Bevezetés

Ez a bevezetés áttekintést ad a C++ programozási nyelv fő fogalmairól, tulajdonságairól és standard (szabvány) könyvtáráról, valamint bemutatja a könyv szerkezetét és elmagyarázza azt a megközelítést, amelyet a nyelv lehetőségeinek és azok használatának leírásánál alkalmaztunk. Ezenkívül a bevezető fejezetek némi háttérinformációt is adnak a C++-ról, annak felépítéséről és felhasználásáról.

Fejezetek

1. Megjegyzések az olvasóhoz
2. Kirándulás a C++-ban
3. Kirándulás a standard könyvtárban

1

Megjegyzések az olvasóhoz

„Szólt a Rozmár:
Van ám elég, miről mesélni jó: ...”
(L. Carroll – ford. Tótfalusi István)

A könyv szerkezete • Hogyan tanuljuk a C++-t? • A C++ jellemzői • Hatékonyság és szerkezet • Filozófiai megjegyzés • Történeti megjegyzés • Mire használjuk a C++-t? • C és C++ • Javaslatok C programozóknak • Gondolatok a C++ programozásról • Tanácsok • Hivatkozások

1.1. A könyv szerkezete

A könyv hat részből áll:

Bevezetés: Az 1–3. fejezetek áttekintik a C++ nyelvet, az általa támogatott fő programozási stílusokat, és a C++ standard könyvtárát.
Első rész: A 4–9. fejezetek oktató jellegű bevezetést adnak a C++ beépített típusairól és az alapszolgáltatásokról, melyekkel ezekből programot építhetünk.
Második rész: A 10–15. fejezetek bevezetést adnak az objektumorientált és az általánosított programozásba a C++ használatával.

Harmadik rész: A 16–22. fejezetek bemutatják a C++ standard könyvtárát.
Negyedik rész: A 23–25. fejezetek tervezési és szoftverfejlesztési kérdéseket tárgyalnak.
Függelékek: Az A–E függelékek a nyelv technikai részleteit tartalmazzák.

Az 1. fejezet áttekintést ad a könyvről, néhány ötletet ad, hogyan használjuk, valamint háttérinformációkat szolgáltat a C++-ról és annak használatáról. Az olvasó bátran átfuthat rajta, elolvashatja, ami érdekesnek látszik, és visszatérhet ide, miután a könyv más részeit elolvasta.

A 2. és 3. fejezet áttekinti a C++ programozási nyelv és a standard könyvtár fő fogalmait és nyelvi alaptulajdonságait, megmutatva, mit lehet kifejezni a teljes C++ nyelvvel. Ha semmi mást nem tesznek, e fejezetek meg kell győzzék az olvasót, hogy a C++ nem (csupán) C, és hogy a C++ hosszú utat tett meg e könyv első és második kiadása óta. A 2. fejezet magas szinten ismertet meg a C++-szal. A figyelmet azokra a nyelvi tulajdonságokra irányítja, melyek támogatják az elvont adatábrázolást, illetve az objektumorientált és az általánosított programozást. A 3. fejezet a standard könyvtár alapelveibe és fő szolgáltatásaiba vezet be, ami lehetővé teszi, hogy a szerző a standard könyvtár szolgáltatásait használhassa a következő fejezetekben, valamint az olvasónak is lehetőséget ad, hogy könyvtári szolgáltatásokat használjon a gyakorlatokhoz és ne kelljen közvetlenül a beépített, alacsony szintű tulajdonságokra hagyatkoznia.

A bevezető fejezetek egy, a könyv folyamán általánosan használt eljárás példáját adják: ahhoz, hogy egy módszert vagy tulajdonságot még közvetlenebb és valószerűbb módon vizsgálhassunk, alkalmanként először röviden bemutatunk egy fogalmat, majd később behatóbban tárgyaljuk azt. Ez a megközelítés lehetővé teszi, hogy konkrét példákat mutassunk be, mielőtt egy témát általánosabban tárgyalnánk. A könyv felépítése így tükrözi azt a megfigyelést, hogy rendszerint úgy tanulunk a legjobban, ha a konkréttól haladunk az elvont felé – még ott is, ahol visszatekintve az elvont egyszerűnek és magától értetődőnek látszik.

Az I. rész a C++-nak azt a részhalmazát írja le, mely a C-ben vagy a Pascalban követett hagyományos programozási stílusokat támogatja. Tárgyalja a C++ programokban szereplő alapvető típusokat, kifejezéseket, vezérlési szerkezeteket. A modularitást, mint a névterek, forrásfájlok és a kivételkezelés által támogatott tulajdonságot, szintén tárgyalja. Feltételezzük, hogy az olvasónak már ismerősek az I. fejezetben használt alapvető programozási fogalmak, így például bemutatjuk a C++ lehetőségeit a rekurzió és iteráció kifejezésére, de nem sokáig magyarázzuk, milyen hasznosak ezek.

A II. rész a C++ új típusok létrehozását és használatát segítő szolgáltatásait írja le. Itt (10. és 12. fejezet) mutatjuk be a konkrét és absztrakt osztályokat (felületeket), az operátor-túlterheléssel (11. fejezet), a többalakúsággal (polimorfizmussal) és az osztályhierarchiák hasz-

nálatával (12. és 15. fejezet) együtt. A 13. fejezet a sablonokat (template) mutatja be, vagyis a C++ lehetőségeit a típus- és függvénycsaládok létrehozására, valamint szemlélteti a tárolók előállítására (pl. listák), valamint az általánosított (generikus) programozás támogatására használt alapvető eljárásokat. A 14. fejezet a kivételkezelést, a hibakezelési módszereket tárgyalja és a hibatűrés biztosításához ad irányelveket. Feltételezzük, hogy az olvasó az objektumorientált és az általánosított programozást nem ismeri jól, illetve hasznát látná egy magyarázatnak, hogyan támogatja a C++ a fő elvonatkoztatási (absztrakciós) eljárásokat. Így tehát nemcsak bemutatjuk az elvonatkoztatási módszereket támogató nyelvi tulajdonságokat, hanem magukat az eljárásokat is elmagyarázzuk. A IV. rész ebben az irányban halad tovább.

A III. rész a C++ standard könyvtárát mutatja be. Célja: megértetni, hogyan használjuk a könyvtárat; általános tervezési és programozási módszereket szemléltetni és megmutatni, hogyan bővítsük a könyvtárat. A könyvtár gondoskodik tárolókról (konténerek – *list*, *vector*, *map*, 18. és 19. fejezet), szabványos algoritmusokról (*sort*, *find*, *merge*, 18. és 19. fejezet), karakterlánc-típusokról és -műveletekről (20. fejezet), a bemenet és kimenet kezeléséről (input/output, 21. fejezet), valamint a számokkal végzett műveletek („numerikus számítás") támogatásáról (22. fejezet).

A IV. rész olyan kérdéseket vizsgál, melyek akkor merülnek fel, amikor nagy szoftverrendszerek tervezésénél és kivitelezésénél a C++-t használjuk. A 23. fejezet tervezési és vezetési kérdésekkel foglalkozik. A 24. fejezet a C++ programozási nyelv és a tervezési kérdések kapcsolatát vizsgálja, míg a 25. fejezet az osztályok használatát mutatja be a tervezésben.

Az „A" függelék a C++ nyelvtana, néhány jegyzettel. A „B" függelék a C és a C++ közti és a szabványos C++ (más néven ISO C++, ANSI C++) illetve az azt megelőző C++-változatok közti rokonságot vizsgálja. A „C" függelék néhány nyelvtechnikai példát mutat be, A „D" függelék pedig a kulturális eltérések kezelését támogató standard könyvtárbeli elemeket mutatja be. Az „E" függelék a standard könyvtár kivételkezeléssel kapcsolatos garanciáit és követelményeit tárgyalja.

1.1.1. Példák és hivatkozások

Könyvünk az algoritmusok írása helyett a program felépítésére fekteti a hangsúlyt. Következésképpen elkerüli a ravasz vagy nehezebben érthető algoritmusokat. Egy egyszerű eljárás alkalmasabb az egyes fogalmak vagy a programszerkezet egy szempontjának szemléltetésére. Például Shell rendezést használ, ahol a valódi kódban jobb lenne gyorsrendezést *(quicksort)* használni. Gyakran jó gyakorlat lehet a kód újraírása egy alkalmasabb algoritmussal. A valódi kódban általában jobb egy könyvtári függvény hívása, mint a könyvben használt, a nyelvi tulajdonságok szemléltetésére használt kód.

A tankönyvi példák szükségszerűen egyoldalú képet adnak a programfejlesztésről. Tisztázva és egyszerűsítve a példákat a felmerült bonyolultságok eltűnnek. Nincs, ami helyettesítené a valódi programok írását, ha benyomást akarunk kapni, igazából milyen is a programozás és egy programozási nyelv. Ez a könyv a nyelvi tulajdonságokra és az alapvető eljárásokra összpontosít, amelyekből minden program összetevődik, valamint az összeépítés szabályaira.

A példák megválasztása tükrözi fordítóprogramokkal, alapkönyvtárakkal, szimulációkkal jellemezhető hátteremet. A példák egyszerűsített változatai a valódi kódban találhatóknak. Egyszerűsítésre van szükség, hogy a programozási nyelv és a tervezés lényeges szempontjai el ne vesszenek a részletekben. Nincs „ügyes” példa, amelynek nincs megfelelője a valódi kódban. Ahol csak lehetséges, a „C” függelékben lévő nyelvtechnikai példákat olyan alakra hoztam, ahol a változók *x* és *y*, a típusok A és *B*, a függvények *f()* és *g()* nevűek.

A kódpéldákban az azonosítókhoz változó szélességű betűket használunk. Például:

```
#include<iostream>

int main()
{
   std::cout << "Helló, világ!\n";
}
```

Első látásra ez „természetellenesnek” tűnhet a programozók számára, akik hozzászoktak, hogy a kód állandó szélességű betűkkel jelenik meg. A változó szélességű betűket általában jobbnak tartják szöveghez, mint az állandó szélességűt. A változó szélességű betűk használata azt is lehetővé teszi, hogy a kódban kevesebb legyen a logikátlan sortörés. Ezenkívül saját kísérleteim azt mutatják, hogy a legtöbb ember kis idő elteltével könnyebben olvashatónak tartja az új stílust.

Ahol lehetséges, a C++ nyelv és könyvtár tulajdonságait a kézikönyvek száraz bemutatási módja helyett a felhasználási környezetben mutatjuk be. A bemutatott nyelvi tulajdonságok és leírásuk részletessége a szerző nézetét tükrözik, aki a legfontosabb kérdésnek a következőt tartja: mi szükséges a C++ hatékony használatához? A nyelv teljes leírása – a könnyebb megközelítés céljából jegyzetekkel ellátva – a *The Annotated C++ Language Standard* című kézikönyvben található, mely Andrew Koenig és a szerző műve. Logikusan kellene hogy legyen egy másik kézikönyv is, a *The Annotated C++ Standard Library*. Mivel azonban mind az idő, mind írási kapacitásom véges, nem tudom megígérni, hogy elkészül.

A könyv egyes részeire való hivatkozások §2.3.4 (2. fejezet, 3.szakasz, 4. bekezdés), §B.5.6 („B” függelék, 5.6. bekezdés és §6.[10](6. fejezet, 10. gyakorlat) alakban jelennek meg. A dőlt betűket kiemelésre használjuk (pl. „egy karakterlánc-literál *nem* fogadható el”), fon-

tos fogalmak első megjelenésénél (pl. *többalakúság*), a C++ nyelv egyes szimbólumainál (pl. *for* utasítás), az azonosítóknál és kulcsszavaknál, illetve a kódpéldákban lévő megjegyzéseknél.

1.1.2. Gyakorlatok

Az egyes fejezetek végén gyakorlatok találhatók. A gyakorlatok főleg az „írj egy programot" típusba sorolhatók. Mindig annyi kódot írjunk, ami elég ahhoz, hogy a megoldás fordítható és korlátozott körülmények között futtatható legyen. A gyakorlatok nehézségben jelentősen eltérőek, ezért becsült nehézségi fokukat megjelöltük. A nehézség hatványozottan nő, tehát ha egy (*1) gyakorlat 10 percet igényel, egy (*2) gyakorlat egy órába, míg egy (*3) egy napba kerülhet. Egy program megírása és ellenőrzése inkább függ az ember tapasztaltságától, mint magától a gyakorlattól.

1.1.3. Megjegyzés az egyes C++-változatokhoz

A könyvben használt nyelv „tiszta C++", ahogyan a C++ szabványban leírták [C++, 1998]. Ezért a példáknak futniuk kell minden C++-változaton. A könyvben szereplő nagyobb programrészleteket több környezetben is kipróbáltuk, azok a példák azonban, melyek a C++-ba csak nemrégiben beépített tulajdonságokat használnak fel, nem mindenhol fordíthatók le. (Azt nem érdemes megemlíteni, mely változatokon mely példákat nem sikerült lefordítani. Az ilyen információk hamar elavulnak, mert a megvalósításon igyekvő programozók keményen dolgoznak azon, hogy nyelvi változataik helyesen fogadjanak el minden C++ tulajdonságot.) A „B" függelékben javaslatok találhatók, hogyan birkózzunk meg a régi C++ fordítókkal és a C fordítókra írott kóddal.

1.2. Hogyan tanuljuk a C++-t?

A C++ tanulásakor a legfontosabb, hogy a fogalmakra összpontosítsunk és ne vesszünk el a részletekben. A programozási nyelvek tanulásának célja az, hogy jobb programozóvá váljunk; vagyis hatékonyabbak legyünk új rendszerek tervezésénél, megvalósításánál és régi rendszerek karbantartásánál. Ehhez sokkal fontosabb a programozási és tervezési módszerek felfedezése, mint a részletek megértése; az utóbbi idővel és gyakorlattal megszerezhető.

A C++ sokféle programozási stílust támogat. Ezek mind az erős statikus típusellenőrzésen alapulnak és legtöbbjük a magas elvonatkoztatási szint elérésére és a programozó elképzeléseinek közvetlen leképezésére irányul. Minden stílus el tudja érni a célját, miközben hatékony marad futási idő és helyfoglalás tekintetében. Egy más nyelvet (mondjuk C, Fortran, Smalltalk, Lisp, ML, Ada, Eiffel, Pascal vagy Modula-2) használó programozó észre kell hogy vegye, hogy a C++ előnyeinek kiaknázásához időt kell szánnia a C++ programozási stílusok és módszerek megtanulására és megemésztésére. Ugyanez érvényes azon programozókra is, akik a C++ egy régebbi, kevésbé kifejezőképes változatát használták.

Ha gondolkodás nélkül alkalmazzuk az egyik nyelvben hatékony eljárást egy másik nyelvben, rendszerint nehézkes, gyenge teljesítményű és nehezen módosítható kódot kapunk. Az ilyen kód írása is csalódást okoz, mivel minden sor kód és minden fordítási hiba arra emlékeztet, hogy a nyelv, amit használunk, más, mint „a régi nyelv". Írhatunk Fortran, C, Smalltalk stb. stílusban bármely nyelven, de ez egy más filozófiájú nyelvben nem lesz sem kellemes, sem gazdaságos. Minden nyelv gazdag forrása lehet az ötleteknek, hogyan írjunk C++ programot. Az ötleteket azonban a C++ általános szerkezetéhez és típusrendszeréhez kell igazítani, hogy hatékony legyen az eltérő környezetben. Egy nyelv alaptípusai felett csak pürroszi győzelmet arathatunk.

A C++ támogatja a fokozatos tanulást. Az, hogy hogyan közelítsünk egy új nyelv tanulásához, attól függ, mit tudunk már és mit akarunk még megtanulni. Nem létezik egyetlen megközelítés sem, amely mindenkinek jó lenne. A szerző feltételezi, hogy az olvasó azért tanulja a C++-t, hogy jobb programozó és tervező legyen. Vagyis nem egyszerűen egy új nyelvtant akar megtanulni, mellyel a régi megszokott módon végzi a dolgokat, hanem új és jobb rendszerépítési módszereket akar elsajátítani. Ezt fokozatosan kell csinálni, mert minden új képesség megszerzése időt és gyakorlást igényel. Gondoljuk meg, mennyi időbe kerülne jól megtanulni egy új természetes nyelvet vagy megtanulni jól játszani egy hangszeren. Könnyen és gyorsan lehetünk jobb rendszertervezők, de nem annyival könnyebben és gyorsabban, mint ahogy azt a legtöbben szeretnénk.

Következésképpen a C++-t – gyakran valódi rendszerek építésére – már azelőtt használni fogjuk, mielőtt megértenénk minden nyelvi tulajdonságot és eljárást. A C++ – azáltal, hogy több programozási modellt is támogat (2. fejezet) – különböző szintű szakértelem esetén is támogatja a termékeny programozást. Minden új programozási stílus újabb eszközt ad eszköztárunkhoz, de mindegyik magában is hatékony és mindegyik fokozza a programozói hatékonyságot. A C++-t úgy alkották meg, hogy a fogalmakat nagyjából sorban egymás után tanulhassuk meg és eközben gyakorlati haszonra tehessünk szert. Ez fontos, mert a haszon a kifejtett erőfeszítéssel arányos.

A folytatódó vita során – kell-e C-t tanulni a C++-szal való ismerkedés előtt – szilárd meggyőződésemmé vált, hogy legjobb közvetlenül a C++-ra áttérni. A C++ biztonságosabb, kifejezőbb, csökkenti annak szükségét, hogy a figyelmet alacsonyszintű eljárásokra irányítsuk. Könnyebb a C-ben a magasabb szintű lehetőségek hiányát pótló trükkösebb részeket megtanulni, ha előbb megismertük a C és a C++ közös részhalmazát és a C++ által közvetlenül támogatott magasabb szintű eljárásokat. A „B" függelék vezérfonalat ad azoknak a programozóknak, akik a C++ ismeretében váltanak a C-re, például azért, hogy régebbi kódot kezeljenek.

Több egymástól függetlenül fejlesztett és terjesztett C++-változat létezik. Gazdag választék kapható eszköztárakból, könyvtárakból, programfejlesztő környezetekből is. Rengeteg tankönyv, kézikönyv, folyóirat, elektronikus hirdetőtábla, konferencia, tanfolyam áll rendelkezésünkre a C++ legfrissebb fejlesztéseiről, használatáról, segédeszközeiről, könyvtárairól, megvalósításairól és így tovább. Ha az olvasó komolyan akarja a C++-t használni, tanácsos az ilyen források között is böngészni. Mindegyiknek megvan a saját nézőpontja, elfogultsága, ezért használjunk legalább kettőt közülük. Például lásd [Barton,1994], [Booch,1994], [Henricson, 1997], [Koenig, 1997], [Martin, 1995].

1.3. A C++ jellemzői

Az egyszerűség fontos tervezési feltétel volt; ahol választani lehetett, hogy a nyelvet vagy a fordítót egyszerűsítsük-e, az előbbit választottuk. Mindenesetre nagy súlyt fektettünk arra, hogy megmaradjon a C-vel való összeegyeztethetőség, ami eleve kizárta a C nyelvtan kisöprését.

A C++-nak nincsenek beépített magasszintű adattípusai, sem magasszintű alapműveletei. A C++-ban például nincs mátrixtípus inverzió operátorral, karakterlánc-típus összefűző művelettel. Ha a felhasználónak ilyen típusra van szüksége, magában a nyelvben definiálhat ilyet. Alapjában véve a C++-ban a legelemibb programozási tevékenység az általános célú vagy alkalmazásfüggő típusok létrehozása. Egy jól megtervezett felhasználói típus a beépített típusoktól csak abban különbözik, milyen módon határozták meg, abban nem, hogyan használják. A III. részben leírt standard könyvtár számos példát ad az ilyen típusokra és használatukra. A felhasználó szempontjából kevés a különbség egy beépített és egy standard könyvtárbeli típus között.

A C++-ban kerültük az olyan tulajdonságokat, melyek akkor is a futási idő növekedését vagy a tár túlterhelését okoznák, ha nem használjuk azokat. Nem megengedettek például azok a szerkezetek, melyek „háztartási információ" tárolását tennék szükségessé minden objektumban, így ha a felhasználó például két 16 bites mennyiségből álló szerkezetet ad meg, az egy 32 bites regiszterbe tökéletesen belefér.

A C++-t hagyományos fordítási és futási környezetben való használatra tervezték, vagyis a UNIX rendszer C programozási környezetére. Szerencsére a C++ sohasem volt a UNIX-ra korlátozva, a UNIX-ot és a C-t csupán modellként használtuk a nyelv, a könyvtárak, a fordítók, a szerkesztők, a futtatási környezetek stb. rokonsága alapján. Ez a minimális modell segítette a C++ sikeres elterjedését lényegében minden számítógépes platformon. Jó okai vannak azonban a C++ használatának olyan környezetekben, melyek jelentősen nagyobb támogatásról gondoskodnak. Az olyan szolgáltatások, mint a dinamikus betöltés, a fokozatos fordítás vagy a típusmeghatározások adatbázisa, anélkül is jól használhatók, hogy befolyásolnák a nyelvet.

A C++ típusellenőrzési és adatrejtési tulajdonságai a programok fordítási idő alatti elemzésére támaszkodnak, hogy elkerüljék a véletlen adatsérüléseket. Nem gondoskodnak titkosításról vagy az olyan személyek elleni védelemről, akik szándékosan megszegik a szabályokat. Viszont szabadon használhatók és nem járnak a futási idő vagy a szükséges tárhely növekedésével. Az alapelv az, hogy ahhoz, hogy egy nyelvi tulajdonság hasznos legyen, nemcsak elegánsnak, hanem valódi programon belül is elhelyezhetőnek kell lennie.

A C++ jellemzőinek rendszerezett és részletes leírását lásd [Stroustrup, 1994].

1.3.1. Hatékonyság és szerkezet

A C++-t a C programozási nyelvből fejlesztettük ki és – néhány kivételtől eltekintve – a C-t, mint részhalmazt, megtartotta. Az alapnyelvet, a C++ C részhalmazát, úgy terveztük, hogy nagyon szoros megfelelés van típusai, műveletei, utasításai, és a számítógépek által közvetlenül kezelhető objektumok (számok, karakterek és címek) között. A *new*, *delete*, *typeid*, *dynamic_cast* és *throw* operátorok és – a *try* blokk – kivételével, az egyes C++ kifejezések és utasítások nem kívánnak futási idejű támogatást.

A C++ ugyanolyan függvényhívási és visszatérési módokat használhat, mint a C – vagy még hatékonyabbakat. Amikor még az ilyen, viszonylag hatékony eljárások is túl költségesek, a C++ függvényt a fordítóval kifejtethetjük helyben (inline kód), így élvezhetjük a függvények használatának kényelmét, a futási idő növelése nélkül.

A C egyik eredeti célja az assembly kód helyettesítése volt a legigényesebb rendszerprogramozási feladatokban. Amikor a C++-t terveztük, vigyáztunk, ne legyen megalkuvás e téren. A C és a C++ közti különbség elsősorban a típusokra és adatszerkezetekre fektetett súly mértékében van. A C kifejező és elnéző. A C++ még kifejezőbb. Ezért a jobb kifejezőképességért cserébe azonban nagyobb figyelmet kell fordítanunk az objektumok típusára. A fordító az objektumok típusának ismeretében helyesen tudja kezelni a kifejezéseket akkor is, ha egyébként kínos precizitással kellett volna megadni a műveleteket. Az objektumok típusának ismerete arra is képessé teszi a fordítót, hogy olyan hibákat fedjen fel, melyek máskülönben egészen a tesztelésig vagy még tovább megmaradnának. Vegyük észre, hogy a típusrendszer használata függvényparaméterek ellenőrzésére – az adatok véletlen sérüléstől való megvédésére, új típusok vagy operátorok előállítására és így tovább – a C++-ban nem növeli a futási időt vagy a szükséges helyet.

A C++-ban a szerkezetre fektetett hangsúly tükrözi a C megtervezése óta megírt programok „súlygyarapodását". Egy kis – mondjuk 1000 soros – programot megírhatunk „nyers erővel", még akkor is, ha felrúgjuk a jó stílus minden szabályát. Nagyobb programoknál ez egyszerűen nincs így. Ha egy 100 000 soros programnak rossz a felépítése, azt fogjuk találni, hogy ugyanolyan gyorsan keletkeznek az újabb hibák, mint ahogy a régieket eltávolítjuk. A C++-t úgy terveztük, hogy lehetővé tegye nagyobb programok ésszerű módon való felépítését, így egyetlen személy is sokkal nagyobb kódmennyiséggel képes megbirkózni. Ezenkívül célkitűzés volt, hogy egy átlagos sornyi C++ kód sokkal többet fejezzen ki, mint egy átlagos Pascal vagy C kódsor. A C++ mostanra megmutatta, hogy túl is teljesíti ezeket a célkitűzéseket.

Nem minden kódrészlet lehet jól szerkesztett, hardverfüggetlen vagy könnyen olvasható. A C++-nak vannak tulajdonságai, melyeket arra szántak, hogy közvetlen és hatékony módon kezelhessük a hardver szolgáltatásait, anélkül, hogy a biztonságra vagy az érthetőségre káros hatással lennénk. Vannak olyan lehetőségei is, melyekkel az ilyen kód elegáns és biztonságos felületek mögé rejthető.

A C++ nagyobb programokhoz való használata természetszerűen elvezet a C++ nyelv programozócsoportok általi használatához. A C++ által a modularitásra, az erősen típusos felületekre és a rugalmasságra fektetett hangsúly itt fizetődik ki. A C++-nak éppen olyan jól kiegyensúlyozott szolgáltatásai vannak nagy programok írására, mint bármely nyelvnek. Ahogy nagyobbak lesznek a programok, a fejlesztésükkel és fenntartásukkal, módosításukkal kapcsolatos problémák a „nyelvi probléma" jellegtől az eszközök és a kezelés általánosabb problémái felé mozdulnak el. A IV. rész ilyen jellegű kérdéseket is tárgyal.

Könyvünk kiemeli az általános célú szolgáltatások, típusok és könyvtárak készítésének módjait. Ezek éppúgy szolgálják a kis programok íróit, mint a nagy programokéit. Ezen túlmenően, mivel minden bonyolultabb program sok, félig-meddig független részből áll, az ilyen részek írásához szükséges módszerek ismerete jó szolgálatot tesz minden alkalmazásprogramozónak.

Az olvasó azt gondolhatja, a részletesebb típusszerkezetek használata nagyobb forrásprogramhoz vezet. A C++ esetében ez nem így van. Egy C++ program, amely függvényparaméter-típusokat vezet be vagy osztályokat használ, rendszerint kissé rövidebb, mint a vele egyenértékű C program, amely nem használja e lehetőségeket. Ott, ahol könyvtárakat használnak, egy C++ program sokkal rövidebb lesz, mint a megfelelő C program, feltéve természetesen, hogy készíthető működőképes C-beli megfelelő.

1.3.2. Filozófiai megjegyzés

A programozási nyelvek két rokon célt szolgálnak: a programozónak részben eszközt adnak, amellyel végrehajtható műveleteket adhat meg, ugyanakkor egy sereg fogódzót is rendelkezésére bocsátanak, amikor arról gondolkodik, mit lehet tenni. Az első cél ideális esetben „gépközeli" nyelvet kíván, amellyel a számítógép minden fontos oldala egyszerűen és hatékonyan kezelhető, a programozó számára ésszerű, kézenfekvő módon. A C nyelvet elsősorban ebben a szellemben tervezték. A második cél viszont olyan nyelvet követel meg, mely „közel van a megoldandó problémához", hogy a megoldás közvetlenül és tömören kifejezhető legyen.

A nyelv, melyben gondolkodunk/programozunk és a problémák, megoldások, melyeket el tudunk képzelni, szoros kapcsolatban állnak egymással. Ezért a nyelvi tulajdonságok megszorítása azzal a szándékkal, hogy kiküszöböljük a programozói hibákat, a legjobb esetben is veszélyes. A természetes nyelvekhez hasonlóan nagy előnye van annak, ha az ember legalább két nyelvet ismer. A nyelv ellátja a programozót a megfelelő eszközökkel, ha azonban ezek nem megfelelőek a feladathoz, egyszerűen figyelmen kívül hagyjuk azokat. A jó tervezés és hibamentesség nem biztosítható csupán az egyedi nyelvi tulajdonságok jelenlétével vagy távollétével.

A típusrendszer különösen összetettebb feladatok esetében jelent segítséget. A C++ osztályai valóban erős eszköznek bizonyultak.

1.4. Történeti megjegyzés

A szerző alkotta meg a C++-t, írta meg első definícióit, és készítette el első változatát. Megválasztotta és megfogalmazta a C++ tervezési feltételeit, megtervezte fő szolgáltatásait, és ő volt a felelős a C++ szabványügyi bizottságban a bővítési javaslatok feldolgozásáért.

Világos, hogy a C++ sokat köszönhet a C-nek [Kernighan, 1978]. A C néhány, a típusellenőrzés terén tapasztalt hiányosságát kivéve megmaradt, részhalmazként (lásd „B" függelék). Ugyancsak megmaradt az a C-beli szándék, hogy olyan szolgáltatásokra fektessen hangsúlyt, melyek elég alacsony szintűek ahhoz, hogy megbirkózzanak a legigényesebb rendszerprogramozási feladatokkal is. A C a maga részéről sokat köszönhet ősének, a BCPL-nek [Richards, 1980]; a BCPL *//* megjegyzés-formátuma (újra) be is került a C++-ba. A C++ másik fontos forrása a Simula67 volt [Dahl, 1970] [Dahl, 1972]; az osztály fogalmát (a származtatott osztályokkal és virtuális függvényekkel) innen vettem át. A C++ operátor-túlterhelési lehetősége és a deklarációk szabad elhelyezése az utasítások között az Algol68-ra emlékeztet [Woodward, 1974].

A könyv eredeti kiadása óta a nyelv kiterjedt felülvizsgálatokon és finomításokon ment keresztül. A felülvizsgálatok fő területe a túlterhelés feloldása, az összeszerkesztési és tárkezelési lehetőségek voltak. Ezenkívül számos kisebb változtatás történt a C-vel való kompatibilitás növelésére. Számos általánosítás és néhány nagy bővítés is belekerült: ezek a többszörös öröklés, a *static* és *const* tagfüggvények, a *protected* tagok, a sablonok, a kivételkezelés, a futási idejű típusazonosítás és a névterek. E bővítések és felülvizsgálatok átfogó feladata a C++ olyan nyelvvé fejlesztése volt, mellyel jobban lehet könyvtárakat írni és használni. A C++ fejlődésének leírását lásd [Stroustrup, 1994].

A sablonok (template) bevezetésének elsődleges célja a statikus típusú tárolók (konténerek – *list*, *vector*, *map*) és azok hatékony használatának (általánosított vagy generikus programozás) támogatása, valamint a makrók és explicit típuskényszerítések (casting) szükségének csökkentése volt. Inspirációt az Ada általánosító eszközei (mind azok erősségei, illetve gyengeségei), valamint részben a Clu paraméteres moduljai szolgáltattak. Hasonlóan, a C++ kivételkezelési eljárásainak elődjei is többé-kevésbé az Ada [Ichbiah, 1979], a Clu [Liskov, 1979] és az ML [Wikstrm, 1987]. Az 1985-1995 között bevezetett egyéb fejlesztések – többszörös öröklés, tisztán virtuális függvények és névterek – viszont nem annyira más nyelvekből merített ötletek alapján születtek, inkább a C++ használatának tapasztalataiból leszűrt általánosítások eredményei.

A nyelv korábbi változatait (összefoglaló néven az *osztályokkal bővített C*-t [Stroustrup, 1994]) 1980 óta használják. Kifejlesztésében eredetileg szerepet játszott, hogy olyan eseményvezérelt szimulációkat szerettem volna írni, melyekhez a Simula67 ideális lett volna,

ha eléggé hatékony. Az „osztályokkal bővített C" igazi területét a nagy programok jelentették, ahol a lehető leggyorsabbnak kell lenni és a lehető legkevesebb helyet foglalni. Az első változatokból még hiányzott az operátor-túlterhelés, valamint hiányoztak a referenciák, a virtuális függvények, a sablonok, a kivételek és sok egyéb. A C++-t nem kísérleti körülmények között először 1983-ban használták.

A C++ nevet Rick Mascitti adta a nyelvnek az említett év nyarán. A név kifejezi mindazt a forradalmi újítást, amit az új nyelv a C-hez képest hozott: a ++ a C növelő műveleti jele. (A „C+"-t is használják, de az egy másik, független nyelv.) A C utasításformáit jól ismerők rámutathatnak, hogy a „C++" kifejezés nem olyan „erős", mint a „++C". Mindazonáltal a nyelv neve nem is D, hiszen a C-nek csupán bővítéséről van szó, amely az ott felmerült problémák elhárításához az eredeti nyelv szolgáltatásai közül egyet sem vet el. A C++ név más megközelítésű elemzéséhez lásd [Orwell, 1949, függelék].

A C++ megalkotásának fő oka azonban az volt, hogy barátaimmal együtt nem szerettünk volna assembly, C vagy más modern, magas szintű nyelven programozni. Csak annyit akartunk elérni, hogy könnyebben és élvezetesebben írhassunk jól használható programokat. Kezdetben nem vetettük papírra rendszerezetten a fejlesztési terveket: egyszerre terveztünk, dokumentáltunk és alkottunk. Nem volt „C++ projekt" vagy „C++ tervezőbizottság". A C++ a felhasználók tapasztalatai és a barátaimmal, munkatársaimmal folytatott viták során fejlődött ki.

A C++ későbbi robbanásszerű elterjedése szükségszerűen változásokat hozott magával. Valamikor 1987-ben nyilvánvalóvá vált, hogy a C++ hivatalos szabványosítása immár elkerülhetetlen és haladéktalanul meg kell kezdenünk az ilyen irányú munka előkészítését [Stroustrup, 1994]. Folyamatosan próbáltuk tartani a kapcsolatot mind hagyományos, mind elektronikus levélben, illetve személyesen, konferenciákat tartva a különböző C++ fordítók készítőivel és a nyelv fő felhasználóival.

Ebben a munkában nagy segítséget nyújtott az AT&T Bell Laboratories, lehetővé téve, hogy vázlataimat és a C++ hivatkozási kézikönyv újabb és újabb változatait megoszthassam a fejlesztőkkel és felhasználókkal. Segítségük nem alábecsülendő, ha tudjuk, hogy az említettek nagy része olyan vállalatoknál dolgozott, amelyek az AT&T vetélytársainak tekinthetők. Egy kevésbé „felvilágosult" cég komoly problémákat okozhatott volna és a nyelv „tájszólásokra" töredezését idézte volna elő, pusztán azáltal, hogy nem tesz semmit. Szerencsére a tucatnyi cégnél dolgozó mintegy száz közreműködő elolvasta és megjegyzésekkel látta el a vázlatokat, melyekből az általánosan elfogadott hivatkozási kézikönyv és a szabványos ANSI C++ alapdokumentuma megszületett. A munkát segítők neve megtalálható a *The Annotated C++ Reference Manual*-ban [Ellis, 1989]. Végül az ANSI X3J16 bizottsága a Hewlett-Packard kezdeményezésére 1989 decemberében összeült, 1991 júniusában pedig már

annak örülhettünk, hogy az ANSI (az amerikai nemzeti szabvány) C++ az ISO (nemzetközi) C++ szabványosítási kezdeményezés részévé vált. 1990-től ezek a szabványügyi bizottságok váltak a nyelv fejlesztésének és pontos körülhatárolásának fő fórumaivá. Magam mindvégig részt vettem e bizottságok munkájában; a bővítményekkel foglalkozó munkacsoport elnökeként közvetlenül feleltem a C++-t érintő lényegbevágó módosítási javaslatok és az új szolgáltatások bevezetését szorgalmazó kérelmek elbírálásáért. Az első szabványvázlat 1995 áprilisában került a nagyközönség elé, a végleges ISO C++ szabványt (ISO/IEC 14882) pedig 1998-ban fogadták el.

A könyvben bemutatott kulcsfontosságú osztályok némelyike a C++-szal párhuzamosan fejlődött. A *complex*, *vector* és *stack* osztályokat például az operátor-túlterhelési eljárásokkal egyidőben dolgoztam ki. A karakterlánc- és listaosztályokat *(string, list)* Jonathan Shopironak köszönhetjük (azért én is közreműködtem). Jonathan hasonló osztályai voltak az elsők, amelyeket egy könyvtár részeként széles körben használtak; ezekből a régi kísérletekből fejlesztettük ki a C++ standard könyvtárának *string* osztályát. A [Stroustrup, 1987] és a §12.7[11] által leírt *task* könyvtár egyike volt az „osztályokkal bővített C" nyelven először írt programoknak. (A könyvtárat és a kapcsolódó osztályokat én írtam a Simula stílusú szimulációk támogatásához.) A könyvtárat később Jonathan Shopiro átdolgozta, és még ma is használják. Az első kiadásban leírt *stream* könyvtárat én terveztem és készítettem el, Jerry Schwarz pedig Andrew Koenig formázó eljárása (§21.4.6) és más ötletek felhasználásával az e könyv 21. fejezetében bemutatandó *iostreams* könyvtárrá alakította. A szabványosítás során a könyvtár további finomításon esett át; a munka dandárját Jerry Schwarz, Nathan Myers és Norihiro Kumagai végezték. A sablonok lehetőségeit az Andrew Koenig, Alex Stepanov, személyem és mások által tervezett *vector*, *map*, *list* és *sort* sablonok alapján dolgoztuk ki. Alex Stepanovnak a sablonokkal történő általánosított programozás terén végzett munkája emellett elvezetett a tárolók bevezetéséhez és a C++ standard könyvtárának egyes algoritmusaihoz is (§16.3, 17. fejezet, 18. fejezet §19.2). A számokkal végzett műveletek *valarray* könyvtára (22. fejezet) nagyrészt Kent Budge munkája.

1.5. A C++ használata

A C++-t programozók százezrei használják, lényegében minden alkalmazási területen. Ezt a használatot támogatja tucatnyi független megvalósítás, többszáz könyvtár és tankönyv, számos műszaki folyóirat, konferencia, és számtalan konzultáns. Oktatás és képzés minden szinten, széles körben elérhető.

A régebbi alkalmazások erősen a rendszerprogramozás felé hajlottak. Több nagy operációs rendszer íródott C++-ban: [Campbell, 1987] [Rozier, 1988] [Hamilton, 1993] [Berg, 1995] [Parrington, 1995] és sokan mások kulcsfontosságú részeket írtak. A szerző lényegesnek tekinti a C++ engedmény nélküli gépközeliségét, ami lehetővé teszi, hogy C++-ban írhassunk eszközmeghajtókat és más olyan programokat, melyek valósidejű, közvetlen hardverkezelésre támaszkodnak. Az ilyen kódban a működés kiszámíthatósága legalább annyira fontos, mint a sebesség és gyakran így van az eredményül kapott rendszer tömörségével is. A C++-t úgy terveztük, hogy minden nyelvi tulajdonság használható legyen a komoly időbeli és helyfoglalásbeli megszorításoknak kitett kódban is. [Stroustrup, 1994, §4.5].

A legtöbb programban vannak kódrészletek, melyek létfontosságúak az elfogadható teljesítmény tekintetében. A kód nagyobb részét azonban nem ilyen részek alkotják. A legtöbb kódnál a módosíthatóság, a könnyű bővíthetőség és tesztelhetőség a kulcskérdés. A C++ ilyen téren nyújtott támogatása vezetett el széleskörű használatához ott, ahol kötelező a megbízhatóság, és ahol az idő haladtával jelentősen változnak a követelmények. Példaként a bankok, a kereskedelem, a biztosítási szféra, a távközlés és a katonai alkalmazások szolgálhatnak. Az USA távolsági telefonrendszere évek óta a C++-ra támaszkodik és minden 800-as hívást (vagyis olyan hívást, ahol a hívott fél fizet) C++ program irányít [Kamath, 1993]. Számos ilyen program nagy méretű és hosszú életű. Ennek eredményképpen a stabilitás, a kompatibilitás és a méretezhetőség állandó szempontok a C++ fejlesztésében. Nem szokatlanok a millió soros C++ programok.

A C-hez hasonlóan a C++-t sem kifejezetten számokkal végzett műveletekhez tervezték. Mindazonáltal sok számtani, tudományos és mérnöki számítást írtak C++-ban. Ennek fő oka, hogy a számokkal való hagyományos munkát gyakran grafikával és olyan számításokkal kell párosítani, melyek a hagyományos Fortran mintába nem illeszkedő adatszerkezetekre támaszkodnak [Budge, 1992] [Barton, 1994]. A grafika és a felhasználói felület olyan területek, ahol erősen használják a C++-t. Bárki, aki akár egy Apple Macintosht, akár egy Windowst futtató PC-t használt, közvetve a C++-t használta, mert e rendszerek elsődleges felhasználói felületeit C++ programok alkotják. Ezenkívül a UNIX-ban az X-et támogató legnépszerűbb könyvtárak némelyike is C++-ban íródott. Ilyenformán a C++ közösen választott nyelve annak a hatalmas számú alkalmazásnak, ahol a felhasználói felület kiemelt fontosságú.

Mindezen szempontok mellett lehet, hogy a C++ legnagyobb erőssége az a képessége, hogy hatékonyan használható olyan programokhoz, melyek többféle alkalmazási területen igényelnek munkát. Egyszerű olyan alkalmazást találni, melyben LAN és WAN hálózatot, számokkal végzett műveleteket, grafikát, felhasználói kölcsönhatást és adatbázis-hozzáférést használunk. Az ilyen alkalmazási területeket régebben különállóknak tekintették és általában különálló fejlesztőközösségek szolgálták ki, többféle programozási nyelvet használva.

A C++-t széles körben használják oktatásra és kutatásra. Ez néhány embert meglepett, akik – helyesen – rámutattak, hogy a C++ nem a legkisebb és legtisztább nyelv, amelyet valaha terveztek. Mindazonáltal a C++

- elég tiszta ahhoz, hogy az alapfogalmakat sikeresen tanítsuk,
- elég valószerű, hatékony és rugalmas az igényes projektekhez is,
- elérhető olyan szervezetek és együttműködő csoportok számára, melyek eltérő fejlesztési és végrehajtási környezetekre támaszkodnak,
- elég érthető ahhoz, hogy bonyolult fogalmak és módszerek tanításának hordozója legyen és
- elég „kereskedelmi", hogy segítse a tanultak gyakorlatban való felhasználását.

A C++ olyan nyelv, mellyel gyarapodhatunk.

1.6. C és C++

A C++ alapnyelvének a C nyelvet választottuk, mert

- sokoldalú, tömör, és viszonylag alacsony szintű,
- megfelel a legtöbb rendszerprogramozási feladatra,
- mindenütt és mindenen fut, és
- illeszkedik a UNIX programozási környezetbe.

A C-nek megvannak a hibái, de egy újonnan készített nyelvnek is lennének, a C problémáit pedig már ismerjük. Nagy jelentősége van, hogy C-vel való munka vezetett el a hasznos (bár nehézkes) eszközzé váló „osztályokkal bővített C"-hez, amikor először gondoltunk a C bővítésére Simula-szerű osztályokkal.

Ahogy szélesebb körben kezdték használni a C++-t és az általa nyújtott, a C lehetőségeit felülmúló képességek jelentősebbek lettek, újra és újra felmerült a kérdés, megtartsuk-e a két nyelv összeegyeztethetőségét. Világos, hogy néhány probléma elkerülhető lett volna, ha némelyik C örökséget elutasítjuk (lásd pl. [Sethi, 1981]). Ezt nem tettük meg, a következők miatt:

1. Több millió sornyi C kód van, mely élvezheti a C++ előnyeit, feltéve, hogy szükségtelen a C-ről C++-ra való teljes átírás.
2. Több millió sornyi C-ben írt könyvtári függvény és eszközillesztő kód van, melyet C++ programokból/programokban használni lehet, feltéve, hogy a C++ program összeszerkeszthető és formailag összeegyeztethető a C programmal.
3. Programozók százezrei léteznek, akik ismerik a C-t és ezért csak a C++ új tulajdonságait kell megtanulniuk, vagyis nem kell az alapokkal kezdeniük.
4. A C++-t és a C-t ugyanazok, ugyanazokon a rendszereken fogják évekig használni, tehát a különbségek vagy nagyon nagyok, vagy nagyon kicsik lesznek, hogy a hibák és a keveredés lehetősége a lehető legkisebbre csökkenjen.

A C++-t felülvizsgáltuk, hogy biztosítsuk, hogy azon szerkezetek, melyek mind a C-ben, mind a C++-ban megengedettek, mindkét nyelvben ugyanazt jelentsék (§B.2).

A C nyelv maga is fejlődött, részben a C++ fejlesztésének hatására [Rosler, 1984]. Az ANSI C szabvány [C,1990] a függvénydeklarációk formai követelményeit az „osztályokkal bővített C"-ből vette át. Az átvétel mindkét irányban előfordul: a *void** mutatótípust például az ANSI C-hez találták ki, de először a C++-ban valósították meg. Mint ahogy e könyv első kiadásában megígértük, a C++-t felülvizsgáltuk, hogy eltávolítsuk az indokolatlan eltéréseket, így a C++ ma jobban illeszkedik a C-hez, mint eredetileg. Az elképzelés az volt, hogy a C++ olyan közel legyen az ANSI C-hez, amennyire csak lehetséges – de ne közelebb [Koenig, 1989]. A száz százalékos megfelelőség soha nem volt cél, mivel ez megalkuvást jelentene a típusbiztonságban, valamint a felhasználói és beépített típusok zökkenésmentes egyeztetésében.

A C tudása nem előfeltétele a C++ megtanulásának. A C programozás sok olyan módszer és trükk használatára biztat, melyeket a C++ nyelvi tulajdonságai szükségtelenné tettek. Az explicit típuskényszerítés például ritkábban szükséges a C++-ban, mint a C-ben (§1.6.1). A *jó* C programok azonban hajlanak a C++ programok felé. A Kernighan és Ritchie féle *A C programozási nyelv* (Műszaki könyvkiadó, második kiadás, 1994) [Kernighan,1988] című kötetben például minden program C++ program. Bármilyen statikus típusokkal rendelkező nyelvben szerzett tapasztalat segítséget jelent a C++ tanulásánál.

1.6.1. Javaslatok C programozóknak

Minél jobban ismeri valaki a C-t, annál nehezebbnek látja annak elkerülését, hogy C stílusban írjon C++ programot, lemondva ezáltal a C++ előnyeiről. Kérjük, vessen az olvasó egy pillantást a „B" függelékre, mely leírja a C és a C++ közti különbségeket. Íme néhány terület, ahol a C++ fejlettebb, mint a C:

1. A C++-ban a makrókra majdnem soha sincs szükség. A névvel ellátott állandók meghatározására használjunk konstanst (*const*) (§5.4) vagy felsorolást (*enum*) (§4.8), a függvényhívás okozta többletterhelés elkerülésére helyben kifejtett függvényeket (§7.1.1), a függvény- és típuscsaládok leírására *sablonokat* (13. fejezet), a névütközések elkerülésére pedig *névtereket* (§8.2).
2. Ne vezessünk be egy változót, mielőtt szükség van rá, így annak azonnal kezdőértéket is adhatunk. Deklaráció bárhol lehet, ahol utasítás lehet (§6.3.1), így *for* utasítások (§6.3.3) és elágazások feltételeiben (§6.3.2.1) is.
3. Ne használjunk *malloc()*-ot, a *new* operátor (§6.2.6) ugyanazt jobban elvégzi. A *realloc()* helyett próbáljuk meg a *vector*-t (§3.8).
4. Próbáljuk elkerülni a *void** mutatókkal való számításokat, az uniókat és típuskonverziókat (típusátalakításokat), kivéve, ha valamely függvény vagy osztály megvalósításának mélyén találhatók. A legtöbb esetben a típuskonverzió a tervezési hiba jele. Ha feltétlenül erre van szükség, az „új cast-ok" (§6.2.7) egyikét próbáljuk használni szándékunk pontosabb leírásához.
5. Csökkentsük a lehető legkevesebbre a tömbök és a C stílusú karakterláncok használatát. A C++ standard könyvtárának *string* (§3.5) és *vector* (§3.7.1) osztályai a hagyományos C stílushoz képest gyakrabban használhatók a programozás egyszerűbbé tételére. Általában ne próbáljunk magunk építeni olyat, ami megvan a standard könyvtárban.

Ahhoz, hogy eleget tegyünk a C szerkesztési szabályainak, a C++ függvényeket úgy kell megadnunk, hogy szerkesztésük C módú legyen. (§9.2.4). A legfontosabb, hogy úgy próbáljunk egy programot elképzelni, mint egymással kölcsönhatásban lévő fogalmakat, melyeket osztályok és objektumok képviselnek, nem pedig úgy, mint egy halom adatszerkezetet, a bitekkel zsonglőrködő függvényekkel.

1.6.2. Javaslatok C++ programozóknak

Sokan már egy évtized óta használják a C++-t. Még többen használják egyetlen környezetben és tanultak meg együtt élni a korai fordítók és első generációs könyvtárak miatti korlátozásokkal. Ami a tapasztalt C++ programozók figyelmét gyakran elkerüli, nem is annyira az új eszközök megjelenése, mint inkább ezen eszközök kapcsolatainak változása, ami alapjaiban új programozási módszereket követel meg. Más szóval, amire annak idején nem gondoltunk vagy haszontalannak tartottunk, ma már kiváló módszerré válhatott, de ezekre csak az alapok újragondolásával találunk rá.

Olvassuk át a fejezeteket sorban. Ha már ismerjük a fejezet tartalmát, gondolatban ismételjük át. Ha még nem ismerjük, valami olyat is megtanulhatunk, amire eredetileg nem számítottunk. Én magam elég sokat tanultam e könyv megírásából, és az a gyanúm, hogy kevés C++ programozó ismeri az összes itt bemutatott összes eszközt és eljárást. Ahhoz, hogy helyesen használjunk egy nyelvet, behatóan kell ismernünk annak eszközeit, módszereit. Felépítése és példái alapján ez a könyv megfelelő rálátást biztosít.

1.7. Programozási megfontolások a C++-ban

A programtervezést ideális esetben három fokozatban közelítjük meg. Először tisztán érthetővé tesszük a problémát (elemzés, analízis), ezután azonosítjuk a fő fogalmakat, melyek egy megoldásban szerepelnek (tervezés), végül a megoldást egy programban fejezzük ki (programozás). A probléma részletei és a megoldás fogalmai azonban gyakran csak akkor válnak tisztán érthetővé, amikor egy elfogadhatóan futtatható programban akarjuk kifejezni azokat. Ez az, ahol számít, milyen programozási nyelvet választunk.

A legtöbb alkalmazásban vannak fogalmak, melyeket nem könnyű a kapcsolódó adatok nélkül az alaptípusok egyikével vagy függvénnyel ábrázolni. Ha adott egy ilyen fogalom, hozzunk létre egy osztályt, amely a programban képviselni fogja. A C++ osztályai típusok, melyek meghatározzák, hogyan viselkednek az osztályba tartozó objektumok, hogyan jönnek létre, hogyan kezelhetők és hogyan szűnnek meg. Az osztály leírhatja azt is, hogyan jelennek meg az objektumok, bár a programtervezés korai szakaszában ez nem szükségszerűen fő szempont. Jó programok írásánál az a legfontosabb, hogy úgy hozzunk létre osztályokat, hogy mindegyikük egyetlen fogalmat, tisztán ábrázoljon. Ez általában azt jelenti, hogy a következő kérdésekre kell összpontosítani: Hogyan hozzuk létre az osztály objektumait? Másolhatók-e és/vagy megsemmisíthetők-e az osztály objektumai? Milyen műveletek alkalmazhatók az objektumokra? Ha nincsenek jó válaszok e kérdésekre, az a legvalószínűbb, hogy a fogalom nem „tiszta". Ekkor jó ötlet, ha tovább gondolkodunk a problémán és annak javasolt megoldásán, ahelyett, hogy azonnal elkezdenénk a kód kidolgozását.

A legkönnyebben kezelhető fogalmak azok, amelyeknek hagyományos matematikai megfogalmazásuk van: mindenfajta számok, halmazok, geometriai alakzatok stb. A szövegközpontú bemenet és kimenet, a karakterláncok, az alaptárolók, az ezekre a tárolókra alkalmazható alap-algoritmusok, valamint néhány matematikai osztály a C++ standard könyvtárának részét képezik (3. fejezet, §16.1.2). Ezenkívül elképesztő választékban léteznek könyvtárak, melyek általános és részterületekre szakosodott elemeket támogatnak.

Az egyes fogalmak (és a hozzájuk kapcsolódó elemek) nem légüres térben léteznek, mindig rokonfogalmak csoportjába tartoznak. Az osztályok közti kapcsolatok szervezése egy programon belül – vagyis az egy megoldásban szereplő különböző elemek közti pontos kapcsolatok meghatározása – gyakran nehezebb, mint az egyes osztályokat kijelölni. Jobb, ha az eredmény nem rendetlenség, melyben minden osztály függ minden másiktól. Vegyünk két osztályt: A-t és B-t. Az olyan kapcsolatok, mint az „A hív B-beli függvényeket", „A létrehoz B-ket" és „A-nak van egy B tagja" ritkán okoznak nagy problémát, míg az olyanok, mint az „A használ B-beli adatot" rendszerint kiküszöbölhetők.

Az összetettség kezelésének egyik legerősebb eszköze a hierarchikus rendezés, vagyis a rokon elemek faszerkezetbe szervezése, ahol a fa gyökere a legáltalánosabb elem. A C++-ban a származtatott osztályok ilyen fastruktúrákat képviselnek. Egy program gyakran úgy szervezhető, mint fák halmaza, vagy mint osztályok irányított körmentes gráfja. Vagyis a programozó néhány alaposztályt hoz létre, melyekhez saját származtatott osztályaik halmaza tartozik. Az elemek legáltalánosabb változatának (a bázisosztálynak) a kezelését végző műveletek meghatározására a virtuális függvényeket (§2.5.5, §12.2.6) használhatjuk. Szükség esetén ezen műveletek megvalósítása az egyedi esetekben (a származtatott osztályoknál) finomítható.

Néha még az irányított körmentes gráf sem látszik kielégítőnek a programelemek szervezésére; egyes elemek kölcsönös összefüggése öröklöttnek tűnik. Ilyen esetben megpróbáljuk a ciklikus függőségeket behatárolni, hogy azok ne befolyásolják a program átfogó rendszerét. Ha nem tudjuk kiküszöbölni vagy behatárolni az ilyen kölcsönös függéseket, valószínű, hogy olyan gondban vagyunk, melyből nincs programozási nyelv, amely kisegítene. Hacsak ki nem tudunk eszelni könnyen megállapítható kapcsolatokat az alapfogalmak között, valószínű, hogy a program kezelhetetlenné válik. A függőségi gráfok kibogozásának egyik eszköze a felület (interfész) tiszta elkülönítése a megvalósítástól (implementáció). A C++ erre szolgáló legfontosabb eszközei az absztrakt osztályok (§2.5.4, §12.3).

A közös tulajdonságok kifejezésének másik formája a sablon (template, §2.7, 13. fejezet). Az osztálysablonok osztályok családját írják le. Egy listasablon például a „T elemek listáját" határozza meg, ahol „T" bármilyen típus lehet. A sablon tehát azt adja meg, hogyan hozhatunk létre egy típust egy másik típus, mint paraméter átadásával. A legszokásosabb sablonok az olyan tárolóosztályok, mint a listák, tömbök és asszociatív tömbök, valamint az ilyen tárolókat használó alap-algoritmusok. Rendszerint hiba, ha egy osztály és a vele kapcsolatos függvények paraméterezését öröklést használó típussal fejezzük ki. A legjobb sablonokat használni.

Emlékeztetünk arra, hogy sok programozási feladat egyszerűen és tisztán elvégezhető elemi típusok, adatszerkezetek, világos függvények és néhány könyvtári osztály segítségével. Az új típusok leírásában szereplő teljes apparátust nem szabad használni, kivéve, ha valóban szükség van rá.

A „Hogyan írjunk C++-ban jó programot?" nagyon hasonlít a „Hogyan írjunk jó prózát?" kérdésre. Két válasz van: „Tudnunk kell, mit akarunk mondani" és „Gyakoroljunk. Színleljük a jó írást." Mind a kettő éppúgy helytálló a C++, mint bármely természetes nyelv esetében – és tanácsukat éppolyan nehéz követni.

1.8. Tanácsok

Íme néhány „szabály", amelyet figyelembe vehetünk a C++ tanulásakor. Ahogy jártasabbak leszünk, továbbfejleszthetjük ezeket saját programfajtáinkhoz, programozási stílusunkhoz illeszkedően. A szabályok szándékosan nagyon egyszerűek, így nélkülözik a részleteket. Ne vegyük őket túlzottan komolyan: a jó programok írásához elsősorban intelligencia, ízlés, türelem kell. Ezeket nem fogjuk elsőre elsajátítani. Kísérletezzünk!

[1] Amikor programozunk, valamilyen probléma megoldására született ötleteink konkrét megvalósítását hozzuk létre. Tükrözze a program szerkezete olyan közvetlenül ezeket az ötleteket, amennyire csak lehetséges:
   a) Ha valamire úgy gondolunk, mint külön ötletre, tegyük osztállyá.
   b) Ha különálló egyedként gondolunk rá, tegyük egy osztály objektumává.
   c) Ha két osztálynak van közös felülete, tegyük ezt a felületet absztrakt osztállyá.
   d) Ha két osztály megvalósításában van valami közös, tegyük bázisosztállyá e közös tulajdonságokat.
   e) Ha egy osztály objektumok tárolója, tegyük sablonná.
   f) Ha egy függvény egy tároló számára való algoritmust valósít meg, tegyük függvénysablonná, mely egy tárolócsalád algoritmusát írja le.
   g) Ha osztályok, sablonok stb. egy halmazán belül logikai rokonság van, tegyük azokat közös névtérbe.

[2] Ha olyan osztályt hozunk létre, amely nem matematikai egyedet ír le (mint egy mátrix vagy komplex szám) vagy nem alacsonyszintű típust (mint egy láncolt lista)
   a) ne használjunk globális adatokat (használjunk tagokat),
   b) ne használjunk globális függvényeket,

c) ne használjunk nyilvános adattagokat,
d) ne használjunk „barát” (friend) függvényeket, kivéve a) vagy c) elkerülésére,
e) ne tegyünk egy osztályba típusazonosító mezőket, használjunk inkább virtuális függvényeket,
f) ne használjunk helyben kifejtett függvényeket, kivéve ha jelentős optimalizálásról van szó.

Egyedi és részletesebb gyakorlati szabályokat az egyes fejezetek „Tanácsok” részében találhatunk. Emlékeztetjük az olvasót, hogy ezek a tanácsok csak útmutatásul szolgálnak, nem megváltoztathatatlan törvények. A tanácsokat csak ott kövessük, ahol értelme van. Nincs pótszere az intelligenciának, a tapasztalatnak, a józan észnek és a jó ízlésnek.

A „soha ne tegyük ezt” alakú szabályokat haszontalannak tekintem. Következésképpen a legtöbb tanácsot javaslatként fogalmaztam meg; azt írtam le, mit tegyünk. A „negatív javaslatokat” pedig nem úgy kell érteni, mint tiltásokat: nem tudok a C++ olyan fő tulajdonságáról, melyet ne láttam volna jól felhasználni. A „Tanácsok” nem tartalmaznak magyarázatokat. Helyette minden tanács mellett hivatkozás található a könyv megfelelő részére. Ahol negatív tanács szerepel, a hivatkozott rész rendszerint alternatív javaslatot tartalmaz.

1.8.1. Hivatkozások

Barton, 1994 John J. Barton and Lee R. Nackman: Scientific and Engineering C++. Addison-Wesley. Reading, Mass. 1994. ISBN 1-201-53393-6.

Berg, 1995 William Berg, Marshall Cline, and Mike Girou: Lessons Learned from the OS/400 OO Project. CACM. Vol. 38 No. 10. October 1995.

Booch, 1994 Grady Booch: Object-Oriented Analysis and Design. Benjamin/Cummings. Menlo Park, Calif. 1994. ISBN 0-8053-5340-2.

Budge, 1992 Kent Budge, J. S. Perry, an A. C. Robinson: High-Performance Scientific Computation using C++. Proc. USENIX C++Conference. Portland, Oregon. August 1992.

C, 1990 X3 Secretariat: Standard - The C Language. X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.

C++, 1998 X+ Secretariat: International Standard- The C++ Language. X3J16-14882. Information Technology Council (NSITC). Washington, DC, USA.

Campbell, 1987 Roy Campbell, et al.: The Design of a Multirocessor Operating System. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

Coplien, 1995 James O. Coplien and Douglas C. Schmidt (editors): Pattern Languages of Program Design. Addison-Wesley. Reading, Mass. 1995. ISBN 1-201-60734-4.

Dahl, 1970 O-J. Dahl, B. Myrhaug, and K. Nygaard: SIMULA Common Base Language. Norwegian Computing Center S-22. Oslo, Norway. 1970.

Dahl, 1972 O-J. Dahl, and C. A. R. Hoare: Hierarchical Program Consturction in Structured Programming. Academic Press, New York. 1972.

Ellis, 1989 Margaret A. Ellis and Bjarne Stroustrup: The Annotated C++ Reference Manual. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

Gamma, 1995 Erich Gamma, et al.: Design Patterns. Addison-Wesley. Reading, Mass. 1995. ISBN 0-201-63361-2.

Goldberg, 1983 A. Goldberg and D. Robson: SMALLTALK- 80 - The Language and Its Implementation. Addison-Wesley. Reading, Mass. 1983.

Griswold, 1970 R. E. Griswold, et al.: The Snobol4 Programming Language. Prentice-Hall. Englewood Cliffs, New Jersey. 1970.

Griswold, 1983 R. E. Grisswold and M. T. Griswold: The ICON Programming Language. Prentice-Hall. Englewood Cliffs, New Jersey. 1983.

Hamilton, 1993 G. Hamilton and P. Kougiouris: The Spring Nucleus: A Microkernel for Objects. Proc. 1993 Summer USENIX Conference. USENIX.

Henricson, 1997 Mats Henricson and Erik Nyquist: Industrial Strenght C++: Rules and Recommendations. Prentice-Hall. Englewood Cliffs, New Jersey. 1997. ISBN 0-13-120965-5.

Ichbiah, 1979 Jean D. Ichbiah, et al.: Rationale for the Design of the ADA Programming Language. SIGPLAN Notices. Vol. 14 No. 6. June 1979.

Kamath, 1993 Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: Reaping Benefits with Object-Oriented Technology. AT&T Technical Journal. Vol. 72 No. 5. September/October 1993.

Kernighan, 1978 Brian W. Kernighan and Dennis M. Ritchie: The C Programming Language. Prentice-Hall. Englewood Cliffs, New Jersey. 1978.

Kernighan, 1988 Brian W. Kernighan and Dennis M. Ritchie: The C Programming Language (Second Edition). Prentice-Hall. Enlewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

Koenig, 1989 Andrew Koenig and Bjarne Stroustrup: C++: As close to C as possible - but no closer. The C++ Report. Vol. 1 No. 7. July 1989.

Koenig, 1997 Andrew Koenig and Barbara Moo: Ruminations on C++. Addison Wesley Longman. Reading, Mass. 1997. ISBN 1-201-42339-1.

Knuth, 1968 Donald Knuth: The Art of Computer Programming. Addison-Wesley. Reading, Mass.

Liskowv, 1979 Barbara Liskov et al.: Clu Reference Manual. MIT/LCS/TR-225. MIT Cambridge. Mass. 1979.

Martin, 1995 Robert C. Martin: Designing Object-Oriented C++ Applications Using the Booch Method. Prentice-Hall. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.

Orwell, 1949 George Orwell: 1984. Secker and Warburg. London. 1949.

Parrington, 1995 Graham Parrington et al.: The Design and Implementation of Arjuna. Computer Systems. Vol. 8 No. 3. Summer 1995.

Richards, 1980 Martin Richards and Colin Whitby-Strevens: BCPL - The Language and Its Compiler. Cambridge University Press, Cambridge. England. 1980. ISBN 0-521-21965-5.

Rosler, 1984 L. Rosler: The Evolution of C - Past and Future. AT&T Bell Laboratories Technical Journal. Vol. 63 No. 8. Part 2. October 1984.

Rozier, 1988 M. Rozier, et al.: CHORUS Distributed Operating Systems. Computing Systems. Vol. 1 no. 4. Fall 1988.

Sethi, 1981 Ravi Sethi: Uniform Syntax for Type Expressions and Declarations. Software Practice & Experience. Vol. 11. 1981.

Stepanov, 1994 Alexander Stepanov and Meng Lee: The Standard Template Library. HP Labs Technical Report HPL-94-34 (R. 1). August, 1994.

Stroustrup, 1986 Bjarne Stroustrup: The C++ Programming Language. Addison-Wesley. Reading, Mass. 1986. ISBN 0-201-12078-X.

Stroustrup, 1987 Bjarne Stroustrup and Jonathan Shopiro: A Set of C Classes for Co-Routine Style Programming. Proc. USENIX C++ conference. Santa Fe, New Mexico. November 1987.

Stroustrup, 1991 Bjarne Stroustrup: The C++ Programming Language (Second Edition) Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.

Strostrup, 1994 Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

Tarjan, 1983 Robert E. Tarjan: Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics. Philadelphia, Penn. 1983. ISBN 0-898-71187-8.

Unicode, 1996 The Unicode Consortium: The Unicode Standard, Version 2.0. Addison-Wesley Developers Press. Reading, Mass. 1996. ISBN 0-201-48345-9.

UNIX, 1985 UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition. AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

Wilson, 1996 Gregory V. Wilson and Paul Lu (editors): Parallel Progrmming Using C++. The MIT Press. Cambridge. Mass. 1996. ISBN 0-262-73118-5.

Wikström, 1987 Ake Wikström: Functional Programming Using ML. Prentice-Hall. Englewood Cliffs, New Jersey. 1987.

Woodward, 1974 P. M. Woodward and S. G. Bond: Algol 68-R Users Guide. Her Majesty's Stationery Office. London. England. 1974.

2

Kirándulás a C++-ban

„Az első tennivalónk: öljünk meg minden törvénytudót”
(Shakespeare: VI. Henrik, II. rész – ford. Németh László)

Mi a C++? • Programozási megközelítések • Eljárásközpontú programozás • Modularitás • Külön fordítás • Kivételkezelés • Elvont adatábrázolás • Felhasználói típusok • Konkrét típusok • Absztrakt típusok • Virtuális függvények • Objektumorientált programozás • Általánosított programozás • Tárolók • Algoritmusok • Nyelv és programozás • Tanácsok

2.1. Mi a C++?

A C++ általános célú programozási nyelv, melynek fő alkalmazási területe a rendszerprogramozás és

- egy jobbfajta C,
- támogatja az elvont adatábrázolást,
- támogatja az objektumorientált programozást, valamint
- az általánosított programozást.

Ez a fejezet elmagyarázza, mit jelentenek a fentiek, anélkül, hogy belemenne a nyelv meghatározásának finomabb részleteibe. Célja általános áttekintést adni a C++-ról és használatának fő módszereiről, *nem* pedig a C++ programozás elkezdéséhez szükséges részletes információt adni az olvasónak.

Ha az olvasó túl elnagyoltnak találja e fejezet némelyik részének tárgyalásmódját, egyszerűen ugorja át és lépjen tovább. Később mindenre részletes magyarázatot kap. Mindenesetre, ha átugrik részeket a fejezetben, tegye meg magának azt a szívességet, hogy később visszatér rájuk.

A nyelvi tulajdonságok részletes megértése – még ha a nyelv *összes* tulajdonságáé is – nem ellensúlyozhatja azt, ha hiányzik az átfogó szemléletünk a nyelvről és használatának alapvető módszereiről.

2.2. Programozási megközelítések

Az objektumorientált (objektumközpontú) programozás egy programozási mód – a „jó" programok írása közben felmerülő sereg probléma megoldásának egy megközelítése (paradigma). Ha az „objektumorientált programozási nyelv" szakkifejezés egyáltalán jelent valamit, olyan programozási nyelvet kell hogy jelentsen, amely az objektumokat középpontba helyező programozási stílust támogató eljárásokról gondoskodik.

Itt fontos megkülönböztetnünk két fogalmat: egy nyelvről akkor mondjuk, hogy *támogat* egy programozási stílust, ha olyan szolgáltatásai vannak, melyek által az adott stílus használata kényelmes (könnyű, biztonságos és hatékony) lesz. A támogatás hiányzik, ha kivételes erőfeszítés vagy ügyesség kell az ilyen programok írásához; ekkor a nyelv csupán *megengedi*, hogy az adott megközelítést használjuk. Lehet strukturált programot írni Fortran77-ben és objektumközpontút C-ben, de szükségtelenül nehezen, mivel ezek a nyelvek nem támogatják közvetlenül az említett megközelítéseket.

Az egyes programozási módok támogatása nem csak az adott megközelítés közvetlen használatát lehetővé tévő nyelvi szolgáltatások magától értetődő formájában rejlik, hanem a fordítási/futási időbeni ellenőrzések finomabb formáiban, melyek védelmet adnak a stílustól való akaratlan eltérés ellen. A típusellenőrzés erre a legkézenfekvőbb példa, de a kétértelműség észlelése és a futási idejű ellenőrzések szintén a programozási módok nyelvi támogatásához tartoznak. A nyelven kívüli szolgáltatások, mint a könyvtárak és programozási környezetek, további támogatást adnak az egyes megközelítési módokhoz.

Egy nyelv nem szükségszerűen jobb, mint egy másik, csak azért, mert olyan tulajdonságokkal rendelkezik, amelyek a másikban nem találhatók meg. Sok példa van ennek az ellenkezőjére is. A fontos kérdés nem annyira az, milyen tulajdonságai vannak egy nyelvnek, hanem inkább az, hogy a meglévő tulajdonságok elegendőek-e a kívánt programozási stílusok támogatására a kívánt alkalmazási területeken. Ennek kívánalmai a következők:

1. Minden tulajdonság tisztán és „elegánsan" a nyelv szerves része legyen.
2. A tulajdonságokat egymással párosítva is lehessen használni, hogy olyan megoldást adjanak, melyhez egyébként külön nyelvi tulajdonságok lennének szükségesek.
3. A lehető legkevesebb legyen az ál- és „speciális célú" tulajdonság.
4. Az egyes tulajdonságok megvalósítása nem okozhat jelentős többletterhelést olyan programoknál, melyek nem igénylik azokat.
5. A felhasználónak csak akkor kell tudnia a nyelv valamely részhalmazáról, ha kifejezetten használja azt egy program írásához.

A fentiek közül az első elv az esztétikához és a logikához való folyamodás. A következő kettő a minimalizmus gondolatának kifejezése, az utolsó kettő pedig így összesíthető: „amiről nem tudunk, az nem fáj".

A C++-t úgy terveztük, hogy az elvont adatábrázolást, illetve az objektumorientált és az általánosított programozást támogassa, mégpedig az e megszorítások mellett támogatott hagyományos C programozási módszereken kívül. *Nem* arra szolgál, hogy minden felhasználóra egyetlen programozási stílust kényszerítsen.

A következőkben néhány programozási stílust és az azokat támogató főbb tulajdonságokat vesszük számba. A bemutatás egy sor programozási eljárással folytatódik, melyek az eljárásközpontú (procedurális) programozástól elvezetnek az objektumorientált programozásban használt osztályhierarchiáig és a sablonokat használó általánosított (generikus) programozásig. Minden megközelítés az elődjére épül, mindegyik hozzátesz valamit a C++ programozók eszköztárához, és mindegyik egy bevált tervezési módot tükröz.

A nyelvi tulajdonságok bemutatása nem teljes. A hangsúly a tervezési megközelítéseken és a programok szerkezeti felépítésén van, nem a nyelvi részleteken. Ezen a szinten sokkal fontosabb, hogy fogalmat kapjunk arról, mit lehet megtenni C++-t használva, mint hogy megértsük, hogyan.

2.3. Eljárásközpontú programozás

Az eredeti programozási alapelv a következő:

> *Döntsd el, mely eljárásokra van szükséged*
> *és használd azokhoz a lehető legjobb algoritmusokat.*

A középpontban az eljárás áll – a kívánt számításhoz szükséges algoritmus. A nyelvek ezt az alapelvet függvényparaméterek átadásával és a függvények által visszaadott értékekkel támogatják. Az e gondolkodásmóddal kapcsolatos irodalom tele van a paraméterátadás és a különböző paraméterfajták megkülönböztetési módjainak (eljárások, rutinok, makrók stb.) tárgyalásával.

A „jó stílus" jellegzetes példája az alábbi négyzetgyök-függvény. Átadva egy kétszeres pontosságú lebegőpontos paramétert, a függvény visszaadja az eredményt. Ezt egy jól érthető matematikai számítással éri el:

```
double sqrt(double arg)
{
  // a négyzetgyök kiszámításának kódja
}

void f()
{
  double root2 = sqrt(2);
  // ...
}
```

A kapcsos zárójelek a C++-ban valamilyen csoportba foglalást fejeznek ki; itt a függvény törzsének kezdetét és a végét jelzik. A kettős törtvonal // egy megjegyzés (comment) kezdete, mely a sor végéig tart. A *void* kulcsszó jelzi, hogy az *f* függvény nem ad vissza értéket.

Programszervezési szempontból a függvényeket arra használjuk, hogy rendet teremtsünk az eljárások labirintusában. Magukat az algoritmusokat függvényhívásokkal és más nyelvi szolgáltatások használatával írjuk meg. A következő alpontok vázlatos képet adnak a C++ legalapvetőbb szolgáltatásairól a számítások kifejezéséhez.

2.3.1. Változók és aritmetika

Minden névnek és kifejezésnek típusa van, amely meghatározza a végrehajtható műveleteket:

```
int inch;
```

A fenti deklaráció például azt adja meg, hogy *inch* típusa *int* (vagyis *inch* egy egész típusú változó).

A *deklaráció* olyan utasítás, mely a programba egy nevet vezet be. Ehhez a névhez egy típust rendel. A *típus* egy név vagy kifejezés megfelelő használatát határozza meg.

A C++ több alaptípussal rendelkezik, melyek közvetlen megfelelői bizonyos hardverszolgáltatásoknak. Például:

```
bool        // logikai típus, lehetséges értékei: true (igaz) és false (hamis)
char        // karakter, például 'a', 'z', vagy '9'
int         // egész érték, például 1, 42, vagy 1216
double      // kétszeres pontosságú lebegőpontos szám, például 3.14 vagy 299793.0
```

A *char* változók természetes mérete egy karakter mérete az adott gépen (rendesen egy bájt), az *int* változóké az adott gépen működő egész típusú aritmetikához igazodik (rendszerint egy gépi szó).

Az aritmetikai műveletek e típusok bármilyen párosítására használhatók:

```
+           // összeadás vagy előjel, egy- és kétoperandusú is lehet
-           // kivonás vagy előjel, egy- és kétoperandusú is lehet
*           // szorzás
/           // osztás
%           // maradékképzés
```

Ugyanígy az összehasonlító műveletek is:

```
==          // egyenlő
!=          // nem egyenlő
<           // kisebb
>           // nagyobb
<=          // kisebb vagy egyenlő
>=          // nagyobb vagy egyenlő
```

Értékadásokban és aritmetikai műveletekben a C++ az alaptípusok között elvégez minden értelmes átalakítást, így azokat egymással tetszés szerint keverhetjük:

```
void some_function()          // értéket vissza nem adó függvény
{
  double d = 2.2;             // lebegőpontos szám kezdeti értékadása
  int i = 7;                  // egész kezdeti értékadása
  d = d+i;                    // összeg értékadása
  i = d*i;                    // szorzat értékadása
}
```

Itt = az értékadó művelet jele és == az egyenlőséget teszteli, mint a C-ben.

2.3.2. Elágazások és ciklusok

A C++ az elágazások és ciklusok kifejezésére rendelkezik a hagyományos utasításkészlettel. Íme egy egyszerű függvény, mely a felhasználótól választ kér és a választól függő logikai értéket ad vissza:

```
bool accept()
{
  cout << "Do you want to proceed (y or n)?\n";        // kérdés kiírása

  char answer = 0;
  cin >> answer;                                        // válasz beolvasása

  if (answer == 'y') return true;
  return false;
}
```

A << („tedd bele") műveleti jelet kimeneti operátorként használtuk; a *cout* a szabványos kimeneti adatfolyam. A >> („olvasd be") a bemenet műveleti jele, a *cin* a szabványos bemenő adatfolyam. A >> jobb oldalán álló kifejezés határozza meg, milyen bemenet fogadható el és ez a beolvasás célpontja. A *\n* karakter a kiírt karakterlánc végén új sort jelent.

A példa kissé javítható, ha egy *'n'* választ is számításba veszünk:

```
bool accept2()
{
  cout << "Do you want to proceed (y or n)?\n";        // kérdés kiírása

  char answer = 0;
  cin >> answer;                                        // válasz beolvasása
```

```
  switch (answer) {
  case 'y':
          return true;
  case 'n':
          return false;
  default:
          cout << "I'll take that for a no.\n";  // nemleges válasznak veszi
          return false;
  }
}
```

A *switch* utasítás egy értéket ellenőriz, állandók halmazát alapul véve. A *case* konstansoknak különállóknak kell lenniük, és ha az érték egyikkel sem egyezik, a vezérlés a *default* címkére kerül. A programozónak nem kell alapértelmezésről (default) gondoskodnia.

Kevés programot írnak ciklusok nélkül. Esetünkben szeretnénk lehetőséget adni a felhasználónak néhány próbálkozásra:

```
bool accept3()
{
  int tries = 1;
  while (tries < 4) {
          cout << "Do you want to proceed (y or n)?\n";        // kérdés kiírása
          char answer = 0;
          cin >> answer;                                       // válasz beolvasása

          switch (answer) {
          case 'y':
                  return true;
          case 'n':
                  return false;
          default:
                  cout << "Sorry, I don't understand that.\n" ;  // nem érti a választ
                  tries = tries + 1;
          }
  }
  cout << "I'll take that for a no.\n";  // nemleges válasznak veszi
  return false;
}
```

A *while* utasítás addig hajtódik végre, amíg a feltétele hamis nem lesz.

2.3.3. Mutatók és tömbök

Egy tömböt így határozhatunk meg:

```
char v[10];          // 10 karakterből álló tömb
```

Egy mutatót így:

```
char* p;             // mutató karakterre
```

A deklarációkban a *[ ]* jelentése „tömbje” (array of), míg a * jelentése „mutatója” (pointer to). Minden tömbnek *0* az alsó határa, tehát *v*-nek tíz eleme van, *v[0]...v[9]*. A mutató változó a megfelelő típusú objektum címét tartalmazhatja:

```
p = &v[3];           // p a v negyedik elemére mutat
```

A „címe” jelentéssel bíró operátor az egyoperandusú *&*. Lássuk, hogyan másolhatjuk át egy tömb tíz elemét egy másik tömbbe:

```
void another_function()
{
  int v1[10];
  int v2[10];
  // ...
  for (int i=0; i<10; ++i) v1[i]=v2[i];
}
```

A *for* utasítás így olvasható: „állítsuk *i*-t *0*-ra, amíg *i* kisebb, mint *10*, az *i*-edik elemet másoljuk át, és növeljük meg *i*-t”. Ha egész típusú változóra alkalmazzuk, a ++ növelő műveleti jel az értéket egyszerűen eggyel növeli.

2.4. Moduláris programozás

Az évek során a programtervezés súlypontja az eljárások felől az adatszervezés irányába tolódott el. Egyebek mellett ez a programok nagyobb méretében tükröződik. Az egymással rokon eljárásokat az általuk kezelt adatokkal együtt gyakran *modul*-nak nevezzük. A megközelítés alapelve ez lesz:

> *Döntsd el, mely modulokra van szükség és oszd fel*
> *a programot úgy, hogy az adatokat modulokba helyezed.*

Ez az *adatrejtés* elve. Ahol az eljárások nincsenek az adatokkal egy csoportban, az eljárás-központú programozás megfelel a célnak. Az egyes modulokon belüli eljárásokra a „jó eljárások" tervezésének módja is alkalmazható. A modulra a legközönségesebb példa egy verem (stack) létrehozása. A megoldandó fő problémák:

1. Gondoskodni kell a verem felhasználói felületéről (pl. *push()* és *pop()* függvények).
2. Biztosítani kell, hogy a verem megjelenítése (pl. az elemek tömbje) csak ezen a felhasználói felületen keresztül legyen hozzáférhető.
3. Biztosítani kell a verem első használat előtti előkészítését (inicializálását).

A C++ egymással rokon adatok, függvények stb. különálló névterekbe való csoportosítására ad lehetőséget. Egy *Stack* modul felhasználói felülete például így adható meg és használható:

```
namespace Stack {              // felület
  void push(char);
  char pop();
}

void f()
{
  Stack::push('c');
  if (Stack::pop() != 'c') error("lehetetlen");
}
```

A *Stack::* minősítés azt jelzi, hogy a *push()* és a *pop()* a *Stack* névtérhez tartoznak. E nevek máshol történő használata nem lesz befolyással erre a programrészre és nem fog zavart okozni.

A *Stack* kifejtése lehet a program külön fordítható része:

```
namespace Stack {              // megvalósítás
  const int max_size = 200;
  char v[max_size];
  int top = 0;
```

```
  void push(char c) { /* túlcsordulás ellenőrzése és c behelyezése */ }
  char pop() { /* alulcsordulás ellenőrzése és a legfelső elem kiemelése */ }
}
```

A *Stack* modul fontos jellemzője, hogy a felhasználói kódot a *Stack::push()*-t és a *Stack::pop()*-ot megvalósító kód elszigeteli a *Stack* adatábrázolásától. A felhasználónak nem kell tudnia, hogy a verem egy tömbbel van megvalósítva, a megvalósítás pedig anélkül módosítható, hogy hatással lenne a felhasználói kódra.

Mivel az adat csak egyike az „elrejtendő" dolgoknak, az adatrejtés elve az *információrejtés* elvévé terjeszthető ki; vagyis a függvények, típusok stb. nevei szintén modulba helyezhetők. Következésképpen a C++ bármilyen deklaráció elhelyezését megengedi egy névtérben (§8.2.).

A fenti *Stack* modul a verem egy ábrázolásmódja. A következőkben többféle vermet használunk a különböző programozói stílusok szemléltetésére.

2.4.1. Külön fordítás

A C++ támogatja a C külön fordítási elvét. Ezt arra használhatjuk, hogy egy programot részben független részekre bontsunk.

Azokat a deklarációkat, melyek egy modul felületét írják le, jellemzően egy fájlba írjuk, melynek neve a használatot tükrözi. Ennek következtében a

```
namespace Stack {                    // felület
  void push(char);
  char pop();
}
```

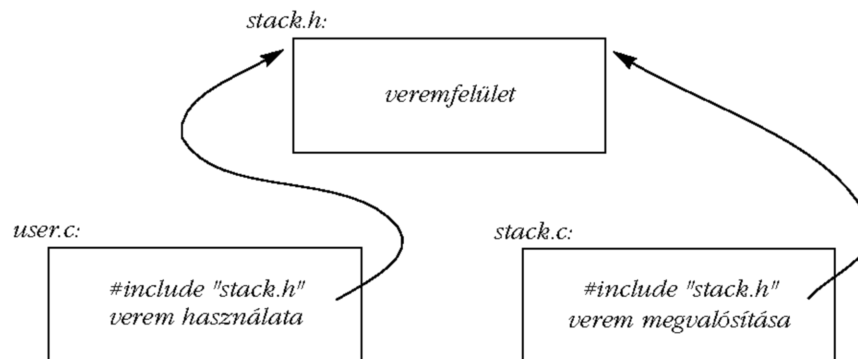
a *stack.h* nevű fájlba kerül, a felhasználók pedig ezt az úgynevezett fejállományt *(header)* beépítik *(#include)*:

```
#include "stack.h"                   // a felület beépítése

void f()
{
  Stack::push('c');
  if (Stack::pop() != 'c') error("impossible");
}
```

Ahhoz, hogy a fordítónak segítsünk az egységesség és következetesség biztosításában, a *Stack* modult megvalósító fájl szintén tartalmazza a felületet:

```
#include "stack.h"                // a felület beépítése

namespace Stack {                 // ábrázolás
  const int max_size = 200;
  char v[max_size];
  int top = 0;
}

void Stack::push(char c) { /* túlcsordulás ellenőrzése és c behelyezése */ }

char Stack::pop() { /* alulcsordulás ellenőrzése és a legfelső elem kiemelése */ }
```

A felhasználói kód egy harmadik fájlba kerül (*user.c*). A *user.c* és a *stack.c* fájlokban lévő kód közösen használja a *stack.h*-ban megadott veremfelületet, de a két fájl egyébként független és külön-külön lefordítható. A program részei a következőképpen ábrázolhatók:



A külön fordítás követelmény minden valódi (tényleges használatra szánt) program esetében, nem csak a moduláris felépítésűeknél (mint pl. a *Stack*). Pontosabban, a külön fordítás használata nem nyelvi követelmény; inkább annak módja, hogyan lehet egy adott nyelvi megvalósítás előnyeit a legjobban kihasználni. A legjobb, ha a modularitást a lehető legnagyobb mértékig fokozzuk, nyelvi tulajdonságok által ábrázoljuk, majd külön-külön hatékonyan fordítható fájlokon keresztül valósítjuk meg (8. és 9. fejezet).

2.4.2. Kivételkezelés

Ha egy programot modulokra bontunk, a hibakezelést a modulok szintjén kell elvégeznünk. A kérdés, melyik modul felelős az adott hiba kezeléséért? A hibát észlelő modul gyakran nem tudja, mit kell tennie. A helyreállítási tevékenység a műveletet kezdeményező modultól függ, nem attól, amelyik észlelte a hibát, miközben megkísérelte a művelet végrehajtását. A programok növekedésével – különösen kiterjedt könyvtárhasználat esetén – a hibák (vagy általánosabban: a „kivételes események") kezelési szabványai egyre fontosabbá válnak.

Vegyük megint a *Stack* példát. Mit kell tennünk, amikor túl sok karaktert próbálunk *push()*-sal egy verembe rakni? A verem modul írója nem tudja, hogy a felhasználó mit akar tenni ilyen esetben, a felhasználó pedig nem mindig észleli a hibát (ha így lenne, a túlcsordulás nem történne meg). A megoldás: a *Stack* írója kell, hogy tudatában legyen a túlcsordulás veszélyének és ezután az (ismeretlen) felhasználóval tudatnia kell ezt. A felhasználó majd megteszi a megfelelő lépést:

```
namespace Stack {            // felület
  void push(char);
  char pop();

  class Overflow { };        // túlcsordulást ábrázoló típus
}
```

Túlcsordulás észlelésekor a *Stack::Push()* meghívhat egy kivételkezelő kódot; vagyis „egy *Overflow* kivételt dobhat":

```
void Stack::push(char c)
{
  if (top == max_size) throw Overflow();
  // c behelyezése
}
```

A *throw* a vezérlést a *Stack::Overflow* típusú kivételkezelőnek adja át, valamilyen függvényben, mely közvetve vagy közvetlenül meghívta a *Stack::Push()*-t. Ehhez „visszatekerjük" a függvényhívási vermet, ami ahhoz szükséges, hogy visszajussunk a hívó függvény környezetéhez. Így a *throw* úgy működik, mint egy többszintű *return*. Például:

```
void f()
{
  // ...
  try {      // a kivételekkel az alább meghatározott kezelő foglalkozik
```

```
            while (true) Stack::push('c');
    }
    catch (Stack::Overflow) {
            // hoppá: verem-túlcsordulás; a megfelelő művelet végrehajtása
    }
    // ...
}
```

A *while* ciklus örökké ismétlődne, ezért ha valamelyik *Stack::push()* hívás egy *throw*-t vált ki, a vezérlés a *Stack::Overflow*-t kezelő *catch* részhez kerül.

A kivételkezelő eljárások használata szabályosabbá és olvashatóbbá teheti a hibakezelő kódot. További tárgyalás, részletek és példák: §8.3, 14. fejezet, „E" függelék.

2.5. Elvont adatábrázolás

A modularitás alapvető szempont minden sikeres nagy programnál. E könyv minden tervezési vizsgálatában ez marad a középpontban. Az előzőekben leírt alakú modulok azonban nem elegendőek ahhoz, hogy tisztán kifejezzünk összetett rendszereket. Az alábbiakban a modulok használatának egyik módjával foglalkozunk (felhasználói típusok létrehozása), majd megmutatjuk, hogyan küzdjünk le problémákat a felhasználói típusok közvetlen létrehozása révén.

2.5.1. Típusokat leíró modulok

A modulokkal való programozás elvezet az összes azonos típusú adatnak egyetlen típuskezelő modul általi központosított kezeléséhez. Ha például sok vermet akarunk – az előbbi *Stack* modulban található egyetlen helyett – megadhatunk egy veremkezelőt, az alábbi felülettel:

```
namespace Stack {
    struct Rep;                    // a verem szerkezetének meghatározása máshol található
    typedef Rep& stack;

    stack create();                // új verem létrehozása
    void destroy(stack s);         // s törlése
```

```
    void push(stack s, char c);          // c behelyezése az s verembe
    char pop(stack s);                   // s legfelső elemének kiemelése
}
```

A következő deklaráció azt mondja, hogy *Rep* egy típus neve, de későbbre hagyja a típus meghatározását (§5.7).

```
struct Rep;
```

Az alábbi deklaráció a *stack* nevet adja egy „*Rep* referenciának" (részletek §5.5-ben).

```
typedef Rep& stack;
```

Az ötlet az, hogy a vermet saját *Stack::stack*-jével azonosítjuk és a további részleteket a felhasználó elől elrejtjük. A *Stack::stack* működése nagyon hasonlít egy beépített típuséhoz:

```
struct Bad_pop { };

void f()
{
    Stack::stack s1 = Stack::create();      // új verem létrehozása
    Stack::stack s2 = Stack::create();      // még egy verem létrehozása

    Stack::push(s1,'c');
    Stack::push(s2,'k');

    if (Stack::pop(s1) != 'c') throw Bad_pop();
    if (Stack::pop(s2) != 'k') throw Bad_pop();

    Stack::destroy(s1);
    Stack::destroy(s2);
}
```

Ezt a *Stack*-et többféleképpen megvalósíthatnánk. Fontos, hogy a felhasználónak nem szükséges tudnia, hogyan tesszük ezt. Amíg a felületet változatlanul hagyjuk, a felhasználó nem fogja észrevenni, ha úgy döntünk, hogy átírjuk a *Stack*-et. Egy megvalósítás például előre lefoglalhatna néhány verempéldányt és a *Stack::create()* egy nem használt példányra való hivatkozást adna át. Ezután a *Stack::destroy()* egy ábrázolást „nem használt"-ként jelölhet meg, így a *Stack::create()* újra használhatja azt:

```
namespace Stack {               // ábrázolás

    const int max_size = 200;
```

```
    struct Rep {
        char v[max_size];
        int top;
    };

    const int max = 16;      // vermek maximális száma

    Rep stacks[max];         // előre lefoglalt verempéldányok
    bool used[max];          // used[i] igaz, ha stacks[i] használatban van

    typedef Rep& stack;
}

void Stack::push(stack s, char c) { /* s túlcsordulásának ellenőrzése és c behelyezése */ }

char Stack::pop(stack s) { /* s alulcsordulásának ellenőrzése és a legfelső elem kiemelése */ }

Stack::stack Stack::create()
{
        // használaton kívüli Rep kiválasztása, használtként
        // megjelölése, előkészítése, rá mutató hivatkozás visszaadása
}

void Stack::destroy(stack s) { /* s megjelölése nem használtként */ }
```

Amit tettünk, az ábrázoló típus becsomagolása felületi függvények készletébe. Az, hogy az eredményül kapott „stack típus" hogyan viselkedik, részben attól függ, hogyan adtuk meg ezeket a felületi függvényeket, részben attól, hogyan mutattuk be a *Stack*-et ábrázoló típust a verem felhasználóinak, részben pedig magától az ábrázoló típustól.

Ez gyakran kevesebb az ideálisnál. Jelentős probléma, hogy az ilyen „műtípusoknak" a felhasználók részére való bemutatása az ábrázoló típus részleteitől függően nagyon változó lehet – a felhasználókat viszont el kell szigetelni az ábrázoló típus ismeretétől. Ha például egy jobban kidolgozott adatszerkezetet választottunk volna a verem azonosítására, a *Stack::stack*-ek értékadási és előkészítési (inicializálási) szabályai drámai módon megváltoztak volna (ami néha valóban kívánatos lehet). Ez azonban azt mutatja, hogy a kényelmes vermek szolgáltatásának problémáját egyszerűen áttettük a *Stack* modulból a *Stack::stack* ábrázoló típusba.

Még lényegesebb, hogy azok a felhasználói típusok, melyeket az adott megvalósító típushoz hozzáférést adó modul határozott meg, nem úgy viselkednek, mint a beépített típusok, és kisebb vagy más támogatást élveznek, mint azok. Azt például, hogy mikor használható egy *Stack::Rep*, a *Stack::create()* és a *Stack::destroy()* függvény ellenőrzi, nem a szokásos nyelvi szabályok.

2.5.2. Felhasználói típusok

A C++ ezt a problémát úgy küzdi le, hogy engedi, hogy a felhasználó közvetlenül adjon meg típusokat, melyek közel úgy viselkednek, mint a beépített típusok. Az ilyen típusokat gyakran *elvont* vagy *absztrakt adattípusoknak* (abstract data type, ADT) nevezzük. A szerző inkább a *felhasználói típus* (user-defined type) megnevezést kedveli. Az elvont adattípus kifejezőbb meghatározásához „absztrakt" matematikai leírás kellene. Ha adva volna ilyen, azok, amiket itt *típusoknak* nevezünk, az ilyen valóban elvont egyedek konkrét példányai lennének. A programozási megközelítés most ez lesz:

> *Döntsd el, mely típusokra van szükség*
> *és mindegyikhez biztosíts teljes műveletkészletet.*

Ott, ahol egy típusból egy példánynál többre nincs szükség, elegendő a modulokat használó adatrejtési stílus. Az olyan aritmetikai típusok, mint a racionális és komplex számok, közönséges példái a felhasználói típusnak. Vegyük az alábbi kódot:

```
class complex {
   double re, im;
public:
   complex(double r, double i) { re=r; im=i; }      // complex létrehozása két skalárból
   complex(double r) { re=r; im=0; }                // complex létrehozása egy skalárból
   complex() { re = im = 0; }                       // alapértelmezett complex: (0,0)

   friend complex operator+(complex, complex);
   friend complex operator-(complex, complex);             // kétoperandusú
   friend complex operator-(complex);                      // egyoperandusú
   friend complex operator*(complex, complex);
   friend complex operator/(complex, complex);

   friend bool operator==(complex, complex);               // egyenlő
   friend bool operator!=(complex, complex);               // nem egyenlő
   // ...
};
```

A *complex* osztály (vagyis felhasználói típus) deklarációja egy komplex számot és a rajta végrehajtható műveletek halmazát ábrázolja. Az ábrázolás *privát* (private); vagyis a *re* és az *im* csak a *complex* osztály bevezetésekor megadott függvények által hozzáférhető. Ezeket az alábbi módon adhatjuk meg:

```
complex operator+(complex a1, complex a2)
{
  return complex(a1.re+a2.re,a1.im+a2.im);
}
```

Az a tagfüggvény, melynek neve megegyezik az osztályéval, a *konstruktor*. A konstruktor írja le az osztály egy objektumának előkészítési-létrehozási módját. A *complex* osztály három konstruktort tartalmaz. Egyikük egy *double*-ból csinál *complex*-et, egy másik egy *double* párból, a harmadik alapértelmezett érték alapján. A *complex* osztály így használható:

```
void f(complex z)
{
  complex a = 2.3;
  complex b = 1/a;
  complex c = a+b*complex(1,2.3);
  // ...
  if (c != b) c = -(b/a)+2*b;
}
```

A fordító a komplex számokhoz kapcsolt műveleti jeleket megfelelő függvényhívásokká alakítja. A *c!=b* jelentése például *operator!=(c,b)*, az *1/a* jelentése *operator/ (complex(1),a)*. A legtöbb – de nem minden – modul jobban kifejezhető felhasználói típusként.

2.5.3. Konkrét típusok

Felhasználói típusok változatos igények kielégítésére készíthetők. Vegyünk egy felhasználói veremtípust a *complex* típus soraival együtt. Ahhoz, hogy kissé valósághűbbé tegyük a példát, ez a *Stack* típus paraméterként elemei számát kapja meg:

```
class Stack {
  char* v;
  int top;
  int max_size;
public:
  class Underflow { };          // kivétel
  class Overflow { };           // kivétel
  class Bad_size { };           // kivétel

  Stack(int s);                 // konstruktor
  ~Stack();                     // destruktor
```

```
    void push(char c);
    char pop();
};
```

A *Stack(int)* konstruktor meghívódik, valahányszor létrehozzuk az osztály egy példányát. Ez a függvény gondoskodik a kezdeti értékadásról. Ha bármilyen „takarításra” van szükség, amikor az osztály egy objektuma kikerül a hatókörből, megadhatjuk a konstruktor ellentétét, a *destruktort:*

```
Stack::Stack(int s)    // konstruktor
{
    top = 0;
    if (s<0 || 10000<s) throw Bad_size();          // "||" jelentése "vagy"
    max_size = s;
    v = new char[s];    // az elemek szabad tárba helyezése
}

Stack::~Stack()        // destruktor
{
    delete[ ] v;        // elemek törlése, hely felszabadítása újrafelhasználás céljára (§6.2.6)
}
```

A konstruktor egy új *Stack* változót hoz létre. Ehhez lefoglal némi helyet a szabad tárból (heap – halom, kupac – vagy dinamikus tár) a *new* operátor használatával. A destruktor takarít, felszabadítva a tárat. Az egész a *Stack*-ek felhasználóinak beavatkozása nélkül történik. A felhasználók a vermeket ugyanúgy hozzák létre és használják, ahogy a beépített típusú változókat szokták. Például:

```
Stack s_var1(10);                    // 10 elemet tárolni képes globális verem

void f(Stack& s_ref, int i)          // hivatkozás a Stack veremre
{
    Stack s_var2(i);                 // lokális verem i számú elemmel
    Stack* s_ptr = new Stack(20);    // mutató a szabad tárban levő Stack-re

    s_var1.push('a');
    s_var2.push('b');
    s_ref.push('c');
    s_ptr->push('d');
    // ...
}
```

Ez a *Stack* típus ugyanolyan névadásra, hatókörre, élettartamra, másolásra stb. vonatkozó szabályoknak engedelmeskedik, mint az *int* vagy a *char* beépített típusok.

Természetszerűen a *push()* és *pop()* tagfüggvényeket valahol szintén meg kell adni:

```
void Stack::push(char c)
{
  if (top == max_size) throw Overflow();
  v[top] = c;
  top = top + 1;
}

char Stack::pop()
{
  if (top == 0) throw Underflow();
  top = top - 1;
  return v[top];
}
```

A *complex* és *Stack* típusokat ***konkrét típusnak*** nevezzük, ellentétben az ***absztrakt típusokkal***, ahol a felület tökéletesebben elszigeteli a felhasználót a megvalósítás részleteitől.

2.5.4. Absztrakt típusok

Amikor a *Stackről*, mint egy modul (§2.5.1) által megvalósított „műtípusról" áttértünk egy saját típusra (§2.5.3), egy tulajdonságot elvesztettünk. Az ábrázolás nem válik el a felhasználói felülettől, hanem része annak, amit be kellene építeni *(#include)* a vermeket használó programrészbe. Az ábrázolás privát, ezért csak a tagfüggvényeken keresztül hozzáférhető, de jelen van. Ha bármilyen jelentős változást szenved, a felhasználó újra le kell, hogy fordítsa. Ezt az árat kell fizetni, hogy a konkrét típusok pontosan ugyanúgy viselkedjenek, mint a beépítettek. Nevezetesen egy típusból nem lehetnek valódi lokális (helyi) változóink, ha nem tudjuk a típus ábrázolásának méretét.

Azon típusoknál, melyek nem változnak gyakran, és ahol lokális változók gondoskodnak a szükséges tisztaságról és hatékonyságról, ez elfogadható és gyakran ideális. Ha azonban teljesen el akarjuk szigetelni az adott verem felhasználóját a megvalósítás változásaitól, a legutolsó *Stack* nem elegendő. Ekkor a megoldás leválasztani a felületet az ábrázolásról és lemondani a valódi lokális változókról.

Először határozzuk meg a felületet:

```
class Stack {
public:
  class Underflow { };          // kivétel
  class Overflow { };           // kivétel
```

```
    virtual void push(char c) = 0;
    virtual char pop() = 0;
};
```

A *virtual* szó a Simulában és a C++-ban azt jelenti, hogy „az adott osztályból származtatott osztályban később felülírható”. Egy *Stack*-ből származtatott osztály a *Stack* felületet valósítja meg. A furcsa *=0* kifejezés azt mondja, hogy a veremből származtatott osztálynak meg *kell* határoznia a függvényt. Ilyenformán a *Stack* felületként szolgál bármilyen osztály részére, mely tartalmazza a *push()* és *pop()* függvényeket.Ezt a *Stack*-et így használhatnánk:

```
void f(Stack& s_ref)
{
    s_ref.push('c');
    if (s_ref.pop() != 'c') throw Bad_pop();
}
```

Vegyük észre, hogyan használja *f()* a *Stack* felületet, a megvalósítás mikéntjéről mit sem tudva. Az olyan osztályt, mely más osztályoknak felületet ad, gyakran *többalakú* (polimorf) *típusnak* nevezzük.

Nem meglepő, hogy a megvalósítás a konkrét *Stack* osztályból mindent tartalmazhat, amit kihagytunk a *Stack* felületből:

```
class Array_stack : public Stack {        // Array_stack megvalósítja Stack-et
    char* p;
    int max_size;
    int top;
public:
    Array_stack(int s);
    ~Array_stack();

    void push(char c);
    char pop();
};
```

A *„:public”* olvasható úgy, mint „származtatva …-ból”, „megvalósítja …-t”, vagy „…altípusa …-nak”.

Az *f()* függvény részére, mely a megvalósítás ismeretének teljes hiányában egy *Stacket* akar használni, valamilyen másik függvény kell létrehozzon egy objektumot, amelyen az *f()* műveletet hajthat végre:

```
void g()
{
```

```
  Array_stack as(200);
  f(as);
}
```

Mivel *f()* nem tud az *Array_stack*-ekről, csak a *Stack* felületet ismeri, ugyanolyan jól fog működni a *Stack* egy másik megvalósításával is:

```
class List_stack : public Stack {        // List_stack megvalósítja Stack-et
  list<char> lc;                         // (standard könyvtárbeli) karakterlista (§3.7.3)
public:
  List_stack() { }

  void push(char c) { lc.push_front(c); }
  char pop();
};

char List_stack::pop()
{
  char x = lc.front();                   // az első elem lekérése
  lc.pop_front();                        // az első elem eltávolítása
  return x;
}
```

Itt az ábrázolás egy karakterlista. Az *lc.push_front(c)* beteszi *c*-t, mint *lc* első elemét, az *lc.pop_front* hívás eltávolítja az első elemet, az *lc.front()* pedig *lc* első elemére utal.

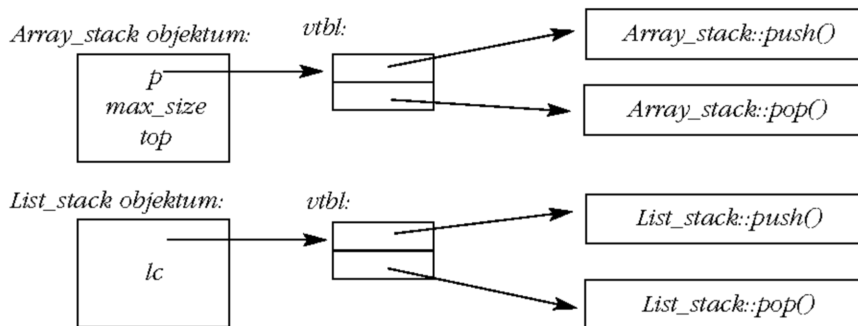
Egy függvény létre tud hozni egy *List_stack*-et és *f()* használhatja azt:

```
void h()
{
  List_stack ls;
  f(ls);
}
```

2.5.5. Virtuális függvények

Hogyan történik az *f()*-en belüli *s_ref.pop()* hívás feloldása a megfelelő függvénydefiníció hívására? Amikor *h()*-ból hívjuk *f()*-et, a *List_stack::pop()*-ot kell meghívni, amikor *g()*-ből, az *Array_stack::pop()*-ot. Ahhoz, hogy ezt feloldhassuk, a *Stack* objektumnak információt kell tartalmaznia arról, hogy futási időben mely függvényt kell meghívni. A fordítóknál szokásos eljárás egy virtuális függvény nevének egy táblázat valamely sorszámértékévé alakítása, amely táblázat függvényekre hivatkozó mutatókat tartalmaz. A táblázatot „virtuális

függvénytáblának" vagy egyszerűen *vtbl*-nek szokás nevezni. Minden virtuális függvényeket tartalmazó osztálynak saját *vtbl*-je van, mely azonosítja az osztály virtuális függvényeit. Ez grafikusan így ábrázolható:



A *vtbl*-ben lévő függvények lehetővé teszik, hogy az objektumot akkor is helyesen használjuk, ha a hívó nem ismeri annak méretét és adatainak elrendezését. A hívónak mindössze a *vtbl* helyét kell tudnia a *Stack*-en belül, illetve a virtuális függvények sorszámát. Ez a virtuális hívási eljárás lényegében ugyanolyan hatékonnyá tehető, mint a „normális függvényhívás". Többlet helyszükséglete: a virtuális függvényeket tartalmazó osztály minden objektumában egy-egy mutató, valamint egy-egy *vtbl* minden osztályhoz.

2.6. Objektumorientált programozás

Az elvont adatábrázolás a jó tervezéshez alapfontosságú, a könyvben pedig a tervezés végig központi kérdés marad. A felhasználói típusok azonban önmagukban nem elég rugalmasak ahhoz, hogy kiszolgálják igényeinket. E részben először egyszerű felhasználói típusokkal mutatunk be egy problémát, majd megmutatjuk, hogyan lehet azt megoldani osztályhierarchiák használatával.

2.6.1. Problémák a konkrét típusokkal

A konkrét típusok – a modulokban megadott „műtípusokhoz" hasonlóan – egyfajta „fekete dobozt" írnak le. Ha egy fekete dobozt létrehozunk, az nem lép igazi kölcsönhatásba a program többi részével. Nincs mód arra, hogy új felhasználáshoz igazítsuk, kivéve, ha de-

finícióját módosítjuk. Ez a helyzet ideális is lehet, de komoly rugalmatlansághoz is vezethet. Vegyük például egy grafikus rendszerben használni kívánt *Shape* (Alakzat) típus meghatározását. Tegyük fel, hogy pillanatnyilag a rendszernek köröket, háromszögeket és négyzeteket kell támogatnia. Tegyük fel azt is, hogy léteznek az alábbiak:

```
class Point { /* ... */ };
class Color { /* ... */ };
```

A /* és */ egy megjegyzés kezdetét, illetve végét jelöli. A jelölés többsoros megjegyzésekhez is használható.

Egy alakzatot az alábbi módon adhatunk meg:

```
enum Kind { circle, triangle, square }; // felsorolás (§4.8)

class Shape {
  Kind k;                              // típusmező
  Point center;
  Color col;
  // ...

public:
  void draw();
  void rotate(int);
  // ...
};
```

A *k* típusazonosító mező azért szükséges, hogy az olyan műveletek számára, mint a *draw()* (rajzolás) vagy a *rotate()* (forgatás) meghatározhatóvá tegyük, milyen fajta alakzattal van dolguk. (A Pascal-szerű nyelvekben egy változó rekordtípust használhatnánk, *k* címkével). A *draw()* függvényt így adhatnánk meg:

```
void Shape::draw()
{
  switch (k) {
  case circle:
          // kör rajzolása
          break;

  case triangle:
          // háromszög rajzolása
          break;
```

```
    case square:
            // négyzet rajzolása
            break;
    }
}
```

Ez azonban „rendetlenség”. A függvényeknek – mint a *draw()* – tudniuk kell arról, milyen alakzatfajták léteznek. Ezért az ilyen függvénynél mindig növekszik a kód, valahányszor a rendszerhez egy új alakzatot adunk. Ha új alakzatot hozunk létre, minden műveletet meg kell vizsgálni és (lehetőség szerint) módosítani kell azokat. A rendszerhez nem adhatunk új alakzatot, hacsak hozzá nem férünk minden művelet forráskódjához. Mivel egy új alakzat hozzáadása magával vonja minden fontos alakzat-művelet kódjának módosítását, az ilyen munka nagy ügyességet kíván és hibákat vihet be a más (régebbi) alakzatokat kezelő kódba. Az egyes alakzat-ábrázolások kiválasztását komolyan megbéníthatja az a követelmény, hogy az ábrázolásoknak (legalább is néhánynak) illeszkednie kell abba a – jellemzően rögzített méretű – keretbe, melyet az általános *Shape* típus leírása képvisel.

2.6.2. Osztályhierarchiák

A probléma az, hogy nincs megkülönböztetés az egyes alakzatok általános tulajdonságai (szín, rajzolhatóság stb.) és egy adott alakzatfajta tulajdonságai közt. (A kör például olyan alakzat, melynek sugara van, egy körrajzoló függvénnyel lehet megrajzolni stb.). E megkülönböztetés kifejezése és előnyeinek kihasználása az objektumorientált programozás lényege. Azok a nyelvek, melyek e megkülönböztetés kifejezését és használatát lehetővé tévő szerkezetekkel rendelkeznek, támogatják az objektumközpontúságot, más nyelvek nem. A megoldásról a Simulából kölcsönzött *öröklés* gondoskodik. Először létrehozunk egy osztályt, mely minden alakzat általános tulajdonságait leírja:

```
class Shape {
  Point center;
  Color col;
  // ...
public:
  Point where() { return center; }
  void move(Point to) { center = to; /* ... */ draw(); }

  virtual void draw() = 0;
  virtual void rotate(int angle) = 0;
  // ...
};
```

Akárcsak a §2.5.4 absztrakt *Stack* típusában, azokat a függvényeket, melyeknél a hívási felület meghatározható, de a konkrét megvalósítás még nem ismert, virtuálisként *(virtual)* vezetjük be.

A fenti meghatározás alapján már írhatunk általános függvényeket, melyek alakzatokra hivatkozó mutatókból álló vektorokat kezelnek:

```
void rotate_all(vector<Shape*>& v, int angle)    // v elemeinek elforgatása angle szöggel
{
    for (int i = 0; i<v.size(); ++i) v[i]->rotate(angle);
}
```

Egy konkrét alakzat meghatározásához meg kell mondanunk, hogy alakzatról van szó és meg kell határoznunk konkrét tulajdonságait (beleértve a virtuális függvényeket is):

```
class Circle : public Shape {
    int radius;
public:
    void draw() { /* ... */ }
    void rotate(int) {}            // igen, üres függvény
};
```

A C++-ban a *Circle* osztályról azt mondjuk, hogy a *Shape* osztályból *származik* (derived), a *Shape* osztályról pedig azt, hogy a *Circle* osztály őse ill. *bázisosztálya* (alaposztálya, base). Más szóhasználat szerint a *Circle* és a *Shape* alosztály (subclass), illetve főosztály (superclass). A származtatott osztályról azt mondjuk, hogy örökli (inherit) a bázisosztály tagjait, ezért a bázis- és származtatott osztályok használatát általában *öröklésként* említjük.A programozási megközelítés itt a következő:

> *Döntsd el, mely osztályokra van szükséged,*
> *biztosíts mindegyikhez teljes műveletkészletet,*
> *az öröklés segítségével pedig határold körül pontosan*
> *a közös tulajdonságokat.*

Ahol nincs ilyen közös tulajdonság, elegendő az elvont adatábrázolás. A típusok közti, öröklés és virtuális függvények használatával kiaknázható közösség mértéke mutatja, mennyire alkalmazható egy problémára az objektumorientált megközelítés. Némely területen, például az interaktív grafikában, világos, hogy az objektumközpontúságnak óriási le-

hetőségei vannak. Más területeken, mint a klasszikus aritmetikai típusoknál és az azokon alapuló számításoknál, alig látszik több lehetőség, mint az elvont adatábrázolás, így az objektumközpontúság támogatásához szükséges szolgáltatások feleslegesnek tűnnek.

Az egyes típusok közös tulajdonságait megtalálni nem egyszerű. A kihasználható közösség mértékét befolyásolja a rendszer tervezési módja. Amikor egy rendszert tervezünk – és akkor is, amikor a rendszerkövetelményeket leírjuk – aktívan kell keresnünk a közös tulajdonságokat. Osztályokat lehet kifejezetten más típusok építőkockáiként tervezni, a létező osztályokat pedig meg lehet vizsgálni, mutatnak-e olyan hasonlóságokat, amelyeket egy közös bázisosztályban kihasználhatnánk. Az objektumorientált programozás konkrét programozási nyelvi szerkezetek nélkül való elemzésére irányuló kísérleteket lásd [Kerr,1987] és [Booch, 1994] a §23.6-ban.

Az osztályhierarchiák és az absztrakt osztályok (§2.5.4) nem kölcsönösen kizárják, hanem kiegészítik egymást (§12.5), így az itt felsorolt irányelvek is inkább egymást kiegészítő, kölcsönösen támogató jellegűek. Az osztályok és modulok például függvényeket tartalmaznak, míg a modulok osztályokat és függvényeket. A tapasztalt tervező sokféle megközelítést használ – ahogy a szükség parancsolja.

2.7. Általánosított programozás

Ha valakinek egy verem kell, nem feltétlenül karaktereket tartalmazó veremre van szüksége. A verem általános fogalom, független a karakter fogalmától. Következésképpen függetlenül kell ábrázolni is.

Még általánosabban, ha egy algoritmus az ábrázolástól függetlenül és logikai torzulás nélkül kifejezhető, akkor így is kell tenni. A programozási irányelv a következő:

> *Döntsd el, mely algoritmusokra van szükség,*
> *és úgy lásd el azokat paraméterekkel, hogy minél több típussal*
> *és adatszerkezettel működjenek.*

2.7.1. Tárolók

Egy karakterverem-típust általánosíthatunk, ha sablont (template) hozunk létre belőle és a konkrét *char* típus helyett sablonparamétert használunk. Például:

```
template<class T> class Stack {
  T* v;
  int max_size;
  int top;
public:
  class Underflow { };
  class Overflow { };

  Stack(int s);        // konstruktor
  ~Stack();            // destruktor

  void push(T);
  T pop();
};
```

A *template<class T>* előtag T-t az utána következő deklaráció paraméterévé teszi.

Hasonlóképpen adhatjuk meg a tagfüggvényeket is:

```
template<class T> void Stack<T>::push(T c)
{
  if (top == max_size) throw Overflow();
  v[top] = c;
  top = top + 1;
}

template<class T> T Stack<T>::pop()
{
  if (top == 0) throw Underflow();
  top = top - 1;
  return v[top];
}
```

Ha a definíciók adottak, a vermet az alábbi módon használhatjuk:

```
Stack<char> sc(200);              // verem 200 karakter számára
Stack<complex> scplx(30);         // verem 30 komplex szám részére
Stack< list<int> > sli(45);       // verem 45, egészekből álló lista számára

void f()
{
  sc.push('c');
  if (sc.pop() != 'c') throw Bad_pop();

  scplx.push(complex(1,2));
  if (scplx.pop() != complex(1,2)) throw Bad_pop();
}
```

Hasonló módon, sablonokként adhatunk meg listákat, vektorokat, asszociatív tömböket (map) és így tovább. Az olyan osztályt, amely valamilyen típusú elemek gyűjteményét tartalmazza, általában *container class*-nak vagy egyszerűen *tárolónak* (konténernek) hívjuk.

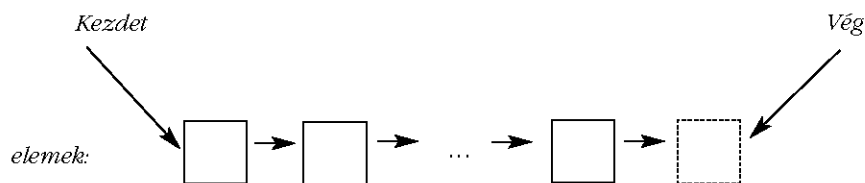
A sablonoknak fordítási időben van jelentőségük, tehát használatuk a „kézzel írott” kódhoz képest nem növeli a futási időt.

2.7.2. Általánosított algoritmusok

A C++ standard könyvtára többféle tárolóról gondoskodik, de a felhasználók sajátokat is írhatnak (3., 17. és 18. fejezetek). Ismét használhatjuk tehát az általánosított (generikus) programozás irányelveit algoritmusok tárolók általi paraméterezésére. Tegyük fel, hogy vektorokat, listákat és tömböket akarunk rendezni, másolni és átkutatni, anélkül, hogy minden egyes tárolóra megírnánk a *sort()*, *copy()* és *search()* függvényeket. Konvertálni nem akarunk egyetlen adott adatszerkezetre sem, melyet egy konkrét *sort* függvény elfogad, ezért találnunk kell egy általános módot a tárolók leírására, mégpedig olyat, amely megengedi, hogy egy tárolót anélkül használjunk, hogy pontosan tudnánk, milyen fajta tárolóról van szó.

Az egyik megoldás, amelyet a C++ standard könyvtárában a tárolók és nem numerikus algoritmusok megközelítéséből (18. fej. §3.8) vettünk át, a *sorozatokra* összpontosít és azokat bejárókkal (iterator) kezeli.

Íme a sorozat fogalmának grafikus ábrázolása:



A sorozatnak van egy kezdete és egy vége. A bejáró (iterator) valamely elemre hivatkozik és gondoskodik arról a műveletről, melynek hatására legközelebb a sorozat soron következő elemére fog hivatkozni. A sorozat vége ugyancsak egy bejáró, mely a sorozat utolsó elemén túlra hivatkozik. A „vége” fizikai ábrázolása lehet egy „őr” (sentinel) elem, de elképzelhető más is. A lényeg, hogy a sorozat számos módon ábrázolható, így listákkal és tömbökkel is.

Az olyan műveletekhez, mint „egy bejáró által férjünk hozzá egy elemhez" és„ a bejáró hivatkozzon a következő elemre" szükségünk van valamilyen szabványos jelölésre. Ha az alapötletet megértettük, az első helyén kézenfekvő választás a * dereferencia (hivatkozó vagy mutató) operátort használni, a másodiknál pedig a ++ növelő műveleti jelet.

A fentieket adottnak tekintve, az alábbi módon írhatunk kódot:

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
  while (from != too_far) {
        *to = *from;        // hivatkozott elemek másolása
        ++to;               // következő cél
        ++from;             // következő forrás
  }
}
```

Ez átmásol bármilyen tárolót, amelyre a formai követelmények betartásával bejárót adhatunk meg. A C++ beépített, alacsonyszintű tömb és mutató típusai rendelkeznek a megfelelő műveletekkel:

```
char vc1[200];      // 200 karakter tömbje
char vc2[500];      // 500 karakter tömbje

void f()
{
  copy(&vc1[0],&vc1[200],&vc2[0]);
}
```

Ez *vc1*-et első elemétől az utolsóig *vc2*-be másolja, *vc2* első elemétől kezdődően.

Minden standard könyvtárbeli tároló (17. Fej., §16.3) támogatja ezt a bejáró- (iterator) és sorozatjelölést. A forrás és a cél típusait egyetlen paraméter helyett két sablonparaméter, az *In* és *Out* jelöli. Ezt azért tesszük, mert gyakran akarunk másolni egy fajta tárolóból egy másik fajtába. Például:

```
complex ac[200];

void g(vector<complex>& vc, list<complex>& lc)
{
  copy(&ac[0],&ac[200],lc.begin());
  copy(lc.begin(),lc.end(),vc.begin());
}
```

Itt a tömböt a *list*-be másoljuk, a *list*-et pedig a *vector*-ba. Egy szabványos tárolónál a *begin()* a bejáró (iterator), amely az első elemre mutat.

2.8. Utóirat

Egyetlen programozási nyelv sem tökéletes. Szerencsére egy programozási nyelvnek nem kell tökéletesnek lennie ahhoz, hogy jó eszközként szolgáljon nagyszerű rendszerek építéséhez. Valójában egy általános célú programozási nyelv nem is lehet minden feladatra tökéletes, amire csak használják. Ami egy feladatra tökéletes, gyakran komoly fogyatékosságokat mutathat egy másiknál, mivel az egy területen való tökéletesség magával vonja a szakosodást. A C++-t ezért úgy terveztük, hogy jó építőeszköz legyen a rendszerek széles választékához és a fogalmak széles körét közvetlenül kifejezhessük vele.

Nem mindent lehet közvetlenül kifejezni egy nyelv beépített tulajdonságait felhasználva. Valójában ez nem is lenne ideális. A nyelvi tulajdonságok egy sereg programozási stílus és módszer támogatására valók. Következésképpen egy nyelv megtanulásánál a feladat a nyelv sajátos és természetes stílusainak elsajátítására való összpontosítás, nem az összes nyelvi tulajdonság minden részletre kiterjedő megértése.

A gyakorlati programozásnál kevés az előnye, ha ismerjük a legrejtettebb nyelvi tulajdonságokat vagy ha a legtöbb tulajdonságot kihasználjuk. Egyetlen nyelvi tulajdonság önmagában nem túl érdekes. Csak akkor lesz jelentékeny, ha az egyes programozási módszerek és más tulajdonságok környezetében tekintjük. Amikor tehát az Olvasó a következő fejezeteket olvassa, kérjük, emlékezzen arra, hogy a C++ részletekbe menő vizsgálatának igazi célja az, hogy képesek legyünk együttesen használni a nyelv szolgáltatásait, jó programozási stílusban, egészséges tervezési környezetben.

2.9. Tanácsok

[1] Ne essünk kétségbe! Idővel minden kitisztul. §2.1.
[2] Jó programok írásához nem kell ismernünk a C++ minden részletét. §1.7.
[3] A programozási módszerekre összpontosítsunk, ne a nyelvi tulajdonságokra. §2.1.

3

Kirándulás a standard könyvtárban

„Minek vesztegessük az időt tanulásra,
mikor a tudatlanság azonnali?”
(Hobbes)

Szabványos könyvtárak • Kimenet • Karakterláncok • Bemenet • Vektorok • Tartományellenőrzés • Listák • Asszociatív tömbök • Tárolók (áttekintés) • Algoritmusok • Bejárók • Bemeneti/kimeneti bejárók • Bejárások és predikátumok • Tagfüggvényeket használó algoritmusok • Algoritmusok (áttekintés) • Komplex számok • Vektoraritmetika • A standard könyvtár (áttekintés) • Tanácsok

3.1. Bevezetés

Nincs olyan jelentős program, mely csak a puszta programnyelven íródik. Először a nyelvet támogató könyvtárakat fejlesztik ki, ezek képezik a további munka alapját.

A 2. fejezet folytatásaként ez a fejezet gyors körutazást tesz a fő könyvtári szolgáltatásokban, hogy fogalmat adjon, mit lehet a C++ és standard könyvtárának segítségével megtenni. Bemutat olyan hasznos könyvtári típusokat, mint a *string*, *vector*, *list* és *map*, valamint használatuk legáltalánosabb módjait. Ez lehetővé teszi, hogy a következő fejezetekben jobb

példákat és gyakorlatokat adjak az olvasónak. A 2. fejezethez hasonlóan bátorítani akarom az olvasót, ne zavarja, ne kedvetlenítse el, ha a részleteket nem érti tökéletesen. E fejezet célja, hogy megízleljük, mi következik, és megértsük a leghasznosabb könyvtári szolgáltatások legegyszerűbb használatát. A standard könyvtárat részletesebben a §16.1.2 mutatja be.

Az e könyvben leírt standard könyvtárbeli szolgáltatások minden teljes C++-változat részét képezik. A C++ standard könyvtárán kívül a legtöbb megvalósítás a felhasználó és a program közti párbeszédre grafikus felhasználói felületeket is kínál, melyeket gyakran GUI-knak vagy ablakozó rendszernek neveznek. Hasonlóképpen, a legtöbb programfejlesztő környezetet alapkönyvtárakkal (foundation library) is ellátták, amelyek a szabványos fejlesztési és/vagy futtatási környezeteket támogatják. Ilyeneket nem fogunk leírni. A szándékunk a C++ önálló leírását adni, úgy, ahogy a szabványban szerepel, és megőrizni a példák „hordozhatóságát" (más rendszerekre való átültetésének lehetőségét), kivéve a külön megjelölteket. Természetesen biztatjuk az olvasót, fedezze fel a legtöbb rendszerben meglévő, kiterjedt lehetőségeket – ezt azonban a gyakorlatokra hagytuk.

3.2. Helló, világ!

A legkisebb C++ program:

```
int main() { }
```

A program megadja a *main* nevű függvényt, melynek nincsenek paraméterei és nem tesz semmit. Minden C++ programban kell, hogy legyen egy *main()* nevű függvény. A program e függvény végrehajtásával indul. A *main()* által visszaadott *int* érték, ha van ilyen, a program visszatérési értéke „a rendszerhez". Ha nincs visszatérési érték, a rendszer a sikeres befejezést jelző értéket kap vissza. Ha a *main()* nem nulla értéket ad vissza, az hibát jelent.

A programok jellemzően valamilyen kimenetet állítanak elő. Íme egy program, amely kiírja: *Helló, világ!*:

```
#include <iostream>

int main()
{
  std::cout << "Helló, világ!\n";
}
```

Az *#include <iostream>* utasítja a fordítót, hogy illessze be az *iostream*-ben található adatfolyam-bemeneti és -kimeneti szolgáltatások deklarációját a forráskódba. E deklarációk nélkül az alábbi kifejezés

```
std::cout << "Helló, világ!\n"
```

értelmetlen volna. A << („tedd bele") kimeneti operátor második operandusát beírja az elsőbe. Ebben az esetben a *„Helló, világ!\n"* karakterliterál a szabványos kimeneti adatfolyamba, az *std::cout*-ba íródik. A karakterliterál egy *" "* jelek közé zárt karaktersorozat. A benne szereplő \ (backslash, fordított perjel) az utána következő karakterrel valamilyen egyedi karaktert jelöl. Esetünkben az *\n* az új sor jele, tehát a kiírt „Helló, világ!" szöveget sortörés követi.

3.3. A standard könyvtár névtere

A standard könyvtár az *std* névtérhez (§2.4, §8.2) tartozik. Ezért írtunk *std::cout*-ot *cout* helyett. Ez egyértelműen a *standard cout* használatát írja elő, nem valamilyen más *cout*-ét.

A standard könyvtár minden szolgáltatásának beépítéséről valamilyen, az *<iostream>*-hez hasonló szabványos fejállomány által gondoskodhatunk:

```
#include<string>
#include<list>
```

Ez rendelkezésre bocsátja a szabványos *string*-et és *list*-et. Használatukhoz az *std::* előtagot alkalmazhatjuk:

```
std::string s = "Négy láb jó, két láb rossz!";
std::list<std::string> slogans;
```

Az egyszerűség kedvéért a példákban ritkán írjuk ki az *std::* előtagot, illetve a szükséges *#include <fejállomány>*-okat. Az itt közölt programrészletek fordításához és futtatásához a megfelelő fejállományokat be kell építeni (*#include*, amint a §3.7.5, §8.6 és a 16. fejezetben szereplő felsorolásokban szerepelnek). Ezenkívül vagy az *std::* előtagot kell használni, vagy globálissá kell tenni minden nevet az *std* névtérből (§8.2.3):

```
#include<string>            // a szabványos karakterlánc-szolgáltatások elérhetővé tétele
using namespace std;         // std nevek elérhetővé tétele az std:: előtag nélkül

string s = "A tudatlanság erény!";        // rendben: a string jelentése std::string
```

Általában szegényes ízlésre vall egy névtérből minden nevet a globális névtérbe helyezni. Mindazonáltal, a nyelvi és könyvtári tulajdonságokat illusztráló programrészletek rövidre fogása érdekében elhagytuk az ismétlődő *#include*-okat és *std::* minősítéseket. E könyvben majdnem kizárólag a standard könyvtárat használjuk, ha tehát egy nevet használunk onnan, azt vagy a szabvány ajánlja, vagy egy magyarázat része (hogyan határozható meg az adott szabványos szolgáltatás).

3.4. Kimenet

Az *iostream* könyvtár minden beépített típusra meghatároz kimenetet, de felhasználói típushoz is könnyen megadhatjuk. Alapértelmezésben a *cout*-ra kerülő kimeneti értékek karaktersorozatra alakítódnak át. A következő kód például az *1* karaktert a *0* karakterrel követve a szabványos kimeneti adatfolyamba helyezi.

```
void f()
{
  cout << 10;
}
```

Ugyanezt teszi az alábbi kód is:

```
void g()
{
  int i = 10;
  cout << i;
}
```

A különböző típusú kimenetek természetesen párosíthatók:

```
void h(int i)
{
  cout << "i értéke ";
  cout << i;
  cout << '\n';
}
```

Ha *i* értéke *10*, a kimenet a következő lesz:

```
i értéke 10
```

A karakterkonstans egy karakter, egyszeres idézőjelek közé zárva. Vegyük észre, hogy a karakterkonstansok nem számértékként, hanem karakterként íródnak ki:

```
void k()
{
  cout << 'a';
  cout << 'b';
  cout << 'c';
}
```

A fenti kód kimenete például *abc* lesz.Az ember hamar belefárad a kimeneti adatfolyam nevének ismétlésébe, amikor több rokon tételt kell kiírni. Szerencsére maguk a kimeneti kifejezések eredményei felhasználhatók további kimenetekhez:

```
void h2(int i)
{
  cout << "i értéke " << i << '\n';
}
```

Ez egyenértékű *h()*-val. Az adatfolyamok részletes magyarázata a 21. fejezetben található.

3.5. Karakterláncok

A standard könyvtár gondoskodik a *string* (karakterlánc) típusról, hogy kiegészítse a korábban használt karakterliterálokat. A *string* típus egy sereg hasznos karakterlánc-műveletet biztosít, ilyen például az *összefűzés* :

```
string s1 = "Helló";
string s2 = "világ";

void m1()
{
  string s3 = s1 + ", " + s2 + "!\n";

  cout << s3;
}
```

Az *s3* kezdeti értéke itt a következő karaktersorozat (új sorral követve):

Helló, világ!

A karakterláncok összeadása összefűzést jelent. A karakterláncokhoz karakterliterálokat és karaktereket adhatunk. Sok alkalmazásban az összefűzés legáltalánosabb formája valamit egy karakterlánc végéhez fűzni. Ezt a += művelet közvetlenül támogatja:

```
void m2(string& s1, string& s2)
{
  s1 = s1 + '\n';      // sortörés
  s2 += '\n';          // sortörés
}
```

A lánc végéhez való hozzáadás két módja egyenértékű, de az utóbbit előnyben részesítjük, mert tömörebb és valószínűleg hatékonyabban valósítható meg.Természetesen a karakterláncok összehasonlíthatók egymással és literálokkal is:

```
string incantation;                    //varázsszó

void respond(const string& answer)
{
  if (answer == incantation) {
                                       // varázslás megkezdése
  }
  else if (answer == "yes") {
         // ...
  }
  // ...
}
```

A standard könyvtár *string* osztályát a 20. fejezet írja le. Ez más hasznos tulajdonságai mellett lehetővé teszi a részláncok (substring) kezelését is. Például:

```
string name = "Niels Stroustrup";

void m3()
{
  string s = name.substr(6,10);        // s = "Stroustrup"
  name.replace(0,5,"Nicholas");        // a név új értéke "Nicholas Stroustrup" lesz
}
```

A *substr()* művelet egy olyan karakterláncot ad vissza, mely a paramétereivel megadott részlánc másolata. Az első paraméter egy, a karakterlánc egy adott helyére mutató sorszám,

a második a kívánt részlánc hossza. Mivel a sorszámozás (az index) *0*-tól indul, *s* a *Stroustrup* értéket kapja.A *replace()* művelet a karakterlánc egy részét helyettesíti egy másik karakterlánccal. Ebben az esetben a *0*-val induló, *5* hosszúságú részlánc a *Niels*; ez helyettesítődik a *Nicholas*-szal. A *name* végső értéke tehát *Nicholas Stroustrup*. Vegyük észre, hogy a helyettesítő karakterláncnak nem kell ugyanolyan méretűnek lennie, mint az a részlánc, amelyet helyettesít.

3.5.1.C stílusú karakterláncok

A C stílusú karakterlánc egy nulla karakterrel végződő karaktertömb (§5.5.2). Meg fogjuk mutatni, hogy egy C stílusú karakterláncot könnyen bevihetünk egy *string*-be. A C stílusú karakterláncokat kezelő függvények meghívásához képesnek kell lennünk egy *string* értékének C stílusú karakterlánc formában való kinyerésére. A *c_str()* függvény ezt teszi (§20.3.7). A *name*-et a *printf()* kiíró C-függvénnyel (§21.8) például az alábbi módon írathatjuk ki:

```
void f()
{
  printf("name: %s\n",name.c_str());
}
```

3.6. Bemenet

A standard könyvtár bemenetre az *istreams*-et ajánlja. Az *ostreams*-hez hasonlóan az *istreams* is a beépített típusok karaktersorozatként történő ábrázolásával dolgozik és könnyen bővíthető, hogy felhasználói típusokkal is meg tudjon birkózni. A >> („olvasd be") műveleti jelet bemeneti operátorként használjuk; a *cin* a szabványos bemeneti adatfolyam. A >> jobb oldalán álló típus határozza meg, milyen bemenet fogadható el és mi a beolvasó művelet célpontja. Az alábbi kód egy számot, például *1234*-et olvas be a szabványos bemenetről az *i* egész változóba és egy lebegőpontos számot, mondjuk *12.34e5*-öt ír a kétszeres pontosságú, lebegőpontos *d* változóba:

```
void f()
{
  int i;
  cin >> i;          // egész szám beolvasása i-be

  double d;
  cin >> d;          // kétszeres pontosságú lebegőpontos szám beolvasása d-be
}
```

A következő példa hüvelykről centiméterre és centiméterről hüvelykre alakít. Bemenetként egy számot kap, melynek végén egy karakter jelzi az egységet (centiméter vagy hüvelyk). A program válaszul kiadja a másik egységnek megfelelő értéket:

```
int main()
{
  const float factor = 2.54;          // 1 hüvelyk 2.54 cm-rel egyenlő
  float x, in, cm;
  char ch = 0;

  cout << "Írja be a hosszúságot: ";

  cin >> x;                           // lebegőpontos szám beolvasása
  cin >> ch;                          // mértékegység beolvasása

  switch (ch) {
  case 'i':                           // inch (hüvelyk)
          in = x;
          cm = x*factor;
          break;
  case 'c':                           // cm
          in = x/factor;
          cm = x;
          break;
  default:
          in = cm = 0;
          break;
  }

  cout << in << " in = " << cm << " cm\n";
}
```

A *switch* utasítás egy értéket hasonlít össze állandókkal. A *break* utasítások a *switch* utasításból való kilépésre valók. A *case* konstansoknak egymástól különbözniük kell. Ha az ellenőrzött érték egyikkel sem egyezik, a vezérlés a *default*-ot választja. A programozónak nem kell szükségszerűen erről az alapértelmezett lehetőségről gondoskodnia.

Gyakran akarunk karaktersorozatot olvasni. Ennek kényelmes módja egy *string*-be való helyezés:

```
int main()
{
  string str;
```

```
    cout << "Írja be a nevét!\n";
    cin >> str;
    cout << "Helló, " << str << "!\n";
}
```

Ha begépeljük a következőt

```
Erik
```

a válasz az alábbi lesz:

```
Helló, Erik!
```

Alapértelmezés szerint az olvasást egy „üreshely” (whitespace) karakter (§5.5.2), például egy szóköz fejezi be, tehát ha beírjuk a hírhedt király nevét

```
Erik a Véreskezű
```

a válasz marad

```
Helló, Erik!
```

A *getline()* függvény segítségével egész sort is beolvashatunk:

```
int main()
{
    string str;

    cout << "Írja be a nevét!\n";
    getline(cin,str);
    cout << "Helló, " << str << "!\n";
}
```

E programmal az alábbi bemenet

```
Erik a Véreskezű
```

a kívánt kimenetet eredményezi:

```
Helló, Erik a Véreskezű!
```

A szabványos karakterláncoknak megvan az a szép tulajdonságuk, hogy rugalmasan bővítik a tartalmukat azzal, amit beviszünk, tehát ha néhány megabájtnyi pontosvesszőt adunk meg, a program valóban több oldalnyi pontosvesszőt ad vissza – hacsak gépünk vagy az operációs rendszer valamilyen kritikus erőforrása előbb el nem fogy.

3.7. Tárolók

Sok számítás jár különböző objektumformákból álló gyűjtemények (collection) létrehozásával és kezelésével. Egy egyszerű példa karakterek karakterláncba helyezése, majd a karakterlánc kiíratása. Az olyan osztályt, melynek fő célja objektumok tárolása, általánosan *tárolónak* (container, konténer) nevezzük. Adott feladathoz megfelelő tárolókról gondoskodni és ezeket támogatni bármilyen program építésénél nagy fontossággal bír.

A standard könyvtár leghasznosabb tárolóinak bemutatására nézzünk meg egy egyszerű programot, amely neveket és telefonszámokat tárol. Ez az a fajta program, amelynek változatai az eltérő hátterű emberek számára is „egyszerűnek és maguktól értetődőnek” tűnnek.

3.7.1. Vektor

Sok C programozó számára alkalmas kiindulásnak látszana egy beépített (név- vagy szám-) párokból álló tömb:

```
struct Entry {
  string name;
  int number;
};

Entry phone_book[1000];

void print_entry(int i)          // egyszerű használat
{
  cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
```

A beépített tömbök mérete azonban rögzített. Ha nagy méretet választunk, helyet pazarolunk; ha kisebbet, a tömb túl fog csordulni. Mindkét esetben alacsonyszintű tárkezelő kódot kell írnunk. A standard könyvtár a *vector* típust (§16.3) bocsátja rendelkezésre, amely megoldja a fentieket:

```
vector<Entry> phone_book(1000);

void print_entry(int i)                 // egyszerű használat, mint a tömbnél
{
  cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}

void add_entries(int n)                 // méret növelése n-nel
{
  phone_book.resize(phone_book.size()+n);
}
```

A *vector size()* tagfüggvénye megadja az elemek számát. Vegyük észre a *()* zárójelek használatát a *phone_book* definíciójában. Egyetlen *vector<Entry>* típusú objektumot hoztunk létre, melynek megadtuk a kezdeti méretét. Ez nagyon különbözik a beépített tömbök bevezetésétől:

```
vector<Entry> book(1000);    // vektor 1000 elemmel
vector<Entry> books[1000];   // 1000 üres vektor
```

Ha hibásan *[ ]* –t (szögletes zárójelet) használnánk ott, ahol egy *vector* deklarálásában *()*-t értettünk, a fordító majdnem biztos, hogy hibaüzenetet ad, amikor a *vector*-t használni próbáljuk.

A *vector* egy példánya objektum, melynek értéket adhatunk:

```
void f(vector<Entry>& v)
{
  vector<Entry> v2 = phone_book;
  v = v2;
  // ...
}
```

A *vector*-ral való értékadás az elemek másolásával jár. Tehát *f()*-ben az előkészítés (inicializálás) és értékadás után *v* és *v2* is egy-egy külön másolatot tartalmaz a *phone_book*-ban lévő minden egyes *Entry*-ről. Ha egy vektor sok elemet tartalmaz, az ilyen ártatlannak látszó értékadások megengedhetetlenül „költségesek”. Ahol a másolás nem kívánatos, referenciákat (hivatkozásokat) vagy mutatókat kell használni.

3.7.2. Tartományellenőrzés

A standard könyvtárbeli *vector* alapértelmezés szerint nem gondoskodik tartományellenőrzésről (§16.3.3). Például:

```
void f()
{
  int i = phone_book[1001].number;    // az 1001 kívül esik a tartományon
  // ...
}
```

A kezdeti értékadás valószínűleg inkább valamilyen véletlenszerű értéket tesz *i*-be, mint hogy hibát okoz. Ez nem kívánatos, ezért a soron következő fejezetekben a *vector* egy egyszerű tartományellenőrző átalakítását fogjuk használni, *Vec* néven. A *Vec* olyan, mint a *vector*, azzal a különbséggel, hogy *out_of_range* típusú kivételt vált ki, ha egy index kifut a tartományából.

A *Vec*-hez hasonló típusok megvalósítási módjait és a kivételek hatékony használatát §11.12, §8.3 és a 14. fejezet tárgyalja. Az itteni definíció azonban elegendő a könyv példáihoz:

```
template<class T> class Vec : public vector<T> {
public:
  Vec() : vector<T>() { }
  Vec(int s) : vector<T>(s) { }

  T& operator[ ](int i) { return at(i); }                     // tartományellenőrzés
  const T& operator[ ](int i) const { return at(i); }         // tartományellenőrzés
};
```

Az *at(i)* egy *vector* indexművelet, mely *out_of_range* típusú kivételt vált ki, ha paramétere kifut a *vector* tartományából (§16.3.3).

Visszatérve a nevek és telefonszámok tárolásának problémájához, most már használhatjuk a *Vec*-et, biztosítva, hogy a tartományon kívüli hozzáféréseket elkapjuk:

```
Vec<Entry> phone_book(1000);

void print_entry(int i)                    // egyszerű használat, mint a vektornál
{
  cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
```

A tartományon kívüli hozzáférés kivételt fog kiváltani, melyet a felhasználó elkaphat:

```
void f()
{
  try {
        for (int i = 0; i<10000; i++) print_entry(i);
  }
  catch (out_of_range) {
        cout << "tartományhiba\n";
  }
}
```

A kivétel „dobása” majd elkapása akkor történik, amikor a *phone_book[i]*-re *i==1000* értékkel történik hozzáférési kísérlet. Ha a felhasználó nem kapja el ezt a fajta kivételt, a program meghatározott módon befejeződik; nem folytatja futását és nem vált ki meghatározatlan hibát. A kivételek okozta meglepetések csökkentésének egyik módja, ha a *main()* törzsében egy *try* blokkot hozunk létre:

```
int main()
try {
  // saját kód
}
catch (out_of_range) {
  cerr << "tartományhiba\n";
}
catch (...) {
  cerr << "ismeretlen kivétel\n";
}
```

Ez gondoskodik az alapértelmezett kivételkezelőkről, tehát ha elmulasztunk elkapni egy kivételt, a *cerr* szabványos hibakimeneti adatfolyamon hibajelzés jelenik meg (§21.2.1).

3.7.3. Lista

A telefonkönyv-bejegyzések beszúrása és törlése általánosabb lehet, ezért egy egyszerű telefonkönyv ábrázolására egy lista jobban megfelelne, mint egy vektor:

```
list<Entry> phone_book;
```

Amikor listát használunk, az elemekhez nem sorszám alapján szeretnénk hozzáférni, ahogy a vektorok esetében általában tesszük. Ehelyett átkutathatjuk a listát, adott értékű elemet keresve.

Ehhez kihasználjuk azt a tényt, hogy a *list* egy sorozat (§3.8):

```
void print_entry(const string& s)
{
  typedef list<Entry>::const_iterator LI;

  for (LI i = phone_book.begin(); i != phone_book.end(); ++i) {
        Entry& e = *i;                                  // rövidítés referenciával
        if (s == e.name) {
              cout << e.name << ' ' << e.number << '\n';
              return;
        }
  }
}
```

Az *s* keresése a lista elejénél kezdődik, és addig folytatódik, míg az *s*-t megtaláljuk vagy elérünk a lista végéhez. Minden standard könyvtárbeli tároló tartalmazza a *begin()* és *end()* függvényeket, melyek egy bejárót (iterátort) adnak vissza az első, illetve az utolsó utáni elemre (§16.3.2). Ha adott egy *i* bejáró, a következő elem *++i* lesz. Az *i* változó a **i* elemre hivatkozik. A felhasználónak nem kell tudnia, pontosan milyen típusú egy szabványos tároló bejárója. A típus a tároló leírásának része és név szerint lehet hivatkozni rá. Ha nincs szükségünk egy tárolóelem módosítására, a *const_iterator* az a típus, ami nekünk kell. Különben a sima *iterator* típust (§16.3.1) használjuk.

Elemek hozzáadása egy *list*-hez igen könnyű:

```
void add_entry(const Entry& e, list<Entry>::iterator i)
{
  phone_book.push_front(e);          // hozzáadás a lista elejéhez
  phone_book.push_back(e);           // hozzáadás a lista végéhez
  phone_book.insert(i,e);            // hozzáadás az 'i' által mutatott elem elé
}
```

3.7.4. Asszociatív tömbök

Egy név- vagy számpárokból álló listához kereső kódot írni valójában igen fáradságos munka. Ezenkívül a sorban történő keresés – a legrövidebb listák kivételével – nagyon rossz hatékonyságú. Más adatszerkezetek közvetlenül támogatják a beszúrást, a törlést és az érték szerinti keresést. A standard könyvtár nevezetesen a *map* típust biztosítja erre a feladatra (§17.4.1). A *map* egy értékpár-tároló. Például:

```
map<string,int> phone_book;
```

Más környezetekben a *map* mint asszociatív tömb vagy szótár szerepel. Ha első típusával (a *kulccsal,* key) indexeljük, a map a második típus (az *érték*, vagyis a *hozzárendelt típus*, mapped type) megfelelő értékét adja vissza:

```
void print_entry(const string& s)
{
  if (int i = phone_book[s]) cout << s << ' ' << i << '\n';
}
```

Ha nem talál illeszkedést az *s* kulcsra, a *phone_book* egy alapértéket ad vissza. A *map* alapértéke *int* típusra *0*. Itt feltételezzük, hogy a *0* nem érvényes telefonszám.

3.7.5. Szabványos tárolók

Az asszociatív tömb, a lista és a vektor mind használható telefonkönyv ábrázolására. Mindegyiknek megvannak az erős és gyenge oldalai. A vektorokat indexelni „olcsó" és könnyű. Másrészt két eleme közé egyet beszúrni költségesebb lehet. A lista tulajdonságai ezzel pontosan ellentétesek. A *map* emlékeztet egy (kulcs–érték) párokból álló listára, azzal a kivétellel, hogy értékei a kulcs szerinti kereséshez a legmegfelelőbbek.

A standard könyvtár rendelkezik a legáltalánosabb és leghasználhatóbb tárolótípusokkal, ami lehetővé teszi, hogy a programozók olyan tárolót válasszanak, mely az adott alkalmazás igényeit a legjobban kiszolgálja:

Szabványos tárolók összefoglalása	
Vector<T>	Változó hosszúságú vektor (§16.3)
list<T>	Kétirányú láncolt lista (§17.2.2)
Queue<T>	Sor (§17.3.2)
Stack<T>	Verem (§17.3.1)
Deque<T>	Kétvégű sor (§17.2.3)
Priority_queue<T>	Érték szerint rendezett sor (§17.3.3)
set<T>	Halmaz (§17.4.3)
Multiset<T>	Halmaz, melyben egy érték többször is előfordulhat (§17.4.4)
Map<kulcs,érték>	Asszociatív tömb (§17.4.1)
Multimap<kulcs,érték>	Asszociatív tömb, melyben egy kulcs többször előfordulhat (§17.4.2)

A szabványos tárolókat §16.2, §16.3 és a 17. fejezet mutatja be. A tárolók az *std* névtérhez tartoznak, leírásuk a *<vector>*, *<list>*, *<map>* stb. fejállományokban szerepel. A szabványos tárolók és alapműveleteik jelölés szempontjából hasonlóak, továbbá a műveletek jelentése a különböző tárolókra nézve egyforma. Az alapműveletek általában mindenfajta tárolóra alkalmazhatók. A *push_back()* például – meglehetősen hatékony módon – egyaránt használható elemeknek egy vektor vagy lista végéhez fűzésére, és minden tárolónak van *size()* tagfüggvénye, mely visszaadja az elemek a számát.Ez a jelölésbeli és jelentésbeli egységesség lehetővé teszi, hogy a programozók új tárolótípusokat készítsenek, melyek a szabványos típusokhoz nagyon hasonló módon használhatók. A *Vec* tartományellenőrzéssel ellátott vektor (§3.7.6) ennek egy példája. A 17. fejezet bemutatja, hogyan lehet egy *hash_map*-et a szerkezethez hozzátenni. A tárolófelületek egységes volta emellett lehetővé teszi azt is, hogy az egyes tárolótípusoktól függetlenül adjunk meg algoritmusokat.

3.8. Algoritmusok

Az adatszerkezetek, mint a *list* vagy a *vector*, önmagukban nem túl hasznosak. Használatukhoz olyan alapvető hozzáférési műveletekre van szükség, mint az elemek hozzáadása és eltávolítása. Emellett ritkán használunk egy tárolót pusztán tárolásra. Rendezzük, kiíratjuk, részhalmazokat vonunk ki belőlük, elemeket távolítunk el, objektumokat keresünk bennük és így tovább. Emiatt a standard könyvtár az általános tárolótípusokon kívül biztosítja a tárolók legáltalánosabb eljárásait is. A következő kódrészlet például egy *vector*-t rendez és minden egyedi *vector* elem másolatát egy *list*-be teszi:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
  sort(ve.begin(),ve.end());
  unique_copy(ve.begin(),ve.end(),le.begin());
}
```

A szabványos algoritmusok leírását a 18. fejezetben találjuk. Az algoritmusok elemek sorozatával működnek (§2.7.2). Az ilyen sorozatok bejáró-párokkal ábrázolhatók, melyek az első, illetve az utolsó utáni elemet adják meg. A példában a *sort()* rendezi a sorozatot, *ve.begin()*-től *ve.end()*-ig, ami éppen a *vector* összes elemét jelenti. Íráshoz csak az első írandó elemet szükséges megadni. Ha több mint egy elemet írunk, a kezdő elemet követő elemek felülíródnak. Ha az új elemeket egy tároló végéhez kívánnánk adni, az alábbit írhatnánk:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
  sort(ve.begin(),ve.end());
  unique_copy(ve.begin(),ve.end(),back_inserter(le));      // hozzáfűzés le-hez
}
```

A *back_inserter()* elemeket ad egy tároló végéhez, bővítve a tárolót, hogy helyet csináljon részükre (§19.2.4). A szabványos tárolók és a *back_inserter()*-ek kiküszöbölik a hibalehetőséget jelentő, C stílusú *realloc()*-ot használó tárkezelést (§16.3.5). Ha a hozzáfűzéskor elfelejtjük a *back_inserter()*-t használni, az hibákhoz vezethet:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
  copy(ve.begin(),ve.end(),le);                    // hiba: le nem bejáró
  copy(ve.begin(),ve.end(),le.end());              // rossz: túlír a végén
  copy(ve.begin(),ve.end(),le.begin());            // elemek felülírása
}
```

3.8.1. Bejárók használata

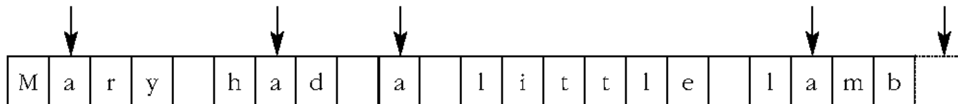
Amikor először találkozunk egy tárolóval, megkaphatjuk néhány hasznos elemére hivatkozó bejáróját (iterátorát); a legjobb példa a *begin()* és az *end()*. Ezenkívül sok algoritmus ad vissza bejárókat. A *find* szabványos algoritmus például egy sorozatban keres egy értéket és azt a bejárót adja vissza, amely a megtalált elemre mutat. Ha a *find*-ot használjuk, megszámlálhatjuk valamely karakter előfordulásait egy karakterláncban:

```
int count(const string& s, char c)       // c előfordulásainak megszámlálása s-ben
{
  int n = 0;
  string::const_iterator i = find(s.begin(),s.end(),c);
  while (i != s.end()) {
            ++n;
            i = find(i+1,s.end(),c);
  }
  return n;
}
```

A *find* algoritmus valamely érték egy sorozatban való első előfordulására vagy a sorozat utolsó utáni elemére mutató bejárót ad vissza. Lássuk, mi történik a *count* egyszerű meghívásakor:

```
void f()
{
  string m = "Mary had a little lamb";
  int a_count = count(m,'a');
}
```

A *find()* első hívása megtalálja '*a*'-t a *Mary*-ben. A bejáró tehát e karakterre mutat és nem az *s.end()*-re, így belépünk a ciklusba. A ciklusban *i+1*-gyel kezdjük a keresést; vagyis eggyel a megtalált '*a*' után. Ezután folytatjuk a ciklust és megtaláljuk a másik három '*a*'-t. A *find()* ekkor elér a sorozat végéhez és az *s.end()*-et adja vissza, így nem teljesül az *i!=s.end()* feltétel és kilépünk a ciklusból. Ezt a *count()* hívást grafikusan így ábrázolhatnánk:



A nyilak az *i* kezdeti, közbenső és végső értékeit mutatják.

Természetesen a *find* algoritmus minden szabványos tárolón egyformán fog működni. Következésképpen ugyanilyen módon általánosíthatnánk a *count()* függvényt:

```
template<class C, class T> int count(const C& v, T val)
{
  typename C::const_iterator i = find(v.begin(),v.end(),val);      // typename, lásd §C.13.5
  int n = 0;
  while (i != v.end()) {
          ++n;
          ++i;                                    // az előbb megtalált elem átugrása
          i = find(i,v.end(),val);
  }
  return n;
}
```

Ez működik, így mondhatjuk:

```
void f(list<complex>& lc, vector<string>& vc, string s)
{
  int i1 = count(lc,complex(1,3));
  int i2 = count(vc,"Diogenész");
  int i3 = count(s,'x');
}
```

A *count* sablont azonban nem kell definiálnunk. Az elemek előfordulásainak megszámlálása annyira általános és hasznos, hogy a standard könyvtár tartalmazza ezt a műveletet. Hogy teljesen általános legyen a megoldás, a standard könyvtári *count* paraméterként tároló helyett egy sorozatot kap:

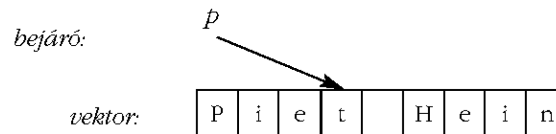
```
void f(list<complex>& lc, vector<string>& vs, string s)
{
   int i1 = count(lc.begin(),lc.end(),complex(1,3));
   int i2 = count(vs.begin(),vs.end(),"Diogenész");
   int i3 = count(s.begin(),s.end(),'x');
}
```

A sorozat használata megengedi, hogy a számlálást beépített tömbre használjuk és azt is, hogy egy tároló valamely részét számláljuk:

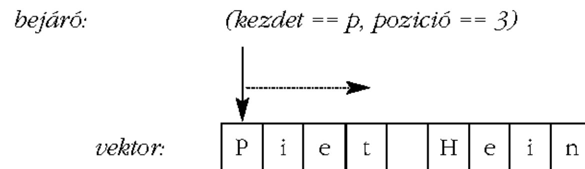
```
void g(char cs[ ], int sz)
{
   int i1 = count(&cs[0],&cs[sz],'z');       // 'z'-k a tömbben
   int i2 = count(&cs[0],&cs[sz/2],'z');     // 'z'-k a tömb első felében
}
```

3.8.2. Bejárótípusok

Valójában mik is azok a *bejárók* (iterátorok)? Minden bejáró valamilyen típusú objektum. Azonban sok különböző típusuk létezik, mert a bejárónak azt az információt kell tárolnia, melyre az adott tárolótípusnál feladata ellátásához szüksége van. Ezek a bejárótípusok olyan különbözők lehetnek, mint a tárolók és azok az egyedi igények, melyeket kiszolgálnak. Egy *vector* bejárója például minden bizonnyal egy közönséges mutató, mivel az nagyon ésszerű hivatkozási mód a *vector* elemeire:

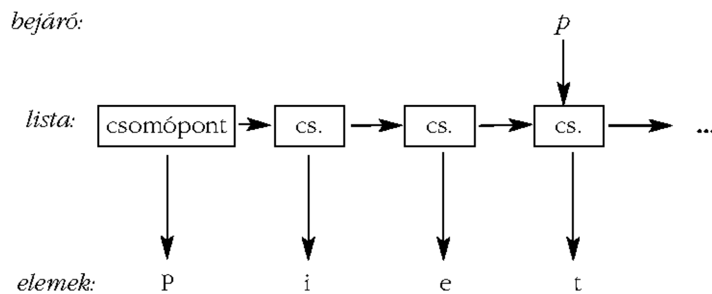


Egy *vector*-bejáró megvalósítható úgy is, mint a *vector*-ra hivatkozó mutató, meg egy sorszám:



Az ilyen bejárók használata tartományellenőrzésre is lehetőséget ad (§19.3).

A listák bejáróinak valamivel bonyolultabbnak kell lenniük, mint egy egyszerű mutató a listában tárolt elemre, mivel egy ilyen elem általában nem tudja, hol van a lista következő tagja. A lista-bejáró tehát inkább a lista valamely csomópontjára hivatkozó mutató:



Ami közös minden bejárónál, az a jelentésük és a műveleteik elnevezése. A ++ alkalmazása bármely bejáróra például olyan bejárót ad, mely a következő elemre hivatkozik. Hasonlóképpen * azt az elemet adja meg, melyre a bejáró hivatkozik. Valójában bejáró lehet bármely objektum, amely néhány, az előzőekhez hasonló egyszerű szabálynak eleget tesz (§19.2.1). Továbbá, a felhasználó ritkán kell, hogy ismerje egy adott bejárótípusát. Saját bejáró-típusait minden tároló ismeri és azokat az egyezményes *iterator* és *const_iterator* neveken rendelkezésre bocsátja. Például a *list<Entry>::iterator* a *list<Entry>* általános bejárótípusa. Ritkán kell aggódnunk az adott típus meghatározása miatt.

3.8.3. Bemeneti és kimeneti bejárók

A bejárók fogalma általános és hasznos a tárolók elemeiből álló sorozatok kezelésénél. A tárolók azonban nem az egyetlen helyet jelentik, ahol elemek sorozatát találjuk. A bemeneti adatfolyamok is értékek sorozatából állnak, és értékek sorozatát írjuk a kimeneti adatfolyamba is. Következésképpen a bejárók hasznosan alkalmazhatók a bemenetnél és kimenetnél is.

Egy *ostream_iterator* létrehozásához meg kell határoznunk, melyik adatfolyamot használjuk és milyen típusú objektumokat írunk bele. Megadhatunk például egy bejárót, mely a *cout* szabványos kimeneti adatfolyamra hivatkozik:

```
ostream_iterator<string> oo(cout);
```

Az értékadás **oo*-nak azt jelenti, hogy az értékadó adatot a *cout*-ra írjuk ki. Például:

```
int main()
{
   *oo = "Helló, ";     // jelentése cout << "Helló, "
   ++oo;
   *oo = "világ!\n";   // jelentése cout << "világ!\n"
}
```

Ez egy újabb mód szabványos üzenetek kiírására a szabványos kimenetre. A *++oo* célja: utánozni egy tömbbe mutatón keresztül történő írást. A szerző elsőnek nem ezt a módot választaná erre az egyszerű feladatra, de ez az eszköz alkalmas arra, hogy a kimenetet úgy kezeljük, mint egy csak írható tárolót, ami hamarosan magától értetődő lesz – ha eddig még nem lenne az.

Hasonlóképpen az *istream_iterator* olyasvalami, ami lehetővé teszi, hogy a bemeneti adatfolyamot úgy kezeljük, mint egy csak olvasható tárolót. Itt is meg kell adnunk a használni kívánt adatfolyamot és a várt értékek típusát:

```
istream_iterator<string> ii(cin);
```

Mivel a bejárók mindig párokban ábrázolnak egy sorozatot, a bemenet végének jelzéséhez egy *istream_iterator*-ról kell gondoskodni. Az alapértelmezett *istream_iterator* a következő:

```
istream_iterator<string> eos;
```

Most újra beolvashatnánk a „*Helló, világot!*”-ot a bemenetről és kiírathatnánk az alábbi módon:

```
int main()
{
  string s1 = *ii;
  ++ii;
  string s2 = *ii;

  cout << s1 << ' ' << s2 << '\n';
}
```

Valójában az *istream_iterator*-okat és *ostream_ iterator*-okat nem közvetlen használatra találták ki. Jellemzően algoritmusok paramétereiként szolgálnak. Írjunk például egy egyszerű programot, mely egy fájlból olvas, az olvasott adatokat rendezi, a kétszer szereplő elemeket eltávolítja, majd az eredményt egy másik fájlba írja:

```
int main()
{
  string from, to;
  cin >> from >> to;                  // a forrás- és célfájl nevének beolvasása

  ifstream is(from.c_str());          // bemeneti adatfolyam (c_str(), lásd §3.5.1 és §20.3.7)
  istream_iterator<string> ii(is);    // bemeneti bejáró az adatfolyam számára
  istream_iterator<string> eos;       // bemenet-ellenőrzés


  vector<string> b(ii,eos);    // b egy vektor, melynek a bemenetről adunk kezdőértéket
  sort(b.begin(),b.end());     // az átmeneti tár (b) rendezése

  ofstream os(to.c_str());     // kimeneti adatfolyam
  ostream_iterator<string> oo(os,"\n");    // kimeneti bejáró az adatfolyam számára

  unique_copy(b.begin(),b.end(),oo);       // b tartalmának a kimenetre másolása,
                                           // a kettőzött értékek elvetése

  return !is.eof() || !os;                 // hibaállapot visszaadása (§3.2, §21.3.3)
}
```

Az *ifstream* egy *istream*, mely egy fájlhoz kapcsolható, az *ofstream* pedig egy *ostream*, mely szintén egy fájlhoz kapcsolható. Az *ostream_iterator* második paramétere a kimeneti értékeket elválasztó jel.

3.8.4. Bejárások és predikátumok

A bejárók lehetővé teszik, hogy ciklusokat írjunk egy sorozat bejárására. A ciklusok megírása azonban fáradságos lehet, ezért a standard könyvtár módot ad arra, hogy egy adott függvényt a sorozat minden egyes elemére meghívjunk. Tegyük fel, hogy írunk egy programot, mely a bemenetről szavakat olvas és feljegyzi előfordulásuk gyakoriságát. A karakterláncok és a hozzájuk tartozó gyakoriságok kézenfekvő ábrázolása egy *map*-pel történhet:

```
map<string,int> histogram;
```

Az egyes karakterláncok gyakoriságának feljegyzésére természetes művelet a következő:

```
void record(const string& s)
{
  histogram[s]++;     // "s" gyakoriságának rögzítése
}
```

Ha beolvastuk a bemenetet, szeretnénk az összegyűjtött adatokat a kimenetre küldeni. A *map* (*string,int*) párokból álló sorozat. Következésképpen szeretnénk meghívni az alábbi függvényt

```
void print(const pair<const string,int>& r)
{
  cout << r.first << ' ' << r.second << '\n';
}
```

a *map* minden elemére (a párok (*pair*) első elemét *first*-nek, második elemét *second*-nak nevezzük). A *pair* első eleme *const string*, nem sima *string*, mert minden *map* kulcs konstans.

A főprogram tehát a következő:

```
int main()
{
  istream_iterator<string> ii(cin);
  istream_iterator<string> eos;

  for_each(ii,eos,record);
  for_each(histogram.begin(),histogram.end(),print);
}
```

Vegyük észre, hogy nem kell rendeznünk a *map*-et ahhoz, hogy a kimenet rendezett legyen. A *map* rendezve tárolja az elemeket és a ciklus is (növekvő) sorrendben járja végig a *map*-et.

Sok programozási feladat szól arról, hogy meg kell keresni valamit egy tárolóban, ahelyett, hogy minden elemen végrehajtanánk egy feladatot. A *find* algoritmus (§18.5.2) kényelmes módot ad egy adott érték megkeresésére. Ennek az ötletnek egy általánosabb változata olyan elemet keres, mely egy bizonyos követelménynek felel meg. Például meg akarjuk keresni egy *map* első *42*-nél nagyobb értékét:

```
bool gt_42(const pair<const string,int>& r)
{
  return r.second>42;
}

void f(map<string,int>& m)
{
  typedef map<string,int>::const_iterator MI;
  MI i = find_if(m.begin(),m.end(),gt_42);
  // ...
}
```

Máskor megszámlálhatnánk azon szavakat, melyek gyakorisága nagyobb, mint *42*:

```
void g(const map<string,int>& m)
{
  int c42 = count_if(m.begin(),m.end(),gt_42);
  // ...
}
```

Az olyan függvényeket, mint a *gt_42()*, melyet az algoritmus vezérlésére használunk, *predikátumnak* („állítmány", vezérlőfüggvény, predicate) nevezzük. Ezek minden elemre meghívódnak és logikai értéket adnak vissza, melyet az algoritmus szándékolt tevékenységének elvégzéséhez felhasznál. A *find_if()* például addig keres, amíg a predikátuma *true*-t nem ad vissza, jelezvén, hogy a kért elemet megtalálta. Hasonló módon a *count_if()* annyit számlál, ahányszor a predikátuma *true*.

A standard könyvtár néhány hasznos predikátumot is biztosít, valamint olyan sablonokat, melyek továbbiak alkotására használhatók (§18.4.2).

3.8.5. Tagfüggvényeket használó algoritmusok

Sok algoritmus alkalmaz függvényt egy sorozat elemeire. Például §3.8.4-ben a

```
for_each(ii,eos,record);
```

meghívja a *record()*-ot minden egyes, a bemenetről beolvasott karakterláncra.

Gyakran mutatók tárolóival van dolgunk, és sokkal inkább a hivatkozott objektum egy tagfüggvényét szeretnénk meghívni, nem pedig egy globális függvényt, a mutatót paraméterként átadva. Tegyük fel, hogy a *Shape::draw()* tagfüggvényt akarjuk meghívni egy *list<Shape>** elemeire. A példa kezelésére egyszerűen egy nem tag függvényt írunk, mely meghívja a tagfüggvényt:

```
void draw(Shape* p)
{
  p->draw();
}

void f(list<Shape*>& sh)
{
  for_each(sh.begin(),sh.end(),draw);
}
```

A módszert így általánosíthatjuk:

```
void g(list<Shape*>& sh)
{
  for_each(sh.begin(),sh.end(),mem_fun(&Shape::draw));
}
```

A standard könyvtári *mem_fun()* sablon (§18.4.4.2) paraméterként egy tagfüggvény mutatóját kapja (§15.5) és valami olyasmit hoz létre, amit a tag osztályára hivatkozó mutatón keresztül hívhatunk meg. A *mem_fun(&Shape::draw)* eredménye egy *Shape** paramétert kap és visszaadja, amit a *Shape::draw()* visszaad.

A *mem_fun()* azért fontos, mert megengedi, hogy a szabványos algoritmusokat többalakú (polimorf) objektumok tárolóira használjuk.

3.8.6. A standard könyvtár algoritmusai

Mi az algoritmus? Általános meghatározása szerint „szabályok véges halmaza, mely adott problémahalmaz megoldásához műveletek sorozatát határozza meg és öt fontos jellemzője van: végesség, meghatározottság, bemenet, kimenet, hatékonyság" [Knuth,1968, §1.1]. A C++ standard könyvtárának viszonylatában az algoritmus elemek sorozatán műveleteket végző sablonok (template-ek) halmaza.

A standard könyvtár több tucat algoritmust tartalmaz. Az algoritmusok az *std* névtérhez tartoznak, leírásuk az *<algorithm>* fejállományban szerepel. Íme néhány, melyeket különösen hasznosnak találtam:

Válogatott szabványos algoritmusok	
for_each()	Hívd meg a függvényt minden elemre (§18.5.1)
find()	Keresd meg a paraméterek első előfordulását (§18.5.2)
find_if()	Keresd meg a predikátumra az első illeszkedést (§18.5.2)
count()	Számláld meg az elem előfordulásait (§18.5.3)
count_if()	Számláld meg az illeszkedéseket a predikátumra (§18.5.3)
replace()	Helyettesítsd be az elemet új értékkel (§18.6.4)
replace_if()	Helyettesítsd be a predikátumra illeszkedő elemet új értékkel (§18.6.4)
copy()	Másold az elemeket (§18.6.1)
unique_copy()	Másold a csak egyszer szereplő elemeket (§18.6.1)
sort()	Rendezd az elemeket (§18.7.1)
equal_range()	Keresd meg az összes egyező értékű elemet (§18.7.2)
merge()	Fésüld össze a rendezett sorozatokat (§18.7.3)

3.9. Matematika

A C-hez hasonlóan a C++ nyelvet sem elsősorban számokkal végzett műveletekre tervezték. Mindemellett rengeteg numerikus munkát végeztek C++-ban és ez tükröződik a standard könyvtárban is.

3.9.1. Komplex számok

A standard könyvtár a komplex számok egy típuscsaládját tartalmazza, a §2.5.2-ben leírt *complex* osztály alakjában. Az egyszeres pontosságú lebegőpontos *(float)*, a kétszeres pontosságú *(double)* stb. skalárokat tartalmazó komplex számok támogatására a standard könyvtárbeli *complex* egy sablon:

```
template<class scalar> class complex {
public:
  complex(scalar re, scalar im);
  // ...
};
```

A szokásos aritmetikai műveletek és a leggyakrabban használt matematikai függvények komplex számokkal is működnek:

```
// szabványos exponenciális függvény a <complex> sablonból:
template<class C> complex<C> pow(const complex<C>&, int);

void f(complex<float> fl, complex<double> db)
{
  complex<long double> ld = fl+sqrt(db);
  db += fl*3;
  fl = pow(1/fl,2);
  // ...
}
```

Részletesebben lásd §22.5.

3.9.2. Vektoraritmetika

A §3.7.1-ben leírt *vector*-t általános értéktárolásra tervezték; kellően rugalmas és illeszkedik a tárolók, bejárók és algoritmusok szerkezetébe, ugyanakkor nem támogatja a matematikai vektorműveleteket. Ilyen műveleteket könnyen be lehetett volna építeni a *vector*-ba, de az általánosság és rugalmasság eleve kizár olyan optimalizálásokat, melyeket komolyabb, számokkal végzett munkánál gyakran lényegesnek tekintünk. Emiatt a standard könyvtárban megtaláljuk a *valarray* nevű vektort is, mely kevésbé általános és a számműveletekhez jobban megfelel:

```
template<class T> class valarray {
  // ...
  T& operator[](size_t);
  // ...
};
```

A *size_t* előjel nélküli egész típus, melyet a nyelv tömbök indexelésére használ. A szokásos aritmetikai műveleteket és a leggyakoribb matematikai függvényeket megírták *a valarray*-kre is:

```
// szabványos abszolútérték-függvény a <valarray> sablonból:
template<class T> valarray<T> abs(const valarray<T>&);

void f(valarray<double>& a1, valarray<double>& a2)
{
  valarray<double> a = a1*3.14+a2/a1;
  a2 += a1*3.14;
  a = abs(a);
  double d = a2[7];
  // ...
}
```

Részletesebben lásd: §22.4

3.9.3. Alapszintű numerikus támogatás

A standard könyvtár a lebegőpontos típusokhoz természetesen tartalmazza a leggyakoribb matematikai függvényeket (*log()*, *pow()* és *cos()*, lásd §2.2.3). Ezenkívül azonban tartalmaz olyan osztályokat is, melyek beépített típusok tulajdonságait – például egy *float* kitevőjének lehetséges legnagyobb értékét –írják le (lásd §22.2).

3.10. A standard könyvtár szolgáltatásai

A standard könyvtár szolgáltatásait az alábbi módon osztályozhatjuk:

1. Alapvető futási idejű támogatás (pl. tárlefoglalás és futási idejű típusinformáció), lásd §16.1.3.
2. A szabványos C könyvtár (nagyon csekély módosításokkal, a típusrendszer megsértésének elkerülésére), lásd §16.1.2.
3. Karakterláncok és bemeneti/kimeneti adatfolyamok (nemzetközi karakterkészlet és nyelvi támogatással), lásd 20. és 21. fejezet.
4. Tárolók (*vector*, *list* és *map*) és tárolókat használó algoritmusok (általános bejárások, rendezések és összefésülések) rendszere, lásd 16., 17., 18. és 19.fejezet.

5. Számokkal végzett műveletek támogatása (komplex számok és vektorok aritmetikai műveletekkel), BLAS-szerű és általánosított szeletek, valamint az optimalizálást megkönnyítő szerkezetek, lásd 22. fejezet.

Annak fő feltétele, hogy egy osztály bekerülhet-e a könyvtárba, az volt, hogy valamilyen módon használta-e már majdnem minden C++ programozó (kezdők és szakértők egyaránt), hogy általános alakban megadható-e, hogy nem jelent-e jelentős többletterhelést ugyanennek a szolgáltatásnak valamely egyszerűbb változatához viszonyítva, és hogy könnyen megtanulható-e a használata. A C++ standard könyvtára tehát az alapvető adatszerkezeteket és az azokon alkalmazható alapvető algoritmusokat tartalmazza.

Minden algoritmus átalakítás nélkül működik minden tárolóra. Ez az egyezményesen STL-nek (Standard Template Library, szabványos sablonkönyvtár) [Stepanov, 1994] nevezett váz bővíthető, abban az értelemben, hogy a felhasználók a könyvtár részeként megadottakon kívül könnyen készíthetnek saját tárolókat és algoritmusokat, és ezeket azonnal működtethetik is a szabványos tárolókkal és algoritmusokkal együtt.

3.11. Tanácsok

[1] Ne találjunk fel a melegvizet – használjunk könyvtárakat.
[2] Ne higgyünk a csodákban. Értsük meg, mit tesznek könyvtáraink, hogyan teszik, és milyen áron teszik.
[3] Amikor választhatunk, részesítsük előnyben a standard könyvtárat más könyvtárakkal szemben.
[4] Ne gondoljuk, hogy a standard könyvtár mindenre ideális.
[5] Ne felejtsük el beépíteni *(#include)* a felhasznált szolgáltatások fejállományait. §3.3.
[6] Ne felejtsük el, hogy a standard könyvtár szolgáltatásai az *std* névtérhez tartoznak. §3.3.
[7] Használjunk *string*-et *char** helyett. §3.5, §3.6.
[8] Ha kétségeink vannak, használjunk tartományellenőrző vektort (mint a *Vec*). §3.7.2.
[9] Részesítsük előnyben a *vector<T>*-t, a *list<T>*-t és a *map<key,value>*-t a *T[ ]*-vel szemben. §3.7.1, §3.7.3, §3.7.4.

[10] Amikor elemeket teszünk egy tárolóba, használjunk *push_back()*-et vagy *back_inserter()*-t. §3.7.3, §3.8.
[11] Használjunk vektoron *push_back()*-et a *realloc()* tömbre való alkalmazása helyett. §3.8.
[12] Az általános kivételeket a *main()*-ben kapjuk el. §3.7.2.

Első rész

Alapok

Ez a rész a C++ beépített típusait és azokat az alapvető lehetőségeket írja le, amelyekkel programokat hozhatunk létre. A C++-nak a C nyelvre visszautaló része a hagyományos programozási stílusok támogatásával együtt kerül bemutatásra, valamint ez a rész tárgyalja azokat az alapvető eszközöket is, amelyekkel C++ programot hozhatunk létre logikai és fizikai elemekből.

Fejezetek

4. Típusok és deklarációk
5. Mutatók, tömbök és struktúrák
6. Kifejezések és utasítások
7. Függvények
8. Névterek és kivételek
9. Forrásfájlok és programok

4

Típusok és deklarációk

„Ne fogadj el semmit, ami nem tökéletes!”
(ismeretlen szerző)

„A tökéletesség csak az összeomlás pontján érhető el.”
(C.N. Parkinson)

Típusok • Alaptípusok • Logikai típusok • Karakterek • Karakterliterálok • Egészek • Egész literálok • Lebegőpontos típusok • Lebegőpontos literálok • Méretek • *void* • Felsoroló típusok • Deklarációk • Nevek • Hatókörök • Kezdeti értékadás • Objektumok • *typedef*-ek • Tanácsok • Gyakorlatok

4.1. Típusok

Vegyük az

```
x = y+f(2);
```

kifejezést. Hogy ez értelmes legyen valamely C++ programban, az *x*, *y* és *f* neveket megfelelően definiálni kell, azaz a programozónak meg kell adnia, hogy ezek az *x*, *y*, és *f* nevű egyedek léteznek és olyan típusúak, amelyekre az = (értékadás), a + (összeadás) és a *()* (függvényhívás) rendre értelmezettek.

A C++ programokban minden névnek (azonosítónak) van típusa. Ez a típus határozza meg, milyen műveleteket lehet végrehajtani a néven (azaz az egyeden, amelyre a név hivatkozik) és ezek a műveletek mit jelentenek. Például a

```
float x;        // x lebegőpontos változó
int y = 7;      // y egész típusú változó, kezdőértéke 7
float f(int);   // f egész paramétert váró és lebegőpontos számot visszaadó függvény
```

deklarációk már értelmessé teszik a fenti példát. Mivel *y*-t *int*-ként adtuk meg, értékül lehet adni, használni lehet aritmetikai kifejezésekben stb. Másfelől *f*-et olyan függvényként határoztuk meg, amelynek egy *int* paramétere van, így meg lehet hívni a megfelelő paraméterrel.

Ez a fejezet az alapvető típusokat (§4.1.1) és deklarációkat (§4.9) mutatja be. A példák csak a nyelv tulajdonságait szemléltetik, nem feltétlenül végeznek hasznos dolgokat. A terjedelmesebb és valósághűbb példák a későbbi fejezetekben kerülnek sorra, amikor már többet ismertettünk a C++-ból. Ez a fejezet egyszerűen csak azokat az alapelemeket írja le, amelyekből a C++ programok létrehozhatók. Ismernünk kell ezeket az elemeket, a velük járó elnevezéseket és formai követelményeket, ahhoz is, hogy valódi C++ programot készíthessünk, de főleg azért, hogy el tudjuk olvasni a mások által írt kódot. A többi fejezet megértéséhez azonban nem szükséges teljesen átlátni ennek a fejezetnek minden apró részletét. Következésképp az olvasó jobban teszi, ha csak átnézi, hogy megértse a fontosabb fogalmakat, és később visszatér, hogy megértse a részleteket, amint szükséges.

4.1.1. Alaptípusok

A C++ rendelkezik azokkal az alaptípusokkal, amelyek megfelelnek a számítógép leggyakoribb tárolási egységeinek és adattárolási módszereinek:

§4.2 Logikai típus (*bool*)
§4.3 Karaktertípusok (mint a *char*)
§4.4 Egész típusok (mint az *int*)
§4.5 Lebegőpontos típusok (mint a *double*)

Továbbá a felhasználó megadhat:

§4.8 felsoroló típusokat adott értékhalmazok jelölésére (*enum*)

és létezik a

§4.7 *void* típus is, melyet az információ hiányának jelzésére használunk.

Ezekből a típusokból más típusokat is létrehozhatunk. Ezek a következők:

§5.1 Mutatótípusok (mint az *int**)
§5.2 Tömbtípusok (mint a *char[ ]*)
§5.5 Referencia-típusok (mint a *double&*)
§5.7 Adatszerkezetek és osztályok (10. fejezet)

A logikai, karakter- és egész típusokat együtt ***integrális típusoknak*** nevezzük, az integrális és lebegőpontos típusokat pedig közösen ***aritmetikai típusoknak***. A felsoroló típusokat és az osztályokat (10. fejezet) ***felhasználói adattípusokként*** emlegetjük, mert a felhasználónak kell azokat meghatároznia; előzetes bevezetés nélkül nem állnak rendelkezésre, mint az alaptípusok. A többi típust ***beépített típusnak*** nevezzük.

Az integrális és lebegőpontos típusok többfajta mérettel adottak, lehetővé téve a programozónak, hogy kiválaszthassa a felhasznált tár nagyságát, a pontosságot, és a számítási értéktartományt (§4.6). Azt feltételezzük, hogy a számítógép bájtokat biztosít a karakterek tárolásához, gépi szót az egészek tárolására és az azokkal való számolásra, léteznek alkalmas egyedek lebegőpontos számításokhoz és címek számára, hogy hivatkozhassunk ezekre az egyedekre. A C++ alaptípusai a mutatókkal és tömbökkel együtt az adott nyelvi megvalósítástól függetlenül biztosítják ezeket a gépszintű fogalmakat a programozó számára.

A legtöbb programban a logikai értékekhez egyszerűen *bool*-t használhatunk, karakterekhez *char*-t, egészekhez *int*-et, lebegőpontos értékekhez pedig *double*-t. A többi alaptípus hatékonysági és más egyedi célokra használatos, így legjobb azokat elkerülni addig, amíg ilyen igények fel nem merülnek, de a régi C és C++ kódok olvasásához ismernünk kell őket.

4.2. Logikai típusok

A logikai típusoknak (*bool*), két értéke lehet: *true* vagy *false* (igaz, illetve hamis). A logikai típusokat logikai műveletek eredményének kifejezésére használjuk:

```
void f(int a, int b)
{
  bool b1 = a==b;          // = értékadás, == egyenlőségvizsgálat
  // ...
}
```

Ha a és *b* értéke ugyanaz, akkor *b1* igaz lesz, máskülönben hamis.

A *bool* gyakran használatos olyan függvény visszatérési értékeként, amely valamilyen feltételt ellenőriz (predikátum):

```
bool is_open(File*);

bool greater(int a, int b) { return a>b; }
```

Egész számra alakítva a *true* értéke *1* lesz, a *false*-é pedig *0*. Ugyanígy az egészek is logikai típusúvá alakíthatók: a nem nulla egészek *true*, a 0 *false* logikai értékké alakulnak:

```
bool b = 7;        // bool(7) igaz, így b igaz
int i = true;      // int(true) értéke 1, így i értéke 1
```

Aritmetikai és logikai kifejezésekben a logikai típusok egésszé alakulnak; az aritmetikai és logikai egész-műveletek az átalakított értékeken hajtódnak végre. Ha az eredmény visszaalakul logikai típusúvá, a 0 *false* lesz, a nem nulla egészek pedig *true* értéket kapnak.

```
void g()
{
    bool a = true;
    bool b = true;

    bool x = a+b;        // a+b értéke 2, így x igaz
    bool y = a|b;        // a|b értéke 1, így y igaz
}
```

Logikai típusúvá mutatókat is alakíthatunk (§C.6.2.5). A nem nulla mutatók *true*, a „nulla mutatók” *false* értékűek lesznek.

4.3. Karaktertípusok

A *char* típusú változók egy karaktert tárolhatnak az adott nyelvi megvalósítás karakterkészletéből:

```
char ch = 'a';
```

A *char* típus általában 8 bites, így 256 különböző értéket tárolhat. A karakterkészlet jellemzően az ISO-646 egy változata, például ASCII, így a billentyűzeten megjelenő karaktereket tartalmazza. Sok probléma származik abból, hogy ezeket a karakterkészleteket csak részben szabványosították (§C.3).

Jelentősen eltérnek azok a karakterkészletek, amelyek természetes nyelveket támogatnak, és azok is, amelyek másféleképpen támogatják ugyanazt a természetes nyelvet. Itt azonban csak az érdekel minket, hogy ezek a különbségek hogyan befolyásolják a C++ szabályait. Ennél fontosabb kérdés, hogyan programozzunk többnyelvű, több karakterkészletes környezetben, ez azonban a könyv keretein túlmutat, bár számos helyen említésre kerül (§20.2, §21.7, §C3.3, §D).

Biztosan feltehető, hogy az adott C++-változat karakterkészlete tartalmazza a decimális számjegyeket, az angol ábécé 26 betűjét és néhány általános írásjelet. Nem biztos, hogy egy 8 bites karakterkészletben nincs 127-nél több karakter (néhány karakterkészlet 255 karaktert biztosít), hogy nincs több alfabetikus karakter, mint az angolban (a legtöbb európai nyelvben több van), hogy az ábécé betűi összefüggőek (az EBCDIC lyukat hagy az „i” és a „j” között), vagy hogy minden karakter rendelkezésre áll, ami a C++ kód írásához szüksé-

ges (néhány nemzeti karakterkészlet nem biztosítja a *{ } [ ] | * karaktereket, §C.3.1). Amikor csak lehetséges, nem szabad semmit feltételeznünk az objektumok ábrázolásáról és ez az általános szabály a karakterekre is vonatkozik.

Minden karakternek van egy egész értéke, a „b"-é például az ASCII karakterkészletben 98. Íme egy kis program, amely a begépelt karakter egész értékét mutatja meg:

```
#include <iostream>

int main()
{
   char c;
   std::cin >> c;
   std::cout << "A(z) '" << c << "' értéke " << int(c) << '\n';
}
```

Az *int(c)* jelölés a *c* karakter egész értékét adja. Az a lehetőség, hogy karaktert egésszé lehet alakítani, felvet egy kérdést: a *char* előjeles vagy előjel nélküli? A 8 bites bájton ábrázolt 256 értéket úgy lehet értelmezni, mint *0*-tól *255*-ig vagy *-127*-től *127*-ig terjedő értékeket. Sajnos az adott fordítóprogram dönti el, melyiket választja egy sima *char* esetében (§C.1, §C.3.4). A C++ azonban ad két olyan típust, amelyekre a kérdés biztosan megválaszolható: a *signed char*-t (előjeles karakter), amely legalább a *-127* és *127* közti értékeket képes tárolni és az *unsigned char* (előjel nélküli karakter) típust, amely legalább *0*-tól *255*-ig tud értékeket tárolni. Szerencsére csak a *0-127* tartományon kívüli értékekben lehet különbség és a leggyakoribb karakterek a tartományon belül vannak.

Azok a *0-127* tartományon túli értékek, amelyeket egy sima *char* tárol, nehezen felderíthető „hordozhatósági" problémákat okozhatnak. Lásd még a §C.3.4-et arra az esetre, ha többféle *char* típus szükséges, vagy ha *char* típusú változókban szeretnénk egészeket tárolni.

A nagyobb karakterkészletek – például a Unicode – karaktereinek tárolására a *wchar_t* áll rendelkezésünkre, amely önálló típus. Mérete az adott C++-változattól függ, de elég nagy ahhoz, hogy a szükséges legnagyobb karakterkészletet tárolhassa (lásd §21.7 és §C.3.3). A különös név még a C-ből maradt meg. A C-ben a *wchar_t* egy *typedef* (§4.9.7), vagyis típus-álnév, nem pedig beépített típus. Az *_t* toldalék a szabványos *typedef*-ektől való megkülönböztetést segíti.

Jegyezzük meg, hogy a karaktertípusok integrális típusok (§4.1.1), így alkalmazhatóak rájuk az aritmetikai és logikai műveletek (§6.2) is.

4.3.1. Karakterliterálok

A karakterliterál, melyet gyakran karakterkonstansnak is hívnak, egy egyszeres idézőjelek közé zárt karakter, például *'a'* és *'0'*. A karakterliterálok típusa *char*. Valójában szimbolikus konstansok (jelképes állandók), melyek értéke annak a számítógépnek a karakterkészletében lévő karakter egész értéke, amin a C++ program fut. Ha például ASCII karakterkészlettel rendelkező számítógépet használunk, a *'0'* értéke *48* lesz. A program hordozhatóságát javítja, ha decimális jelölés helyett karakterliterálokat használunk. Néhány karakternek szintén van szabványos neve, ezek a \ fordított perjelt használják ún. escape karakterként. Például a *\n* az új sort, a *\t* pedig a vízszintes tabulátort (behúzást) jelenti. Az escape karakterekről részletesebben lásd §C.3.2-t.

A „széles" karakterliterálok *L'ab'* alakúak, ahol az egyszeres idézőjelek között lévő karakterek számát és jelentését az adott C++-megvalósítás a *wchar_t* típushoz igazítja, mivel a széles karakterliterálok típusa *wchar_t*.

4.4. Egész típusok

A *char*-hoz hasonlóan az egész típusok is háromfélék: „sima" *int*, *signed int*, és *unsigned int*. Az egészek három méretben adottak: *short int*, „sima" *int* , illetve *long int*. Egy *long int*-re lehet sima *long*-ként hivatkozni. Hasonlóan, a *short* a *short int*, az *unsigned* az *unsigned int*, a *signed* pedig a *signed int* szinonimája.

Az *előjel nélküli* (unsigned) egész típusok olyan felhasználásra ideálisak, amely úgy kezeli a tárat, mint egy bittömböt. Szinte soha nem jó ötlet az előjel nélküli típust használni *int* helyett, hogy egy vagy több bitet nyerjünk pozitív egészek tárolásához. Azok a próbálkozások, amelyek úgy kísérlik meg biztosítani valamilyen érték nem negatív voltát, hogy a változót *unsigned*-ként adják meg, általában meghiúsulnak a mögöttes konverziós szabályok miatt (§C.6.1, §C.6.2.1).

A sima *char*-ral ellentétben a sima *int*-ek mindig előjelesek. A *signed int* típusok csak világosabban kifejezett szinonimái a nekik megfelelő sima *int* típusoknak.

4.4.1. Egész literálok

Az egész literálok négyféle alakban fordulnak elő: decimális, oktális, hexadecimális és karakterliterálként. A decimális literálok a leginkább használatosak és úgy néznek ki, ahogy elvárjuk tőlük:

7 1234 976 12345678901234567890

A fordítóprogramnak figyelmeztetnie kell olyan literálok esetében, amelyek túl hosszúak az ábrázoláshoz. A nullával kezdődő és *x*-szel folytatódó (*0x*) literálok hexadecimális (16-os számrendszerbeli) számok. Ha a literál nullával kezdődik és számjeggyel folytatódik, oktális (8-as számrendszerbeli) számról van szó:

decimális:		*2*	*63*	*83*
oktális:	*0*	*02*	*077*	*0123*
hexadecimális:	*0x0*	*0x2*	*0x3f*	*0x53*

Az *a*, *b*, *c*, *d*, *e* és *f* betűk, illetve nagybetűs megfelelőik rendre *10*-et, *11*-et, *12*-t, *13*-at, *14*-et és 15-öt jelentenek. Az oktális és hexadecimális jelölés leginkább bitminták kifejezésénél hasznos. Meglepetéseket okozhat, ha ezekkel a jelölésekkel valódi számokat fejezünk ki. Egy olyan gépen például, ahol az *int* egy kettes komplemensű 16 bites egészként van ábrázolva, *0xffff* a *-1* negatív decimális szám lesz. Ha több bitet használtunk volna az egész ábrázolására, akkor ez *65 535* lett volna.

Az *U* utótag használatával előjel nélküli (unsigned) literálokat adhatunk meg. Hasonlóan, az *L* utótag használatos a *long* literálokhoz. Például *3* egy *int*, *3U* egy *unsigned int* és *3L* egy *long int*. Ha nincs megadva utótag, a fordító egy olyan egész literált ad, amelynek típusa megfelel az értéknek és a megvalósítás egész-méreteinek (§C.4).

Jó ötlet korlátozni a nem maguktól értetődő állandók használatát néhány, megjegyzésekkel megfelelően ellátott *const* (§5.4) vagy felsoroló típusú (§4.8) kezdeti értékadására.

4.5. Lebegőpontos típusok

A lebegőpontos típusok lebegőpontos (valós) számokat ábrázolnak. Az egészekhez hasonlóan ezek is háromfajta méretűek lehetnek: *float* (egyszeres pontosságú), *double* (kétszeres pontosságú), és *long double* (kiterjesztett pontosságú).

Az egyszeres, kétszeres és kiterjesztett pontosság pontos jelentése az adott C++-változattól függ. A megfelelő pontosság kiválasztása egy olyan problémánál, ahol fontos a választás, a lebegőpontos számítások mély megértését követeli meg. Ha nem értünk a lebegőpontos aritmetikához, kérjünk tanácsot, szánjunk időt a megtanulására, vagy használjunk *double*-t és reméljük a legjobbakat.

4.5.1. Lebegőpontos literálok

Alapértelmezés szerint a lebegőpontos literálok *double* típusúak. A fordítónak itt is figyelmeztetnie kell, ha a lebegőpontos literálok az ábrázoláshoz képest túl nagyok. Íme néhány lebegőpontos literál:

```
1.23   .23   0.23   1.   1.0   1.2e10   1.23e-15
```

Jegyezzük meg, hogy szóköz nem fordulhat elő egy lebegőpontos literál közepén.
A *65.43 e-21* például nem lebegőpontos literál, hanem négy különálló nyelvi egység (ami formai hibát okoz):

```
65.43   e   -   21
```

Ha *float* típusú lebegőpontos literált akarunk megadni, akkor azt az *f* vagy *F* utótag használatával tehetjük meg:

```
3.14159265f   2.0f   2.997925F   2.9e-3f
```

Ha *long double* típusú lebegőpontos literált szeretnénk megadni, használjuk az *l* vagy *L* utótagot:

```
3.14159265L   2.0L   2.997925L   2.9e-3L
```

4.6. Méretek

A C++ alaptípusainak néhány jellemzője, mint például az *int* mérete, a C++ adott megvalósításától függ (§C.2). Rámutatok ezekre a függőségekre és gyakran ajánlom, hogy kerüljük őket vagy tegyünk lépéseket annak érdekében, hogy hatásukat csökkentsük. Miért kellene ezzel foglalkozni? Azok, akik különböző rendszereken programoznak vagy többféle fordí-

tót használnak, kénytelenek törődni ezzel, mert ha nem tennék, rákényszerülnének arra, hogy időt pazaroljanak nehezen megfogható programhibák megtalálására és kijavítására. Azok, akik azt állítják, hogy nem érdekli őket a hordozhatóság, általában azért teszik ezt, mert csak egy rendszert használnak és úgy érzik, megengedhetik maguknak azt a hozzáállást, miszerint „a nyelv az, amit a fordítóm megvalósít". Ez beszűkült látásmód. Ha egy program sikeres, akkor valószínű, hogy átviszik más rendszerre, és valakinek meg kell találnia és ki kell javítania a megvalósítás sajátosságaiból adódó problémákat. A programokat továbbá gyakran újra kell fordítani más fordítókkal ugyanarra a rendszerre és még a kedvenc fordítónk későbbi változata is másképpen csinálhat néhány dolgot, mint a mostani. Sokkal egyszerűbb ismerni és korlátozni az adott fordító használatának hatását, amikor egy programot megírnunk, mint megpróbálni később kibogozni a problémát.

A megvalósításból eredő nyelvi sajátosságok hatását viszonylag könnyű korlátozni, a rendszerfüggő könyvtárakét azonban sokkal nehezebb. Az egyik módszer az lehet, hogy lehetőleg csak a standard könyvtár elemeit használjuk.

Annak, hogy több egész, több előjel nélküli, és több lebegőpontos típus van, az az oka, hogy ez lehetőséget ad a programozónak, hogy a hardver jellemzőit megfelelően kihasználhassa. Sok gépen jelentős különbségek vannak a memóriaigényben, a memória hozzáférési idejében, és a többfajta alaptípussal való számolási sebességben. Ha ismerjük a gépet, általában könnyen kiválaszthatjuk például a megfelelő egész típust egy adott változó számára, igazán hordozható kódot írni azonban sokkal nehezebb.

A C++ objektumainak mérete mindig a *char* méretének többszöröse, így a *char* mérete 1. Egy objektum méretét a *sizeof* operátorral kaphatjuk meg (§6.2). Az alaptípusok méretére vonatkozóan a következők garantáltak:

1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
1 ≤ sizeof(bool) ≤ sizeof(long)
sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
sizeof(N) ≡ sizeof(signed N) ≡ sizeof(unsigned N)

A fentiekben *N* lehet *char*, *short int*, *int* vagy *long int*, továbbá biztosított, hogy a *char* legalább 8, a *short* legalább 16, a *long* pedig legalább 32 bites. A *char* a gép karakterkészletéből egy karaktert tárolhat. Az alábbi ábra az alaptípusok egy lehetséges halmazát és egy minta-karakterláncot mutat:

char: 'a'

bool: 1

short: 756

int: 100000000

int*: &c1

double: 1234567e34

char[14]: Hello, world!\0

Ugyanilyen méretarányban (0,5 cm egy bájt) egy megabájt memória körülbelül öt kilométernyire lógna ki a jobb oldalon.

A *char* típust az adott nyelvi változatnak úgy kell megválasztania, hogy a karakterek tárolására és kezelésére egy adott számítógépen a legmegfelelőbb legyen; ez jellemzően egy 8 bites bájt. Hasonlóan, az *int* típusnak a legmegfelelőbbnek kell lennie az egészek tárolására és kezelésére; ez általában egy 4 bájtos (32 bites) gépi szó. Nem bölcs dolog többet feltételezni. Például vannak olyan gépek, ahol a *char* 32 bites. Ha szükségünk van rá, az adott C++-változat egyedi tulajdonságait megtalálhatjuk a *<limits>* fejállományban (§22.2). Például:

```
#include <limits>
#include <iostream>

int main()
{
  std::cout << "A legnagyobb lebegőpontos szám == " << std::numeric_limits<float>::max()
            << ", a char előjeles == " << std::numeric_limits<char>::is_signed << '\n';
}
```

Az alaptípusok értékadásokban és kifejezésekben szabadon párosíthatók. Ahol lehetséges, az értékek úgy alakítódnak át, hogy ne legyen adatvesztés (§C.6).

Ha *v* érték pontosan ábrázolható egy *T* típusú változóban, akkor *v* érték *T* típusúvá alakítása megőrzi az értéket és nincs probléma. Legjobb, ha elkerüljük azokat az eseteket, amikor a konverziók nem értékőrzők (§C.6.2.6).

Nagyobb programok készítéséhez az automatikus konverziókat részletesebben meg kell értenünk, főleg azért, hogy képesek legyünk értelmezni a mások által írt kódot, a következő fejezetek olvasásához ugyanakkor ez nem szükséges.

4.7. Void

A *void* formája alapján alaptípus, de csak egy bonyolultabb típus részeként lehet használni; nincsenek *void* típusú objektumok. Vagy arra használjuk, hogy meghatározzuk, hogy egy függvény nem ad vissza értéket, vagy akkor, amikor egy mutató ismeretlen típusú objektumra mutat:

```
void x;        // hiba: nincsenek void objektumok
void f();      // az f függvény nem ad vissza értéket (§7.3)
void* pv;      // ismeretlen típusú objektumra hivatkozó mutató (§5.6)
```

Amikor egy függvényt bevezetünk, meg kell határozni visszatérési értékének típusát is. Logikailag elvárható lenne, hogy a visszatérési típus elhagyásával jelezzük, a függvény nem ad vissza értéket. Ez viszont a nyelvtant („A" függelék) kevésbé szabályossá tenné és ütközne a C-beli gyakorlattal. Következésképpen a *void* „látszólagos visszatérési típus"-ként használatos, annak jelölésére, hogy a függvény nem ad vissza értéket.

4.8. Felsoroló típusok

A *felsoroló típus* (enumeration) olyan típus, amely felhasználó által meghatározott értékeket tartalmaz. Meghatározása után az egész típushoz hasonlóan használható. A felsoroló típusok tagjaiként névvel rendelkező egész konstansokat adhatunk meg. Az alábbi kód például három egész állandót ad meg – ezeket *felsoroló konstansoknak* nevezzük – és értékeket rendel hozzájuk:

```
enum { ASM, AUTO, BREAK };
```

Alapértelmezés szerint az állandók *0*-tól növekvően kapnak értékeket, így *ASM==0*, *AUTO==1*, *BREAK==2*. A felsoroló típusnak lehet neve is:

```
enum keyword { ASM, AUTO, BREAK };
```

Minden felsorolás önálló típus. A felsoroló konstansok típusa a felsorolási típus lesz. Az *AUTO* például *keyword* típusú.

Ha *keyword* típusú változót adunk meg sima *int* helyett, mind a felhasználónak, mind a fordítónak utalunk a változó tervezett használatára:

```
void f(keyword key)
{
  switch (key) {
  case ASM:
          // valamit csinálunk
          break;
  case BREAK:
          // valamit csinálunk
          break;
  }
}
```

A fordító figyelmeztetést adhat, mert a három *keyword* típusú értékből csak kettőt kezeltünk.

A felsoroló konstans a kezdeti értékadáskor integrális típusú (§4.1.1) konstans kifejezéssel (§C.5.) is megadható. A felsoroló típus értékhalmaza összes tagjának értékét tartalmazza, felkerekítve a 2 legközelebbi, azoknál nagyobb hatványánál eggyel kisebb értékig. Ha a legkisebb felsoroló konstans nem negatív, az értékhalmaz *0*-val kezdődik, ha negatív, a 2 legközelebbi, a tagoknál kisebb negatív hatványával. Ez a szabály azt a legkisebb bitmezőt adja meg, amely tartalmazhatja a felsoroló konstansok értékét. Például:

```
enum e1 { dark, light };                  // tartomány: 0:1
enum e2 { a = 3, b = 9 };                 // tartomány: 0:15
enum e3 { min = -10, max = 1000000 };     // tartomány: -1048576:1048575
```

Egy integrális típus értékét meghatározott módon felsoroló típusúvá alakíthatjuk. Ha az érték nem esik a felsoroló típus értékhalmazába, a konverzió eredménye nem meghatározott:

```
enum flag { x=1, y=2, z=4, e=8 };      // tartomány: 0:15

flag f1 = 5;                 // típushiba: 5 nem flag típusú
flag f2 = flag(5);           // rendben: flag(5) flag típusú és annak tartományán belüli

flag f3 = flag(z|e);         // rendben: flag(12) flag típusú és annak tartományán belüli
flag f4 = flag(99);          // meghatározatlan: 99 nem esik a flag tartományán belülre
```

Az utolsó értékadás mutatja, miért nincs automatikus konverzió egészről felsoroló típusra; a legtöbb egész érték ugyanis nem ábrázolható egy adott felsoroló típusban.

A felsoroló típusok értékhalmazának fogalma különbözik a Pascal nyelvcsaládba tartozó nyelvek felsoroló típusainak fogalmától. A C-ben és a C++-ban azonban hosszú hagyománya van azoknak a bitkezelő műveleteknek, amelyek arra építenek, hogy a felsoroló típusok tagjain kívüli értékek jól körülhatároltak.

A felsoroló típusok *sizeof*-ja egy olyan integrális típus *sizeof*-ja, amely képes a felsoroló típus értékhalmazát tárolni és nem nagyobb *sizeof(int)*-nél, hiszen akkor nem lehetne *int*-ként vagy *unsigned int*-ként ábrázolni. Például *sizeof(e1)* lehet *1* vagy talán *4*, de nem lehet *8* egy olyan gépen, ahol *sizeof(int)==4*.

Alapértelmezés szerint a felsoroló típusok aritmetikai műveletek esetében egésszé alakítódnak (§6.2). A felsoroló típus felhasználói típus, így a felhasználók a felsorolásra saját műveleteket adhatnak meg, például a ++ és << operátorokkal (§11.2.3).

4.9. Deklarációk

A C++ programokban a neveket (azonosítókat) használat előtt be kell vezetnünk, azaz meg kell határoznunk típusukat, hogy megmondjuk a fordítónak, a név miféle egyedre hivatkozik. A deklarációk sokféleségét a következő példák szemléltetik:

```
char ch;
string s;
int count = 1;
const double pi = 3.1415926535897932385;
extern int error_number;

char* name = "Natasa";
char* season[ ] = { "tavasz", "nyár", "ősz", "tél" };

struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0 ? -a : a; }

typedef complex<short> Point;
struct User;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

Amint a példákból látható, a deklaráció többet is jelenthet annál, mint hogy egyszerűen egy nevet kapcsol össze a név típusával. A fenti deklarációk többsége definíció is, azaz meg is határozza azt az egyedet, amelyre a név hivatkozik. A *ch* esetében például ez az egyed a megfelelő memóriaterület, amelyet változóként használunk (vagyis ezt a memóriaterületet fogjuk lefoglalni), a *day*-nél a meghatározott függvény, a *pi* állandónál a 3.1415926535897932385 érték, a *Date*-nél egy új típus. A *Point* esetében az egyed a *complex<short>* típus, így a *Point* a *complex<short>* szinonimája lesz. A fenti deklarációk közül csak a

```
double sqrt(double);
extern int error_number;
struct User;
```

deklarációk nem definíciók is egyben: azaz máshol kell definiálni (meghatározni) azokat az egyedeket, amelyekre hivatkoznak. Az *sqrt* függvény kódját (törzsét) más deklarációkkal kell meghatározni, az *int* típusú *error_number* változó számára az *error_number* egy másik deklarációjának kell lefoglalnia a memóriát, és a *User* típus egy másik deklarációjának kell megadnia, hogy a típus hogy nézzen ki. Például:

```
double sqrt(double d) { /* ... */ }
int error_number = 1;

struct User { /* ... */ };
```

A C++ programokban minden név számára mindig pontosan egy definíció (meghatározás) létezhet (az *#include* hatásait lásd §9.2.3-ban). Ugyanakkor a nevet többször is deklarálhatunk (bevezethetjük). Egy egyed minden deklarációja meg kell, hogy egyezzen a hivatkozott egyed típusában. Így a következő részletben két hiba van:

```
int count;
int count;                    // hiba: újbóli definíció

extern int error_number;
extern short error_number;    // hiba: nem megfelelő típus
```

A következőben viszont egy sincs (az *extern* használatáról lásd §9.2):

```
extern int error_number;
extern int error_number;
```

Néhány definíció valamilyen „értéket” is meghatároz a megadott egyedeknek:

```
struct Date { int d, m, y; };
typedef complex<short> Point;
int day(Date* p) { return p->d; }
const double pi = 3.1415926535897932385;
```

Típusok, sablonok, függvények és állandók esetében ez az „érték” nem változik. Nem konstans adattípusok esetében a kezdeti értéket később módosíthatjuk:

```
void f()
{
  int count = 1;
  char* name = "Bjarne";
  // ...
  count = 2;
  name = "Marian";
}
```

A definíciók közül csak az alábbi nem határoz meg értéket:

```
char ch;
string s;
```

(Arról, hogy hogyan és mikor kap egy változó alapértelmezett értéket, lásd §4.9.5-öt és §10.4.2-t.) Minden deklaráció, amely értéket határoz meg, egyben definíciónak is minősül.

4.9.1. A deklarációk szerkezete

A deklarációk négy részből állnak: egy nem kötelező minősítőből, egy alaptípusból, egy deklarátorból, és egy – szintén nem kötelező – kezdőérték-adó kifejezésből. A függvény- és névtér-meghatározásokat kivéve a deklaráció pontosvesszőre végződik:

```
char* kings[ ] = { "Antigónusz", "Szeleukusz", "Ptolemaiosz" };
```

Itt az alaptípus *char*, a deklarátor a **kings[ ]*, a kezdőérték-adó rész pedig az *={…}*.

A minősítő (specifier) egy kulcsszó, mint a *virtual* (§2.5.5, §12.2.6) és az *extern* (§9.2), és a bevezetett elem néhány, nem a típusra jellemző tulajdonságát határozza meg. A deklarátor (declarator) egy névből és néhány nem kötelező operátorból áll. A leggyakoribb deklarátor-operátorok a következők (§A.7.1):

*	*mutató*	*előtag*
*const	*konstans mutató*	*előtag*
&	*referencia*	*előtag*
[ ]	*tömb*	*utótag*
()	*függvény*	*utótag*

Használatuk egyszerű lenne, ha mindegyikük előtagként (prefix) vagy mindegyikük utótagként (postfix) használt operátor volna. A *, a *[ ]* és a *()* operátorokat azonban arra tervezték, hogy kifejezésekben is használhatók legyenek (§6.2), így a * előtag-, a *[ ]* és a *()* pedig utótag operátorok. Az utótagként használt operátorok több megkötéssel járnak, mint az előtagként használtak. Következésképpen a **kings[ ]* egy valamire hivatkozó mutatókból álló vektor, és zárójeleket kell használnunk, ha olyasmit akarunk kifejezni, mint „… függvényre hivatkozó mutató" (lásd az §5.1 példáit). Teljes részletességgel lásd a nyelvtant az „A" függelékben.

Jegyezzük meg, hogy a típus nem hagyható el a deklarációból:

```
const c = 7;                                // hiba: nincs típus
gt(int a, int b) { return (a>b) ? a : b; }  // hiba: nincs visszatérési típus

unsigned ui;    // rendben: 'unsigned' jelentése 'unsigned int'
long li;        // rendben: 'long' jelentése 'long int'
```

A szabványos C++ ebben eltér a C és a C++ régebbi változataitól, amelyek megengedték az első két példát, azt feltételezve, hogy a típus *int*, ha nincs típus megadva (§B.2). Ez az „*implicit int*" szabály sok félreértés és nehezen megfogható hiba forrása volt.

4.9.2. Több név bevezetése

Egyetlen deklarációban több nevet is megadhatunk. A deklaráció ekkor vesszővel elválasztott deklarációk listáját tartalmazza. Két egészet például így vezethetünk be:

```
int x, y;          // int x és int y;
```

Jegyezzük meg, hogy az operátorok csak egyes nevekre vonatkoznak, az ugyanabban a deklarációban szereplő további nevekre nem:

```
int* p, y;           // int* p és int y, NEM int* y
int x, *q;           // int x és int* q
int v[10], *pv;      // int v[10] és int* pv
```

A fentihez hasonló szerkezetek rontják a program olvashatóságát, ezért kerülendők.

4.9.3. Nevek

A név (azonosító) betűk és számok sorozatából áll. Az első karakternek betűnek kell lennie. Az _ (aláhúzás) karaktert betűnek tekintjük. A C++ nem korlátozza a névben használható karakterek számát. A fordítóprogram írója azonban a megvalósítás egyes részeire nincs befolyással (konkrétan a szerkesztőprogramra, a *linker*-re), ez pedig határokat szab. Néhány futási idejű környezet ugyancsak szükségessé teszi, hogy kibővítsük vagy megszorítsuk az azonosítóban elfogadható karakterkészletet. A bővítések (például a *$* engedélyezése egy névben) nem hordozható programokat eredményeznek. A C++ kulcsszavai („A" függelék), mint a *new* és az *int*, nem használhatók felhasználói egyedek neveként. Példák a nevekre:

```
hello      ez_egy_szokatlanul_hosszú_név
DEFINED    foO      bAr      u_name   LoPatko
var0       var1     CLASS    _class   ___
```

És néhány példa olyan karaktersorozatokra, amelyek nem használhatók azonosítóként:

```
012                egy bolond   $sys    class   3var
fizetes.esedekes   foo~bar      .name   if
```

Az aláhúzással kezdődő nevek a nyelvi megvalósítás és a futási idejű környezet egyedi eszközei számára vannak fenntartva, így ezeket nem szabadna használni alkalmazói programokban.

Amikor a fordító olvassa a programot, mindig a leghosszabb olyan sorozatot keresi, amely kiadhat egy nevet. Így a *var10* és nem a *var* név (amit a 10-es szám követ). Hasonlóan, az *elseif* is egy név, nem pedig az *else*, amit az *if* kulcsszó követ.

A kis- és nagybetűket a nyelv megkülönbözteti, így a *Count* és a *count* különböző nevek, de nem bölcs dolog olyan neveket választani, amelyek csak a kezdőbetűben térnek el. A legjobb elkerülni azokat a neveket, amelyek csak kicsit különböznek. Például a nagybetűs o (*O*) és a nulla (*0*) nehezen megkülönböztethető, a kis L (*l*) és az egyes (*1*) szintén. Következésképpen azonosítónak a *l0*, *lO*, *l1* és *ll* nem szerencsés választás.

A nagy hatókörű nevek lehetőleg hosszúak és érthetőek legyenek, mint *vector*, *Window_with_border*, és *Department_number*. A kód viszont érthetőbb lesz, ha a kis hatókörben használt neveknek rövid, hagyományos stílusú nevük van, mint *x*, *i* és *p*. Az osztályok (10. fejezet) és a névterek (§8.2) használhatók arra, hogy a hatókörök kicsik maradjanak. Hasznos dolog viszonylag rövidnek hagyni a gyakran használt neveket, az igazán hosszúakat

pedig megtartani a kevésbé gyakran használatos egyedeknek. Válasszuk meg úgy a neveket, hogy az egyed jelentésére és ne annak megvalósítására utaljanak. A *phone_book* (telefonkönyv) például jobb, mint a *number_list* (számok listája), még akkor is, ha a telefonszámokat listában (§3.7) tároljuk. A jó nevek megválasztása is egyfajta művészet.

Próbáljunk következetes elnevezési stílust fenntartani. Például írjuk nagy kezdőbetűvel a nem standard könyvtárbeli felhasználói típusokat és kisbetűvel azokat a neveket, amelyek nem típusnevek (például *Shape* és *current_token*). Továbbá használjunk csupa nagybetűt makrók esetében (ha makrókat kell használnunk, például *HACK*) és használjunk aláhúzást, ha az azonosítóban szét akarjuk választani a szavakat. Akárhogy is, nehéz elérni a következetességet, mivel a programokat általában különböző forrásokból vett részletek alkotják és számos különböző ésszerű stílus használatos bennük. Legyünk következetesek a rövidítések és betűszavak használatában is.

4.9.4. Hatókörök

A deklaráció a megadott nevet egy hatókörbe (scope) vezeti be, azaz a nevet csak a programszöveg meghatározott részében lehet használni. A függvényeken belül megadott nevek esetében (ezeket gyakran *lokális* vagy *helyi névnek* hívjuk) ez a hatókör a deklaráció helyétől annak a blokknak a végéig tart, amelyben a deklaráció szerepel. A blokk olyan kódrész, amelyet a *{ }* kapcsos zárójelek határolnak.

Egy nevet *globálisnak* nevezünk, ha függvényen, osztályon (10. fejezet) vagy névtéren (§8.2) kívül bevezetett. A globális nevek hatóköre a bevezetés pontjától annak a fájlnak a végéig terjed, amelyben a deklaráció szerepel. A blokkokban szereplő névdeklarációk a körülvevő blokkban lévő deklarációkat és a globális neveket elfedhetik, azaz egy nevet újra meg lehet adni úgy, hogy egy másik egyedre hivatkozzon egy blokkon belül. A blokkból való kilépés után a név visszanyeri előző jelentését:

```
int x;                  // globális x

void f()
{
  int x;                // a lokális x elfedi a globális x-et
  x = 1;                // értékadás a lokális x-nek

  {
            int x;      // elfedi az első lokális x-et
            x = 2;      // értékadás a második lokális x-nek
  }
```

```
  x = 3;              // értékadás az első lokális x-nek
}

int* p = &x;          // a globális x címének felhasználása
```

A nevek elfedése elkerülhetetlen nagy programok írásakor. A kódot olvasónak azonban könnyen elkerüli a figyelmét, hogy egy név többször szerepel. Mivel az ilyen hibák viszonylag ritkán fordulnak elő, nagyon nehezen lehet azokat megtalálni. Következésképpen a névelfedések számát a lehető legkisebbre kell csökkenteni. Ha valaki olyan neveket használ globális változóként vagy egy hosszabb függvény lokális változójaként, mint *i* és *x*, akkor maga keresi a bajt.

Az elfedett globális nevekre a *::* hatókörjelző használatával hivatkozhatunk:

```
int x;

void f2()
{
  int x = 1;          // a globális x elfedése
  ::x = 2;            // értékadás a globális x-nek
  x = 2;              // értékadás a lokális x-nek
  // ...
}
```

Elfedett lokális név használatára nincs mód.

A név hatóköre a név deklarációjának pontjától kezdődik; azaz a teljes deklarátor után és a kezdeti érték(ek)et megadó rész előtt. Ez azt jelenti, hogy egy nevet saját kezdőértékének meghatározására is használhatunk:

```
int x;

void f3()
{
  int x = x;     // perverz: kezdeti értékadás x-nek saját maga (nem meghatározott) értékével
}
```

Ez nem tiltott, csak butaság. Egy jó fordítóprogram figyelmeztetést ad, ha egy változót azelőtt használunk, mielőtt értékét beállítottuk volna (lásd §5.9[9]).

Egy névvel egy blokkban két különböző objektumra a *::* operátor használata nélkül is hivatkozhatunk:

```
int x = 11;

void f4()                  // perverz:
{
  int y = x;               // a globális x felhasználása, y = 11
  int x = 22;
  y = x;                   // a lokális x felhasználása, y = 22
}
```

A függvényparaméterek neveit úgy tekintjük, mint ha a függvény legkülső blokkjában lennének megadva, így az alábbi hibás, mert *x*-et ugyanabban a hatókörben kétszer adtuk meg:

```
void f5(int x)
{
  int x;          // hiba
}
```

Ilyen hiba gyakran előfordul, érdemes figyelnünk rá.

4.9.5. Kezdeti értékadás

Ha egy objektumhoz kezdőérték-adó kifejezést adunk meg, akkor ez határozza meg az objektum kezdeti értékét. Ha nincs megadva ilyen, a globális (§4.9.4), névtér (§8.2), vagy helyi statikus objektumok (§7.1.2, §10.2.4) (melyeket együttesen ***statikus objektumoknak*** nevezünk) a megfelelő típus *0* értékét kapják kezdőértékül:

```
int a;          // jelentése "int a = 0;"
double d;       // jelentése "double d = 0.0;"
```

A lokális változóknak (ezeket néha ***automatikus objektumoknak*** nevezzük) és a szabad tárban létrehozott objektumoknak (***dinamikus*** vagy „heap" ***objektumok***) alapértelmezés szerint nincs kezdőértékük:

```
void f()
{
  int x;    // x értéke nem meghatározott
  // ...
}
```

A tömbök és struktúrák tagjai alapértelmezés szerint kapnak kezdőértéket, függetlenül attól, hogy az adott szerkezet statikus-e vagy sem. A felhasználói típusokhoz magunk is megadhatunk alapértelmezett kezdőértéket (§10.4.2).

A bonyolultabb objektumoknak egynél több értékre van szükségük a kezdeti értékadáshoz. A tömbök és struktúrák C típusú feltöltésekor (§5.2.1, §5.7) ezt egy *{ }* zárójelek által határolt listával érhetjük el. A konstruktorral rendelkező felhasználói típusoknál a függvény stílusú paraméterlisták használatosak (§2.5.2, §10.2.3). Jegyezzük meg, hogy a deklarációkban szereplő *()* üres zárójelek jelentése mindig „függvény":

```
int a[ ] = { 1, 2 };        // tömb kezdeti értékadása
Point z(1,2);               // függvény stílusú kezdeti értékadás (konstruktorral)
int f();                    // függvény-deklaráció
```

4.9.6. Objektumok és balértékek

Névvel nem rendelkező „változókat" is lefoglalhatunk és használhatunk, és értéket is adhatunk nekik furcsa kifejezésekkel (pl. **p[a+10]=7*). Következésképp el kellene neveznünk azt, hogy „valami a memóriában". Ez az objektum legegyszerűbb és legalapvetőbb fogalma. Azaz, az objektum egy folytonos tárterület; a bal oldali érték („***balérték***") pedig egy olyan kifejezés, amely egy objektumra hivatkozik. A balérték (*lvalue*) szót eredetileg arra alkották, hogy a következőt jelentse: „valami, ami egy értékadás bal oldalán szerepelhet". Nem minden balérték lehet azonban az értékadás bal oldalán, egy balérték hivatkozhat állandóra (*const*) is (§5.5). A nem *const*-ként megadott balértéket szokás ***módosítható balértéknek*** (***modifiable lvalue***) is nevezni. Az objektumnak ezt az egyszerű és alacsony szintű fogalmát nem szabad összetéveszteni az osztályobjektumok és többalakú (polimorf típusú) objektumok (§15.4.3) fogalmával.

Hacsak a programozó másképp nem rendelkezik (§7.1.2, §10.4.8), egy függvényben bevezetett változó akkor jön létre, amikor definíciójához érkezünk, és akkor szűnik meg, amikor a neve a hatókörön kívülre kerül (§10.4.4). Az ilyen objektumokat automatikus objektumoknak nevezzük. A globális és névtér-hatókörben bevezetett objektumok és a függvényekben vagy osztályokban megadott *static* objektumok (csak) egyszer jönnek létre és kapnak kezdeti értéket, és a program befejeztéig „élnek"(10.4.9). Az ilyen objektumokat statikus objektumoknak nevezzük. A tömbelemeknek és a nem statikus struktúrák vagy osztályok tagjainak az élettartamát az az objektum határozza meg, amelynek részei. A *new* és *delete* operátorokkal olyan objektumok hozhatók létre, amelyek élettartama közvetlenül szabályozható (§6.2.6).

4.9.7. Typedef

Az a deklaráció, amit a *typedef* kulcsszó előz meg, a típus számára új nevet hoz létre, nem egy adott típusú változót:

```
typedef char* Pchar;
Pchar p1, p2;            // p1 és p2 típusa char*
char* p3 = p1;
```

Az így megadott, gyakran *typedef*-nek (áltípus) nevezett név kényelmes rövidítés lehet egy nehezen használható név helyett. Például az *unsigned char* túlságosan hosszú az igazán gyakori használatra, ezért megadhatjuk a szinonimáját, az *uchar*-t:

```
typedef unsigned char uchar;
```

A *typedef* másik használata az, hogy egy típushoz való közvetlen hozzáférést egy helyre korlátozunk:

```
typedef int int32;
typedef short int16;
```

Ha most az *int32*-t használjuk, amikor egy viszonylag nagy egészre van szükségünk, programunkat átvihetjük egy olyan gépre, ahol a *sizeof(int)* 2-vel egyenlő, úgy, hogy a kódban egyszer szereplő *int32*-t most másképp határozzuk meg:

```
typedef long int32;
```

Végezetül, a *typedef*-ek inkább más típusok szinonimái, mint önálló típusok. Következésképpen a *typedef*-ek szabadon felcserélhetők azokkal a típusokkal, melyeknek szinonimái. Azok, akik ugyanolyan jelentéssel vagy ábrázolással rendelkező önálló típusokat szeretnének, használják a felsoroló típusokat (§4.8) vagy az osztályokat (10. fejezet).

4.10. Tanácsok

[1] A hatókörök legyenek kicsik. §4.9.4.
[2] Ne használjuk ugyanazt a nevet egy hatókörben és az azt körülvevő hatókörben is. §4.9.2.
[3] Deklarációnként (csak) egy nevet adjunk meg. §4.9.3.
[4] A gyakori és helyi nevek legyenek rövidek, a nem helyi és ritkán használt nevek hosszabbak. §4.9.3.
[5] Kerüljük a hasonlónak látszó neveket. §4.9.3.
[6] Elnevezési stílusunk legyen következetes. §4.9.3.
[7] Figyeljünk arra, hogy a névválasztás inkább a jelentésre, mintsem a megvalósításra utaljon. §4.9.3.
[8] Ha a beépített típus, amelyet egy érték ábrázolására használunk, megváltozhat, használjunk *typedef*-et, így a típus számára beszédes nevet adhatunk. §4.9.7
[9] A *typedef*-ekkel típusok szinonimáit adjuk meg; új típusok definiálására használjunk felsoroló típusokat és osztályokat. §4.9.7.
[10] Emlékezzünk arra, hogy minden deklarációban szükséges a típus megadása (nincs „*implicit int*”). §4.9.1.
[11] Kerüljük a karakterek számértékével kapcsolatos szükségtelen feltételezéseket. §4.3.1, §C.6.2.1.
[12] Kerüljük az egészek méretével kapcsolatos szükségtelen feltételezéseket. §4.6.
[13] Kerüljük a szükségtelen feltételezéseket a lebegőpontos típusok értékkészletével kapcsolatban is. § 4.6.
[14] Részesítsük előnyben a sima *int*-et a *short int*-tel vagy a *long int*-tel szemben. §4.6.
[15] Részesítsük előnyben a *double*-t a *float*-tal vagy a *long double*-lal szemben. §4.5.
[16] Részesítsük előnyben a sima *char*-t a *signed char*-ral és az *unsigned char*-ral szemben. §C.3.4.
[17] Kerüljük az objektumok méretével kapcsolatos szükségtelen feltételezéseket. §4.6.
[18] Kerüljük az előjel nélküli aritmetikát. §4.4.
[19] Legyünk óvatosak az előjelesről előjel nélkülire és *unsigned*-ról *signed*-ra való átalakítással. §C.6.2.6.
[20] Legyünk óvatosak a lebegőpontos típusról egészre való átalakítással. § C.6.2.6.
[21] Legyünk óvatosak a kisebb típusokra való átalakításokkal (például *int*-ről *char*-ra). § C.6.2.6.

4.11. Gyakorlatok

1. (*2) Futtassuk le a „Helló, világ!" programot (§3.2). Ha a program fordítása nem úgy sikerül, ahogy kellene, olvassuk el §B.3.1-et.
2. (*1) §4.9 minden deklarációjára végezzük el a következőket: ha a deklaráció nem definíció, írjunk hozzá definíciót. Ha a deklaráció definíció is, írjunk olyan deklarációt, ami nem az.
3. (*1.5) Írjunk programot, amely kiírja az alaptípusok, néhány szabadon választott mutatótípus és néhány szabadon választott felsoroló típus méretét. Használjuk a *sizeof* operátort.
4. (*1.5) Írjunk programot, amely kiírja az *'a'*...*'z'* betűket és a *'0'*...*'9'* számjegyeket, valamint a hozzájuk tartozó egész értékeket. Végezzük el ugyanezt a többi kiírható karakterre is. Csináljuk meg ugyanezt hexadecimális jelöléssel.
5. (*2) Mi a rendszerünkön a legnagyobb és legkisebb értéke a következő típusoknak: *char*, *short*, *int*, *long*, *float*, *double*, *long double* és *unsigned*?
6. (*1) Mi a leghosszabb lokális név, amit a C++ programokban használhatunk a rendszerünkben? Mi a leghosszabb külső név, amit a C++ programokban használhatunk a rendszerünkben? Van-e megszorítás a nevekben használható karakterekre?
7. (*2) Rajzoljunk ábrát az egész és alaptípusokról, ahol egy típus egy másik típusra mutat, ha az első minden értéke minden szabványos megvalósításban ábrázolható a másik típus értékeként. Rajzoljuk meg az ábrát kedvenc C++-változatunk típusaira is.

5

Mutatók, tömbök és struktúrák

„A fenséges és a nevetséges
gyakran annyira összefüggnek,
hogy nehéz őket szétválasztani."
(Tom Paine)

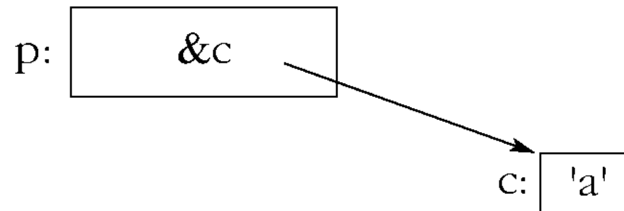
Mutatók • Nulla • Tömbök • Karakterliterálok • Tömbre hivatkozó mutatók • Konstansok • Mutatók és konstansok • Referenciák • *void** • Struktúrák • Tanácsok • Gyakorlatok

5.1. Mutatók

Ha *T* egy típus, *T** a „*T*-re hivatkozó mutató" típus lesz, azaz egy *T** típusú változó egy *T* típusú objektum címét tartalmazhatja. Például:

```
char c = 'a';
char* p = &c;          // a p a c címét tárolja
```

Ábrával:



Sajnos a tömbökre és függvényekre hivatkozó mutatók esetében bonyolultabb jelölés szükséges:

```
int* pi;              // mutató egészre
char** ppc;           // mutató char-ra hivatkozó mutatóra
int* ap[15];          // egészre hivatkozó mutatók 15 elemű tömbje
int (*fp)(char*);     // char* paraméterű függvényre hivatkozó mutató; egészet ad vissza
int* f(char*);        // char* paraméterű függvény; egészre hivatkozó mutatót ad vissza
```

Lásd §4.9.1-et a deklarációk formai követelményeire vonatkozóan, és az „A” függeléket a teljes nyelvtannal kapcsolatban.

A mutatón végezhető alapvető művelet a „dereferencia”, azaz a mutató által mutatott objektumra való hivatkozás. E műveletet *indirekciónak* (közvetett használatnak, hivatkozásnak) is hívják. Az indirekció jele az előtagként használt egyoperandusú * :

```
char c = 'a';
char* p = &c;         // a p a c címét tárolja
char c2 = *p;         // c2 == 'a'
```

A *p* által mutatott változó *c*, a *c*-ben tárolt érték *'a'*, így *c2* értéke *'a'* lesz.A tömbelemekre hivatkozó mutatókon aritmetikai műveleteket is végezhetünk (§5.3), a függvényekre hivatkozó mutatók pedig végtelenül hasznosak, ezeket a §7.7 pontban tárgyaljuk.

A mutatók célja, hogy közvetlen kapcsolatot teremtsenek annak a gépnek a címzési eljárásaival, amin a program fut. A legtöbb gép bájtokat címez meg. Azok, amelyek erre nem képesek, olyan hardverrel rendelkeznek, amellyel a bájtokat gépi szavakból nyerik ki. Másrészről kevés gép tud közvetlenül egy bitet megcímezni, következésképp a legkisebb objektum, amely számára önállóan memóriát foglalhatunk és amelyre beépített típusú mutatóval hivatkozhatunk, a *char*. Jegyezzük meg, hogy a *bool* legalább annyi helyet foglal, mint a *char* (§4.6). Ahhoz, hogy a kisebb értékeket tömörebben lehessen tárolni, logikai operátorokat (§6.2.4) vagy struktúrákban levő bitmezőket (§C.8.1) használhatunk.

5.1.1. Nulla

A nulla (*0*) az *int* típusba tartozik. A szabványos konverzióknak (§C.6.2.3) köszönhetően a *0* integrális (§4.1.1), lebegőpontos, mutató, vagy „tagra hivatkozó mutató" típusú konstansként is használható. A típust a környezet dönti el. A nullát általában (de nem szükségszerűen) a megfelelő méretű „csupa nulla" bitminta jelöli.Nincs olyan objektum, amely számára a *0* címmel foglalnánk helyet. Következésképpen a *0* mutató-literálként viselkedik, azt jelölve, hogy a mutató nem hivatkozik objektumra.A C-ben a nulla mutatót (nullpointer) szokás a *NULL* makróval jelölni. A C++ szigorúbb típusellenőrzése miatt az ilyen *NULL* makrók helyett használjuk a sima *0*-t, ez kevesebb problémához vezet. Ha úgy érezzük, muszáj a *NULL*-t megadnunk, tegyük azt az alábbi módon:

```
const int NULL = 0 ;
```

A *const* minősítő megakadályozza, hogy a *NULL*-t véletlenül újra definiáljuk és biztosítja, hogy a *NULL*–t ott is használni lehessen, ahol állandóra van szükség.

5.2. Tömbök

Ha *T* egy típus, a *T[size]* a „*size* darab *T* típusú elemből álló tömb" típus lesz. Az elemek sorszámozása *0*-tól *size-1*-ig terjed:

```
float v[3];          // három lebegőpontos számból álló tömb: v[0], v[1], v[2]
char* a[32];         // karakterre hivatkozó mutatók 32 elemű tömbje: a[0] .. a[31]
```

A tömb elemeinek száma, mérete vagy dimenziója, konstans kifejezés kell, hogy legyen (§C.5). Ha változó méretre van szükségünk, használjunk vektort (§3.7.1, §16.3):

```
void f(int i)
{
  int v1[i];                 // hiba: a tömb mérete nem konstans kifejezés
  vector<int> v2(i);         // rendben
}
```

A többdimenziós tömbök úgy ábrázolódnak, mint tömbökből álló tömbök:

```
int d2[10][20];        // d2 olyan tömb, amely 10 darab, 20 egészből álló tömböt tartalmaz
```

A más nyelvekben tömbök méretének meghatározására használt vessző jelölés fordítási hibákat eredményez, mert a , (vessző) műveletsorozatot jelző operátor (§6.2.2) és nem megengedett konstans kifejezésekben (§C.5). Például próbáljuk ki ezt:

```
int bad[5,2];                    // hiba: konstans kifejezésben nem lehet vessző
```

A többdimenziós tömböket a §C.7 pontban tárgyaljuk. Alacsonyszintű kódon kívül a legjobb, ha kerüljük őket.

5.2.1. Tömbök feltöltése

A tömböknek értékekből álló listákkal adhatunk kezdőértéket:

```
int v1[ ] = { 1, 2, 3, 4 };
char v2[ ] = { 'a', 'b', 'c', 0 };
```

Amikor egy tömböt úgy adunk meg, hogy a méretét nem határozzuk meg, de kezdőértékeket biztosítunk, a fordítóprogram a tömb méretét a kezdőérték-lista elemeinek megszámlálásával számítja ki. Következésképp *v1* és *v2* típusa rendre *int[4]* és *char[4]* lesz. Ha a méretet megadjuk, a kezdőérték-listában nem szerepelhet annál több elem, mert ez hibának számít:

```
char v3[2] = { 'a', 'b', 0 };      // hiba: túl sok kezdőérték
char v4[3] = { 'a', 'b', 0 };      // rendben
```

Ha a kezdőérték túl kevés elemet ad meg, a tömb maradék elemeire *0* lesz feltételezve:

```
int v5[8] = { 1, 2, 3, 4 };
```

Az előző kód egyenértékű a következővel:

```
int v5[ ] = { 1, 2, 3, 4 , 0, 0, 0, 0 };
```

Jegyezzük meg, hogy a kezdőérték-lista nem helyettesíthető és nem bírálható felül tömbértékadással:

```
void f()
{
   v4 = { 'c', 'd', 0 }; // hiba: tömböt nem lehet értékül adni
}
```

Ha ilyen értékadásra van szükségünk, tömb helyett használjunk *vector*-t (§16.3) vagy *valarray*-t (§22.4).

A karaktertömböket kényelmi okokból karakterliterálokkal (§5.2.2) is feltölthetjük.

5.2.2. Karakterliterálok

A karakterliterál egy macskakörmökkel határolt karaktersorozat:

```
"Ez egy karakterlánc"
```

Egy karakterliterál a látszólagosnál eggyel több karaktert tartalmaz; a *'\0'* null karakterre végződik, melynek értéke *0*:

```
sizeof("Bohr")==5
```

A karakterliterálok típusa „megfelelő számú *const* (állandó) karakterből álló tömb”, így a *"Bohr"* típusa *const char[5]* lesz.

A karakterliterálokat egy *char**-nak is értékül adhatjuk. Ez azért megengedett, mert a karakterliterál típusa a C és a C++ korábbi változataiban *char** volt, így ez szükséges ahhoz, hogy millió sornyi C és C++ kód érvényes maradjon. Az ilyen karakterliterálokat azonban hiba ilyen mutatón keresztül módosítani.

```
void f()
{
    char* p = "Platón";
    p[4] = 'e';            // hiba: értékadás konstansnak; az eredmény nem meghatározott
}
```

Az effajta hibát általában nem lehet a futási időig kideríteni, és a nyelv egyes megvalósításai is különböznek abban, hogy mennyire szereznek érvényt ennek a szabálynak. (Lásd még §B.2.3-at.) Az, hogy a karakterliterálok állandók, nemcsak magától értetődő, hanem azt is lehetővé teszi, hogy a nyelv adott változata jelentősen optimalizálhassa a karakterliterálok tárolásának és hozzáférésének módját.

Ha olyan karakterláncot szeretnénk, amit biztosan módosíthatunk, karaktereit egy tömbbe kell másolnunk:

```
void f()
{
  char p[ ] = "Zénón";                 // p egy 6 karakterből álló tömb
  p[0] = 'R';                          // rendben
}
```

A karakterliterál tárolási helye nem változik (statikus), így egy függvény visszatérési értékeként biztonságosan megadható:

```
const char* error_message(int i)
{
  // ...
  return "tartományhiba";
}
```

A *range_error*-t tartalmazó memóriaterület tartalma nem törlődik az *error_message()* meghívása után.

Az, hogy két egyforma karakterliterál egyetlen memóriaterületen tárolódik-e, az adott nyelvi változattól függ (§C.1):

```
const char* p = "Herakleitosz";
const char* q = "Herakleitosz";

void g()
{
  if (p == q) cout << "Egyezik!\n";    // az eredmény az adott C++-változattól függ
  // ...
}
```

Jegyezzük meg, hogy a mutatókra alkalmazott == a címeket (a mutató értékeket) hasonlítja össze, nem azokat az értékeket, melyekre a mutatók hivatkoznak.

Üres karakterláncot a *""* szomszédos macskaköröm-párral írhatunk le (típusa *const char[1]*).

A nem grafikus karakterek jelölésére használt fordított perjel (§C.3.2) szintén használható egy karakterlánc belsejében. Ez lehetővé teszi az idézőjel (*"*) és a fordított perjel „escape" karakter (\) karakterláncon belüli ábrázolását is. Az *'\n'* (új sor) karakter ezek közül messze a leggyakrabban használt:

```
cout<<"csengő az üzenet végén\a\n";
```

Az *'\a'* karakter az ASCII *BEL* (csengő), amely *alert*-ként is ismert, kiírása pedig valamilyen hangjelzést eredményez.

A karakterláncokban „igazi” sortörés nem szerepelhet:

```
"Ez nem karakterlánc
hanem formai hiba"
```

A hosszú láncok „üreshely” (whitespace) karakterekkel széttördelhetők, hogy a programszöveg szebb legyen:

```
char alpha[ ] = "abcdefghijklmnopqrstuvwxyz"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

A fordítóprogram összefűzi a szomszédos láncokat, így az *alpha* egyetlen karakterlánccal is megadható lett volna:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

A null karaktert elvileg a karakterláncok belsejében is használhatnánk, de a legtöbb program nem feltételezi, hogy utána is vannak karakterek. A *"Jens\000Munk"* karakterláncot például az olyan standard könyvtárbeli függvények, mint a *strcpy()* és a *strlen()*, *"Jens"*-ként fogják kezelni (§20.4.1).

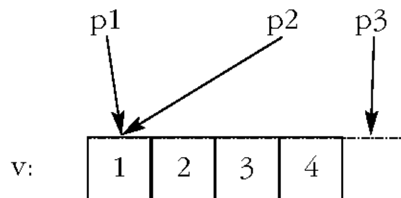
Az *L* előtagú karakterláncok – mint amilyen az *L"angst"* – „széles” karakterekből (*wide char*) állnak (§4.3, §C3.3), típusuk *const wchar_t[ ]*.

5.3. Tömbökre hivatkozó mutatók

A C++-ban a tömbök és mutatók között szoros kapcsolat áll fenn. Egy tömb nevét úgy is használhatjuk, mint egy mutatót, amely a tömb első elemére mutat:

```
int v[ ] = { 1, 2, 3, 4 };
int* p1 = v;          // mutató a kezdőelemre (automatikus konverzió)
int* p2 = &v[0];      // mutató a kezdőelemre
int* p3 = &v[4];      // mutató az "utolsó utáni" elemre
```

Ábrával:



Az biztosan működik, ha a mutatót eggyel a tömb vége utáni elemre állítjuk. Ez sok algoritmus számára fontos (§2.7.2, §18.3). Mivel azonban egy ilyen mutató ténylegesen már nem mutat egy tömb elemére, nem szabad írásra vagy olvasásra használni. Nem meghatározott, hogy mi történik, amikor egy tömb kezdőeleme előtt levő elem címét vesszük, ezért az ilyesmi kerülendő. Egyes számítógépeken a tömbök gyakran a gép címzési határain kerülnek lefoglalásra, így az „eggyel a kezdőelem előtti elem" egyszerűen értelmetlen lesz.

A tömbnevek automatikus (implicit) átalakítása mutatóvá széleskörűen használatos a C stílusú kódokban szereplő függvényhívásoknál:

```
extern "C" int strlen(const char*);      // a <string.h> fejállományból

void f()
{
   char v[] = "Annemarie";
   char* p = v;          // char[] automatikus átalakítása char*-gá
   strlen(p);
   strlen(v);            // char[] automatikus átalakítása char*-gá
   v = p;                // hiba: a tömbnek nem adható érték
}
```

A standard könyvtár *strlen()* függvényének mindkét híváskor ugyanaz az érték adódik át. Az a bökkenő, hogy az automatikus konverziót lehetetlen elkerülni, vagyis nincs mód olyan függvény bevezetésre, amelynek meghívásakor a *v* tömb átmásolódik. Szerencsére mutatóról tömbre való átalakítás nem végezhető sem automatikusan, sem definiált módon.

A tömbparaméter automatikus mutatóvá alakítása azt jelenti, hogy a tömb mérete elvész a függvény számára. A függvénynek azonban valahogy meg kell határoznia a tömb méretét, hogy értelmes műveleteket hajthasson végre rajta. A C standard könyvtárában levő más függvényekhez hasonlóan – amelyek karakterre hivatkozó mutatókat kapnak paraméterként – az *strlen()* is arra számít, hogy a null karakter jelzi a karakterlánc végét, így a *strlen(p)*

a *p* karaktereinek a *0* null karakter végződésig számolt mennyiségét jelenti, nem beleértve a null karakter végződést . Ez meglehetősen alacsony szintű megoldás. A standard könyvtárban lévő *vector* (§16.3.) és *string* (20. fejezet) esetében nincs ilyen probléma.

5.3.1. Tömbök bejárása

Sok algoritmus lényege a tömbökhöz és más hasonló adattípusokhoz való hatékony és „elegáns" hozzáférés (lásd §3.8, 18. fejezet). A hozzáférés egy tömbre hivatkozó mutatóval, illetve egy indexszel vagy egy elemre hivatkozó mutatóval valósítható meg. Íme egy példa egy karakterlánc bejárására index használatával:

```
void fi(char v[])
{
  for (int i = 0; v[i]!=0; i++) use(v[i]);
}
```

Ez egyenértékű a mutatóval történő bejárással:

```
void fp(char v[])
{
  for (char* p = v; *p!=0; p++) use(*p);
}
```

Az előtagként használt * indirekció operátor egy mutató-hivatkozást old fel, így **p* a *p* által mutatott karakter lesz, a ++ pedig úgy növeli a mutatót, hogy az a tömb következő elemére hivatkozzon. Nincs olyan eredendő ok, amiért az egyik változat gyorsabb lenne a másiknál. A modern fordítóprogramoknak ugyanazt a kódot kell létrehozniuk mindkét példa esetében (lásd §5.9[8]-at). A programozók logikai és esztétikai alapon választhatnak a változatok között.

Ha a +, -, ++ vagy -- aritmetikai műveleti jeleket mutatókra alkalmazzuk, az eredmény a mutatók által hivatkozott objektumok típusától függ. Amikor egy *T** típusú *p* mutatóra alkalmazunk egy aritmetikai operátort, akkor *p*-ről feltételezzük, hogy egy *T* típusú objektumokból álló tömb elemére mutat, így *p+1* a tömb következő elemét jelzi, *p-1* pedig az előző elemre mutat. Ez arra utal, hogy *p+1* egész értéke *sizeof(T)*-vel nagyobb lesz, mint *p* egész értéke. Hajtsuk végre a következőt:

```
#include <iostream>

int main ()
{
  int vi[10];
  short vs[10];
```

```
    std::cout << &vi[0] << ' ' << &vi[1] << '\n';
    std::cout << &vs[0] << ' ' << &vs[1] << '\n';
}
```

Ekkor a következőt kapjuk (a mutatók értékének alapértelmezés szerinti, hexadecimális jelölését használva):

```
0x7fffaef0 0x7fffaef4
0x7fffaedc 0x7fffaede
```

Ez azt mutatja, hogy *sizeof(short)* az adott megvalósításban *2*, *sizeof(int)* pedig *4*.

Mutatókat csak akkor vonhatunk ki egymásból definiált módon, ha mindkét mutató ugyanannak a tömbnek az elemeire mutat (bár a nyelvben nincs gyors mód annak ellenőrzésére, hogy valóban arra mutatnak). Amikor kivonunk egy mutatót egy másikból, az eredmény a két mutató között lévő tömbelemek száma (egy egész típusú érték) lesz. A mutatókhoz egész értéket is adhatunk és ki is vonhatunk belőle egészet, az eredmény mindkét esetben egy mutató érték lesz. Ha ez az érték nem ugyanannak a tömbnek egy elemére mutat, amelyre az eredeti mutató, vagy nem eggyel a tömb mögé, az eredményül kapott mutató érték felhasználása kiszámíthatatlan eredményhez vezethet:

```
void f()
{
    int v1[10];
    int v2[10];

    int i1 = &v1[5]-&v1[3];          // i1 = 2
    int i2 = &v1[5]-&v2[3];          // meghatározhatatlan eredmény

    int* p1 = v2+2;                  // p1 = &v2[2]
    int* p2 = v2-2;                  // *p2 nem meghatározott
}
```

A bonyolult mutatóaritmetika rendszerint szükségtelen, ezért legjobb elkerülni. Nincs értelme mutatókat összeadni, és ez nem is megengedett.

A tömbök nem önleírók, mert nem biztos, hogy a tömb elemeinek száma is tárolódik a tömbbel együtt. Ez azt jelenti, hogy ahhoz, hogy bejárjunk egy tömböt, amely nem tartalmaz a karakterláncokéhoz hasonló végződést, valahogy meg kell adnunk a tömb elemeinek számát:

```
void fp(char v[ ], unsigned int size)
{
  for (int i=0; i<size; i++) use(v[i]);

  const int N = 7;
  char v2[N];
  for (int i=0; i<N; i++) use(v2[i]);

}
```

Jegyezzük meg, hogy a legtöbb C++-változat a tömbök esetében nem végez indexhatár-ellenőrzést. A tömb ezen fogalma eredendően alacsony szintű. Fejlettebb tömbfogalmat osztályok használatával valósíthatunk meg (§3.7.1).

5.4. Konstansok

A C++ felkínálja a *const*, azaz a felhasználói állandó fogalmát, hogy lehetőségünk legyen annak kifejezésére, hogy egy érték nem változik meg közvetlenül. Ez számos esetben hasznos lehet. Sok objektumnak a létrehozás után már nem változik meg az értéke. A szimbolikus konstansok (jelképes állandók) könnyebben módosítható kódhoz vezetnek, mint a kódban közvetlenül elhelyezett literálok. Gyakori, hogy egy értéket mutatón keresztül érünk el, de az értéket nem változtatjuk meg. A legtöbb függvényparamétert csak olvassuk, nem írjuk.

A *const* kulcsszó hozzáadható egy objektum deklarációjához, jelezve, hogy az objektumot állandóként határozzuk meg. Mivel egy állandónak később nem lehet értéket adni, kezdeti értékadást kell végeznünk:

```
const int model = 90;                // a model állandó
const int v[ ] = { 1, 2, 3, 4 };     // a v[i] állandó
const int x;                         // hiba: nincs kezdeti értékadás
```

Ha valamit *const*-ként határozunk meg, az biztosíték arra, hogy hatókörén belül értéke nem fog megváltozni:

```
void f()
{
  model = 200;     // hiba
  v[2]++;          // hiba
}
```

Jegyezzük meg, hogy a *const* kulcsszó módosítja a típust és megszorítást ad arra, hogyan használhatunk egy objektumot, de nem határozza meg, hogyan kell az állandó számára helyet foglalni:

```
void g(const X* p)
{
  // itt *p nem módosítható
}

void h()
{
  X val;     // a val módosítható
  g(&val);
  // ...
}
```

Attól függően, hogy a fordítóprogram mennyire okos, számos módon kihasználhatja egy objektum állandó mivoltát. Az állandók kezdeti értéke például gyakran (de nem mindig) egy konstans kifejezés (§C.5), ami fordítási időben kiértékelhető. Továbbá, ha a fordítóprogram tud az állandó minden használatáról, nem kell tárhelyet sem lefoglalnia számára:

```
const int c1 = 1;
const int c2 = 2;
const int c3 = my_f(3);        // c3 értéke fordításkor nem ismert
extern const int c4;           // c4 értéke fordításkor nem ismert
const int* p = &c2;            // c2 számára tárterületet kell foglalni
```

Ekkor a fordítóprogram ismeri *c1* és *c2* értékét, így azokat konstans kifejezésekben felhasználhatjuk. Mivel a *c3* és *c4* értékek fordítási időben nem ismertek (ha csak ebben a fordítási egységben levő információkat használjuk fel, lásd §9.1), *c3*-nak és *c4*-nek tárhelyet kell foglalni. Mivel *c2* címét használjuk, *c2*-nek is helyet kell foglalni. A *c1* konstans példa arra az egyszerű és gyakori esetre, amikor az állandó értéke fordítási időben ismert és számára nem szükséges tárat foglalni. Az *extern* kulcsszó azt jelöli, hogy a *c4*-et máshol definiáltuk (§9.2).

A konstansokból álló tömböknek általában szükséges helyet foglalni, mert a fordítóprogram nem tudja eldönteni, mely tömbelemekre hivatkoznak a kifejezések. Sok gépen azonban még ebben az esetben is növelhetjük a hatékonyságot, úgy, hogy a konstansokból álló tömböt csak olvasható memóriába tesszük.

A *const*-okat gyakran használjuk tömbök indexhatáraként és *case* címkéknél is:

```
const int a = 42;
const int b = 99;
const int max = 128;

int v[max];

void f(int i)
{
  switch (i) {
  case a:
        // ...

  case b:
        // ...
  }
}
```

Ilyen esetekben gyakori, hogy *const* helyett felsoroló konstansokat (§4.8) használunk. Azt, hogy a *const* milyen módon használható osztályok tagfüggvényeivel, a §10.2.6 és §10.2.7 pontokban tárgyaljuk.

A szimbolikus konstansokat rendszeresen használnunk kellene arra, hogy elkerüljük a kódban a „mágikus számokat". Ha egy numerikus állandó, például egy tömb mérete, a kódban ismétlődik, a programot nehéz lesz átnézni, hogy a megfelelő módosításkor az állandó minden egyes előfordulását kicseréljük. A szimbolikus konstansok használata viszont lokálissá teszi az információt. A numerikus konstansok rendszerint valamilyen, a programmal kapcsolatos feltételezést jelölnek. A *4* például egy egészben lévő bájtok számát, a *128* a bemenet átmeneti tárba helyezéséhez (puffereléséhez) szükséges karakterek számát, a *6.24* pedig a dán korona és az amerikai dollár közötti keresztárfolyamot jelölheti. Ha ezeket az értékeket numerikus állandóként hagyjuk a kódban, akkor az, aki a programot karbantartja, nagyon nehezen tudja megtalálni és megérteni azokat. Ezeket az állandókat gyakran nem veszik észre, és érvénytelenné válnak, amikor a programot átviszik más rendszerre vagy ha más változások aláássák az általuk kifejezett feltételezéseket. Ha a feltevéseket megjegyzésekkel megfelelően ellátott szimbolikus konstansokként valósítjuk meg, minimálisra csökkenthetjük az ilyen jellegű karbantartási problémákat.

5.4.1. Mutatók és konstansok

A mutatók használatakor két objektummal kapcsolatos dologról van szó: magáról a mutatóról és az általa mutatott objektumról. Ha a mutató deklarációját a *const* szó előzi meg, akkor az az objektumot, és nem a mutatót határozza meg állandóként. Ahhoz, hogy állandóként egy mutatót, és ne az általa mutatott objektumot vezessük be, a **const* deklarátort kell használnunk a sima * helyett:

```
void f1(char* p)
{
   char s[ ] = "Gorm";

   const char* pc = s;            // mutató állandóra
   pc[3] = 'g';                   // hiba: pc állandóra mutat
   pc = p;                        // rendben

   char *const cp = s;            // konstans mutató
   cp[3] = 'a';                   // rendben
   cp = p;                        // hiba: cp konstans

   const char *const cpc = s;     // konstans mutató állandóra
   cpc[3] = 'a';                  // hiba: cpc állandóra mutat
   cpc = p;                       // hiba: cpc konstans
}
```

A **const* deklarátorjelző teszi állandóvá a mutatót. Nincs azonban *const** deklarátor-operátor, így a * előtt szereplő *const* kulcsszót az alaptípus részének tekintjük:

```
char *const cp;        // konstans mutató karakterre
char const* pc;        // mutató kostans karakterre
const char* pc2;       // mutató kostans karakterre
```

Általában segítséget jelent, ha az ilyen deklarációkat jobbról balra olvassuk ki. Például: „*cp* egy konstans (*const*) mutató, amely egy karakterre (*char*) mutat" és „*pc2* egy mutató, amely egy karakter-konstansra (*char const*) mutat".

Egy objektum, amely állandó akkor, amikor mutatón keresztül férünk hozzá, lehet, hogy módosítható lesz akkor, ha más módon férünk hozzá. Ez különösen hasznos a függvényparaméterek esetében. Azzal, hogy egy mutató-paramétert *const*-ként adunk meg, a függvénynek megtiltjuk, hogy módosítsa a mutató által mutatott objektumot:

```
char* strcpy(char* p, const char* q); // *q nem módosítható
```

Egy változó címét értékül adhatjuk egy konstansra hivatkozó mutatónak, mert ebből még semmi rossz nem következik. Konstans címét azonban nem lehet értékül adni egy nem konstans mutatónak, mert ezzel megengednénk, hogy az objektum értéke megváltozzon:

```
void f4()
{
  int a = 1;
  const int c = 2;
  const int* p1 = &c;          // rendben
  const int* p2 = &a;          // rendben
  int* p3 = &c;                // hiba: kezdeti értékadás int*-nak const int*-gal
  *p3 = 7;                     // kísérlet c értékének módosítására
}
```

A *const*-ra hivatkozó mutatókkal kapcsolatos megszorításokat meghatározott (explicit) típuskonverzióval küszöbölhetjük ki (§10.2.7.1 és §15.4.2.1).

5.5. Referenciák

A *referencia* (hivatkozás) egy objektum „álneve" (alias). Az ilyen hivatkozásokat általában függvények és különösen túlterhelt operátorok (11. fejezet) paramétereinek és visszatérési értékeinek megadására használjuk. Az *X&* jelölés jelentése „referencia *X*-re". Lássunk egy példát:

```
void f()
{
  int i = 1;
  int& r = i;                  // r és i itt ugyanarra az int-re hivatkoznak
  int x = r;                   // x = 1

  r = 2;                       // i = 2
}
```

Azt biztosítandó, hogy a referencia valaminek a neve legyen (azaz tartozzon hozzá objektum), a hivatkozás célpontját már létrehozáskor meg kell határoznunk:

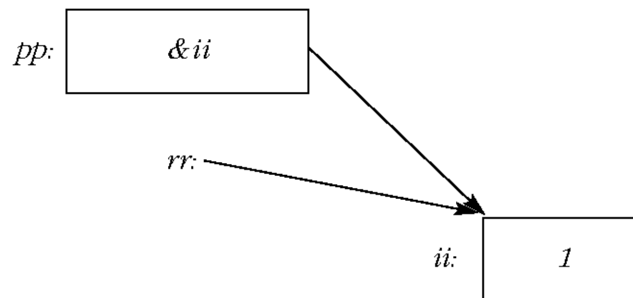
```
int i = 1;
int& r1 = i;                   // rendben: r1 kapott kezdőértéket
int& r2;                       // hiba: kezdeti értékadás hiányzik
extern int& r3;                // rendben: r3 máshol kap kezdőértéket
```

A referencia kezdeti értékadása nagyban különbözik a későbbi értékadástól. A látszat ellenére a referencián nem hajtódik végre egyetlen művelet sem. Például:

```
void g()
{
   int ii = 0;
   int& rr = ii;
   rr++;                    // ii növelése eggyel
   int* pp = &rr;           // pp az ii-re mutat
}
```

Ez nem helytelen, de *rr++* nem az *rr* értékét növeli; a ++ egy *int*-re hajtódik végre (ami itt *ii*). Következésképpen a referenciák értéke már nem módosítható a kezdeti értékadás után; mindig arra az objektumra fognak hivatkozni, amelyre kezdetben beállítottuk azokat. Az *rr* által jelölt objektumra hivatkozó mutatót *&rr*-rel kaphatjuk meg.

A referencia magától értetődő módon megvalósítható (konstans) mutatóként is, amely minden egyes használatakor automatikusan feloldja a mutató-hivatkozást. Nem származhat baj abból, ha így gondolunk a referenciákra, mindaddig, míg el nem felejtjük, hogy nem olyan objektumok, amelyet mutatóként kezelhetnénk:



Egyes fordítóprogramok olyan optimalizációt alkalmazhatnak, amely a referencia számára futási időben szükségtelenné teszi tárterület lefoglalását.

A referencia kezdeti értékadása magától értetődő, ha a kezdőérték egy balérték (vagyis egy olyan objektum, amelynek címére hivatkozhatunk, lásd 4.9.6). Egy „sima” *T&* kezdőértéke *T* típusú balérték kell, hogy legyen. Egy *const T&* esetében ez nem szükséges (sem balértéknek, sem *T* típusúnak nem kell lennie), helyette az alábbiak történnek:

1. Először *T*-re történő automatikus típuskonverzió megy végbe, ha szükséges (lásd §C.6-ot),
2. aztán a kapott érték egy *T* típusú ideiglenes változóba kerül,
3. végül ez az ideiglenes változó lesz a kezdőérték.

Vegyük a következő példát:

```
double& dr = 1;                // hiba: balértékre van szükség
const double& cdr = 1;         // rendben
```

A második a következőképpen értelmezhető:

```
double temp = double(1);    // először létrehozunk egy ideiglenes változót a jobb oldali
                            // értékkel
const double& cdr = temp;   // majd ezt használjuk a cdr kezdeti értékadására
```

A referencia kezdőértékét tároló ideiglenes változó a referencia hatókörének végéig marad fenn. A konstansok és változók hivatkozásait azért különböztetjük meg, mert a változók esetében nagy hibalehetőségeket rejt magában egy ideiglenes változó bevezetése, a változónak való értékadás ugyanis a – nemsokára megszűnő – ideiglenes tárterületnek adna értéket. A konstansok hivatkozásaival nincs ilyen probléma, ami szerencsés, mert ezek gyakran függvényparaméterként játszanak fontos szerepet (§11.6).

A referenciákat olyan függvényparaméterek megadására is használhatjuk, melyeken keresztül a függvény módosíthatja a neki átadott objektum értékét:

```
void increment(int& aa) { aa++; }

void f()
{
   int x = 1;
   increment(x);            // x = 2
}
```

A paraméterátadás a kezdeti értékadáshoz hasonló, így az *increment* meghívásakor az *aa* paraméter az *x* másik neve lesz. Ha azt szeretnénk, hogy a program olvasható maradjon, legjobb, ha elkerüljük az olyan függvényeket, amelyek módosítják paramétereiket. Ehelyett meghatározhatjuk a függvény által visszaadandó értéket vagy mutató paramétert adhatunk neki:

```
int next(int p) { return p+1; }

void incr(int* p) { (*p)++; }
```

```
void g()
{
   int x = 1;
   increment(x);              // x = 2
   x = next(x);               // x = 3
   incr(&x);                  // x = 4
}
```

Az *increment(x)* jelölés az olvasónak semmit sem árul el arról, hogy az *x* értéke módosul, ellentétben az *x=next(x)* és *incr(&x)* jelölésekkel. Következésképpen a „sima" referencia-paramétereket csak olyan esetekben használjuk, amikor a függvény neve határozottan utal arra, hogy ezek módosulnak.

A referenciákat olyan függvények megadására is használhatjuk, amelyek egy értékadás bal és jobb oldalán egyaránt szerepelhetnek. Ez a bonyolultabb felhasználói típusok tervezésekor lehet igazán hasznos. Adjunk meg például egy egyszerű asszociatív tömböt. Először határozzuk meg a *Pair* adatszerkezetet:

```
struct Pair {
   string name;
   double val;
};
```

Az alapötlet az, hogy a *string*-hez tartozik egy lebegőpontos érték. Könnyű elkészíteni a *value()* függvényt, amely egy *Pair*-ből álló adatszerkezetet vet össze különböző karakterláncokkal. Rövidítsük le a példát és használjunk egy nagyon egyszerű (persze nem túl hatékony) megvalósítást:

```
vector<Pair> pairs;

double& value(const string& s)
/*
   Vesszük Pair-ek egy halmazát,
   megkeressük s-t, ha megtaláltuk, visszaadjuk az értékét; ha nem, új Pair-t készítünk és
   visszaadjuk az alapértelmezett 0-át.
*/
{

   for (int i = 0; i < pairs.size(); i++)
            if (s == pairs[i].name) return pairs[i].val;

   Pair p = { s, 0 };
   pairs.push_back(p);          // Pair hozzáadása a végéhez (§3.7.3)

   return pairs[pairs.size()-1].val;
}
```

Ezt a függvényt úgy foghatjuk fel, mint egy lebegőpontos értékekből álló tömböt, amit karakterláncok indexelnek. Adott karakterlánccal összevetve, a *value()* a megfelelő lebegőpontos objektumot (nem pedig annak értékét) találja meg, és az erre vonatkozó referenciát adja vissza:

```
int main()        // az egyes szavak előfordulásának megszámlálása a bemeneten
{
  string buf;

  while (cin>>buf) value(buf)++;

  for (vector<Pair>::const_iterator p = pairs.begin(); p!=pairs.end(); ++p)
          cout << p->name << ": " << p->val << '\n';
}
```

A *while* ciklus minden esetben beolvas egy szót a *cin* szabványos bemenetről és a *buf* karakterláncba helyezi (§3.6.), aztán növeli a hozzá tartozó számlálót. Végül kiírja az eredményül kapott táblázatot, amelyben a bemenetről kapott karakterláncok és azok előfordulásának száma szerepel. Ha a bemenet például a következő:

```
aa bb bb aa aa bb aa aa
```

akkor a program eredménye az alábbi:

```
aa: 5
bb: 3
```

Ezt már könnyű úgy tovább finomítani, hogy valódi asszociatív tömböt kapjunk; ehhez egy sablon osztályt kell használnunk a túlterhelt (§11.8.) *[]* indexelő operátorral. Még könnyebb a dolgunk, ha a standard könyvtár *map* (§17.4.1.) típusát használjuk.

5.6. Void-ra hivatkozó mutatók

Bármilyen típusú objektumra hivatkozó mutatót értékül lehet adni egy *void** típusú változónak, egy *void** típusú változót értékül lehet adni egy másik *void** típusúnak, a *void** típusú változókat össze lehet hasonlítani, hogy egyenlőek-e vagy sem, egy *void** típusú változót pedig meghatározott módon más típusúvá lehet alakítani. A többi művelet nem lenne

biztonságos, mert a fordítóprogram nem tudja, hogy valójában miféle objektumra hivatkozik egy ilyen mutató, ezért a többi művelet fordítási idejű hibát eredményez. Ahhoz, hogy egy *void** típusú változót használhassunk, át kell konvertálnunk azt adott típusú mutatóvá:

```
void f(int* pi)
{
  void* pv = pi;       // rendben: int* automatikus konvertálása void*-gá
  *pv;                 // hiba: nem lehet void*-ra hivatkozni
  pv++;                // hiba: void* nem növelhető (a mutatott objektum mérete ismeretlen)

  int* pi2 = static_cast<int*>(pv);           // explicit visszaalakítás int*-ra

  double* pd1 = pv;                           // hiba
  double* pd2 = pi;                           // hiba
  double* pd3 = static_cast<double*>(pv);     // nem biztonságos
}
```

Általában nem biztonságos olyan mutatót használni, amely olyan típusra konvertálódik (*cast*), amely különbözik a mutató által előzőleg hivatkozott típustól. A gép például feltételezheti, hogy minden *double* 8 bájtos memóriahatáron jön létre. Ha így van, akkor furcsa működés származhat abból, ha a *pi* egy olyan *int*-re mutatott, amely nem így helyezkedett el a memóriában. Az ilyen jellegű explicit típuskényszerítés eredendően csúnya és nem biztonságos, következésképp a használt *static_cast* jelölést is szándékosan csúnyának tervezték.

A *void** elsődlegesen arra használatos, hogy mutatókat adjunk át olyan függvényeknek, amelyek nem feltételeznek semmit az objektumok típusáról, valamint arra, hogy függvények nem típusos objektumokat adjanak vissza. Ahhoz, hogy ilyen objektumokat használjunk, explicit típuskonverziót kell alkalmaznunk. Azok a függvények, amelyek *void** típusú mutatókat használnak, jellemzően a rendszer legalsó szintjén helyezkednek el, ahol az igazi hardver-erőforrásokat kezelik. Például:

```
void* my_alloc(size_t n);     // n bájt lefoglalása saját tárterületen
```

A rendszer magasabb szintjein lévő *void** típusú mutatókat gyanakvással kell figyelnünk, mert tervezési hibát jelezhetnek. Ha a *void**-ot optimalizálásra használjuk, rejtsük típusbiztos felület mögé (§13.5, §24.4.2).

A függvényekre hivatkozó mutatókat (§7.7.) és a tagokra hivatkozó mutatókat (§15.5) nem adhatjuk értékül *void** típusú változónak.

5.7. Struktúrák

A tömbök azonos típusú elemekből állnak, a *struct*-ok (adatszerkezetek, *struktúrák*) majdnem tetszőleges típusúakból:

```
struct address {
  char* name;              // "Jim Dandy"
  long int number;         // 61
  char* street;            // "South St"
  char* town;              // "New Providence"
  char state[2];           // 'N' 'J'
  long zip;                // 7974
};
```

A fenti kód egy *address* (cím) nevű új típust hoz létre, amely levelek küldéséhez szükséges címzési adatokat tartalmaz. Vegyük észre a pontosvesszőt a definíció végén. Ez egyike azon kevés helyeknek a C++-ban, ahol pontosvesszőt kell tenni a kapcsos zárójel után, ezért sokan hajlamosak elfelejteni.

Az *address* típusú változókat pontosan úgy adhatjuk meg, mint más változókat, és az egyes tagokra a . (pont, tagkiválasztó) operátorral hivatkozhatunk:

```
void f()
{
  address jd;
  jd.name = "Jim Dandy";
  jd.number = 61;
}
```

A tömbök kezdeti értékadására használt jelölés a struktúra-típusú változók feltöltésére is használható:

```
address jd = {
  "Jim Dandy",
  61, "South St",
  "New Providence", {'N','J'}, 7974
};
```

Ennél azonban rendszerint jobb megoldás konstruktorokat (§10.2.3) használni. Vegyük észre, hogy a *jd.state*-et nem lehetett volna az *"NJ"* karakterlánccal feltölteni. Mivel a karakterláncok a *'\0'* karakterre végződnek, az *"NJ"* három karakterből áll, ami eggyel több, mint ami a *jd.state*-be belefér.

A struktúrák objektumaira gyakran hivatkozunk mutatókon keresztül a -> (struktúra-mutató) operátorral:

```
void print_addr(address* p)
{
  cout << p->name << '\n'
       << p->number << ' ' << p->street << '\n'
       << p->town << '\n'
       << p->state[0] << p->state[1] << ' ' << p->zip << '\n';
}
```

Ha *p* egy mutató, akkor *p->m* egyenértékű *(*p).m*-mel.

A struktúra-típusú objektumokat értékül adhatjuk, átadhatjuk függvényparaméterként, és visszaadhatjuk függvények visszatérési értékeként is:

```
address current;

address set_current(address next)
{
  address prev = current;
  current = next;
  return prev;
}
```

Más lehetséges műveletek, mint az összehasonlítás (== és !=), nem meghatározottak, de a felhasználó megadhat ilyeneket (11. fejezet).

A struktúra-típusú objektumok mérete nem feltétlenül a tagok méretének összege. Ennek az az oka, hogy sok gép igényli bizonyos típusú objektumok elhelyezését a felépítéstől függő memóriahatárokra, vagy eleve hatékonyabban kezeli az így létrehozott objektumokat. Az egészek például gyakran gépi szóhatárokon jönnek létre. Ezt úgy mondjuk, hogy az ilyen gépeken az objektumok jól illesztettek. Ez a struktúrákon belül „lyukakat" eredményez. Számos gépen a *sizeof(address)* például *24*, nem pedig *22*, ahogy az elvárható lenne. Az elpazarolt helyet a lehető legkevesebbre csökkenthetjük, ha egyszerűen méret szerint rendezzük a struktúra tagjait (a legnagyobb tag lesz az első). A legjobb azonban az, ha olvashatóság szerint rendezzük sorba a tagokat, és csak akkor méret szerint, ha bizonyítottan szükség van optimalizálásra.

Egy típus neve rögtön felhasználható attól a ponttól, ahol először megjelenik, nem csak a teljes deklaráció után:

```
struct Link {
  Link* previous;
  Link* successor;
};
```

A struktúra teljes deklarációjának végéig viszont nem adhatunk meg újabb ilyen típusú objektumokat:

```
struct No_good {
  No_good member;          // hiba: rekurzív definíció
};
```

Ez azért hibás, mert a fordítóprogram nem képes eldönteni a *No_good* méretét. Két (vagy több) struktúra-típus kölcsönös hivatkozásához adjunk meg például egy nevet, amely a típus neve:

```
struct List;             // később meghatározandó

struct Link {
  Link* pre;
  Link* suc;
  Link* member_of;
};

struct List {
  Link* head;
};
```

A *List* első deklarációja nélkül a *List* használata a *Link* deklarációjában formai hibát okozott volna. A struktúra-típus neve a típus meghatározása előtt is felhasználható, feltéve, hogy ez a használat nem igényli egy tag nevének vagy a struktúra méretének ismeretét:

```
class S;     // 'S' valamilyen típus neve

extern S a;
S f();
void g(S);
S* h(S*);
```

A fenti deklarációk közül azonban sok nem használható, hacsak meg nem adjuk az *S* típusát:

```
void k(S* p)
{
  S a;               // hiba: S nem definiált; a helyfoglaláshoz méret kell

  f();               // hiba: S nem definiált; érték visszaadásához méret kell
  g(a);              // hiba: S nem definiált; paraméter átadásához méret kell
  p->m = 7;          // hiba: S nem definiált; a tag neve nem ismert

  S* q = h(p);       // rendben: a mutatók számára foglalható hely és át is adhatók
  q->m = 7;          // hiba: S nem definiált; a tag neve nem ismert
}
```

A *struct* az osztály (10. fejezet) egyszerű formája.

A C történetére visszanyúló okok miatt ugyanazzal a névvel és ugyanabban a hatókörben megadhatunk egy *struct*-ot és egy nem struktúra jellegű típust is:

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

Ebben az esetben a „sima" *stat* név a nem-struktúra neve, az adatszerkezetre pedig a *struct* előtaggal kell hivatkoznunk. Előtagként a *class*, *union* (§C.8.2) és *enum* (§4.8) kulcsszavak is használhatók, ezekkel elkerülhetjük a kétértelműséget. A legjobb azonban, ha nem terheljük túl a neveket.

5.7.1. Egyenértékű típusok

Két struktúra mindig különböző típusú, akkor is, ha tagjaik ugyanazok:

```
struct S1 { int a; };
struct S2 { int a; };
```

A fenti két típus különböző, így

```
S1 x;
S2 y = x;            // hiba: nem megfelelő típus
```

A struktúra-típusok az alaptípusoktól is különböznek, ezért

```
S1 x;
int i = x;   // hiba: nem megfelelő típus
```

Minden *struct*-nak egyértelmű meghatározása kell, hogy legyen a programban (§9.2.3.).

5.8. Tanácsok

[1] Kerüljük a nem magától értetődő mutató-aritmetikát. §5.3.
[2] Ügyeljünk arra, hogy ne írjunk egy tömb indexhatárán túlra. §5.3.1.
[3] Használjunk *0*-át *NULL* helyett. §5.1.1.
[4] Használjuk a *vector*-t és a *valarray*-t a beépített (C stílusú) tömbök helyett. §5.3.1.
[5] Használjunk *string*-et nulla végződésű karaktertömbök helyett. §5.3.
[6] Használjunk a lehető legkevesebb egyszerű referencia-paramétert. §5.5.
[7] Az alacsonyszintű kódot kivéve kerüljük a *void**-ot. §5.6.
[8] Kerüljük a kódban a nem magától értetődő literálokat („mágikus számokat"). Használjunk helyettük jelképes állandókat. §4.8, §5.4.

5.9. Gyakorlatok

1. (*1) Vezessük be a következőket: karakterre hivatkozó mutató, 10 egészből álló tömb, 10 egészből álló tömb referenciája, karakterláncokból álló tömbre hivatkozó mutató, karakterre hivatkozó mutatóra hivatkozó mutató, konstans egész, konstans egészre hivatkozó mutató, egészre hivatkozó konstans mutató. Mindegyiknek adjunk kezdeti értéket.
2. (*1,5) Mik a *char**, *int**, és *void** mutatótípusokra vonatkozó megszorítások a mi rendszerünkön? Lehetne-e például egy *int**-nak furcsa értéke? Segítség: *illesztés*.

3. (*1) Használjunk *typedef*-et a következők meghatározására: *unsigned char*, *const unsigned char*, egészre hivatkozó mutató, karakterre hivatkozó mutatóra hivatkozó mutató, karaktertömbökre hivatkozó mutató; 7 elemű, egészre hivatkozó mutatókból álló tömb; 7 elemű, egészre hivatkozó mutatókból álló tömbre hivatkozó mutató; egészre hivatkozó mutatókat tartalmazó 7 elemű tömbökből álló 8 elemű tömb.
4. (*1) Írjunk egy *swap* nevű függvényt, amely két egészt cserél fel. Használjunk *int** típust a paraméterek típusaként. Írjunk egy másik *swap*-et is, melynek paraméterei *int&* típusúak.
5. (*1,5) Mi az *str* tömb mérete a következő példában?

```
char str[] = "rövid karakterlánc";
```

   Mi a *"rövid karakterlánc"* hossza?
6. (*1) Készítsük el az *f(char)*, *g(char&)* és *h(const char&)* függvényeket. Hívjuk meg őket az *'a'*, *49*, *3300*, *c*, *uc* és *sc* paraméterekkel, ahol *c char*, *uc unsigned char* és *sc signed char* típusú. Mely hívások megengedettek? Mely hívásoknál vezet be a fordítóprogram ideiglenes változót?
7. (*1,5) Készítsünk egy táblázatot, amely a hónapok neveiből és napjaik számából áll. Írjuk ki a táblázatot. Csináljuk meg mindezt kétszer: egyszer használjunk karaktertömböt a nevek és egy tömböt a napok számára, másodszor használjunk struktúrákból álló tömböt, ahol az egyes adatszerkezetek a hónap nevét és a benne levő napok számát tárolják.
8. (*2) Futtassunk le néhány tesztet, hogy megnézzük, a fordítóprogram tényleg egyenértékű kódot hoz-e létre a mutatók használatával és az indexeléssel való tömbbejáráshoz (§5.3.1). Ha különböző mértékű optimalizálást lehet használni, nézzük meg, hat-e és hogyan hat ez a létrehozott kód minőségére.
9. (*1,5) Találjunk példát, hol lenne értelme egy nevet a saját kezdőértékében használni.
10. (*1) Adjunk meg egy karakterláncokból álló tömböt, ahol a karakterláncok a hónapok neveit tartalmazzák. Írjuk ki ezeket. Adjuk át a tömböt egy függvénynek, amely kiírja a karakterláncokat.
11. (*2) Olvassuk be a bemenetről szavak egy sorozatát. A bemenetet lezáró szóként használjuk a *Quit*-et. Írjuk ki a beolvasott szavakat. Ne írjuk ki kétszer ugyanazt a szót. Módosítsuk a programot, hogy rendezze a szavakat, mielőtt kiírná azokat.
12. (*2) Írjunk olyan függvényt, amely megszámolja egy betűpár előfordulásait egy karakterláncban, és egy másikat, ami ugyanezt csinálja egy nulla végű karaktertömbben (vagyis egy C stílusú karakterláncban). Az *"ab"* pár például kétszer szerepel az *"xabaacbaxabb"*-ben.
13. (*1,5) Adjunk meg egy *Date* struktúrát dátumok ábrázolásához. Írjunk olyan függvényt, ami *Date*-eket olvas be a bemenetről, olyat, ami *Date*-eket ír a kimenetre, és olyat, ami egy dátummal ad kezdőértéket a *Date*-nek.

6

Kifejezések és utasítások

*„Az idő előtti optimalizálás
minden rossz gyökere."
(D. Knuth)*

*„Másrészről, nem hagyhatjuk
figyelmen kívül
a hatékonyságot."
(John Bentley)*

„Asztali számológép" példa • Bemenet • Parancssori paraméterek • Kifejezések (áttekintés) • Logikai és összehasonlító operátorok • Növelés és csökkentés • Szabad tár • Meghatározott típuskonverziók • Utasítások (áttekintés) • Deklarációk • Elágazó utasítások • Deklarációk a feltételekben • Ciklusutasítások • A hírhedt *goto* • Megjegyzések és behúzás • Tanácsok • Gyakorlatok

6.1. Egy asztali számológép

A kifejezéseket és utasításokat egy asztali számológép példáján keresztül mutatjuk be. A számológép a négy aritmetikai alapműveletet hajtja végre lebegőpontos értékeken. A műveleti jeleket a számok között (infix operátorként) kell megadni. A felhasználó változókat is megadhat. A bemenet legyen a következő:

```
r = 2.5
area = pi * r * r
```

A számológép program az alábbiakat fogja kiírni (*pi* előre meghatározott):

```
2.5
19.635
```

A *2.5* a bemenet első sorának, a *19.635* a bemenet második sorának eredménye.

A számológép négy fő részből áll: egy elemzőből (parser), egy adatbeviteli függvényből, egy szimbólumtáblából és egy vezérlőből. Valójában ez egy miniatűr fordítóprogram, amelyben az elemző végzi a szintaktikai elemzést (vagyis a nyelvi utasítások formai elemzését), az adatbeviteli függvény kezeli a bemenetet és végzi a lexikai elemzést (vagyis a nyelvi elemek értelmezését), a szimbólumtábla tartalmazza a nem változó adatokat és a vezérlő kezeli a kezdeti értékadást, a kimenetet és a hibákat. A számológépet számos szolgáltatással bővíthetjük, hogy még hasznosabbá tegyük (§6.6[20]), de a kód így is elég hosszú lesz, és a legtöbb szolgáltatás csak a kódot növelné, anélkül, hogy további betekintést nyújtana a C++ használatába.

6.1.1. Az elemző

Íme a számológép által elfogadott nyelvtan:

```
program:
        END                          // END a bevitel vége
        expr_list END

expr_list:
        expression PRINT             // PRINT a pontosvessző
        expression PRINT expr_list
```

```
expression:
        expression + term
        expression - term
        term

term:
        term / primary
        term * primary
        primary

primary:
        NUMBER
        NAME
        NAME = expression
        - primary
        ( expression )
```

Más szóval, a program kifejezések sorozata, amelyeket pontosvesszők választanak el egymástól. A kifejezések alapelemei a számok, nevek és a *, /, +, - (akár egy, akár két operandusú) operátorok, és az = . A neveket nem kell használat előtt definiálni.

Az általunk használt szintaktikai elemzés módszerét rendszerint *rekurzív leszállásnak* (recursive descent) nevezik; népszerű és lényegretörő, felülről lefelé haladó eljárás. Egy olyan nyelvben, mint a C++, amelyben a függvényhívások viszonylag „kis költségűek", a módszer hatékony is. A nyelvtan minden szabályára adunk egy függvényt, amely más függvényeket hív meg. A lezáró szimbólumokat (például az *END*, *NUMBER*, + és -) a *get_token()* lexikai elemző, a nem lezáró szimbólumokat pedig az *expr()*, *term()* és *prim()* szintaktikai elemző függvények ismerik fel. Ha egy (rész)kifejezés minkét operandusa ismert, a kifejezés kiértékelődik – egy igazi fordítóprogram esetében ezen a ponton történhetne a kód létrehozása. Az elemző a *get_token()* függvényt használja arra, hogy bemenetet kapjon. Az utolsó *get_token()* hívás eredménye a *curr_tok* globális változóban található. A *curr_tok* változó típusa *Token_value* felsoroló típus:

```
enum Token_value {
    NAME,           NUMBER,         END,
    PLUS='+',       MINUS='-',      MUL='*',        DIV='/',
    PRINT=';',      ASSIGN='=',     LP='(',         RP=')'
};

Token_value curr_tok = PRINT;
```

Az, hogy minden szimbólumot (token) a karakterének megfelelő egész értékkel jelölünk, kényelmes és hatékony megoldás, és segítheti azokat, akik hibakeresőt (debugger) használnak. Ez a módszer addig működik, amíg bemenetként olyan karaktert nem adunk meg, melynek értékét felsoroló konstansként már használjuk. Én pedig nem tudok olyan karakterkészletről, amelyben van olyan kiírható karakter, melynek egész értéke egy számjegyű. Azért választottam a *curr_tok* kezdeti értékeként a *PRINT*-et, mert a *curr_tok* ezt az értéket fogja felvenni, miután a számológép kiértékelt egy kifejezést és kiírta annak értékét. Így alapállapotban „indítjuk el a rendszert”, a lehető legkisebbre csökkentjük annak az esélyét, hogy hibák forduljanak elő és egyedi indítókódra sincs szükségünk.

Minden elemző függvénynek van egy logikai (*bool*) (§4.2) paramétere, amely jelzi, hogy meg kell-e hívnia a *get_token()*-t a következő szimbólum beolvasásához. A függvény kiértékeli a „saját” kifejezését és visszaadja az értékét. Az *expr()* függvény kezeli az összeadást és kivonást. A függvény egyetlen ciklusból áll, amely elemeket (*term*) keres az összeadáshoz vagy kivonáshoz:

```
double expr(bool get)                    // összeadás és kivonás
{
  double left = term(get);

  for (;;)                               // "örökké" (végtelen ciklus)
          switch (curr_tok) {
          case PLUS:
                  left += term(true);
                  break;
          case MINUS:
                  left -= term(true);
                  break;
          default:
                  return left;
          }
}
```

Ez a függvény önmagában nem csinál túl sokat. Egy nagyobb program magasabb szintű függvényeihez hasonló módon más függvényeket hív meg a feladat elvégzéséhez.

A *switch* utasítás azt vizsgálja meg, hogy a *switch* kulcsszó után zárójelben megadott feltétel értéke megegyezik-e a konstansok valamelyikével. A *break*-kel a *switch* utasításból léphetünk ki. A *case* címkéket követő konstansoknak különbözniük kell egymástól. Ha a vizsgált érték nem egyezik egyik *case* címkével sem, a *default* címke választódik ki. A programozónak nem kötelező megadnia a *default* részt.

Figyeljük meg, hogy a *2-3+4*-hez hasonló kifejezések *(2-3)+4*-ként értékelődnek ki, ahogy azt a nyelvtanban meghatároztuk.

A furcsa *for(; ;)* jelölés megszokott módja annak, hogy végtelen ciklust írjunk; úgy mondhatjuk ki, hogy „örökké” (*forever*). Ez a *for* utasítás (§6.3.3) végletes formája; helyette használhatjuk a *while(true)* szerkezetet is. A *switch* utasítás végrehajtása addig ismétlődik, amíg nem talál a +-tól és --tól különböző jelet, amikor is a *default* címke utáni *return* utasítás hajtódik végre.

Az összeadás és kivonás kezelésére a += és -= operátorokat használjuk. Használhatnánk a *left=left+term(true)* és *left=left-term(true)* formát is, a program jelentése nem változna. A *left+=term(true)* és a *left-=term(true)* azonban nemcsak rövidebbek, hanem közvetlenebbül is fejezik ki a kívánt műveletet. Minden értékadó operátor önálló nyelvi egység, így a + = *1* nyelvtanilag hibás a + és az = közötti szóköz miatt.

A következő kétoperandusú műveletekhez léteznek értékadó operátorok:

+ - * / % & | ^ << >>

Így a következő értékadó operátorok lehetségesek:

= += -= *= /= %= &= |= ^= <<= >>=

A % a moduló vagy maradékképző operátor; &, |, és ^ a bitenkénti ÉS, VAGY, illetve kizáró VAGY operátorok; << és >> pedig a balra és jobbra léptető operátorok. A műveleti jeleket és jelentésüket §6.2 foglalja össze. Ha @ egy bináris (kétoperandusú) operátor, akkor *x@=y* jelentése *x=x@y*, azzal a különbséggel, hogy *x* csak egyszer értékelődik ki.

A 8. és a 9. fejezet tárgyalja, hogyan építsünk fel programot modulokból. A számológép példa deklarációit – egy kivétellel – úgy rendezhetjük sorba, hogy mindent csak egyszer és használat előtt adunk meg. A kivétel az *expr()*, ami meghívja a *term()*-et, ami meghívja a *prim()*-et, ami pedig ismét meghívja az *expr()*-et. Ezt a kört valahol meg kell szakítanunk. A *prim()* meghatározása előtti deklaráció erre való.

```
double expr(bool);
```

A *term()* függvény ugyanolyan módon kezeli a szorzást és osztást, mint ahogy az *expr()* kezeli az összeadást és kivonást:

```
double term(bool get)                    // szorzás és osztás
{
  double left = prim(get);

  for (;;)
        switch (curr_tok) {
        case MUL:
              left *= prim(true);
              break;

        case DIV:
              if (double d = prim(true)) {
                    left /= d;
                    break;
              }
              return error("Nullával nem lehet osztani");

        default:
              return left;
        }
}
```

A nullával való osztás nem meghatározott és rendszerint végzetes hibát okoz. Ezért osztás előtt megnézzük, hogy a nevező *0* -e, és ha igen, meghívjuk az *error()*-t. Az *error()* függvényt a §6.1.4-ben ismertetjük. A *d* változót pontosan azon a ponton vezetjük be a programba, ahol az szükséges, és rögtön kezdeti értéket is adunk neki. Egy feltételben bevezetett név hatóköre a feltétel által vezérelt utasítás, az eredményezett érték pedig a feltétel értéke (§6.3.2.1). Következésképpen a *left/=d* osztás és értékadás csak akkor megy végbe, ha *d* nem nulla.

A *prim()* függvény, amely az elemi szimbólumokat kezeli, nagyban hasonlít az *expr()*-re és a *term()*-re, kivéve azt, hogy mivel már lejjebb értünk a hívási hierarchiában, némi valódi munkát kell végezni és nincs szükség ciklusra:

```
double number_value;
string string_value;

double prim(bool get)                    // elemi szimbólumok kezelése
{
  if (get) get_token();

  switch (curr_tok) {
  case NUMBER:                           // lebegőpontos konstans
  {     double v = number_value;
        get_token();
        return v;
  }
```

```
        case NAME:
        {       double& v = table[string_value];
                if (get_token() == ASSIGN) v = expr(true);
                return v;
        }
        case MINUS:                             // egyoperandusú mínusz
                return -prim(true);

        case LP:
        {       double e = expr(true);
                if (curr_tok != RP) return error(")  szükséges");
                get_token();                    // ')' lenyelése
                return e;
        }
        default:
                return error("elemi szimbólum szükséges");
        }
}
```

Amikor egy *NUMBER*-t (azaz egy egész vagy lebegőpontos literált) találunk, visszaadjuk az értékét. A *get_token()* bemeneti eljárás elhelyezi az értéket a *number_value* globális változóban. Globális változó használata a kódban gyakran jelenti, hogy a program szerkezete nem kristálytiszta – valamiféle optimalizációt alkalmaztak rá. Itt is ez történt. Ideális esetben egy nyelvi egység (lexikai szimbólum) két részből áll: egy értékből, amely meghatározza a szimbólum fajtáját (ebben a programban ez a *Token_value*) és (ha szükséges) a *token* értékéből. Itt csak egy egyszerű *curr_tok* változó szerepel, így a *number_value* globális változó szükséges ahhoz, hogy az utolsó beolvasott *NUMBER* értékét tárolja. E kétes szerepű globális változó kiküszöbölését is a feladatok közé tűzzük ki (§6.6[21]). A *number_value* értékét nem feltétlenül szükséges a *v* lokális változóba menteni a *get_token()* meghívása előtt. A számológép a számításhoz minden helyes bemenetnél használatba veszi az első számot, mielőtt egy másikat olvasna be, hiba esetén viszont segítheti a felhasználót, ha mentjük az értéket és helyesen kiírjuk. Hasonlóan ahhoz, ahogy az utolsó beolvasott *NUMBER*-t a *number_value* tárolja, az utolsó beolvasott *NAME* karakterláncot a *string_value* tartalmazza. Mielőtt a számológép bármit kezdene egy névvel, meg kell néznie, hogy a nevet értékül kell-e adnia vagy csak egyszerűen be kell olvasnia. Mindkét esetben a szimbólumtáblához fordul. A szimbólumtábla egy *map* (§3.7.4, §17.4.1):

```
map<string,double> table;
```

Azaz, amikor a *table*-t egy karakterlánccal indexeljük, az eredményül kapott érték az a *double* lesz, ami a karakterlánchoz tartozik. Tegyük fel, hogy a felhasználó a következőket írja be:

```
radius = 6378.388;
```

Ekkor a számológép az alábbiakat hajtja végre:

```
double& v = table["radius"];
// ... expr() kiszámolja az átadandó értéket
v = 6378.388;
```

A *v* referenciát használjuk arra, hogy a *radius*-hoz tartozó *double* értékre hivatkozzunk, amíg az *expr()* a bemeneti karakterekből kiszámítja a 6378.388 értéket.

6.1.2. A bemeneti függvény

A bemenet beolvasása gyakran a program legrendezetlenebb része. Ez azért van így, mert a programnak egy emberrel kell társalognia és meg kell birkóznia annak szeszélyeivel, szokásaival és viszonylag véletlenszerű hibáival. Kellemetlen dolog (jogosan), ha megpróbáljuk a felhasználót rákényszeríteni, hogy úgy viselkedjen, hogy az a gép számára megfelelőbb legyen. Egy alacsonyszintű beolvasó eljárás feladata az, hogy karaktereket olvasson be és magasabb szintű szimbólumokat hozzon létre belőlük. Ezek a szimbólumok később a magasabb szintű eljárások bemeneti egységei lesznek. Itt az alacsonyszintű beolvasást a *get_token()* végzi. Nem feltétlenül mindennapi feladat alacsonyszintű bemeneti eljárásokat írni. Sok rendszer erre a célra szabványos függvényeket nyújt.

Két lépésben építem fel a *get_token()*-t. Először egy megtévesztően egyszerű változatot készítek, amely komoly terhet ró a felhasználóra. Ezután ezt módosítom egy kevésbé elegáns, de jóval használhatóbb változatra.

Az ötlet az, hogy beolvasunk egy karaktert, felhasználjuk arra, hogy eldöntsük, milyen szimbólumot kell létrehozni, majd visszaadjuk a beolvasott *token*-t ábrázoló *Token_value* értéket. A kezdeti utasítások beolvassák az első nem „üreshely" (whitespace, azaz szóköz, tabulátor, új sor stb.) karaktert *ch*-ba, és ellenőrzik, hogy az olvasási művelet sikerült-e:

```
Token_value get_token()
{
  char ch = 0;
  cin>>ch;

  switch (ch) {
  case 0:
          return curr_tok=END;        // értékadás és visszatérés
```

Alapértelmezés szerint a >> operátor átugorja az üreshely karaktereket és *ch* értékét változatlanul hagyja, ha a bemeneti művelet nem sikerül. Következésképpen *ch==0* a bemenet végét jelzi.

Az értékadás egy operátor, az értékadás eredménye pedig annak a változónak az értéke, melynek értéket adunk. Ez megengedi, hogy a *curr_tok* változónak az *END*-et adjam értékül, majd a változót ugyanabban az utasításban adjam vissza. Az, hogy egy utasítást használunk kettő helyett, megkönnyíti a kód későbbi módosítását. Ha az értékadást és a visszaadott értéket különválasztanánk a kódban, lehet, hogy a programozó megváltoztatná az egyiket, de elfelejtené módosítani a másikat.

Nézzünk meg néhány esetet külön-külön, mielőtt a teljes függvénnyel foglalkoznánk. A kifejezések *;* végződését, a zárójeleket és az operátorokat úgy kezeljük, hogy egyszerűen visszaadjuk az értéküket:

```
case ';':
case '*':
case '/':
case '+':
case '-':
case '(':
case ')':
case '=':
        return curr_tok=Token_value(ch);
```

A számokat így kezeljük:

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
        cin.putback(ch);
        cin >> number_value;
        return curr_tok=NUMBER;
```

A *case* címkéket függőleges helyett vízszintesen egy kupacba tenni általában nem jó ötlet, mert ez az elrendezés nehezebben olvasható. Fárasztó lenne azonban minden számjegyet külön sorba írni. Mivel a >> műveleti jel a lebegőpontos konstansokat szabályszerűen egy *double* típusú változóba olvassa, a kód magától értetődő. Először a kezdő karaktert (számjegyet vagy pontot) visszatesszük a *cin*-be, majd a konstanst a *number_value* változóba helyezzük.

A neveket hasonlóan kezeljük:

```
default:                              // NAME, NAME =, vagy hiba
        if (isalpha(ch)) {
                cin.putback(ch);
                cin>>string_value;
                return curr_tok=NAME;
        }
        error("rossz szimbólum");
        return curr_tok=PRINT;
```

A standard könyvtárban levő *isalpha()* függvényt (§20.4.2) használjuk arra, hogy ne kelljen minden karaktert felsorolnunk, mint különböző *case* címkéket. A karakterláncok (ebben az esetben a *string_value*) >> művelete addig olvassa a láncot, amíg üreshelyet nem talál. Következésképpen a felhasználónak szóközzel kell befejeznie az adott nevet azon operátorok előtt, melyek a nevet operandusként használják. Ez nem ideális megoldás, ezért erre a problémára még visszatérünk §6.1.3-ban.

Íme a teljes bemeneti függvény:

```
Token_value get_token()
{
  char ch = 0;
  cin>>ch;

  switch (ch) {
  case 0:
          return curr_tok=END;

  case ';':
  case '*':
  case '/':
  case '+':
  case '-':
  case '(':
  case ')':
  case '=':
          return curr_tok=Token_value(ch);

  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
  case '.':
          cin.putback(ch);
          cin >> number_value;
          return curr_tok=NUMBER;
```

```
    default:                               // NAME, NAME =, vagy hiba
            if (isalpha(ch)) {
                    cin.putback(ch);
                    cin>>string_value;
                    return curr_tok=NAME;
            }
            error("rossz szimbólum");
            return curr_tok=PRINT;
    }
}
```

Egy operátor átalakítása az operátornak megfelelő szimbólumra magától értetődő, mivel az operátorok *token_value* értékét az operátor egész értékeként határoztuk meg (§4.8).

6.1.3. Alacsonyszintű bemenet

Ha a számológépet úgy használjuk, ahogy az eddigiekben leírtuk, fény derül néhány kényelmetlen dologra. Fárasztó emlékezni arra, hogy pontosvesszőt kell tennünk egy kifejezés után, ha ki akarjuk íratni az értékét, és nagyon bosszantó tud lenni, hogy csak üreshellyel lehet egy nevet befejezni. Például *x=7* egy azonosító, és nem *x*, amit az = operátor és a 7-es szám követ. Mindkét problémát úgy oldjuk meg, hogy a *get_token()*-ben a típussal kapcsolatos alapértelmezett bemeneti műveleteket olyan kódra cseréljük, amely egyenként olvassa be a karaktereket. Először is, az „új sor" karaktert azonosként kezeljük a kifejezés végét jelző pontosvesszővel:

```
Token_value get_token()
{
    char ch;

    do {                // üreshelyek átugrása az '\n' kivételével
            if(!cin.get(ch)) return curr_tok = END;
    } while (ch!='\n' && isspace(ch));

    switch (ch) {
    case ';':
    case '\n':
            return curr_tok=PRINT;
```

A *do* utasítást használjuk, amely egyenértékű a *while* utasítással, kivéve, hogy a ciklusmag mindig legalább egyszer végrehajtódik. A *cin.get(ch)* beolvas egy karaktert a szabványos bemeneti adatfolyamból *ch*-ba. Alapértelmezés szerint a *get()* nem ugorja át az üreshelyeket

úgy, ahogy a >> művelet teszi. Az *if(!cin.get(ch))* ellenőrzés sikertelen, ha nem olvasható be karakter a *cin*-ből; ebben az esetben *END*-et adunk vissza, hogy befejezzük a számológép működését. A *! (NEM)* operátort azért használjuk, mert a *get()* igazat ad vissza, ha sikeres.

A standard könyvtár *isspace()* függvénye végzi az üreshelyek (§20.4.2) szabványos vizsgálatát. Ha *c* üreshely, az *isspace(c)* nem nulla értéket ad vissza, más esetben nullát. A vizsgálatot táblázatban való keresésként valósítjuk meg, így az *isspace()* használata sokkal gyorsabb, mint az egyes üreshely karakterek vizsgálata. Hasonló függvényekkel nézhetjük meg, hogy egy karakter számjegy (*isdigit()*), betű (*isalpha()*), esetleg betű vagy szám-e (*isalnum()*).

Miután átugrottuk az üreshelyeket, a következő karaktert arra használjuk, hogy eldöntsük, miféle nyelvi egység jön. A problémát, amit az okoz, hogy a >> addig olvassa a karakterláncot, amíg üreshelyeket nem talál, úgy oldjuk meg, hogy egyszerre egy karaktert olvasunk be, amíg olyan karaktert nem találunk, ami nem szám és nem betű:

```
default:                          // NAME, NAME=, vagy hiba
        if (isalpha(ch)) {
                string_value = ch;
                while (cin.get(ch) && isalnum(ch)) string_value.push_back(ch);
                cin.putback(ch);
                return curr_tok=NAME;
        }
        error("rossz szimbólum");
        return curr_tok=PRINT;
```

Szerencsére mindkét javítás elvégezhető úgy, hogy a kódnak csak egyes helyi érvényességű részeit módosítjuk. Fontos tervezési cél, hogy olyan programokat hozzunk létre, melyek javítását, fejlesztését helyi módosításokkal intézhetjük.

6.1.4. Hibakezelés

Mivel a program ennyire egyszerű, a hibakezeléssel nem kell komolyabban törődnünk. Az *error* függvény egyszerűen megszámolja a hibákat, kiír egy hibaüzenetet, és visszatér:

```
int no_of_errors;

double error(const string& s)
{
  no_of_errors++;
  cerr << "hiba: " << s << '\n';
  return 1;
}
```

A *cerr* egy átmeneti tárba nem helyezett („nem pufferelt") kimeneti adatfolyam, amely rendszerint hibajelzésre használatos (§21.2.1).

Azért adunk vissza értéket, mert a hibák jellemzően valamilyen kifejezés kiértékelése közben történnek, így vagy teljesen abba kellene hagynunk a kiértékelést, vagy olyan értéket kellene visszaadnunk, amely nem valószínű, hogy további hibákat okozna. Ezen egyszerű számológép esetében az utóbbi megoldás megfelelő. Ha a *get_token()* nyomon követte volna a sorok számát, az *error()* tájékoztathatta volna a felhasználót a hiba pontos helyéről, ami akkor lenne hasznos, ha a számológépet nem interaktívan használnánk (§6.6.[19]).

A program futásának gyakran be kell fejeződne, miután hiba történt, mert nincs megadva, milyen ésszerű módon folytathatná működését. Ezt tehetjük meg az *exit()* meghívásával, amely először rendbe rakja az adatfolyamokat és hasonló dolgokat, majd befejezi a programot, melynek visszatérési értéke az *exit()* paramétere lesz (§9.4.1.1).

Kivételek használatával elegánsabb hibakezelő eljárások készíthetők (lásd §8.3 és 14. fejezet), de amit most csináltunk, egy 150 soros számológépnek éppen megfelel.

6.1.5. A vezérlő

Miután a program minden részlete a helyére került, már csak a vezérlő kódra van szükségünk ahhoz, hogy elindítsuk a működést. Ebben az egyszerű példában ezt a *main()* végzi el:

```
int main()
{
  table["pi"] = 3.1415926535897932385;          // előre megadott nevek beillesztése
  table["e"] = 2.7182818284590452354;

  while (cin) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
  }

  return no_of_errors;
}
```

Hagyomány szerint a *main()* *0*-át kell, hogy visszaadjon, ha a program hiba nélkül ér véget, más esetben nem nullát (§3.2). A hibák számának visszaadásával ezt szépen megold-

juk. Ebben az esetben az egyetlen szükséges előkészítés az, hogy a szimbólumtáblába bele kell tennünk az előre megadott neveket. A fő ciklus feladata, hogy beolvassa a kifejezéseket és kiírja a választ. Ezt a következő sor oldja meg:

```
cout << expr(false) << '\n';
```

A *false* paraméter mondja meg az *expr()*-nek, hogy nem kell meghívnia a *get_token()*-t ahhoz, hogy egy újabb szimbólumot kapjon, amellyel dolgozhat.

A *cin* ciklusonként egyszeri ellenőrzése biztosítja, hogy a program befejeződik, ha hiba történik a bemeneti adatfolyammal, az *END* vizsgálata pedig arról gondoskodik, hogy a ciklusból megfelelően lépjünk ki, ha a *get_token()* a fájl végéhez ér. A *break* utasítás a legközelebbi körülvevő *switch* utasításból vagy ciklusból (azaz *for*, *while* vagy *do* utasításból) lép ki. A *PRINT* (azaz *'\n'* és *';'*) vizsgálata megkönnyíti az *expr()* dolgát az üres kifejezések kezelésében. A *continue* utasítás egyenértékű azzal, hogy a ciklus legvégére ugrunk, így ebben az esetben

```
while (cin) {
  // ...
  if (curr_tok == PRINT) continue;
  cout << expr(false) << '\n';
}
```

megegyezik a következővel:

```
while (cin) {
  // ...
  if (curr_tok != PRINT)
          cout << expr(false) << '\n';
}
```

6.1.6. Fejállományok

A számológép a standard könyvtár eszközeit használja. Ezért a megfelelő fejállományokat (header) be kell építenünk (*#include*), hogy befejezzük a programot:

```
#include<iostream>        // bemenet/kimenet
#include<string>          // karakterláncok
#include<map>             // asszociatív tömb
#include<cctype>          // isalpha(), stb.
```

Ezen fejállományok mindegyike az *std* névtérben nyújt szolgáltatásokat, így ahhoz, hogy az általuk nyújtott neveket felhasználhassuk, vagy az *std::* minősítőt kell használnunk, vagy a globális névtérbe kell helyeznünk a neveket a következőképpen:

```
using namespace std;
```

Én az utóbbit választottam, hogy ne keverjem össze a kifejezések tárgyalását a modularitás kérdéskörével. A 8. és a 9. fejezet tárgyalja, milyen módon lehet ezt a számológépet a névterek használatával modulokba szervezni és hogyan lehet forrásfájlokra bontani. A szabványos fejállományoknak számos rendszeren *.h* kiterjesztésű fájl megfelelőjük van, melyek leírják az osztályokat, függvényeket stb. és a globális névtérbe is behelyezik azokat (§9.2.1, §9.2.4, §B.3.1).

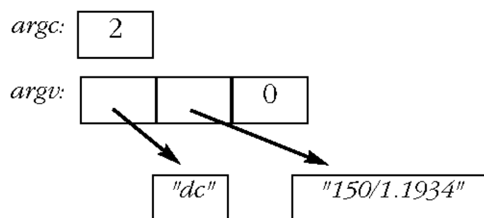
6.1.7. Parancssori paraméterek

Miután a programot megírtam és kipróbáltam, kényelmetlennek találtam, hogy először el kell indítani a programot, aztán be kell gépelni a kifejezéseket, végül ki kell lépni. A leggyakrabban egyetlen kifejezés kiértékelésére használtam a programot. Ha egy kifejezést meg lehetne adni parancssori paraméterként, jónéhány billentyűleütést megtakaríthatnánk.

A program a *main()* (§3.2., §9.4.) meghívásával kezdődik, amely két paramétert kap: az egyik, melyet általában *argc*-nek neveznek, a paraméterek (argumentumok) számát adja meg, a másik a paraméterekből álló tömb, ezt rendszerint *argv*-nek hívják. A paraméterek karakterláncok, ezért *argv* típusa *char*[argc+1]* lesz. A program neve (ahogy az a parancssorban előfordul) *argc[0]*-ként adódik át, így *argc* értéke mindig legalább 1. A paraméterek listáját a null karakter zárja le, így *argv[argc]==0*. Vegyük az alábbi parancsot:

```
dc 150/1.1934
```

Ekkor a paraméterek értéke a következő:



Mivel a *main()* meghívására vonatkozó szabályok a C nyelv követelményeivel azonosak, a híváskor C típusú tömbök és karakterláncok használatosak.

A parancssori paraméterek beolvasása egyszerű, a probléma csak az, hogyan használjuk azokat úgy, hogy minél kevesebbet kelljen programoznunk. Az ötlet a következő: olvassunk ugyanúgy a parancssori karakterláncból, mint a bemeneti adatfolyamokból. A karakterláncból olvasó adatfolyam neve – micsoda meglepetés – *istringstream*. Sajnos nincs elegáns módja annak, hogy *cin*-ként az *istringstream*-re hivatkozhassunk, ezért ki kell találnunk, hogy a számológép bemeneti függvényei hogyan hivatkozzanak vagy az *istringstream*-re vagy a *cin*-re, attól függően, milyen parancssori paramétereket adunk meg. Egyszerű megoldás, ha bevezetünk egy *input* nevű globális mutatót, amely a használandó bemeneti adatfolyamra mutat; minden bemeneti eljárásban ezt fogjuk felhasználni:

```
istream* input;      // mutató bemeneti adatfolyamra

int main(int argc, char* argv[])
{

  switch (argc) {
  case 1:                                   // olvasás a szabványos bemenetről
          input = &cin;
          break;
  case 2:                                   // a karakterlánc paraméter beolvasása
          input = new istringstream(argv[1]);
          break;
  default:
          error("túl sok paraméter");
          return 1;
  }

  table["pi"] = 3.1415926535897932385;     // előre megadott nevek beillesztése
  table["e"] = 2.7182818284590452354;

  while (*input) {
          get_token();
          if (curr_tok == END) break;
          if (curr_tok == PRINT) continue;
          cout << expr(false) << '\n';
  }

  if (input != &cin) delete input;
  return no_of_errors;
}
```

Az *istringstream* olyan *istream*, amely karakterlánc paraméteréből olvas (§21.5.3). Amikor eléri a lánc végét, pontosan ugyanúgy jelzi azt, mint a többi adatfolyam a bemenet végét (§3.6, §21.3.3). Az *istringstream* használatához be kell építeni az *<sstream>* fejállományt.

Könnyű lenne úgy módosítani a *main()*-t, hogy több parancssori paramétert is elfogadjon, de erre nincs szükség, mert egyetlen paraméterként több kifejezést is átadhatunk:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Azért használok idézőjeleket, mert a *;* a UNIX rendszerekben parancs-elválasztóként használatos. Más rendszerek szabályai a program indításakor paraméterek megadására vonatkozóan eltérőek.

Nem volt elegáns dolog úgy módosítani a bemeneti eljárásokat, hogy *cin* helyett **input*-ot használjanak, hogy ezzel rugalmasabbak legyenek és különböző bemeneti forrásokkal működhessenek. A változtatás elkerülhető lett volna, ha kellő előrelátással már a kezdetektől bevezetünk valamilyen, az *input*-hoz hasonló dolgot. Általánosabb és hasznosabb megoldást készíthetünk, ha észrevesszük, hogy a bemenet forrása valójában a számológép modul paramétere kell, hogy legyen. Az alapvető probléma, amit ezzel a számológéppel érzékeltetni akartam, az, hogy a „számológép" csak függvények és adatok gyűjteménye. Nincs olyan modul (§2.4) vagy objektum (§2.5.2), amely kifejezett és egyértelmű módon ábrázolja a számológépet. Ha egy számológép modul vagy számológép típus tervezése lett volna a célom, akkor természetesen meggondoltam volna, milyen paraméterei lehetnek a modulnak/típusnak (§8.5[3], §10.6[16]).

6.1.8. Megjegyzés a stílussal kapcsolatban

A standard könyvtárbeli *map* szimbólumtáblaként való használata majdnem „csalásnak" tűnhet azoknak a programozóknak a szemében, akik nem ismerik az asszociatív tömböket. De nem az. A standard könyvtár és más könyvtárak arra valók, hogy használják azokat. A könyvtárak tervezéskor és megvalósításkor általában nagyobb figyelmet kapnak, mint amennyit egy programozó megengedhet magának, amikor saját kezűleg olyan kódot ír, amit csak egyetlen program használ fel.

Ha megnézzük a számológép kódját (különösen az első változatot), láthatjuk, hogy nem sok hagyományos C stílusú, alacsonyszintű kód található benne. Számos hagyományos trükköt helyettesítettünk azzal, hogy olyan standard könyvtárbeli osztályokat használtunk, mint az *ostream*, *string*, és *map* (§3.4, §3.5, §3.7.4, 17.fejezet).

Vegyük észre, hogy az aritmetika, a ciklusok, sőt az értékadások is viszonylag ritkán fordulnak elő. Általában ilyennek kellene lennie egy olyan kódnak, amely nem kezeli a hardvert közvetlenül és nem él alacsonyszintű elvont adatábrázolásokkal.

6.2. Operátorok – áttekintés

Ez a rész összefoglalja a kifejezéseket és bemutat néhány példát. Minden operátort egy vagy több név követ, amely példaként szolgál az általánosan használt megnevezésekre és a szokásos használatra. A táblázatokban az *osztálynév* egy osztály neve, a *tag* egy tag neve, az *objektum* egy olyan kifejezés, amelynek az eredménye osztályobjektum, a *mutató* egy mutató eredményű kifejezés, a *kif* egy kifejezés, és a *balérték* egy olyan kifejezés, amely nem konstans objektumot jelöl.

A *típus* csak akkor lehet egy teljesen általános típusnév (*-gal, ()-lel stb.), ha zárójelek közé van zárva; máshol megszorítások vonatkoznak rá (§A.5).

A kifejezések formája független az operandusok típusától. Az itt bemutatott jelentések arra az esetre vonatkoznak, amikor az operandusok beépített típusúak (§4.1.1). A felhasználói típusú operandusokra alkalmazott operátorok jelentését magunk határozhatjuk meg (§2.5.2, 11. fejezet).

A táblázat minden cellájában azonos erősségű (precedenciájú) operátorok találhatók. A felsőbb cellákban levő operátorok az alsó cellákban levőkkel szemben előnyt élveznek. Például a+b*c jelentése a+(b*c), nem pedig (a+b)*c, mert a * magasabb precedenciájú, mint a +.

Az egyoperandusú (unáris) és az értékadó operátorok jobbról balra, az összes többi balról jobbra értelmezendő. Például *a=b=c* jelentése *a=(b=c)*, *a+b+c* jelentése *(a+b)+c*, *p++ jelentése pedig *(p++), nem (*p)++.

Néhány nyelvtani szabályt nem lehet kifejezni a precedenciával és az asszociativitással (kötéssel). Például *a=b<c?d=e:f=g* jelentése *a=((b<c)?(d=e):(f=g))*, de ahhoz, hogy ezt eldönthessük, meg kell néznünk a nyelvtant (§A.5).

Operátor – áttekintés	
hatókör-feloldás	*osztálynév :: tag*
hatókör-feloldás	*névtér_név :: tag*
globális hatókör	*:: név*
globális hatókör	*:: minősített_név*
tagkiválasztás	*objektum . tag*
tagkiválasztás	*mutató -> tag*
indexelés	*mutató [kif]*
függvényhívás	*kif (kif_lista)*
érték létrehozása	*típus (kif_lista)*
növelés utótaggal	*balérték* ++
csökkentés utótaggal	*balérték* --
típusazonosítás	typeid *(típus)*
futási idejű típusazonosítás	typeid *(kif)*
futási időben ellenőrzött típuskényszerítés	dynamic_cast *<típus> (kif)*
fordítási időben ellenőrzött típuskényszerítés	static_cast *<típus> (kif)*
nem ellenőrzött típuskényszerítés	reinterpret_cast *<típus> (kif)*
konstans típuskényszerítés	const_cast *<típus> (kif)*
objektum mérete	sizeof *kif*
típus mérete	sizeof *(típus)*
növelés előtaggal	++ *balérték*
csökkentés előtaggal	-- *balérték*
komplemensképzés	~ *kif*
(logikai) nem	! *kif*
mínusz előjel	- *kif*
plusz előjel	+ *kif*
cím operátor	& *balérték*
indirekció	* *kif*
létrehozás (memóriafoglalás)	new *típus*
létrehozás (memóriafoglalás és kezdeti értékadás)	new *(kif_lista)*
létrehozás (elhelyezés)	new *(kif_lista) típus*
létrehozás (elhelyezés és kezdeti értékadás)	new *(kif_lista) típus (kif_lista)*
felszámolás (felszabadítás)	delete *mutató*
tömb felszámolása	delete *[ ] mutató*
típuskonverzió	*(típus) kif*

Operátor – áttekintés (folytatás)	
tagkiválasztás	*objektum .*tagra_hivatkozó_mutató*
tagkiválasztás	*mutató -> *tagra_hivatkozó_mutató*
szorzás	*kif * kif*
osztás	*kif / kif*
moduló (maradékképzés)	*kif % kif*
összeadás (plusz)	*kif + kif*
kivonás (mínusz)	*kif - kif*
balra léptetés	*kif << kif*
jobbra léptetés	*kif >> kif*
kisebb	*kif < kif*
kisebb vagy egyenlő	*kif <= kif*
nagyobb	*kif > kif*
nagyobb vagy egyenlő	*kif >= kif*
egyenlő	*kif == kif*
nem egyenlő	*kif != kif*
bitenkénti ÉS	*kif & kif*
bitenkénti kizáró VAGY	*kif ^ kif*
bitenkénti megengedő VAGY	*kif | kif*
logikai ÉS	*kif && kif*
logikai megengedő VAGY	*kif || kif*
feltételes kifejezés	*kif ? kif : kif*

Operátor – áttekintés (folytatás)	
egyszerű értékadás	*balérték* = *kif*
szorzás és értékadás	*balérték* *= *kif*
osztás és értékadás	*balérték* /= *kif*
maradékképzés és értékadás	*balérték* %= *kif*
összeadás és értékadás	*balérték* += *kif*
kivonás és értékadás	*balérték* -= *kif*
balra léptetés és értékadás	*balérték* <<= *kif*
jobbra léptetés és értékadás	*balérték* >>= *kif*
ÉS és értékadás	*balérték* &= *kif*
megengedő VAGY és értékadás	*balérték* |= *kif*
kizáró VAGY és értékadás	*balérték* ^= *kif*
kivétel kiváltása	`throw` *kif*
vessző (műveletsor)	*kif* , *kif*

6.2.1. Eredmények

Az aritmetikai műveletek eredményének típusát az a szabályhalmaz dönti el, amelyet „általános aritmetikai átalakítások”-nak nevezünk (§C.6.3). A fő cél az, hogy a „legtágabb” operandustípussal megegyező eredmény jöjjön létre. Ha egy bináris operátor operandusa például lebegőpontos, a számítást lebegőpontos aritmetikával végezzük és az eredmény egy lebegőpontos érték lesz. Ha *long* típusú operandusa van, a számítás hosszú egész (*long*) aritmetikával történik, az eredmény pedig *long* érték lesz. Az *int*-nél kisebb operandusok (mint a *bool* és a *char*) *int*-té alakulnak, mielőtt az operátort alkalmazzuk rájuk.

Az ==, <= stb. relációs (összehasonlító) operátorok logikai értékeket adnak vissza. A felhasználó által megadott operátorok jelentését és eredményét deklarációjuk határozza meg (§11.2).

Ha egy operátornak balérték operandusa van, akkor – ha ez logikailag lehetséges – az operátor eredménye egy olyan balérték lesz, amely a balérték operandust jelöli:

```
void f(int x, int y)
{
    int j = x = y;              // x=y értéke az x értékadás utáni értéke
    int* p = &++x;              // p x-re mutat
    int* q = &(x++);            // hiba: x++ nem balérték
    int* pp = &(x>y?x:y);       // a nagyobb értékű int címe
}
```

Ha a *? :* második és harmadik operandusa is balérték és ugyanolyan típusúak, az eredmény a megfelelő típusú balérték lesz. Az, hogy ilyen módon megőrizzük a balértékeket, nagy rugalmasságot ad az operátorok használatában. Ez különösen akkor fontos, ha olyan kódot írunk, amelynek egyformán és hatékonyan kell működnie beépített és felhasználói típusok esetében is (például ha olyan sablonokat vagy programokat írunk, amelyek C++ kódot hoznak létre).

A *sizeof* eredménye a *size_t* nevű előjel nélküli integrális típus, melynek meghatározása a *<cstddef>* fejállományban szerepel, a mutató-kivonásé pedig egy előjeles integrális típus, amit *ptrdiff_t*-nek hívnak és szintén a *<cstddef>* fejállomány írja le.

A fordítónak nem kell ellenőriznie az aritmetikai túlcsordulást és általában nem is teszi meg. Például:

```
void f()
{
    int i = 1;
    while (0 < i) i++;
    cout << "Az i negatív lett!" << i << '\n';
}
```

A ciklus előbb-utóbb az *i* értékét a legnagyobb egész értéken túl növeli. Ami ekkor történik, nem meghatározott; az érték jellemzően egy negatív számig „ér körbe" (az én gépemen ez *-2147483648*). Hasonlóan, a nullával osztás eredménye sem meghatározott, ez viszont rendszerint a program hirtelen befejeződését eredményezi. Az alulcsordulás, a túlcsordulás és a nullával való osztás nem vált ki szabványos kivételeket (§14.10).

6.2.2. Kiértékelési sorrend

A kifejezéseken belüli részkifejezések kiértékelési sorrendje nem meghatározott, így nem tételezhetjük fel például azt sem, hogy a kifejezés kiértékelése balról jobbra történik:

```
int x = f(2)+g(3);    // nem meghatározott, hogy f() vagy g() hívódik meg először
```

Jobb kódot készíthetünk, ha a kifejezések kiértékelési sorrendje nem kötött, de a kiértékelési sorrendre vonatkozó megszorítások hiánya előre nem meghatározott eredményekhez vezethet:

```
int i = 1;
v[i] = i++;  // nem meghatározott eredmény
```

A fenti kifejezés vagy *v[1]=1*-ként, vagy *v[2]=1*-ként értékelődik ki, esetleg még furcsábban viselkedik. A fordítóprogramok figyelmeztethetnek az ilyen kétértelműségekre, sajnos, a legtöbb ezt nem teszi meg.

A , (vessző), a && (logikai ÉS), és a || (logikai VAGY) operátorok esetében biztosított, hogy a bal oldali operandus a jobb oldali előtt értékelődik ki. A *b=(a=2,a+1)* például a *b*-nek *3*-at ad értékül. A || és a && használatára vonatkozó példák a §6.2.3-ban találhatók. Beépített típusoknál a && második operandusa csak akkor értékelődik ki, ha az első operandus *true*, a || második operandusa pedig csak akkor, ha az első operandus értéke *false*; ezt néha rövid vagy „rövidzáras" kiértékelésnek (short-circuit evaluation) nevezik. Jegyezzük meg, hogy a , (vessző) műveletsor-jelző logikailag különbözik attól a vesszőtől, amit arra használunk, hogy a függvényhívásoknál elválasszuk a paramétereket. Nézzük az alábbi példát:

```
f1(v[i],i++);          // két paraméter
f2( (v[i],i++) );      // egy paraméter
```

Az *f1* meghívásának két paramétere van, *v[i]* és *i++*, a paraméter-kifejezések kiértékelési sorrendje pedig nem meghatározott. Az olyan megoldás, amely függ a paraméter-kifejezések sorrendjétől, nagyon rossz stílusról árulkodik és eredménye nem meghatározott. Az *f2* meghívásához egy paramétert adtunk meg; a *(v[i], i++)* „vesszős" kifejezés, amely *i++*-szal egyenértékű.

A csoportosítás kikényszerítésére zárójeleket használhatunk. Például *a*b/c* jelentése *(a*b)/c*, ezért zárójeleket kell használnunk, ha *a*(b/c)*-t akarunk kapni. Az *a*(b/c)* kifejezés csak akkor értékelődhet ki *(a*b)/c*-ként, ha a felhasználó nem tud különbséget tenni köztük. Az *a*(b/c)* és az *(a*b)/c* számos lebegőpontos számításnál jelentősen különbözik, így a fordítóprogram pontosan úgy fogja az ilyen kifejezéseket kiértékelni, ahogy azokat leírtuk.

6.2.3. Az operátorok sorrendje

A precedencia és a „kötési" (asszociativitási) szabályok a leggyakoribb használatot tükrözik. Például

```
if (i<=0 || max<i) // ...
```

azt jelenti, hogy „ha *i* kisebb vagy egyenlő *0*-nál VAGY *max* kisebb *i*-nél". Ez egyenértékű az alábbival:

```
if ( (i<=0) || (max<i) ) // ...
```

Az alábbi – értelmetlen, de szabályos – kifejezéssel viszont nem:

```
if (i <= (0||max) < i) // ...
```

Zárójeleket használni azonban mindig hasznos, ha a programozónak kétségei vannak ezekkel a szabályokkal kapcsolatban. A zárójelek használata még gyakoribb, ha a részkifejezések bonyolultabbak. A bonyolult részkifejezések mindig hiba forrásai lehetnek, ezért – ha úgy érezzük, hogy szükségünk van zárójelekre – fontoljuk meg, hogy nem kellene-e egy külön változó használatával szétbontanunk a kifejezést. Vannak olyan esetek, amikor az operátorok sorrendje nem a „magától értetődő" értelmezést eredményezi:

```
if (i&mask == 0)     // hoppá! == kifejezés & operandusaként
```

Ekkor nem az történik, hogy alkalmazzuk a *mask*-ot az *i*-re, majd megnézzük, hogy az eredmény *0*-e. Mivel az == előnyt élvez az & (kétoperandusú) művelettel szemben, a kifejezés *i&(mask==0)*-ként lesz értelmezve. Szerencsére a fordítóprogram könnyen figyelmeztethet az ilyen hibákra. Ebben az esetben a zárójelek fontosak:

```
if ((i&mask) == 0) // ...
```

Érdemes megjegyezni, hogy a következő nem úgy működik, ahogy egy matematikus elvárná:

```
if (0 <= x <= 99) // ...
```

Ez megengedett, de értelmezése *(0<=x)<=99*, ahol az első összehasonlítás eredménye vagy *true* vagy *false*. A logikai értéket a fordítóprogram aztán automatikusan *1*-re vagy *0*-ra alakítja, amit aztán összehasonlítva *99*-cel *true*-t kapunk. A következőképpen vizsgálhatjuk meg, hogy *x* a *0..99* tartományban van-e:

```
if (0<=x && x<=99) // ...
```

Gyakori hiba kezdőknél, hogy a feltételekben =-t (értékadást) használnak == (egyenlő) helyett:

```
if (a = 7)   // hoppá! konstans értékadás a feltételben
```

Ez természetes, mert az = jelentése sok nyelvben „egyenlő". A fordítóprogramok általában figyelmeztetnek is erre.

6.2.4. Bitenkénti logikai műveletek

Az &, |, ^, ~, >> és << bitenkénti logikai operátorokat integrális (egész típusú) objektumokra és felsorolásokra alkalmazzuk – azaz a *bool*, *char*, *short*, *int*, *long* típusokra, ezek előjel nélküli (*unsigned*) megfelelőire és az *enum* típusokra. Az eredmény típusát a szokásos aritmetikai átalakítások (§C.6.3.) döntik el.

A bitenkénti logikai operátorok jellemző felhasználása a kis halmazok (bitvektorok) fogalmának megvalósítása. Ebben az esetben egy előjel nélküli egész minden bitje a halmaz egy elemét jelöli, és a bitek száma korlátozza a halmaz elemeinek számát. Az & bináris operátort metszetként, a | operátort unióként, a ^-ot szimmetrikus differenciaként, a ~-t pedig komplemensként értelmezzük. Felsoroló típust arra használhatunk, hogy megnevezzük egy ilyen halmaz elemeit. Íme egy rövid példa, melyet az *ostream* megvalósításából vettünk kölcsön:

```
enum ios_base::iostate {
  goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

Az adatfolyam az állapotot így állíthatja be és ellenőrizheti:

```
state = goodbit;
// ...
if (state&(badbit|failbit))      // nem megfelelő adatfolyam
```

A külön zárójelek azért szükségesek, mert az *&* előnyt élvez a *|* műveleti jellel szemben.

Egy függvény így jelezheti, hogy elérte a bemenet végét:

```
state |= eofbit;
```

A *|=* operátort arra használjuk, hogy az állapothoz hozzáadjunk valamilyen új információt. Az egyszerű *state=eofbit* értékadás kitörölt volna minden más bitet.

Ezek az adatfolyam-állapotjelzők megfigyelhetők a folyam megvalósításán kívül is. Például így nézhetjük meg, hogyan különbözik két adatfolyam állapota:

```
int diff = cin.rdstate()^cout.rdstate();          // rdstate() az állapotot adja vissza
```

Az adatfolyam-állapotok különbségeinek kiszámítása nem túl gyakori, más hasonló típusoknál viszont alapvető művelet. Vegyük például azt az esetet, amikor össze kell hasonlítanunk azt a bitvektort, amely a kezelt megszakítások halmazát jelöli, egy másik bitvektorral, amely olyan megszakítások halmazát ábrázolja, melyek arra várnak, hogy kezeljék őket.

Jegyezzük meg, hogy ezt a „zsonglőrködést" a bitekkel az *iostream* megvalósításából vettük és nem a felhasználói felületből. A kényelmes bitkezelés nagyon fontos lehet, de a megbízhatóság, a módosíthatóság, vagy a hordozhatóság érdekében a rendszer alacsonyabb szintjein kell tartanunk. Általánosabb halmazfogalomra nézzük meg a standard könyvtárbeli *set*-et (§17.4.3), *bitset*-et (§17.5.3), és a *vector<bool>*-t (§16.3.11).

A mezők (§C.8.1) használata igazán kényelmes módon rövidíti le azt a műveletet, amikor léptetéssel és maszkolással veszünk ki bitmezőket egy szóból. Ezt természetesen megtehetjük a bitenkénti logikai operátorokkal is. Egy 32 bites *long* középső 16 bitjét például így vehetjük ki:

```
unsigned short middle(long a) { return (a>>8)&0xffff; }
```

A bitenkénti logikai operátorokat ne keverjük össze az *&&*, *||* és *!* logikai operátorokkal. Az utóbbiak vagy *true*-t, vagy *false*-t adnak vissza, és elsődlegesen akkor hasznosak, amikor egy

if, *while*, vagy *for* utasításban (§6.3.2, §6.3.3) feltételt írunk. Például az *!0* (nem nulla) *true* érték, míg a *~0* (a nulla komplemense) egy csupa egyesből álló bitminta, amely a *-1* érték kettes komplemensbeli ábrázolása.

6.2.5. Növelés és csökkentés

A ++ operátort arra használjuk, hogy egy érték növelését közvetlenül, és nem az összeadás és értékadás párosításával fejezzük ki. Definíció szerint a *++lvalue* jelentése *lvalue+=1*, ez pedig *lvalue=lvalue+1*-et jelent, feltéve, hogy a balértéknek nincs „mellékhatása". A növelendő objektumot jelölő kifejezés (csak) egyszer értékelődik ki. A csökkentést ugyanígy a -- operátor fejezi ki. A ++ és -- operátorokat használhatjuk előtagként és utótagként is. A *++x* értéke az *x* új (megnövelt) értéke lesz, például az *y=++x* egyenértékű az *y=(x+=1)*-gyel. Az *x++* értéke azonban az *x* régi értéke: az *y=x++* egyenértékű az *y=(t=x,x+=1,t)*-vel, ahol *t* egy *x*-szel azonos típusú változó.

A mutatók összeadásához és kivonásához hasonlóan a mutatókra alkalmazott ++ és -- működését azok a tömbelemek határozzák meg, amelyekre a mutató hivatkozik; *p++* a *p*-t a következő tömbelemre állítja (§5.3.1).

A növelő operátorok különösen a ciklusokban használatosak, változók növelésére vagy csökkentésére. Egy nulla végződésű karakterláncot például a következőképpen másolhatunk át:

```
void cpy(char* p, const char* q)
{
  while (*p++ = *q++) ;
}
```

A C-hez hasonlóan a C++-t is szeretik és gyűlölik azért, mert megengedi az ilyen tömör, kifejezésközpontú kódolást. Mivel a

```
while (*p++ = *q++) ;
```

kifejezés meglehetősen zavaros a nem C programozók számára, ez a kódolási stílus viszont nem ritka a C-ben és a C++-ban, megéri közelebbről megvizsgálnunk. Vegyük először a karaktertömbök másolásának egy hagyományosabb módját:

```
int length = strlen(q);
for (int i = 0; i<=length; i++) p[i] = q[i];
```

Ez pazarlás. A nulla végződésű karakterlánc hosszát úgy határozzuk meg, hogy a nulla végződést keresve végigolvassuk azt. Így kétszer olvassuk végig a teljes láncot: egyszer azért, hogy meghatározzuk a hosszát, egyszer pedig azért, hogy átmásoljuk. Ezért inkább próbáljuk ezt:

```
int i;
for (i = 0; q[i]!=0 ; i++) p[i] = q[i];
p[i] = 0;                // lezáró nulla
```

Az *i* változót indexelésre használjuk, de ki lehet küszöbölni, mert *p* és *q* mutatók:

```
while (*q != 0) {
   *p = *q;
   p++;                  // léptetés a következő karakterre
   q++;                  // léptetés a következő karakterre
}
*p = 0;                  // lezáró nulla
```

Mivel az utótagként használt növelő operátor megengedi, hogy először felhasználjuk az értéket, és csak azután növeljük meg, a következőképpen írhatjuk újra a ciklust:

```
while (*q != 0) {
   *p++ = *q++;
}
*p = 0;                  // lezáró nulla
```

A **p++ = *q++* értéke **q*, ezért a példát így módosíthatjuk:

```
while ((*p++ = *q++) != 0) { }
```

Ebben az esetben addig nem vesszük észre, hogy **q* nulla, amíg be nem másoljuk **p*-be és meg nem növeljük *p*-t. Következésképpen elhagyhatjuk az utolsó értékadást, amiben a nulla végződést adjuk értékül. Végül tovább rövidíthetjük a példát azzal, hogy észrevesszük, nincs szükségünk az üres blokkra és hogy felesleges a *„!=0”* vizsgálat, mert egy mutató vagy integrális feltétel mindig összehasonlítódik *0*-val. Így megkapjuk azt a változatot, amelyet célul tűztünk ki.

```
while (*p++ = *q++) ;
```

Ez a változat vajon kevésbé olvasható, mint az előző? Egy tapasztalt C vagy C++ programozó számára nem. Hatékonyabb időben és tárterületben, mint az előző? Az első változatot ki-

véve, ami meghívta az *strlen()*-t, nem igazán. Az, hogy melyik változat a leghatékonyabb, a gép felépítésétől és a fordítóprogramtól függ, a nulla végződésű karakterláncok másolásának leghatékonyabb módja viszont általában a standard könyvtárbeli másoló függvény.

```
char* strcpy(char*, const char*);        // a <string.h> fejállományból
```

Általánosabb másolásra a szabványos *copy* algoritmust (§2.7.2, §18.6.1) használhatjuk. Ahol lehetséges, részesítsük előnyben a standard könyvtár lehetőségeit a mutatókkal és bájtokkal való ügyeskedéssel szemben. A standard könyvtár függvényei lehetnek helyben kifejtett függvények (§7.1.1) vagy egyedi gépi utasításokkal megvalósítottak. Ezért gondosan fontoljuk meg, mielőtt elhinnénk, hogy valamilyen kézzel írt kódrészlet felülmúlja a könyvtári függvények teljesítményét.

6.2.6. Szabad tár

A névvel rendelkező objektumok élettartamát (lifetime) hatókörük (§4.9.4) dönti el, gyakran azonban hasznos, ha olyan objektumot hozunk létre, amely függetlenül létezik attól a hatókörtől, ahol létrehoztuk. Nevezetesen gyakori, hogy olyan objektumokat hozunk létre, amelyek akkor is felhasználhatók, miután visszatértünk abból a függvényből, ahol létrehoztuk azokat. Az ilyen objektumokat a *new* operátor hozza létre és a *delete* operátort használhatjuk felszámolásukra. A *new* által létrehozott objektumokra azt mondjuk, hogy „a szabad tárban vannak" (free store), vagy azt, hogy „kupac-objektumok" (heap), vagyis „a dinamikus memóriában vannak".

Nézzük meg, hogyan írnánk meg egy fordítóprogramot olyan stílusban, ahogy az asztali számológépnél tettük (§6.1). A szintaktikai elemző függvények felépíthetnek egy kifejezésfát a kódkészítő számára:

```
struct Enode {
  Token_value oper;
  Enode* left;
  Enode* right;
  // ...
};

Enode* expr(bool get)
{
  Enode* left = term(get);
```

```
    for (;;)
        switch(curr_tok) {
        case PLUS:
        case MINUS:
        {       Enode* n = new Enode;       // Enode létrehozása a szabad tárban
                n->oper = curr_tok;
                n->left = left;
                n->right = term(true);
                left = n;
                break;
        }
        default:
                return left;                // csomópont visszaadása
        }
}
```

A kódkészítő aztán felhasználná az eredményül kapott csomópontokat (node) és törölné azokat:

```
void generate(Enode* n)
{
  switch (n->oper) {
  case PLUS:
        // ...
        delete n;        // Enode törlése a szabad tárból
  }
}
```

A *new* által létrehozott objektum addig létezik, amíg kifejezetten meg nem semmisítjük a *delete*-tel. Ezután a *new* újra felhasználhatja az objektum által lefoglalt tárhelyet. A C++-változatok nem garantálják, hogy van „szemétgyűjtő” (garbage collector), amely megkeresi azokat az objektumokat, amelyekre nincs már hivatkozás és újra felhasználhatóvá teszi azok helyét. Következésképpen feltételezzük, hogy a *new* által létrehozott objektumokat magunknak kell megsemmisítenünk, a *delete*-et használva. Ha van szemétgyűjtő, a *delete*-ek a legtöbb esetben elhagyhatók (§C.9.1).

A *delete* operátort csak a *new* által visszaadott mutatóra vagy nullára lehet alkalmazni. Ha a *delete*-et nullára alkalmazzuk, nem lesz hatása.

A *new* operátornak egyedi változatait is meghatározhatjuk (§15.6).

6.2.6.1. Tömbök

A *new* használatával létrehozhatunk objektumokból álló tömböt is:

```
char* save_string(const char* p)
{
  char* s = new char[strlen(p)+1];
  strcpy(s,p);                    // másolás p-ből s-be
  return s;
}

int main(int argc, char* argv[ ])
{
  if (argc < 2) exit(1);
  char* p = save_string(argv[1]);
  // ...
  delete[ ] p;
}
```

A „sima" *delete* operátort arra használhatjuk, hogy egyes objektumokat felszámoljuk, a *delete[ ]* tömbök felszámolására használatos.

Ha vissza akarjuk nyerni a *new* által lefoglalt tárhelyet, a *delete*-nek vagy a *delete[ ]*-nek meg kell tudni állapítani, mekkora a lefoglalt objektum mérete. Ebből az következik, hogy a szabványos *new* operátorral létrehozott objektumok valamivel több helyet foglalnak, mint a statikus objektumok. Az objektum méretét általában egy gépi szó tárolja.

Jegyezzük meg, hogy a *vector* (§3.7.1, §16.3) valódi objektum, ezért létrehozására és felszámolására a sima *new*-t és *delete*-et használhatjuk:

```
void f(int n)
{
  vector<int>* p = new vector<int>(n);          // önálló objektum
  int* q = new int[n];                          // tömb
  // ...
  delete p;
  delete[ ] q;
}
```

A *delete[ ]* operátort csak a *new* által visszaadott mutatóra vagy nullára alkalmazhatjuk. Ha a *delete[ ]*-et nullára alkalmazzuk, nem lesz hatása.

6.2.6.2. Memória-kimerülés

A szabad tár *new*, *delete*, *new[ ]* és *delete[ ]* operátorai függvényekként vannak megvalósítva:

```
void* operator new(size_t);          // hely az önálló objektum számára
void operator delete(void*);

void* operator new[](size_t);        // hely a tömb számára
void operator delete[](void*);
```

Ha a *new* operátornak egy objektum számára kell helyet foglalnia, az *operator new()*-t hívja meg, hogy az megfelelő számú bájtot foglaljon le. Ha tömb számára foglal helyet, az *operator new[ ]()* meghívására kerül sor. Az *operator new()* és az *operator new [ ]()* szabványos megvalósítása a visszaadott memóriát nem tölti fel kezdőértékkel.

Mi történik, ha a *new* nem talál lefoglalható helyet? Alapértelmezés szerint a lefoglaló *bad_alloc* kivételt vált ki (a másik lehetőséget illetően lásd §19.4.5-öt):

```
void f()
{
  try {
        for(;;) new char[10000];
  }
  catch(bad_alloc) {
        cerr << "Elfogyott a memória!\n";
  }
}
```

Akármennyi memória áll a rendelkezésünkre, a kód végül meg fogja hívni a *bad_alloc* eseménykezelőjét.

Magunk is meghatározhatjuk, mit csináljon a *new*, amikor kifogy a memória. Ha a *new* nem jár sikerrel, először azt a függvényt hívja meg, amelyet a *<new>* fejállományban bevezetett *set_new_handler()* függvénnyel előzőleg beállítottunk (amennyiben ezt megtettük):

```
void out_of_store()
{
  cerr << "Az operator new nem járt sikerrel: nincs tárhely\n";
  throw bad_alloc();
}
```

```
int main()
{
  set_new_handler(out_of_store);        // out_of_store lesz a new_handler
  for (;;) new char[10000];
  cout << "kész\n";
}
```

Ez azonban soha nem fog elérni addig, hogy kiírja a *„kész"*-t. Ehelyett a következőt fogja kiírni:

Az operator new nem járt sikerrel: nincs tárhely

Lásd §14.4.5-öt az *operator new()* egy olyan lehetséges megvalósításáról, amely megvizsgálja, létezik-e meghívható kezelőfüggvény, és ha nem talál ilyet, *bad_alloc*-ot vált ki. Egy *new_handler* azonban valami okosabbat is tehet, mint hogy egyszerűen befejezi a programot. Ha tudjuk, hogyan működik a *new* és a *delete* – például azért, mert saját *operator new()*-t és *operator delete()*-et írtunk –,a kezelőfüggvény megpróbálhat valamennyi memóriát keresni, hogy a *new* visszatérhessen, vagyis a felhasználó gondoskodhat szemétgyűjtőről, így elhagyhatóvá teheti a *delete*-et (bár ez kétségtelenül nem kezdőknek való feladat). Majdnem mindenkinek, akinek automatikus szemétgyűjtőre van szüksége, a leghasznosabb, ha szerez egy már megírt és ellenőrzött terméket (§C.9.1).

Azzal, hogy új *new_handler*-t állítunk be, felvállaljuk, hogy nekünk kell törődnünk a memória kimerülésével kapcsolatos problémákkal a *new* minden használatakor. A memóriafoglalásnak kétféle útja létezik: vagy gondoskodunk nem szabványos lefoglaló és felszabadító függvényekről (§15.6) a *new* szabályos használata számára, vagy a felhasználó által adott további foglalási adatokra támaszkodunk (§10.4.11, §19.4.5.).

6.2.7. Meghatározott típuskonverziók

Néha „nyers memóriával" kell dolgoznunk, azaz a tár olyan objektumot tartalmaz vagy fog tartalmazni, melynek típusa ismeretlen a fordítóprogram számára. Ilyen eset, amikor a memóriafoglaló (allokátor) egy újonnan lefoglalt memóriaterületre hivatkozó *void** típusú mutatót ad vissza, vagy ha azt akarjuk kifejezni, hogy egy adott egész értéket úgy kell kezelni, mint egy I/O eszköz címét:

```
void* malloc(size_t);

void f()
{
  int* p = static_cast<int*>(malloc(100)); // a new által lefoglalt helyet int-ként használjuk
  IO_device* d1 = reinterpret_cast<IO_device*>(0Xff00);  // eszköz a 0Xff00 címen
  // ...
}
```

A fordítóprogram nem ismeri a *void** által mutatott objektum típusát. Azt sem tudja, vajon a *0Xff00* érvényes cím-e. Következésképpen az átalakítások helyessége teljes mértékben a programozó kezében van. Az explicit (pontosan meghatározott) típuskényszerítések (*casting*) néha szükségesek, de hagyományosan túl sokszor használják azokat, és jelentős hibaforrások.

A *static_cast* operátor egymással kapcsolatban levő típusok közötti konverziót végez, például két, ugyanazon osztályhierarchiában lévő mutatótípus, integrális típus és felsoroló típus, vagy lebegőpontos típus és integrális típus közöttit. A *reinterpret_class* olyan típusok átalakítását hajtja végre, amelyek nincsenek kapcsolatban, például egészről mutatóra vagy mutatótípusról egy másik, nem rokon mutatóra konvertál. Ez a megkülönböztetés lehetővé teszi, hogy a fordítóprogram elvégezzen bizonyos minimális típusellenőrzést a *static_cast* esetében, és megkönnyíti, hogy a programozó megtalálja a veszélyesebb átalakításokat, melyeket a *reinterpret_cast* jelöl. Néhány *static_cast* „hordozható", a *reinterpret_cast*-ok közül viszont csak kevés. A *reinterpret_cast* esetében nem sok dolog biztos; általában új típust hoz létre, amelynek ugyanaz a bitmintája, mint a paraméteréé. Ha a cél legalább annyi bites, mint az eredeti érték, az eredményt a *reinterpret_cast*-tal az eredeti típusra alakíthatjuk és használhatjuk azt. A *reinterpret_cast* eredményét csak akkor lehet biztosan felhasználni, ha annak típusa pontosan az a típus, amelyet az érték meghatározására használtunk.

Ha kísértést érzünk, hogy pontosan meghatározott típuskényszerítést alkalmazzunk, szánjunk időt arra, hogy meggondoljuk, vajon tényleg szükséges-e. A C++-ban az explicit típuskényszerítés a legtöbb esetben szükségtelen olyankor, amikor a C-ben szükség lenne rá (§1.6), és sok olyan esetben is, ahol a C++ korai változataiban szükséges volt (§1.6.2, §B.2.3). Számos programban az ilyen típuskonverzió teljesen elkerülhető; máshol néhány eljárásra korlátozhatjuk a használatát. Ebben a könyvben explicit típuskényszerítést valósághű helyzetekben csak a §6.2.7, §7.7, §13.5, §15.4, és §25.4.1 pontokban használunk.

A futási időben ellenőrzött konverziók egyik formája a *dynamic_cast* (§15.4.1). A *const* minősítőt eltávolító „konstanstalanító" *const_cast* (§15.4.2.1) operátort szintén használhatjuk.

A C++ a C-ből örökölte a *(T)e* jelölést, amely bármilyen átalakítást elvégez, amit ki lehet fejezni a *static_cast*, *reinterpret_cast* és a *const_cast* kombinációjaként. Eredményül *T* típusú érték jön létre (§B.2.3). Ez a C stílusú konverzió sokkal veszélyesebb, mint a fent említettek, mert a jelölést nehezebben lehet észrevenni egy nagy programban és a programozó szándéka szerinti átalakítás fajtája nem nyilvánvaló. Azaz a *(T)e* lehet, hogy „hordozható" átalakítást végez egymással kapcsolatban levő típusok között, de nem hordozhatót a nem rokon típusok között, esetleg egy mutatótípusról eltávolítja a *const* minősítőt. Ha nem tudjuk *T* és *e* pontos típusát, ezt nem tudjuk eldönteni.

6.2.8. Konstruktorok

Egy *T* típusú érték létrehozása egy *e* értékből a *T(e)* függvényjelöléssel fejezhető ki:

```
void f(double d)
{
  int i = int(d);                    // d csonkolása
  complex z = complex(d);            // complex létrehozása d-ből
  // ...
}
```

A *T(e)* szerkezetet néha függvény stílusú konverziónak nevezik. Sajnos, beépített *T* típusokra *T(e)* egyenértékű *(T)e*-vel, ami azt vonja maga után, hogy a *T(e)* használata nem mindig biztonságos. Aritmetikai típusok esetében az értékek csonkulhatnak, és még egy hosszabb egész típusról egy rövidebbre (például *long*-ról *char*-ra) való átalakítás is nem meghatározott viselkedést eredményezhet. A jelölést megpróbálom kizárólag ott használni, ahol az érték létrehozása pontosan meghatározott, azaz a szűkítő aritmetikai átalakításoknál (§C.6), az egészekről felsoroló típusra való átalakításoknál (§4.8), és a felhasználói típusok objektumainak létrehozásánál (§2.5.2, §10.2.3).

A mutató-konverziókat a *T(e)* jelölést használva nem fejezhetjük ki közvetlenül. A *char*(2)* például formai hibának számít. Sajnos az a védelem, amit a konstruktor jelölés nyújt az ilyen veszélyes átalakítások ellen, kikerülhető ha a mutatótípusokra *typedef* neveket (§4.9.7) használunk.

A *T* alapértelmezett értékének kifejezésére a *T()* konstruktor jelölés használatos:

```
void f(double d)
{
  int j = int();                     // alapértelmezett int érték
  complex z = complex();             // alapértelmezett complex érték
  // ...
}
```

A beépített típusok konstruktorának értéke a *0*, amit a fordító az adott típusra konvertál (§4.9.5). Ezért az *int()* egy másfajta módja a *0* írásának. A *T* felhasználói típusra *T()*-t az alapértelmezett konstruktor (§10.4.2) határozza meg, ha létezik ilyen.

A konstruktor jelölés használata beépített típusokra sablonok írásakor különösen fontos. Ekkor a programozó nem tudhatja, hogy a sablon (template) paramétere beépített vagy felhasználói típusra vonatkozik-e majd (§16.3.4, §17.4.1.2).

6.3. Utasítások – áttekintés

Íme a C++ utasítások összefoglalása, néhány példával:

Az utasítások formai szabályai

utasítás:
  deklaráció
  { utasítás_lista$_{nem\ kötelező}$*}*
  `try` *{ utasítás_lista*$_{nem\ kötelező}$*} kezelő_lista*
  kif$_{nem\ kötelező}$*;*
  `if` *(feltétel) utasítás*
  `if` *(feltétel) utasítás* `else` *utasítás*
  `switch` *(feltétel) utasítás*
  `while` *(feltétel) utasítás*
  `do` *utasítás* `while` *(kif);*
  `for` *(kezdőérték_meghatározó feltétel*$_{nem\ kötelező}$*;kif*$_{nem\ kötelező}$*) utasítás*
  `case` *konstans_kif : utasítás*
  `default` *: utasítás*
  `break` *;*
  `continue` *;*
  `return` *kif*$_{nem\ kötelező}$*;*
  `goto` *azonosító;*
  azonosító : utasítás

utasítás_lista:
  utasítás utasítás_lista$_{nem\ kötelező}$

feltétel:
  kif
  típusazonosító deklarátor = kif

kezelő_lista:
  `catch` *(kif_deklaráció) { utasítás_lista*$_{nem\ kötelező}$*}*
  kezelő_lista kezelő_lista$_{nem\ kötelező}$

Jegyezzük meg, hogy a deklaráció egy utasítás, értékadó és eljáráshívó utasítások pedig nincsenek: az értékadások és a függvényhívások kifejezések. A kivételek kezelésére vonatkozó utasításokat – a *try* blokkokat – a §8.3.1 pontban tárgyaljuk.

6.3.1. Deklarációk mint utasítások

A deklaráció utasítás. Hacsak egy változót *static*-ként nem adunk meg, minden esetben kezdőértéket fog kapni, amikor a vezérlés áthalad a deklarációján (lásd még §10.4.8). A deklarációkat azért engedjük meg minden olyan helyen, ahol utasítást használhatunk (és még pár további helyen, §6.3.2.1, §6.3.3.1), hogy lehetővé tegyük a programozónak a kezdőérték nélküli változókból származó hibák csökkentését és a változók hatókörének lehető legnagyobb szűkítését a kódban. Ritkán van ok új változó bevezetésére, mielőtt lenne egy olyan érték, amit a változónak tartalmaznia kell:

```
void f(vector<string>& v, int i, const char* p)
{
  if (p==0) return;
  if (i<0 || v.size()<=i) error("rossz index");
  string s = v[i];
  if (s == p) {
            // ...
  }
  // ...
}
```

A lehetőség, hogy a deklarációkat végrehajtható kód után is elhelyezhetjük, alapvető fontosságú sok konstans esetében, illetve az olyan egyszeri értékadásos programozási stílusnál, ahol egy objektum értéke nem változik meg annak létrehozása és kezdeti értékadása után. Felhasználói típusoknál a változó meghatározásának elhalasztása addig, amíg egy megfelelő kezdeti érték rendelkezésre nem áll jobb teljesítményhez is vezethet:

```
string s; /* ... */ s = "A legjobb a jó ellensége.";
```

A fenti könnyen előfordulhat, hogy sokkal lassabb, mint a következő:

```
string s = "Voltaire";
```

Kezdeti érték nélkül általában akkor adunk meg egy változót, ha a változónak utasításra van szüksége a kezdeti értékadáshoz. Ilyenek például a bemeneti változók és a tömbök.

6.3.2. Kiválasztó utasítások

Egy értéket az *if* vagy a *switch* utasítással vizsgálhatunk meg:

```
if (feltétel) utasítás
if (feltétel) utasítás else utasítás
switch (feltétel) utasítás
```

Az alábbi összehasonlító operátorok a logikai (*bool*) típusú *true* értéket adják vissza, ha az összehasonlítás igaz, és *false*-t, ha hamis:

```
==    !=    <    <=    >    >=
```

Az *if* utasításban az első (és egyetlen) utasítás akkor hajtódik végre, ha a kifejezés nem nulla. Ha nulla, a második utasításra ugrunk (ha megadtunk ilyet). Ebből az következik, hogy bármilyen aritmetikai vagy mutató kifejezést lehet feltételként használni. Például ha *x* egy egész, akkor

```
if (x) // ...
```

azt jelenti, hogy

```
if (x != 0) // ...
```

A *p* mutató esetében az alábbi egy közvetlen utasítás, ami azt a vizsgálatot fejezi ki, hogy „*p* egy érvényes objektumra mutat":

```
if (p) // ...
```

A következő közvetett módon ugyanezt a kérdést fogalmazza, úgy, hogy összehasonlítja egy olyan értékkel, amelyről tudjuk, hogy nem mutat objektumra:

```
if (p != 0) // ...
```

Jegyezzük meg, hogy a *0* mutatót nem minden gép ábrázolja „csupa nullával" (§5.1.1). Minden fordítóprogram, amivel találkoztam, ugyanazt a kódot készítette mindkét vizsgálatra.

A

```
&&    ||    !
```

logikai operátorok leggyakrabban feltételekben használatosak. Az *&&* és a || műveletek nem értékelik ki a második paramétert, csak ha szükség van rá:

```
if (p && 1<p->count) // ...
```

A fenti utasítás például először megvizsgálja, hogy *p* nem nulla-e, és csak akkor nézi meg, hogy *1<p->count* teljesül-e, ha *p* nem nulla.

Néhány *if* utasítást kényelmesen feltételes kifejezésekre cserélhetünk. Például az

```
if (a <= b)
  max = b;
else
  max = a;
```

jobban kifejezhető így:

```
max = (a<=b) ? b : a;
```

A feltétel körül lévő zárójelek nem szükségesek, én azonban úgy gondolom, a kód könnyebben olvasható lesz tőlük.

A *switch* utasítás *if* utasítások sorozataként is leírható. Például a

```
switch (val) {
case 1:
  f();
  break;
case 2:
  g();
  break;
default:
  h();
  break;
}
```

így is kifejezhető:

```
if (val == 1)
  f();
else if (val == 2)
  g();
else
  h();
```

A jelentés ugyanaz, de az első (*switch*) változatot részesítjük előnyben, mert a művelet természete (egy értéket állandók halmazával hasonlítunk össze) így világosabb. A *switch* utasítás olvashatóbb olyan példáknál, amelyek nem maguktól értetődőek, és jobb kódot is hozhatunk létre vele.

Vigyázzunk arra, hogy a *switch case*-ét mindig fejezzük be valahogy, hacsak nem akarjuk a végrehajtást a következő *case*-nél folytatni. Vegyük a következőt:

```
switch (val) {                              // vigyázat!
case 1:
  cout << "1. eset\n";
case 2:
  cout << "2.eset\n";
default:
  cout << "Alapértelmezés: nincs ilyen eset\n";
}
```

Ha *val==1*-gyel hívjuk meg, a következőket írja ki:

```
1. eset
2. eset
Alapértelmezés: nincs ilyen eset
```

Ez az avatatlanokat nagy meglepetésként érheti. Jó ötlet, ha megjegyzésekkel látjuk el azon (ritka) eseteket, amikor a *case*-ek közötti továbblépés szándékos, így egy nem magyarázott továbblépésről feltételezhetjük, hogy programhiba. A *case* befejezésének leggyakoribb módja a *break* használata, de a *return* is hasznos lehet (§6.1.1).

6.3.2.1. Deklarációk feltételekben

A véletlen hibás működés elkerülésére általában jó ötlet a változókat a legkisebb lehetséges hatókörben bevezetni. Nevezetesen, rendszerint legjobb elhalasztani egy ideális változó bevezetését addig, amíg kezdeti értéket nem tudunk adni neki. Így nem kerülhetünk olyan helyzetbe, hogy a változót még azelőtt használjuk, mielőtt kezdeti értékét beállítottuk volna.

Az említett két elv egyik legelegánsabb felhasználása, ha a változót egy feltételben adjuk meg. Vegyük a következő példát:

```
if (double d = prim(true)) {
        left /= d;
        break;
}
```

Itt *d* deklarált és kezdőértéket is kap, amit a feltétel értékével hasonlítunk össze. A *d* hatóköre a deklaráció pontjától annak az utasításnak a végéig terjed, amit a feltétel vezérel. Ha volna egy *else* ág az *if* utasításban, a *d* hatóköre mindkét ágra kiterjedne.

A másik hagyományos és kézenfekvő megoldás, ha a *d*-t a feltétel előtt vezetjük be. Így viszont nagyobb lesz a *d* használatának hatóköre; kiterjedhet a kezdeti értékadás elé vagy a *d* szándékolt hasznos élettartama után is:

```
double d;
// ...

d2 = d;   // hoppá!
// ...

if (d = prim(true)) {
        left /= d;
        break;
}
// ...

d = 2.0;              // d két, egymástól független használata
```

A változók feltételekben történő megadásának nemcsak logikai haszna van, tömörebb forráskódot is eredményez.

A feltételben lévő deklarációnak egyetlen változót vagy konstanst kell megadnia és feltöltenie kezdőértékkel.

6.3.3. Ciklusutasítások

A ciklusokat *for*, *while* vagy *do* utasítással fejezhetjük ki:

```
while ( feltétel ) utasítás
do utasítás while ( kifejezés ) ;
for ( kezdőérték_meghatározó feltétel_nem kötelező ; kifejezés_nem kötelező ) utasítás
```

Ezen utasítások mindegyike ismételten végrehajt egy utasítást (amit vezérelt (*controlled*) utasításnak vagy ciklusmagnak nevezünk), amíg a feltétel hamissá nem válik vagy a programozó más módon ki nem lép a ciklusból.

A *for* utasítás szabályos ciklusok kifejezésére való. A ciklusváltozót, a ciklusfeltételt, és a ciklusváltozót módosító kifejezést egyetlen sorban írhatjuk le, ami nagyon megnövelheti az olvashatóságot és ezzel csökkentheti a hibák gyakoriságát. Ha nem szükséges kezdeti értékadás, a *kezdőérték_meghatározó* (inicializáló) utasítás üres is lehet. Ha a feltételt elhagyjuk, a *for* utasítás örökké a ciklusban marad, hacsak a felhasználó kifejezetten kilépésre nem kényszeríti egy *break*, *return*, *goto*, vagy *throw* utasítással, vagy valami kevésbé egyszerű módon, például az *exit()* (§9.4.1.1) meghívásával. Ha a kifejezést elhagyjuk, a ciklusmagban kell módosítanunk egy ciklusváltozót. Ha a ciklus nem az egyszerű „bevezetünk egy ciklusváltozót, megvizsgáljuk a feltételt, módosítjuk a ciklusváltozót" fajtából való, általában jobb, ha *while* utasítással fejezzük ki, de a *for* is segíthet olyan ciklusok írásánál, melyeknek nincs meghatározott leállási feltétele:

```
for(;;) {      // "örökké" (végtelen ciklus)
  // ...
}
```

A *while* utasítás egyszerűen végrehajtja a ciklusmagot, amíg feltétele hamissá nem válik. Akkor hajlok arra, hogy a *while*-t részesítsem előnyben a *for*-ral szemben, amikor nincs magától értetődő ciklusváltozó vagy amikor a ciklusváltozó módosítása természetes módon a ciklusmag közepén történik. A bemeneti ciklus egy olyan ciklusra példa, amelyben nincs magától értetődő ciklusváltozó:

```
while(cin>>ch) // ...
```

Tapasztalatom szerint a *do* utasítás könnyen hibák és tévedések forrása lehet. Ennek az az oka, hogy a ciklusmag mindig végrehajtódik egyszer, mielőtt a feltétel kiértékelődik. Ahhoz azonban, hogy a ciklusmag megfelelően működjön, valamilyen feltételnek már az első alkalommal is teljesülnie kell. A vártnál sokkal gyakrabban vettem észre azt, hogy egy felté-

tel nem úgy teljesült, ahogy az elvárható lett volna; vagy amikor a programot először megírták és tesztelték, vagy később, amikor a kódot módosították. Ezenkívül jobban szeretem a feltételt „elöl, ahol jól láthatom”. Következésképpen én magam próbálom elkerülni a *do* utasításokat.

6.3.3.1. Deklarációk a for utasításban

Változókat a *for* utasítás kezdőérték-adó részében adhatunk meg. Ha ez deklaráció, akkor az általa bevezetett változó (vagy változók) hatóköre a *for* utasítás végéig terjed:

```
void f(int v[ ], int max)
{
  for (int i = 0; i<max; i++) v[i] = i*i;
}
```

Ha az index végső értékét tudni kell a *for* ciklusból való kilépés után, a ciklusváltozót a cikluson kívül kell megadni (pl. §6.3.4.).

6.3.4. Goto

A C++-ban megtalálható a hírhedt *goto*:

```
goto azonosító ;
azonosító : utasítás
```

A *goto* az általános magasszintű programozásban kevés dologra használható, de nagyon hasznos lehet, amikor a C++ kódot program és nem közvetlenül egy személy készíti; használhatjuk például olyan elemzőben, melyet egy kódkészítő program (kódgenerátor) hozott létre valamilyen nyelvtan alapján. A *goto* akkor is hasznos lehet, ha a hatékonyság alapvető követelmény, például valamilyen valós idejű alkalmazás belső ciklusában.

A *goto* kevés értelmes használatának egyike a mindennapi kódban az, hogy kilépünk egy beágyazott ciklusból vagy *switch* utasításból (a *break* csak a legbelső ciklusból vagy *switch* utasításból lép ki):

```
void f()
{
  int i;
  int j;
```

```
  for (i = 0; i<n; i++)
          for (j = 0; j<m; j++) if (nm[i][j] == a) goto found;
  // nem található
  // ...
found:
  // nm[i][j] == a
}
```

A ciklus végére ugró *continue* utasítás működésével a §6.1.5-ben foglalkoztunk.

6.4. Megjegyzések és behúzás

A program olvasását és megértését sokkal kellemesebbé teheti, ha okosan használjuk a megjegyzéseket és a behúzást. Számos behúzási stílus használatos és nem látok alapvető okot arra, hogy egyiket a másikkal szemben előnyben részesítsük (bár a legtöbb programozóhoz hasonlóan nekem is van választott stílusom – a könyv nyilván tükrözi is azt). Ugyanez vonatkozik a megjegyzések stílusára is.

A megjegyzéseket számos módon lehet rosszul használni, ami így nagymértékben rontja a program olvashatóságát. A fordítóprogram nem érti a megjegyzések tartalmát, ezért nincs mód arra, hogy biztosítsa azt, hogy egy megjegyzés

[1] értelmes,
[2] a programmal összhangban álló és
[3] időszerű legyen.

Számos program olyan megjegyzéseket tartalmaz, melyek érthetetlenek, félreérthetőek, vagy egyszerűen hibásak. A rossz megjegyzések rosszabbak, mint ha egyáltalán nem használnánk megjegyzést.

Ha valamit leírhatunk magával a programnyelvvel, akkor tegyük azt, ne megjegyzésben említsük meg. Ez az észrevétel az ilyenfajta megjegyzésekre vonatkozik:

```
// a "v" változónak kezdőértéket kell adni

// a "v" változót csak az "f()" függvény használhatja

// az "init()" függvényt minden más függvény előtt meg kell hívni ebben a fájlban
```

```
// a "cleanup()" függvényt meg kell hívni a program végén

// a "weird()" függvényt ne használjuk

// az "f()" függvénynek két paramétert kell adni
```

A C++ megfelelő használata az ilyen megjegyzéseket általában szükségtelenné teszi. A fentieket például kiválthatjuk, ha alkalmazzuk az összeszerkesztési (§9.2) vagy az osztályokra vonatkozó láthatósági, kezdőérték-adási és felszámolási szabályokat (§10.4.1).

Mihelyt valamit világosan leírtunk a nyelvvel, másodszor már nem kell megemlítenünk egy megjegyzésben:

```
a = b+c;    // a-ból b+c lesz
count++;    // növeljük a számlálót
```

Az ilyen megjegyzések még az egyszerűen feleslegeseknél is rosszabbak, mert növelik az elolvasandó szöveg hosszát, gyakran összezavarják a program szerkezetét, és lehet, hogy hibásak. Meg kell azonban jegyeznünk, hogy az ilyen megjegyzések széleskörűen használatosak tanítási célokra az olyan programozási nyelvekről szóló könyvekben, mint amilyen ez is. Ez az egyik, amiben egy könyvben lévő program különbözik egy igazi programtól.

Én a következőket szeretem:

1. Minden forrásfájlban van megjegyzés, amely leírja, mi a közös a fájlban levő deklarációkban, utal a segédanyagokra, általános ötleteket ad a kód módosításával kapcsolatban stb.
2. Minden osztályhoz, sablonhoz és névtérhez tartozik megjegyzés.
3. Minden nem magától értetődő függvényhez van olyan megjegyzés, amely leírja a függvény célját, a felhasznált algoritmust (ha az nem nyilvánvaló), és esetleg azt, hogy mit feltételez környezetéről.
4. Minden globális és névtér-változóhoz, illetve konstanshoz van megjegyzés.
5. Van néhány megjegyzés ott, ahol a kód nem nyilvánvaló és/vagy más rendszerre nem átültethető.
6. A fentieken kívül kevés megjegyzés van.

Például:

```
// tbl.c: Implementation of the symbol table.

/*
  Gaussian elimination with partial pivoting.
  See Ralston: "A first course ..." pg 411.
*/

// swap() assumes the stack layout of an SGI R6000.

/*******************************

  Copyright (c) 1997 AT&T, Inc.
  All rights reserved

*******************************/
```

A jól megválasztott és jól megírt megjegyzések alapvető részét képezik a jó programnak. Jó megjegyzéseket írni legalább olyan nehéz, mint megírni magát a programot. Olyan művészet, melyet érdemes művelni.

Jegyezzük meg azt is, hogy ha kizárólag a // megjegyzéseket használjuk egy függvényben, akkor ennek a függvénynek bármely részét megjegyzésbe tehetjük a /* */ jelöléssel (ez fordítva is igaz).

6.5. Tanácsok

[1] Részesítsük előnyben a standard könyvtárat a többi könyvtárral és a „kézzel írt kóddal" szemben. §6.1.8.
[2] Kerüljük a bonyolult kifejezéseket. §6.2.3.
[3] Ha kétségeink vannak az operátorok precedenciájával kapcsolatban, zárójelezzünk. §6.2.3.
[4] Kerüljük a típuskényszerítést (cast). §6.2.7.
[5] Ha explicit típuskonverzió szükséges, részesítsük előnyben a jobban definiált konverziós operátorokat a C stílusú átalakítással szemben. §6.2.7.
[6] Kizárólag jól meghatározott szerkezeteknél használjuk a *T(e)* jelölést. §6.2.8.
[7] Kerüljük az olyan kifejezéseket, melyek kiértékelési sorrendje nem meghatáro-

zott. §6.2.2.
[8] Kerüljük a *goto*-t. §6.3.4.
[9] Kerüljük a *do* utasítást. §6.3.3.
[10] Ne adjunk meg változót addig, amíg nincs érték, amivel feltölthetnénk. §6.3.1, §6.3.2.1, §6.3.3.1.
[11] A megjegyzéseket frissítsük rendszeresen. §6.4.
[12] Tartsunk fenn következetes stílust. §6.4.
[13] A globális *operator new()* helyettesítésére adjunk meg inkább egy *operator new()* tagot (§15.6). §6.2.6.2.
[14] Bemenet beolvasásakor mindig vegyük számításba a rosszul megadott bemenetet is. §6.1.3.

6.6. Gyakorlatok

1. (*1) Írjuk meg a következő *for* utasítással egyenértékű *while* utasítást:

```
for (i=0; i<max_length; i++) if (input_line[i] == '?') quest_count++;
```

   Ezt írjuk át úgy, hogy ciklusváltozóként mutatót használunk, azaz úgy, hogy a vizsgálat **p=='?'* alakú legyen.
2. (*1) Zárójelezzük teljesen a következő kifejezéseket:

```
a = b + c * d << 2 & 8
a & 077 != 3
a == b || a == c && c < 5
c = x != 0
0 <= i < 7
f(1,2)+3
a = - 1 + + b -- - 5
a = b == c ++
a = b = c = 0
a[4][2] *= * b ? c : * d * 2
a-b,c=d
```

3. (*2) Olvassuk be üreshellyel elválasztott (név- és érték-) párok sorozatát, ahol a név egyetlen üreshellyel elválasztott szó, az érték pedig egy egész vagy lebegőpontos érték. Számítsuk ki és írjuk ki minden névre az értékek összegét és számtani közepét, valamint az összes névre vonatkozó összeget és számtani közepet. Tipp: §6.1.8.
4. (*1) Írjuk meg a bitenkénti logikai operátorok (§6.2.4) értéktáblázatát a *0* és *1*

operandusok összes lehetséges párosítására.

5. (*1,5) Találjunk 5 különböző C++ szerkezetet, melynek jelentése nem meghatározott (§C.2). (*1,5.) Találjunk 5 különböző C++ szerkezetet, melynek jelentése a nyelvi megvalósítástól függ (§C.2).
6. (*1) Adjunk 10 különböző példát nem hordozható C++ kódra.
7. (*2) Írjunk 5 kifejezést, melyek kiértékelési sorrendje nem meghatározott. Hajtsuk őket végre és nézzük meg, mit csinál velük egy – de lehetőleg több – C++-változat.
8. (*1,5) Mi történik a rendszerünkben, ha nullával osztunk? Mi történik túlcsordulás és alulcsordulás esetén?
9. (*1) Zárójelezzük teljesen a következő kifejezéseket:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

10. (*2) Írjuk meg a következő függvényeket: *strlen()*, ami egy C stílusú karakterlánc hosszát adja vissza, *strcpy()*, ami egy karakterláncot másol egy másikba, és *strcmp()*, ami két karakterláncot hasonlít össze. Gondoljuk meg, mi legyen a paraméterek és a visszatérési érték típusa. Ezután hasonlítsuk össze a függvényeket a standard könyvtárban lévő változatokkal, ahogy azok a <*cstring*>-ben (<*string.h*>-ban) szerepelnek és ahogy a §20.4.1 pontban leírtuk azokat.
11. (*1) Nézzük meg, hogyan reagál a fordítóprogramunk ezekre a hibákra:

```
void f(int a, int b)
{
        if (a = 3) // ...
        if (a&077 == 0) // ...
        a := b+1;
}
```

„Készítsünk" több egyszerű hibát és nézzük meg, hogyan reagál a fordítóprogram.

12. (*2) Módosítsuk úgy a §6.6[3] programot, hogy a középső értéket (medián) is kiszámítsa.
13. (*2) Írjuk meg a *cat()* függvényt, amelynek két C stílusú karakterlánc paramétere van és egy olyan karakterláncot ad vissza, amely a paraméterek összefűzéséből áll elő. Az eredményeknek foglaljunk helyet a *new*-val.
14. (*2) Írjuk meg a *rev()* függvényt, amelynek egy karakterlánc paramétere van és

megfordítja a benne lévő karaktereket. Azaz a *rev(p)* lefutása után *p* utolsó karaktere az első lesz és így tovább.

15. (*1,5) Mit csinál a következő példa és miért írna valaki ilyesmit?

```
void send(int* to, int* from, int count)
// Duff programja. A megjegyzéseket szándékosan töröltem.
{
        int n = (count+7)/8;
        switch (count%8) {
        case 0:  do {    *to++ = *from++;
        case 7:          *to++ = *from++;
        case 6:          *to++ = *from++;
        case 5:          *to++ = *from++;
        case 4:          *to++ = *from++;
        case 3:          *to++ = *from++;
        case 2:          *to++ = *from++;
        case 1:          *to++ = *from++;
                 } while (--n>0);
        }
}
```

16. (*2) Írjuk meg az *atoi(const char*)* függvényt, amely egy számokat tartalmazó karakterláncot kap és visszaadja a megfelelő egészet. Például *atoi("123") 123* lesz. Módosítsuk az *atoi()*-t úgy, hogy kezelje a C++ oktális és hexadecimális jelölését is, az egyszerű, tízes számrendszerbeli számokkal együtt. Módosítsuk a függvényt úgy is, hogy kezelje a C++ karakterkonstans jelölést is.
17. (*2) Írjunk egy olyan *itoa(int i, char b[ ])* függvényt, amely létrehozza *b*-ben *i* karakterlánc ábrázolását és visszaadja *b*-t.
18. (*2) Gépeljük be teljes egészében a számológép példát és hozzuk működésbe. Ne takarítsunk meg időt azzal, hogy már begépelt szövegrészeket használunk. A legtöbbet a „kis buta hibák" kijavításából fogunk tanulni.
19. (*2) Módosítsuk a számológépet, hogy kiírja a hibák sorának számát is.
20. (*3) Tegyük lehetővé, hogy a felhasználó függvényeket adhasson meg a számológéphez. Tipp: adjunk meg egy függvényt műveletek sorozataként, úgy, mintha a felhasználó gépelte volna be azokat. Az ilyen sorozatokat karakterláncként vagy szimbólumok (tokenek) listájaként tárolhatjuk. Ezután olvassuk be és hajtsuk végre ezeket a műveleteket, amikor a függvény meghívódik. Ha azt akarjuk, hogy egy felhasználói függvénynek paraméterei legyenek, akkor arra külön jelölést kell kitalálnunk.
21. (*1,5) Alakítsuk át a számológépet, hogy a *symbol* szerkezetet használja és ne a statikus *number_value* és *string_value* változókat.
22. (*2,5) Írjunk olyan programot, amely kiveszi a megjegyzéseket a C++ progra-

mokból. Azaz, olvassunk a *cin*-ről, távolítsuk el mind a *//*, mind a */* */* megjegyzéseket, majd írjuk ki az eredményt a *cout*-ra. Ne törődjünk azzal, hogy a kimenet szép legyen (az egy másik, sokkal nehezebb feladat lenne). Ne törődjünk a hibás programokkal. Óvakodjunk a *//*, */** és **/* használatától a megjegyzésekben, karakterláncokban és karakterkonstansokban.

23. (*2) Nézzünk meg néhány programot, hogy elképzelésünk lehessen a mostanság használatos stílusok (behúzások, elnevezések és megjegyzések) változatosságáról.

7

Függvények

„Ismételni emberi dolog.
Rekurziót írni isteni.”
(L. Peter Deutsch)

Függvénydeklarációk és -definíciók • Paraméterátadás • Visszatérési értékek • Függvénytúlterhelés • A többértelműség feloldása • Alapértelmezett paraméterek • *stdargs* • Függvényekre hivatkozó mutatók • Makrók • Tanácsok • Gyakorlatok

7.1. Függvénydeklarációk

A C++-ban általában úgy végzünk el valamit, hogy meghívunk rá egy függvényt, és a függvény definiciójával írjuk le, hogyan kell azt elvégezni. A függvényt azonban nem hívhatjuk meg úgy, hogy előzetesen nem deklaráltuk. A függvény deklarációja megadja a függvény nevét, visszatérési értékének típusát (ha van ilyen), és azon paraméterek számát és típusát, amelyeket át kell adni a függvény meghívásakor:

```
Elem* next_elem();
char* strcpy(char* to, const char* from);
void exit(int);
```

A paraméterátadás ugyanazt a szerepet tölti be, mint a kezdeti értékadás. A fordítóprogram ellenőrzi a paraméterek típusát és automatikus típuskonverziót végez, ha szükséges:

```
double sqrt(double);

double sr2 = sqrt(2);          // sqrt() meghívása double(2) paraméterrel
double sq3 = sqrt("three");    // hiba: sqrt() double típusú paramétert igényel
```

Az ilyen ellenőrzés és konverzió jelentőségét nem szabad alábecsülni.

A függvénydeklaráció paraméterneveket is tartalmazhat, melyek segítik a program olvasóját. A fordítóprogram ezeket egyszerűen nem veszi figyelembe. Amint §4.7-ben említettük, a *void* visszatérési típus azt jelenti, hogy a függvény nem ad vissza értéket.

7.1.1. Függvénydefiníciók

Minden függvényt, amit meghívunk a programban, valahol (de csak egyszer) meg kell határoznunk. A függvénydefiníció (függvény-meghatározás) olyan deklaráció, amelyben megadjuk a függvény törzsét:

```
extern void swap(int*, int*);      // deklaráció

void swap(int* p, int* q)          // definíció
{
  int t = *p;
  *p = *q;
  *q = t;
}
```

A függvények definícióinak és deklarációinak ugyanazt a típust kell meghatározniuk. A paraméterek nevei azonban nem részei a típusnak, így nem kell azonosaknak lenniük.

Nem ritka az olyan függvénydefiníció, amely nem használja fel az összes paramétert:

```
void search(table* t, const char* key, const char*)
{
  // a harmadik paraméter nem használatos
}
```

Amint látjuk, azt a tényt, hogy egy paramétert nem használunk fel, úgy jelölhetjük, hogy nem nevezzük meg a paramétert. A névtelen paraméterek jellemzően a program egyszerűsítése folytán vagy annak későbbi bővíthetőségét biztosítandó kerülnek a kódba. Mindkét esetben azzal, hogy bár nem használjuk fel, de a helyükön hagyjuk a paramétereket, biztosítjuk, hogy a függvényt meghívókat nem érintik a módosítások.

A függvényeket a fordító által a hívás sorában kifejtendőként (*inline*-ként) is megadhatjuk:

```
inline int fac(int n)
{
  return (n<2) ? 1 : n*fac(n-1);
}
```

Az *inline* kulcsszó javaslat a fordítóprogram számára, hogy a *fac()* meghívásánál próbálja meg annak kódját a hívás sorában létrehozni, ahelyett, hogy előbb létrehozná azt, majd a szokásos függvényhívó eljárás szerint hívná meg. Egy okos fordítóprogram a *fac(6)* meghívásakor létre tudja hozni a *720* konstanst. Az egymást kölcsönösen meghívó (kölcsönösen rekurzív) helyben kifejtett függvények, illetve a bemenettől függően magukat újrahívó vagy nem újrahívó helyben kifejtett függvények lehetősége miatt nem biztos, hogy egy *inline* függvény minden hívása a hívás sorában jön létre. A fordítóprogramok intelligenciájának mértéke nem írható elő, így előfordulhat, hogy az egyik *720*-at, a másik *6*fac(5)*-öt, a harmadik pedig a *fac(6)* nem helyben kifejtett hívást hozza létre.

Ha nem rendelkezünk kivételesen intelligens fordító- és szerkesztő-programmal, a hívás sorában történő létrehozást akkor biztosíthatjuk, ha a függvény kifejtése – és nem csak deklarációja – is a hatókörben szerepel (§9.2). Az *inline* kulcsszó nem befolyásolja a függvény értelmezését. Nevezetesen az ilyen függvényeknek – és *static* változóiknak (§7.1.2.) is – ugyanúgy egyedi címük van.

7.1.2. Statikus változók

A lokális (helyi) változók akkor kapnak kezdőértéket, amikor a végrehajtás elér a definiciójukhoz. Alapértelmezés szerint ez a függvény minden meghívásakor megtörténik, az egyes függvényhívásoknak pedig saját másolatuk van a változóról. Ha egy lokális változót *static*-ként vezetünk be, akkor azt a függvény minden meghívásakor egyetlen, állandó című objektum jelöli majd. A változó csak egyszer kap értéket, amikor a végrehajtás eléri annak első definícióját:

```
void f(int a)
{
  while (a--) {
          static int n = 0;    // egyszer kap kezdőértéket
          int x = 0;          // 'a' alkalommal kap kezdőértéket (az f() minden meghívásakor)

          cout << "n == " << n++ << ", x == " << x++ << '\n';
  }
}

int main()
{
  f(3);
}
```

A fenti a következőket írja ki:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

A statikus változók anélkül biztosítanak „emlékezetet" a függvénynek, hogy globális változót vezetnének be, amelyet más függvények is elérhetnek és módosítással használhatatlanná tehetnek (lásd még §10.2.4).

7.2. Paraméterátadás

Amikor egy függvény meghívódik, a fordítóprogram a formális paraméterek számára tárterületet foglal le, az egyes formális paraméterek pedig a megfelelő valódi (aktuális) paraméter-értékkel töltődnek fel. A paraméterátadás szerepe azonos a kezdeti értékátadáséval. A fordítóprogram ellenőrzi, hogy az aktuális paraméterek típusa megegyezik-e a formális paraméterek típusával, és végrehajt minden szabványos és felhasználói típuskonverziót. A tömbök átadására egyedi szabályok vonatkoznak (§7.2.1), de van lehetőség nem ellenőrzött (§7.6) és alapértelmezett paraméterek (§7.5) átadására is. Vegyük a következő példát:

```
void f(int val, int& ref)
{
  val++;
  ref++;
}
```

Amikor *f()* meghívódik, *val++* az első aktuális paraméter helyi másolatát növeli, *ref++* pedig a második aktuális paramétert. Az alábbi

```
void g()
{
  int i = 1;
  int j = 1;
  f(i,j);
}
```

j-t fogja növelni, de *i*-t nem. Az első paraméter (*i*) érték szerint adódik át, a második (*j*) referencia szerint. Amint §5.5-ben említettük, azok a függvények, melyek módosítják a referencia szerint átadott paramétereiket, nehezen olvashatóvá teszik a programot és általában kerülendők (ellenben lásd §21.3.2-et). Észrevehetően hatékonyabb lehet, ha egy nagy objektumot referencia, nem pedig érték szerint adunk át. Ebben az esetben a paramétert megadhatjuk *const*-ként, hogy jelezzük, csak hatékonysági okokból használunk referenciát és nem szeretnénk lehetővé tenni, hogy a hívott függvény módosíthassa az objektum értékét:

```
void f(const Large& arg)
{
  // "arg" értéke nem módosítható, csak explicit típuskonverzióval
}
```

A referencia-paraméter deklarációjában a *const* elhagyása azt fejezi ki, hogy szándékunkban áll a változót módosítani:

```
void g(Large& arg);          // tételezzük fel, hogy g() módosítja arg-ot
```

Hasonlóan, a *const*-ként megadott mutató paraméter azt jelzi az olvasónak, hogy a paraméter által mutatott objektum értékét a függvény nem változtatja meg:

```
int strlen(const char*);                    // karakterek száma egy C stílusú
                                            // karakterláncban
char* strcpy(char* to, const char* from);   // C stílusú karakterlánc másolása
int strcmp(const char*, const char*);       // C stílusú karakterláncok összehasonlítása
```

A *const* paraméterek használatának fontossága a program méretével együtt nő.

Jegyezzük meg, hogy a paraméterátadás szerepe különbözik a (nem kezdeti) értékadásétól. Ez a *const* paraméterek, a referencia-paraméterek, és néhány felhasználói típusú paraméter esetében lényeges (§10.4.4.1).

Literált, állandót és olyan paramétert, amely átalakítást igényel, át lehet adni *const&* paraméterként, de nem *const&*-ként nem. Mivel a *const T&* paraméterek konverziója megengedett, biztosított, hogy egy ilyen paraméternek pontosan ugyanazokat az értékeket lehet adni, mint egy *T* típusú értéknek, azáltal, hogy az értéket ideiglenes változóban adjuk át (amennyiben ez szükséges):

```
float fsqrt(const float&);        // Fortran stílusú sqrt referencia-paraméterrel

void g(double d)
{
  float r = fsqrt(2.0f);          // a 2.0f-et tartalmazó ideiglenes változóra hivatkozó
                                  // referencia átadása
  r = fsqrt(r);        // r-re hivatkozó referencia átadása
  r = fsqrt(d);        // a  float(d)-t tartalmazó ideiglenes változóra hivatkozó referencia
                       // átadása
}
```

Mivel a nem *const* referencia típusú paraméterek konverziója nem megengedett, az ideiglenes változók bevezetéséből adódó buta hibák elkerülhetők:

```
void update(float& i);

void g(double d, float r)
{
  update(2.0f);        // hiba: konstans paraméter
  update(r);           // r-re hivatkozó referencia átadása
  update(d);           // hiba: típuskonverzió szükséges
}
```

Ha ezek a hívások szabályosak lennének, az *update()* csendben módosította volna azokat az ideiglenes változókat, amelyek azonnal törlődtek. Ez rendszerint kellemetlen meglepetésként érné a programozót.

7.2.1. Tömb paraméterek

Ha függvényparaméterként tömböt használunk, a tömb első elemére hivatkozó mutató adódik át:

```
int strlen(const char*);

void f()
{
  char v[ ] = "egy tömb";
  int i = strlen(v);
  int j = strlen("Nicholas");
}
```

Azaz egy *T[ ]* típusú paraméter *T** típusúvá lesz alakítva, ha paraméterként adódik át. Ebből az következik, hogy ha egy paraméterként alkalmazott tömb egy eleméhez értéket rendelünk, a tömb paraméter is módosul, vagyis a tömbök abban különböznek a többi típustól, hogy nem érték szerint adódnak át (ez nem is lehetséges).

A tömb mérete nem hozzáférhető a hívott függvény számára. Ez bosszantó lehet, de több mód van rá, hogy a problémát megkerüljük. A C stílusú karakterláncok nulla végződésűek, így méretük könnyen kiszámítható. Más tömböknél a méretet egy további paraméterrel adhatjuk meg:

```
void compute1(int* vec_ptr, int vec_size);          // egyik módszer

struct Vec {
   int* ptr;
   int size;
};

void compute2(const Vec& v);                        // másik módszer
```

Választhatjuk azt is, hogy tömbök helyett olyan típusokat használunk, mint a *vector* (§3.7.1, §16.3). A többdimenziós tömbök némileg bonyolultabbak (lásd §C.7), de helyettük gyakran használhatunk mutatókból álló tömböket, melyek nem igényelnek különleges bánásmódot:

```
char* day[ ] = {
   "hétfő", "kedd", "szerda", "csütörtök", "péntek", "szombat", "vasárnap"
};
```

A *vector* és a hozzá hasonló típusok a beépített, alacsonyszintű tömbök és mutatók helyett is használhatók.

7.3. Visszatérési érték

A *main()* kivételével(§3.2) minden nem *void*-ként megadott függvénynek értéket kell visszaadnia. Megfordítva, a *void* függvények nem adhatnak vissza értéket:

```
int f1() { }               // hiba: nincs visszatérési érték
void f2() { }              // rendben
```

```
int f3() { return 1; }          // rendben
void f4() { return 1; }         // hiba: visszatérési érték void függvényben

int f5() { return; }            // hiba: visszatérési érték hiányzik
void f6() { return; }           // rendben
```

A visszatérési értéket a *return* utasítás határozza meg:

```
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
```

Az önmagukat meghívó függvényeket *rekurzív* (újrahívó) függvényeknek nevezzük.

Egy függvényben több *return* utasítás is lehet:

```
int fac2(int n)
{
  if (n > 1) return n*fac2(n-1);
  return 1;
}
```

A paraméterátadáshoz hasonlóan a függvényérték visszaadásának szerepe is azonos a kezdeti értékadáséval. A *return* utasítást úgy tekintjük, hogy az egy visszatérési típusú, név nélküli változónak ad kezdőértéket. A fordítóprogram összehasonlítja a *return* kifejezés típusát a visszatérési típussal és minden szabványos és felhasználói típusátalakítást végrehajt:

```
double f() { return 1; }        // 1 automatikusan double(1)-gyé alakul
```

Minden egyes alkalommal, amikor egy függvény meghívódik, paramétereinek és lokális (automatikus) változóinak egy új másolata jön létre. A tár a függvény visszatérése után ismét felhasználásra kerül, ezért lokális változóra hivatkozó mutatót soha nem szabad visszaadni, mert a mutatott hely tartalma kiszámíthatatlan módon megváltozhat:

```
int* fp() { int local = 1; /* ... */ return &local; }    // rossz
```

Ez a hiba kevésbé gyakori, mint referenciát használó megfelelője:

```
int& fr() { int local = 1; /* ... */ return local; }     // rossz
```

Szerencsére a fordítóprogram általában figyelmeztet, hogy lokális változóra vonatkozó hivatkozást adtunk vissza.

A *void* függvények nem adhatnak vissza értéket, de meghívásuk nem is eredményez ilyet, következésképpen egy *void* függvény *return* utasításában szereplő kifejezésként használhatunk *void* függvényt:

```
void g(int* p);

void h(int* p) { /* ... */ return g(p); }   // rendben: üres visszatérési érték
```

Ez a fajta visszatérés olyan sablon (template) függvények írásánál fontos, ahol a visszatérési típus egy sablonparaméter (lásd §18.4.4.2).

7.4. Túlterhelt függvénynevek

A legtöbb esetben jó ötlet különböző függvényeknek különböző neveket adni, de amikor egyes függvények lényegében ugyanazt a műveletet végzik különböző típusú objektumokon, kényelmesebb lehet ugyanúgy elnevezni azokat. Azt, hogy különböző típusokra vonatkozó műveletekre ugyanazt a nevet használjuk, ***túlterhelésnek*** (*overloading*) nevezzük. Ez a módszer a C++ alapműveleteinél is használatos. Azaz, csak egyetlen „név" van az összeadásra (+), az mégis használható egész, lebegőpontos, és mutató típusú értékek összeadására is. A gondolatot a programozó által készített függvényekre is kiterjeszthetjük:

```
void print(int);            // egész kiírása
void print(const char*);    // C stílusú karakterlánc kiírása
```

Ami a fordítóprogramot illeti, az egyetlen, ami közös az azonos nevű függvényekben, a név. A függvények feltehetően hasonlóak valamilyen értelemben, de a nyelv nem korlátozza és nem is segíti a programozót. Ezért a túlterhelt függvénynevek elsődlegesen jelölésbeli kényelmet adnak. Ez a kényelem az olyan hagyományos nevű függvényeknél igazán lényeges, mint az *sqrt*, *print*, és *open*. Amikor a név jelentése fontos, ez a kényelem alapvetővé válik. Ez történik például a +, * és << operátorok, a konstruktorok (§11.7), valamint az általánosított (generikus) programozás (§2.7.2, 18. fejezet) esetében. Amikor egy *f* függvény meghívódik, a fordítóprogramnak ki kell találnia, melyik *f* nevű függvényt hívja meg. Ezt úgy teszi, hogy minden egyes *f* nevű függvény formális paramétereinek típusát összehasonlítja az aktuális paraméterek típusával, majd azt a függvényt hívja meg, amelynek paraméterei a legjobban illeszkednek, és fordítási idejű hibát ad, ha nincs jól illeszkedő függvény:

```
void print(double);
void print(long);

void f()
{
  print(1L);         // print(long)
  print(1.0);        // print(double)
  print(1);          // hiba, többértelmű: print(long(1)) vagy print(double(1))?
}
```

A fordítóprogram a túlterhelt függvények halmazából úgy választja ki a megfelelő változatot, hogy megkeresi azt a függvényt, amelyiknél a hívás paraméter-kifejezésének típusa a legjobban illeszkedik a függvény formális paramétereire. Ahhoz, hogy mindez elvárásainknak (közelítően) megfelelő módon történjen, az alábbiakat kell megkísérelni (ebben a sorrendben):

1. Pontos illeszkedés: nincs szükség konverzióra vagy csak egyszerű konverziókat kell végezni (például tömb nevét mutatóra, függvény nevét függvényre hivatkozó mutatóra, vagy *T*-t *const T*-re).
2. Kiterjesztést használó illeszkedés: csak egész értékre kiterjesztés (integral promotion) szükséges (*bool*-ról *int*-re, *char*-ról *int*-re, *short*-ról *int*-re, illetve ezek *unsigned* megfelelőiről *int*-re §C.6.1), valamint *float*-ról *double*-ra.
3. Szabványos konverziókat használó illeszkedés: *int*-ről *double*-ra, *double*-ról *int*-re, *Derived**-ról *Base**-ra (§12.2), *T**-ról *void**-ra (§5.6), vagy *int*-ről *unsigned int*-re (§C.6).
4. Felhasználói konverziókat használó illeszkedés (§11.4).
5. Függvénydeklarációban három pontot (...) használó illeszkedés (§7.6).

Ha azon a szinten, ahol először találunk megfelelő illeszkedést, két illeszkedést is találunk, a hívást a fordító többértelműként elutasítja. A túlterhelést feloldó szabályok elsősorban azért ennyire alaposan kidolgozottak, mert figyelembe kellett venni a C és a C++ beépített numerikus típusokra vonatkozó bonyolult szabályait (§C.6). Vegyük a következő példát:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
  print(c);          // pontos illeszkedés: print(char) meghívása
  print(i);          // pontos illeszkedés: print(int) meghívása
```

```
  print(s);           // kiterjesztés egésszé: print(int) meghívása
  print(f);           // float kiterjesztése double-lé: print(double)

  print('a');         // pontos illeszkedés: print(char) meghívása
  print(49);          // pontos illeszkedés: print(int) meghívása
  print(0);           // pontos illeszkedés: print(int) meghívása
  print("a");         // pontos illeszkedés: print(const char*) meghívása
}
```

A *print(0)* hívás a *print(int)*-et hívja meg, mert *0* egy *int*. A *print('a')* hívás a *print(char)*-t, mivel *'a'* egy *char* (§4.3.1). Az átalakítások (konverziók) és a kiterjesztések között azért teszünk különbséget, mert előnyben akarjuk részesíteni a biztonságos kiterjesztéseket (például *char*-ról *int*-re) az olyan, nem biztonságos műveletekkel szemben, mint az *int*-ről *char*-ra történő átalakítás.

A túlterhelés feloldása független a szóba jöhető függvények deklarációs sorrendjétől.

A túlterhelés viszonylag bonyolult szabályrendszeren alapul, így a programozó néha meglepődhet azon, melyik függvény hívódik meg. Ez azonban még mindig a kisebbik rossz. Vegyük figyelembe a túlterhelés alternatíváját: gyakran hasonló műveleteket kell végrehajtanunk többféle típusú objektumon. Túlterhelés nélkül több függvényt kellene megadnunk, különböző nevekkel:

```
void print_int(int);
void print_char(char);
void print_string(const char*);        // C stílusú karakterlánc

void g(int i, char c, const char* p, double d)
{
  print_int(i);                        // rendben
  print_char(c);                       // rendben
  print_string(p);                     // rendben

  print_int(c);                        // rendben? print_int(int(c)) meghívása
  print_char(i);                       // rendben? print_char(char(i)) meghívása
  print_string(i);                     // hiba
  print_int(d);                        // rendben? print_int(int(d)) meghívása
}
```

A túlterhelt *print()*-hez képest több névre és arra is emlékeznünk kell, hogy a neveket helyesen használjuk. Ez fárasztó lehet, meghiúsítja az általánosított programozásra (§2.7.2) irányuló erőfeszítéseket, és általában arra ösztönzi a programozót, hogy viszonylag alacsony

szintű típusokra irányítsa figyelmét. Mivel nincs túlterhelés, ezen függvények paraméterein bármilyen szabványos konverziót elvégezhetünk, ami szintén hibákhoz vezethet. Ebből az következik, hogy a fenti példában a négy „hibás" paraméterrel való hívás közül csak egyet vesz észre a fordítóprogram. A túlterhelés növeli annak az esélyét, hogy egy nem megfelelő paramétert a fordítóprogram elutasít.

7.4.1. A túlterhelés és a visszatérési típus

A túlterhelés feloldásánál a visszatérési típust nem vesszük figyelembe. Ez azért szükséges, hogy az egyes operátorokra (§11.2.1, §11.2.4) vagy függvényhívásra vonatkozó feloldás környezetfüggetlen maradjon. Vegyük a következőt:

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
  float fl = sqrt(da);          // sqrt(double) meghívása
  double d = sqrt(da);          // sqrt(double) meghívása
  fl = sqrt(fla);               // sqrt(float) meghívása
  d = sqrt(fla);                // sqrt(float) meghívása
}
```

Ha a visszatérési típust a fordítóprogram figyelembe venné, többé nem lenne lehetséges, hogy elszigetelten nézzük az *sqrt()* egy hívását és eldöntsük, azzal melyik függvényt hívták meg.

7.4.2. A túlterhelés és a hatókör

A különböző, nem névtér hatókörben megadott függvények nem túlterheltek:

```
void f(int);

void g()
{
  void f(double);
  f(1);                // f(double) meghívása
}
```

Világos, hogy *f(int)* lett volna a legjobb illeszkedés *f(1)*-re, de a hatókörben csak *f(double)* látható. Az ilyen esetekben helyi deklarációkat adhatunk a kódhoz vagy törölhetjük azokat, hogy megkapjuk a kívánt viselkedést. Mint mindig, a szándékos elfedés hasznos módszer lehet, a nem szándékos azonban meglepetéseket okozhat. Ha osztály-hatókörök (§15.2.2) vagy névtér-hatókörök (§8.2.9.2) között átnyúló túlterhelést szeretnénk, *using* deklarációkat vagy *using* utasításokat használhatunk (§8.2.2). Lásd még §8.2.6-ot.

7.4.3. A többértelműség „kézi" feloldása

Többértelműséghez vezethet, ha egy függvénynek túl sok (vagy túl kevés) túlterhelt változatát adjuk meg:

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
{
  f1(i);     // többértelmű: f1(char) vagy f1(long)
  f2(0);     // többértelmű: f2(char*) vagy f2(int*)
}
```

Ahol lehetséges, az ilyen esetekben úgy kell tekintsük egy függvény túlterhelt változatainak halmazát, mint egészet, és gondoskodnunk kell róla, hogy a változatok azonos értelműek legyenek. Ez többnyire úgy oldható meg, ha a függvénynek egy olyan új változatát adjuk hozzá az eddigiekhez, amely feloldja a többértelműséget:

```
inline void f1(int n) { f1(long(n)); }
```

A fenti függvényváltozat hozzáadása feloldja az összes *f1(i)*-hez hasonló többértelműséget a szélesebb *long int* típus javára.

A hívások feloldására típuskényszerítést is használhatunk:

```
f2(static_cast<int*>(0));
```

Ez azonban általában csúnya és ideiglenes szükségmegoldás, hiszen a következő – hamarosan bekövetkező – hasonló függvényhívással ismét foglalkoznunk kell majd.

Néhány kezdő C++ programozót bosszantanak a fordítóprogram által kiírt többértelműségi hibák, a tapasztaltabbak viszont becsülik ezeket az üzeneteket, mert hasznos jelzői a tervezési hibáknak.

7.4.4. Több paraméter feloldása

A túlterhelést feloldó szabályok alapján biztosíthatjuk, hogy a legegyszerűbb algoritmus (függvény) lesz felhasználva, amikor a számítások hatékonysága és pontossága jelentősen különbözik a szóban forgó típusoknál:

```
int pow(int, int);
double pow(double, double);

complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);

void k(complex z)
{
  int i = pow(2,2);                   // pow(int,int) meghívása
  double d = pow(2.0,2.0);            // pow(double,double) meghívása
  complex z2 = pow(2,z);              // pow(double,complex) meghívása
  complex z3 = pow(z,2);              // pow(complex,int) meghívása
  complex z4 = pow(z,z);              // pow(complex,complex) meghívása
}
```

A kettő vagy több paraméterű túlterhelt függvények közötti választás folyamán minden paraméterre kiválasztódik a legjobban illeszkedő függvény, a §7.4 szabályai alapján. Az a függvény hívódik meg, amely az adott paraméterre a legjobban, a többire pedig jobban vagy ugyanúgy illeszkedik. Ha nem létezik ilyen függvény, a hívást a fordító többértelműként elutasítja:

```
void g()
{
  double d = pow(2.0,2);       // hiba: pow(int(2.0),2) vagy pow(2.0,double(2))?
}
```

A függvényhívás azért többértelmű, mert a *pow (double,double)* első paraméterére *2.0*, a *pow(int,int)* második paraméterére pedig *2* a legjobb illeszkedés.

7.5. Alapértelmezett paraméterek

Egy általános függvénynek általában több paraméterre van szüksége, mint amennyi az egyszerű esetek kezeléséhez kell. Nevezetesen az objektumokat (§10.2.3) létrehozó függvények gyakran számos lehetőséget nyújtanak a rugalmasság érdekében. Vegyünk egy függvényt, amely egy egészt ír ki. Ésszerűnek látszik megadni a felhasználónak a lehetőséget, hogy meghatározza, milyen számrendszerben írja ki a függvény az egészt, a legtöbb program azonban az egészeket tízes számrendszer szerint írja ki. Például a

```
void print(int value, int base =10);      // az alapértelmezett alap 10

void f()
{
  print(31);
  print(31,10);
  print(31,16);
  print(31,2);
}
```

ezt a kimenetet eredményezheti:

```
31 31 1f 11111
```

Az alapértelmezett (default) paraméter hatását elérhetjük túlterheléssel is:

```
void print(int value, int base);
inline void print(int value) { print(value,10); }
```

A túlterhelés viszont kevésbé teszi nyilvánvalóvá az olvasó számára, hogy az a szándékunk, hogy legyen egy egyszerű *print* függvényünk és egy rövidítésünk.

Az alapértelmezett paraméter típusának ellenőrzése a függvény deklarációjakor történik és a paraméter a függvény hívásakor értékelődik ki. Alapértelmezett értékeket csak a záró paramétereknek adhatunk:

```
int f(int, int =0, char* =0);    // rendben
int g(int =0, int =0, char*);    // hiba
int h(int =0, int, char* =0);    // hiba
```

Jegyezzük meg, hogy a * és az = közötti szóköz lényeges (a *= értékadó operátor, §6.2):

```
int nasty(char*=0);          // szintaktikus hiba
```

Az alapértelmezett paraméterek ugyanabban a hatókörben egy későbbi deklarációval nem ismételhetők meg és nem módosíthatók:

```
void f(int x = 7);
void f(int = 7);             // hiba: az alapértelmezett paraméter nem adható meg kétszer
void f(int = 8);             // hiba: különböző alapértelmezett paraméterek

void g()
{
  void f(int x = 9);         // rendben: ez a deklaráció elfedi a külsőt
  // ...
}
```

Hibalehetőséget rejt magában, ha egy nevet úgy adunk meg egy beágyazott hatókörben, hogy a név elfedi ugyanannak a névnek egy külső hatókörben levő deklarációját.

7.6. Nem meghatározott számú paraméter

Néhány függvény esetében nem határozható meg a hívásban elvárt paraméterek száma és típusa. Az ilyen függvényeket úgy adhatjuk meg, hogy a paraméter-deklarációk listáját a ... jelöléssel zárjuk le, melynek jelentése „és talán néhány további paraméter":

```
int printf(const char* ...);
```

A fenti azt határozza meg, hogy a C standard könyvtárának *printf()* függvénye (§21.8) meghívásakor legalább egy *char** típusú paramétert vár, de lehet, hogy van más paramétere is:

```
printf("Helló, világ!\n");
printf("A nevem %s %s\n", vezetek_nev, kereszt_nev);
printf("%d + %d = %d\n",2,3,5);
```

Az ilyen függvények olyan adatokra támaszkodnak, amelyek nem elérhetők a fordítóprogram számára, amikor az a paraméterek listáját értelmezi. A *printf()* esetében az első paraméter egy formátum-vezérlő, amely egyedi karaktersorozatokat tartalmaz, lehetővé téve, hogy a *printf()* helyesen kezelje a többi paramétert: a *%s* például azt jelenti, hogy „várj egy *char** paramétert", a *%d* pedig azt, hogy „várj egy *int* paramétert". A fordítóprogram viszont általában nem tudhatja (és nem is biztosíthatja), hogy a várt paraméterek tényleg ott vannak és megfelelő típusúak-e:

```
#include <stdio.h>

int main()
{
  printf("A nevem %s %s\n",2);
}
```

A fenti kódot a fordító lefordítja és (a legjobb esetben) furcsának látszó kimenetet hoz létre. (Próbáljuk ki!)

Természetesen ha egy paraméter nem deklarált, a fordítóprogram nem fog elegendő információval rendelkezni a szabványos típusellenőrzés és -konverzió elvégzéséhez. Ebben az esetben egy *short* vagy egy *char int*-ként adódik át, egy *float* pedig *double*-ként, a programozó pedig nem feltétlenül ezt várja.

Egy jól megtervezett programban legfeljebb néhány olyan függvényre van szükség, melynek paraméterei nem teljesen meghatározottak. A túlterhelt vagy alapértelmezett paramétereket használó függvények arra használhatók, hogy megoldják a típusellenőrzést a legtöbb olyan esetben, amikor a paraméterek típusát szükségből meghatározatlanul hagyjuk. A három pont csak akkor szükséges, ha a paraméterek száma és típusa is változik. Leggyakrabban akkor használjuk, amikor olyan C könyvtári függvényekhez készítünk felületet, amelyek még nem használják ki a C++ által nyújtott újabb lehetőségeket:

```
int fprintf(FILE*, const char* ...);        // a <cstdio> fejállományból
int execl(const char* ...);                 // UNIX fejállományból
```

A *<cstdarg>* fejállományban szabványos makrókat találunk, melyekkel hozzáférhetünk az ilyen függvények nem meghatározott paramétereihez. Képzeljük el, hogy egy olyan hibafüggvényt írunk, amelynek van egy egész paramétere, ami a hiba súlyosságát jelzi, és ezt tetszőleges hosszúságú (több karakterláncból álló) szöveg követi. Elképzelésünk az, hogy úgy hozzuk létre a hibaüzenetet, hogy minden szót külön karakterlánc paraméterként adunk át. Ezen paraméterek listáját egy *char*-ra hivatkozó „nullpointer" kell, hogy lezárja:

```
extern void error(int ...);
extern char* itoa(int, char[ ]);                    // lásd §6.6.[17]

const char* Null_cp = 0;

int main(int argc, char* argv[ ])
{
  switch (argc) {
  case 1:
          error(0,argv[0],Null_cp);
          break;
  case 2:
          error(0,argv[0],argv[1],Null_cp);
          break;
  default:
          char buffer[8];
          error(1,argv[0], "with",itoa(argc-1,buffer),"arguments", Null_cp);
  }
  // ...
}
```

Az *itoa()* azt a karakterláncot adja vissza, amelyik a függvény egész típusú paraméterének felel meg.

Vegyük észre, hogy ha a *0* egész értéket használtuk volna befejezésként, a kód nem lett volna hordozható: néhány nyelvi megvalósítás a nulla egészt és a nulla mutatót (nullpointer) nem azonos módon ábrázolja. Ez a példa szemlélteti a nehézségeket és azt a többletmunkát, amellyel a programozó szembenéz, amikor a típusellenőrzést „elnyomja" a három pont.

A hibafüggvényt így adhatjuk meg:

```
void error(int severity ...)     // a "severity" (súlyosság) után nullával lezárt char*-ok
                                 // következnek
{
  va_list ap;
  va_start(ap,severity);         // kezdeti paraméterek

  for (;;) {
          char* p = va_arg(ap,char*);
          if (p == 0) break;
          cerr << p << ' ';
  }

  va_end(ap);                    // paraméterek visszaállítása

  cerr << '\n';
  if (severity) exit(severity);
}
```

Először meghatározzuk a *va_list*-et és a *va_start()* meghívásával értéket adunk neki. A *va_start* makró paraméterei a *va_list* neve és az utolsó formális paraméter neve. A *va_arg()* makrót arra használjuk, hogy sorba rendezzük a nem megnevezett paramétereket. A programozónak minden egyes híváskor meg kell adnia egy típust; a *va_arg()* feltételezi, hogy ilyen típusú aktuális paraméter került átadásra, de általában nincs mód arra, hogy ezt biztosítani is tudja. Mielőtt még visszatérnénk egy olyan függvényből, ahol a *va_start()*-ot használtuk, meg kell hívnunk a *va_end()*-et. Ennek az az oka, hogy a *va_start()* úgy módosíthatja a vermet, hogy a visszatérést nem lehet sikeresen véghezvinni. A *va_end()* helyreállítja ezeket a módosításokat.

7.7. Függvényre hivatkozó mutatók

A függvényekkel csak két dolgot csinálhatunk: meghívhatjuk őket és felhasználhatjuk a címüket. Amikor a függvény címét vesszük, az így kapott mutatót használhatjuk arra, hogy meghívjuk a függvényt:

```
void error(string s) { /* ... */ }

void (*efct)(string);          // mutató függvényre

void f()
{
  efct = &error;               // efct az error függvényre mutat
  efct("error");               // error meghívása efct-n keresztül
}
```

A fordítóprogram rá fog jönni, hogy *efct* egy mutató és meghívja az általa mutatott függvényt. Azaz, egy függvényre hivatkozó mutatót nem kötelező a * operátorral feloldanunk. Ugyanígy nem kötelező a & használata sem a függvény címének lekérdezésére:

```
void (*f1)(string) = &error;   // rendben
void (*f2)(string) = error;    // ez is jó; jelentése ugyanaz, mint az &error-nak

void g()
{
  f1("Vasa");                  // rendben
  (*f1)("Mary Rose");          // ez is jó
}
```

A függvényekre hivatkozó mutatók paramétereinek típusát ugyanúgy meg kell adnunk, mint a függvényeknél. A mutatókat használó értékadásokban ügyelni kell a teljes függvény típusára:

```
void (*pf)(string);              // mutató void(string)-re
void f1(string);                 // void(string)
int f2(string);                  // int(string)
void f3(int*);                   // void(int*)

void f()
{
  pf = &f1;                      // rendben
  pf = &f2;                      // hiba: rossz visszatérési típus
  pf = &f3;                      // hiba: rossz paramétertípus

  pf("Héra");                    // rendben
  pf(1);                         // hiba: rossz paramétertípus

  int i = pf("Zeusz");           // hiba: void értékadás int-nek
}
```

A paraméterátadás szabályai a közvetlen függvényhívások és a függvények mutatón keresztül történő meghívása esetében ugyanazok. Gyakran kényelmes, ha nevet adunk egy függvényre hivatkozó mutató típusnak, hogy ne mindig a meglehetősen nehezen érthető deklarációformát használjuk. Íme egy példa egy UNIX-os rendszer-fejállományból:

```
typedef void (*SIG_TYP)(int);          // a <signal.h> fejállományból
typedef void (*SIG_ARG_TYP)(int);
SIG_TYP signal(int, SIG_ARG_TYP);
```

A függvényekre hivatkozó mutatókból álló tömbök gyakran hasznosak. Például az egeret használó szövegszerkesztőm menürendszere az egyes műveleteket jelölő függvényekre hivatkozó mutatókból összeállított tömbökkel van megvalósítva. Itt nincs lehetőségünk, hogy a rendszert részletesen ismertessük, de az alapvető ötlet ez:

```
typedef void (*PF)();

PF edit_ops[ ] = {               // szerkesztőműveletek
  &cut, &paste, &copy, &search
};

PF file_ops[ ] = {               // fájlkezelés
  &open, &append, &close, &write
};
```

Az egér gombjaival kiválasztott menüpontokhoz kapcsolódó műveleteket vezérlő mutatókat így határozhatjuk meg és tölthetjük fel értékkel:

```
PF* button2 = edit_ops;
PF* button3 = file_ops;
```

A teljes megvalósításhoz több információra van szükség ahhoz, hogy minden menüelemet meghatározhassunk. Például tárolnunk kell valahol azt a karakterláncot, amelyik meghatározza a kiírandó szöveget. Ahogy a rendszert használjuk, az egérgombok jelentése gyakran megváltozik a környezettel együtt. Az ilyen változásokat (részben) úgy hajtjuk végre, hogy módosítjuk a gombokhoz kapcsolt mutatók értékét. Amikor a felhasználó kiválaszt egy menüpontot (például a 3-as elemet a 2-es gomb számára), a megfelelő művelet hajtódik végre:

```
button2[2]();          // button2 harmadik függvényének meghívása
```

Akkor tudnánk igazán nagyra értékelni a függvényekre hivatkozó mutatók kifejezőerejét, ha nélkülük próbálnánk ilyen kódot írni – és még jobban viselkedő rokonaik, a virtuális függvények (§12.2.6) nélkül. Egy menüt futási időben úgy módosíthatunk, hogy új függvényeket teszünk a művelettáblába, de új menüket is könnyen létrehozhatunk.

A függvényekre hivatkozó mutatók arra is használhatók, hogy a többalakú (polimorf) eljárások – azaz amelyeket több, különböző típusú objektumra lehet alkalmazni – egyszerű formáját adják:

```
typedef int (*CFT)(const void*, const void*);

void ssort(void* base, size_t n, size_t sz, CFT cmp)
/*
  A "base" vektor "n" elemének rendezése növekvő sorrendbe
  a "cmp" által mutatott összehasonlító függvény segítségével.
  Az elemek "sz" méretűek.

  Shell rendezés (Knuth, 3. kötet, 84.o.)
*/

{
  for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
              for (int j=i-gap; 0<=j; j-=gap) {
                    char* b = static_cast<char*>(base); // szükséges típuskényszerítés
                    char* pj = b+j*sz;            // &base[j]
                    char* pjg = b+(j+gap)*sz;     // &base[j+gap]
                    if (cmp(pjg,pj)<0) {          // base[j] és base[j+gap] felcserélése
                          for (int k=0; k<sz; k++) {
                                char temp = pj[k];
                                pj[k] = pjg[k];
                                pjg[k] = temp;
                          }
                    }
              }
}
```

Az *ssort()* nem ismeri azoknak az objektumoknak a típusát, amelyeket rendez, csak az elemek számát (a tömb méretét), az egyes elemek méretét, és azt a függvényt, melyet meg kell hívnia, hogy elvégezze az összehasonlítást. Az *ssort()* típusát úgy választottuk meg, hogy megegyezzen a szabványos C könyvtári *qsort()* rendező eljárás típusával. A valódi programok a *qsort()*-ot, a C++ standard könyvtárának *sort* algoritmusát (§18.7.1), vagy egyedi rendező eljárást használnak. Ez a kódolási stílus gyakori C-ben, de nem a legelegánsabb módja, hogy ezt az algoritmust C++-ban írjuk le (lásd §13.3, §13.5.2).

Egy ilyen rendező függvényt a következőképpen lehetne egy táblázat rendezésére használni:

```
struct User {
    char* name;
    char* id;
    int dept;
};

User heads[ ] = {
    "Ritchie D.M.",          "dmr",    11271,
    "Sethi R.",              "ravi",   11272,
    "Szymanski T.G.",        "tgs",    11273,
    "Schryer N.L.",          "nls",    11274,
    "Schryer N.L.",          "nls",    11275,
    "Kernighan B.W.",        "bwk",    11276
};

void print_id(User* v, int n)
{
    for (int i=0; i<n; i++)
            cout << v[i].name << '\t' << v[i].id << '\t' << v[i].dept << '\n';
}
```

Először meg kell határoznunk a megfelelő összehasonlító függvényeket, hogy rendezni tudjunk. Az összehasonlító függvénynek negatív értéket kell visszaadnia, ha az első paramétere kisebb, mint a második, nullát, ha paraméterei egyenlőek, egyéb esetben pedig pozitív számot:

```
int cmp1(const void* p, const void* q)          // nevek (name) összehasonlítása
{
    return strcmp(static_cast<const User*>(p)->name,static_cast<const User*>(q)->name);
}

int cmp2(const void* p, const void* q)          // osztályok (dept) összehasonlítása
{
    return static_cast<const User*>(p)->dept - static_cast<const User*>(q)->dept;
}
```

Ez a program rendez és kiír:

```
int main()
{
  cout << "Főnökök ábécésorrendben:\n";
  ssort(heads,6,sizeof(User),cmp1);
  print_id(heads,6);
  cout << '\n';

  cout << "Főnökök osztályok szerint:\n";
  ssort(heads,6,sizeof(User),cmp2);
  print_id(heads,6);
}
```

Egy túlterhelt függvény címét úgy használhatjuk fel, hogy egy függvényre hivatkozó mutatóhoz rendeljük vagy annak kezdőértékül adjuk. Ebben az esetben a cél típusa alapján választunk a túlterhelt függvények halmazából:

```
void f(int);
int f(char);

void (*pf1)(int) = &f;        // void f(int)
int (*pf2)(char) = &f;        // int f(char)
void (*pf3)(char) = &f;       // hiba: nincs void f(char)
```

Egy függvényt egy függvényre hivatkozó mutatón keresztül pontosan a megfelelő paraméter- és visszatérési típusokkal kell meghívni. Ezen típusokra vonatkozóan nincs automatikus konverzió, ha függvényekre hivatkozó mutatókat adunk értékül vagy töltünk fel kezdőértékkel. Ez azt jelenti, hogy

```
int cmp3(const mytype*,const mytype*);
```

nem megfelelő paraméter az *ssort()* számára. Ha *cmp3*-at elfogadnánk az *ssort* paramétereként, megszegnénk azt a vállalást, hogy a *cmp3*-at *mytype** típusú paraméterekkel fogjuk meghívni (lásd még §9.2.5-öt).

7.8. Makrók

A makrók nagyon fontosak a C-ben, de kevesebb a hasznuk a C++-ban. Az első makrókra vonatkozó szabály: ne használjuk őket, ha nem szükségesek. Majdnem minden makró a programozási nyelv, a program, vagy a programozó gyenge pontját mutatja. Mivel átrendezik a programkódot, mielőtt a fordítóprogram látná azt, számos programozási eszköz számára komoly problémát jelentenek. Így ha makrót használunk, számíthatunk arra, hogy az olyan eszközök, mint a hibakeresők, kereszthivatkozás-vizsgálók és hatékonyságvizsgálók gyengébb szolgáltatást fognak nyújtani. Ha makrót kell használnunk, olvassuk el figyelmesen C++-változatunk előfordítójának (preprocessor) hivatkozási kézikönyvét és ne próbáljunk túl okosak lenni. Kövessük azt a szokást, hogy a makrókat úgy nevezzük el, hogy sok nagybetű legyen bennük. A makrók formai követelményeit az §A.11 mutatja be.

Egy egyszerű makrót így adhatunk meg:

```
#define NAME a sor maradék része
```

ahol a NAME szimbólum előfordul, ott kicserélődik a sor maradék részére. Például a

```
named = NAME
```

kifejezést a következő váltja fel:

```
named = a sor maradék része
```

Megadhatunk paraméterekkel rendelkező makrót is:

```
#define MAC(x,y) argument1: x argument2: y
```

Amikor *MAC*-ot használjuk, paraméterként meg kell adnunk két karakterláncot. Ezek *x*-et és *y*-t fogják helyettesíteni, amikor *MAC* behelyettesítődik. Például a

```
expanded = MAC(foo bar, yuk yuk)
```

így alakul át:

```
expanded = argument1: foo bar argument2: yuk yuk
```

A makróneveket nem terhelhetjük túl és a makró-előfordító rekurzív hívásokat sem tud kezelni:

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c) /* problémás?: újbóli definíció, nem túlterhelés */

#define FAC(n) (n>1)?n*FAC(n-1):1          /* problémás: rekurzív makró */
```

A makrók karakterláncokat kezelnek, keveset tudnak a C++ nyelvtanáról és semmit sem a C++ típusairól, illetve a hatókörök szabályairól. A fordítóprogram csak a makró behelyettesített formáját látja, így akkor jelzi a makróban lévő esetleges hibát, amikor a makró behelyettesítődik, és nem akkor, amikor a makrót kifejtjük, ami nagyon homályos hibaüzenetekhez vezet.

Íme néhány lehetséges makró:

```
#define CASE break;case
#define FOREVER for(;;)
```

Néhány teljesen fölösleges makró:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

És néhány veszélyes makró:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

Hogy lássuk, miért veszélyesek, próbáljuk meg behelyettesíteni ezt:

```
int xx = 0;                     // globális számláló

void f()
{
   int xx = 0;                  // lokális változó
   int y = SQUARE(xx+2);        // y=xx+2*xx+2 vagyis y=xx+(2*xx)+2
   INCR_xx;                     // a lokális xx növelése
}
```

Ha makrót kell használnunk, használjuk a *::* hatókör-jelzőt, amikor globális nevekre (§4.9.4) hivatkozunk, és a makró paraméterek előfordulásait tegyük zárójelbe, ahol csak lehetséges:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

Ha bonyolult makrókat kell írnunk, amelyek megjegyzésekre szorulnak, bölcs dolog /* */ megjegyzéseket használnunk, mert a C++ eszközök részeként néha C előfordítókat használnak, ezek viszont nem ismerik a *//* jelölést:

```
#define M2(a) something(a)          /* értelmes megjegyzés */
```

Makrók használatával megtervezhetjük saját, egyéni nyelvünket. Ha azonban ezt a „kibővített nyelvet" részesítjük előnyben a sima C++-szal szemben, az a legtöbb C++ programozó számára érthetetlen lesz. Továbbá a C előfordító egy nagyon egyszerű makró-feldolgozó. Ha valami nem magától értetődőt akarunk csinálni, akkor az vagy lehetetlennek, vagy szükségtelenül nehéznek bizonyulhat. A *const*, *inline*, *template*, *enum* és *namespace* megoldásokat arra szánták, hogy a hagyományos előfordított szerkezeteket kiváltsák:

```
const int answer = 42;
template<class T> inline T min(T a, T b) { return (a<b)?a:b; }
```

Amikor makrót írunk, nem ritka, hogy egy új névre van szükségünk valami számára. Két karakterláncot a *##* makróoperátorral összefűzve például új karakterláncot hozhatunk létre:

```
#define NAME2(a,b) a##b

int NAME2(hack,cah)();
```

Ez a következőt eredményezi a fordítóprogram számára:

```
int hackcah();
```

A

```
#undef X
```

utasítás biztosítja, hogy *X* nevű makró nem lesz definiálva – akkor sem, ha az utasítás előtt szerepelt ilyen. Ez bizonyos védelmet ad a nem kívánt makrók ellen, de nem tudhatjuk, hogy egy kódrészletben mit feltételezzünk *X* hatásairól.

7.8.1. Feltételes fordítás

A makrók egy bizonyos használatát majdnem lehetetlen elkerülni. Az *#ifdef azonosító* direktíva arra utasítja a fordítóprogramot, hogy feltételesen minden bemenetet figyelmen kívül hagyjon, amíg az *#endif* utasítással nem találkozik. Például az

```
int f(int a
#ifdef arg_two
,int b
#endif
);
```

kódrészletből a fordítóprogram ennyit lát (kivéve ha az *arg_two* nevű makrót a *#define* előfordító direktívával korábban definiáltuk):

```
int f(int a
);
```

Ez megzavarja azokat az eszközöket, amelyek ésszerű viselkedést teteleznek fel a programozóról.

Az *#ifdef* legtöbb felhasználása kevésbé bizarr, és ha mérséklettel használják, kevés kárt okoz. Lásd még §9.3.3-at.

Az *#ifdef*-et vezérlő makrók neveit figyelmesen kell megválasztani, hogy ne ütközzenek a szokásos azonosítókkal:

```
struct Call_info {
  Node* arg_one;
  Node* arg_two;
  // ...
};
```

Ez az ártatlannak látszó forrásszöveg zavart fog okozni, ha valaki a következőt írja:

```
#define arg_two x
```

Sajnos a szokványos és elkerülhetetlenül beépítendő fejállományok sok veszélyes és szükségtelen makrót tartalmaznak.

7.9. Tanácsok

[1] Legyünk gyanakvóak a nem *const* referencia paraméterekkel kapcsolatban; ha azt akarjuk, hogy a függvény módosítsa paraméterét, használjunk inkább mutatókat és érték szerinti visszaadást. §5.5.
[2] Használjunk *const* referencia paramétereket, ha a lehető legritkábbra kell csökkentenünk a paraméterek másolását. §5.5.
[3] Használjuk a *const*-ot széleskörűen, de következesen. §7.2.
[4] Kerüljük a makrókat. §7.8.
[5] Kerüljük a nem meghatározott számú paraméterek használatát. §7.6.
[6] Ne adjunk vissza lokális változókra hivatkozó mutatókat vagy ilyen referenciákat. §7.3.
[7] Akkor használjuk a túlterhelést, ha a függvények elvben ugyanazt a műveletet hajtják végre különböző típusokon. §7.4.
[8] Amikor egészekre vonatkozik a túlterhelés, használjunk függvényeket, hogy megszüntessük a többértelműséget. §7.4.3.
[9] Ha függvényre hivatkozó mutató használatát fontolgatjuk, vizsgáljuk meg, hogy egy virtuális függvény (§2.5.5) vagy sablon (§2.7.2) használata nem jobb megoldás-e. §7.7.
[10] Ha makrókat kell használnunk, használjunk csúnya neveket, sok nagybetűvel. §7.8.

7.10. Gyakorlatok

1. (*1) Deklaráljuk a következőket: függvény, amelynek egy karakterre hivatkozó mutató és egy egészre mutató referencia paramétere van és nem ad vissza értéket; ilyen függvényre hivatkozó mutató; függvény, amelynek ilyen mutató paramétere van; függvény, amely ilyen mutatót ad vissza. Írjuk meg azt a függvényt, amelynek egy ilyen mutatójú paramétere van és visszatérési értékként paraméterét adja vissza. Tipp: használjunk *typedef*-et.
2. (*1) Mit jelent a következő sor? Mire lehet jó?

```
typedef int (&rifii) (int, int);
```

3. (*1,5) Írjunk egy „Helló, világ!”-szerű programot, ami parancssori paraméterként vesz egy nevet és kiírja, hogy „Helló, *név*!”. Módosítsuk ezt a programot úgy, hogy tetszőleges számú név paramétere lehessen és mondjon hellót minden egyes névvel.
4. (*1,5) Írjunk olyan programot, amely tetszőleges számú fájlt olvas be, melyek nevei parancssori paraméterként vannak megadva, és kiírja azokat egymás után a *cout*-ra. Mivel ez a program összefűzi a paramétereit, hogy megkapja a kimenetet, elnevezhetjük *cat*-nek.
5. (*2) Alakítsunk egy kis C programot C++ programmá. Módosítsuk a fejállományokat úgy, hogy minden meghívott függvény deklarálva legyen és határozzuk meg minden paraméter típusát. Ahol lehetséges, cseréljük ki a *#define* utasításokat *enum*-ra, *const*-ra vagy *inline*-ra. Távolítsuk el az *extern* deklarációkat a *.c* fájlokból, és ha szükséges, alakítsunk át minden függvényt a C++ függvények formai követelményeinek megfelelően. Cseréljük ki a *malloc()* és *free()* hívásokat *new*-ra, illetve *delete*-re. Távolítsuk el a szükségtelen konverziókat.
6. (*2) Írjuk újra az *ssort()*-ot (§7.7) egy hatékonyabb rendezési algoritmus felhasználásával. Tipp: *qsort()*.
7. (*2,5) Vegyük a következőt:

```
struct Tnode {
        string word;
        int count;
        Tnode* left;
        Tnode* right;
};
```

   Írjunk függvényt, amellyel új szavakat tehetünk egy *Tnode*-okból álló fába. Írjunk függvényt, amely kiír egy *Tnode*-okból álló fát. Írjunk olyan függvényt, amely egy *Tnode*-okból álló fát úgy ír ki, hogy a szavak ábécésorrendben vannak. Módosítsuk a *Tnode*-ot, hogy (csak) egy mutatót tároljon, ami egy tetszőlegesen hosszú szóra mutat, amit a szabad tár karaktertömbként tárol, a *new* segítségével. Módosítsuk a függvényeket, hogy a *Tnode* új definicióját használják.
8. (*2,5) Írjunk függvényt, amely kétdimenziós tömböt invertál. Tipp: §C.7.
9. (*2) Írjunk titkosító programot, ami a *cin*-ről olvas és a kódolt karaktereket kiírja a *cout*-ra. Használhatjuk a következő, egyszerű titkosító sémát: *c* karakter titkosított formája legyen *c^key[i]*, ahol *key* egy karakterlánc, amely parancssori paraméterként adott. A program ciklikus módon használja a *key*-ben lévő karaktereket, amíg a teljes bemenetet el nem olvasta. Ha nincs megadva *key* (vagy a paraméter null-karakterlánc), a program ne végezzen titkosítást.

10. (*3,5) Írjunk programot, ami segít megfejteni a §7.10[9]-ben leírt módszerrel titkosított üzeneteket, anélkül, hogy tudná a kulcsot. Tipp: lásd David Kahn: *The Codebreakers*, Macmillan, 1967, New York; 207-213. o.
11. (*3) Írjunk egy *error* nevű függvényt, amely *%s*, *%c* és *%d* kifejezéseket tartalmazó, *printf* stílusú, formázott karakterláncokat vesz paraméterként és ezen kívül tetszőleges számú paramétere lehet. Ne használjuk a *printf()*-et. Nézzük meg a §21.8-at, ha nem tudjuk, mit jelent a *%s*, *%c* és *%d*. Használjuk a *<cstdarg>*-ot.
12. (*1) Hogyan választanánk meg a *typedef* használatával meghatározott függvényekre hivatkozó mutatótípusok neveit?
13. (*2) Nézzünk meg néhány programot, hogy elképzelésünk lehessen a mostanság használatos nevek stílusának változatosságáról. Hogyan használják a nagybetűket? Hogyan használják az aláhúzást? Mikor használnak rövid neveket, mint amilyen az *i* és *x*?
14. (*1) Mi a hiba ezekben a makrókban?

```
#define PI = 3.141593;
#define MAX(a,b) a>b?a:b
#define fac(a) (a)*fac((a)-1)
```

15. (*3) Írjunk makrófeldolgozót, amely egyszerű makrókat definiál és cserél ki (ahogy a C előfordító teszi). Olvassunk a *cin*-ről és írjunk a *cout*-ra. Először ne próbáljunk paraméterekkel rendelkező makrókat kezelni. Tipp: az asztali számológép (§6.1) tartalmaz egy szimbólumtáblát és egy lexikai elemzőt, amit módosíthatunk.
16. (*2) Írjuk meg magunk a *print()* függvényt a §7.5-ből.
17. (*2) Adjunk hozzá a §6.1 pontban lévő asztali számológéphez olyan függvényeket, mint az *sqrt()*, *log()*, és *sin()*. Tipp: adjuk meg előre a neveket, a függvényeket pedig függvényre hivatkozó mutatókból álló tömbön keresztül hívjuk meg. Ne felejtsük el ellenőrizni a függvényhívások paramétereit.
18. (*1) Írjunk olyan faktoriális függvényt, amely nem hívja meg önmagát. Lásd még §11.14[6]-ot.
19. (*2) Írjunk függvényeket, amelyek egy napot, egy hónapot, és egy évet adnak hozzá egy *Date*-hez, ahogy azt a §6.6[13]-ban leírtuk. Írjunk függvényt, ami megadja, hogy egy adott *Date* a hét melyik napjára esik. Írjunk olyan függvényt, ami megadja egy adott *Date*-re következő első hétfő *Date*-jét.

8

Névterek és kivételek

„– Ez a 787-es év!
– I.sz.?”
(Monty Python)

„Nincs olyan általános szabály, ami
alól ne lenne valamilyen kivétel.”
(Robert Burton)

Modulok, felületek és kivételek • Névterek • *using* • *using namespace* • Névütközések feloldása • Nevek keresése • Névterek összefűzése • Névtér-álnevek • Névterek és C kód • Kivételek • *throw* és *catch* • A kivételek és a programok szerkezete • Tanácsok • Gyakorlatok

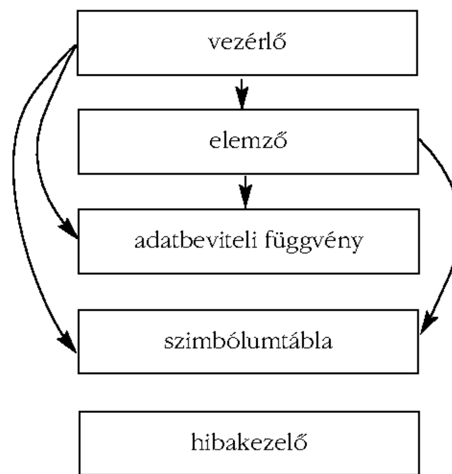
8.1. Modulok és felületek

Minden valóságos program különálló részekből áll. Még az egyszerű „Helló, világ!” program is legalább két részre osztható: a felhasználói kódra, ami a *Helló, világ!* kiírását kéri, és a kiírást végző I/O rendszerre.

Vegyük a számológép példáját a §6.1-ből. Láthatjuk, hogy 5 részből áll:

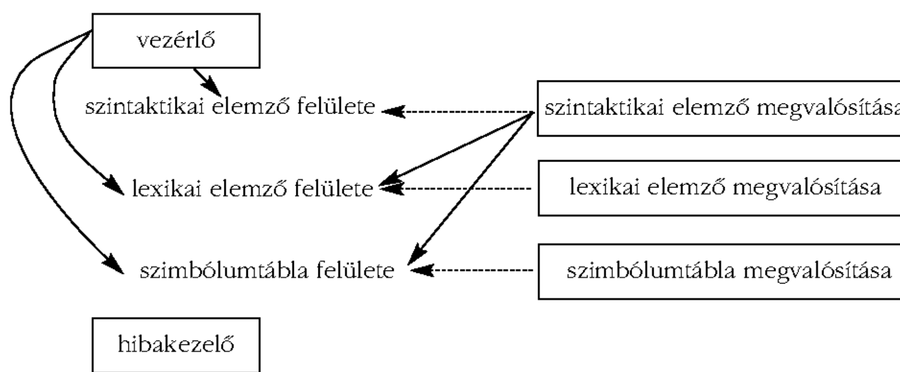
1. A (szintaktikai) elemzőből (parser), ami a szintaktikai elemzést végzi,
2. az adatbeviteli függvényből vagy lexikai elemzőből (lexer), ami a karakterekből szimbólumokat hoz létre
3. a (karakterlánc, érték) párokat tároló szimbólumtáblából
4. a *main()* vezérlőből
5. és a hibakezelőből

Ábrával:



A fenti ábrában a nyíl jelentése: „felhasználja”. Az egyszerűsítés kedvéért nem jelöltem, hogy mindegyik rész támaszkodik a hibakezelésre. Az igazat megvallva a számológépet három részből állóra terveztem, a vezérlőt és a hibakezelőt a teljesség miatt adtam hozzá.

Amikor egy modul felhasznál egy másikat, nem szükséges, hogy mindent tudjon a felhasznált modulról. Ideális esetben a modulok legnagyobb része nem ismert a felhasználó elem számára. Következésképpen különbséget teszünk a modul és a modul felülete (interfész) között. A szintaktikai elemző például közvetlenül csak az adatbeviteli függvény felületére, nem pedig a teljes lexikai elemzőre támaszkodik. Az adatbeviteli függvény csak megvalósítja a felületében közzétett szolgáltatásokat. Ezt ábrával így mutathatjuk be:



A szaggatott vonalak jelentése: „megvalósítja". Ez tekinthető a program valódi felépítésének. Nekünk, programozóknak, az a feladatunk, hogy ezt hű módon adjuk vissza a kódban. Ha ezt tesszük, a kód egyszerű, hatékony, érthető, és könnyen módosítható lesz, mert közvetlenül fogja tükrözni eredeti elképzelésünket.

A következő részben bemutatjuk, hogyan lehet a számológép program logikai felépítését világosan kifejezni, a §9.3 pontban pedig azt, hogyan rendezhetjük el úgy a program forrásszövegét, hogy abból előnyünk származzon. A számológép kis program; a „valódi életben" nem használnám olyan mértékben a névtereket és a külön fordítást (§2.4.1, §9.1), mint itt. Most csak azért használjuk ezeket, hogy nagyobb programok esetében is hasznos módszereket mutassunk be, anélkül, hogy belefulladnánk a kódba. A valódi programokban minden modul, amelyet önálló névtér jelöl, gyakran függvények, osztályok, sablonok stb. százait tartalmazza.

A nyelvi eszközök bő választékának bemutatásához több lépésben bontom modulokra a számológépet. Az igazi programoknál nem valószínű, hogy ezen lépések mindegyikét végrehajtanánk. A tapasztalt programozó már az elején kiválaszthat egy „körülbelül megfelelő" tervet. Ahogy azonban a program az évek során fejlődik, nem ritkák a drasztikus szerkezeti változtatások.

A hibakezelés mindenütt fontos szerepet tölt be a program szerkezetében. Amikor egy programot modulokra bontunk vagy egy programot modulokból hozunk létre, ügyelnünk kell arra, hogy a hibakezelés okozta modulok közötti függőségekből minél kevesebb legyen. A C++ kivételeket nyújt arra a célra, hogy elkülönítsük a hibák észlelését és jelzését azok kezelésétől. Ezért miután tárgyaltuk, hogyan ábrázolhatjuk a modulokat névterekként (§8.2), bemutatjuk, hogyan használhatjuk a kivételeket arra, hogy a modularitást tovább javítsuk (§8.3).

A modularitás fogalma sokkal több módon értelmezhető, mint ahogy ebben és a következő fejezetben tesszük. Programjainkat például részekre bonthatjuk párhuzamosan végrehajtott és egymással kapcsolatot tartó folyamatok segítségével is. Ugyanígy az önálló címterek (address spaces) és a címterek közötti információs kapcsolat is olyan fontos témakörök, amelyeket itt nem tárgyalunk. Úgy gondolom, a modularitás ezen megközelítései nagyrészt egymástól függetlenek és ellentétesek. Érdekes módon minden rendszer könnyen modulokra bontható. A nehézséget a modulok közötti biztonságos, kényelmes és hatékony kapcsolattartás biztosítása jelenti.

8.2. Névterek

A névterek (namespace) mindig valamilyen logikai csoportosítást fejeznek ki. Azaz, ha egyes deklarációk valamilyen jellemző alapján összetartoznak, akkor ezt a tényt kifejezhetjük úgy is, hogy közös névtérbe helyezzük azokat. A számológép elemzőjének (§6.1.1) deklarációit például a *Parser* névtérbe tehetjük:

```
namespace Parser {
  double expr(bool);
  double prim(bool get) { /* ... */ }
  double term(bool get) { /* ... */ }
  double expr(bool get) { /* ... */ }
}
```

Az *expr()* függvényt először deklaráljuk és csak később fejtjük ki, hogy megtörjük a §6.1.1-ben leírt függőségi kört.

A számológép bemeneti részét szintén önálló névtérbe helyezhetjük:

```
namespace Lexer {
  enum Token_value {
          NAME,          NUMBER,          END,
          PLUS='+',      MINUS='-',       MUL='*',      DIV='/',
          PRINT=';',     ASSIGN='=',      LP='(',       RP=')'
  };

  Token_value curr_tok;
  double number_value;
  string string_value;

  Token_value get_token() { /* ... */ }
}
```

A névterek ilyen használata elég nyilvánvalóvá teszi, mit nyújt a lexikai és a szintaktikai elemző a felhasználó programelemnek. Ha azonban a függvények forráskódját is a névterekbe helyeztem volna, a szerkezet zavarossá vált volna. Ha egy valóságos méretű névtér deklarációjába beletesszük a függvénytörzseket is, általában több oldalas (képernyős) információn kell átrágnunk magunkat, mire megtaláljuk, milyen szolgáltatások vannak felkínálva, azaz, hogy megtaláljuk a felületet.

Külön meghatározott felületek helyett olyan eszközöket is biztosíthatunk, amelyek kinyerik a felületet egy modulból, amely a megvalósítást tartalmazza. Ezt nem tekintem jó megoldásnak. A felületek meghatározása alapvető tervezési tevékenység (lásd §23.4.3.4-et), hiszen egy modul a különböző programelemek számára különböző felületeket nyújthat, ráadásul a felületet sokszor már a megvalósítás részleteinek kidolgozása előtt megtervezik.

Íme a *Parser* egy olyan változata, ahol a felületet (interfész) elkülönítjük a megvalósítástól (implementáció):

```
namespace Parser {
   double prim(bool);
   double term(bool);
   double expr(bool);
}

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

Vegyük észre, hogy a felület és a lényegi programrész szétválasztásának eredményeként most minden függvénynek pontosan egy deklarációja és egy definíciója van. A felhasználó programelemek csak a deklarációkat tartalmazó felületet fogják látni. A program megvalósítását – ebben az esetben a függvénytörzseket – a felhasználó elem látókörén kívül helyezzük el.

Láthattuk, hogy egy tagot megadhatunk a névtér meghatározásán belül, és kifejthetjük később, a *névtér_neve::tag_neve* jelölést használva.

A névtér tagjait a következő jelölés használatával kell bevezetni:

```
namespace névtér_név {
   // deklaráció és definíciók
}
```

A névtérdefiníción kívül új tagot nem adhatunk meg minősítő formában:

```
void Parser::logical(bool);       // hiba: nincs logical() a Parser névtérben
```

A cél az, hogy könnyen meg lehessen találni minden nevet a névtérdeklarációban, és hogy a gépelési, illetve az eltérő típusokból adódó hibákat észrevegyük:

```
double Parser::trem(bool);        // hiba: nincs trem() a Parser névtérben
double Parser::prim(int);         // hiba: Parser::prim() logikai paraméterű
```

A névtér (namespace) egyben hatókör (scope), vagyis nagyon alapvető és viszonylag egyszerű fogalom. Minél nagyobb egy program, annál hasznosabbak a névterek, hogy kifejezzék a program részeinek logikai elkülönítését. A közönséges lokális hatókörök, a globális hatókörök és az osztályok maguk is névterek (§C.10.3). Ideális esetben egy program minden eleme valamilyen felismerhető logikai egységhez (modulhoz) tartozik. Ezért – elméletileg – egy bonyolultabb program minden deklarációját önálló névterekbe kellene helyezni, melyek neve a programban betöltött logikai szerepet jelzi. A kivétel a *main()*, amelynek globálisnak kell lennie, hogy a futási idejű környezet felismerje (§8.3.3).

8.2.1. Minősített nevek

A névterek külön hatókört alkotnak. Az általános hatókör-szabályok természetesen rájuk is vonatkoznak, így ha egy nevet előzetesen a névtérben vagy egy körülvevő blokkban adtunk meg, minden további nehézség nélkül használhatjuk. Másik névtérből származó nevet viszont csak akkor használhatunk, ha minősítjük névterének nevével:

```
double Parser::term(bool get)                  // figyeljük meg a Parser:: minősítőt
{
  double left = prim(get);                     // nem kell minősítő

  for (;;)
          switch (Lexer::curr_tok) {           // figyeljük meg a Lexer:: minősítőt
          case Lexer::MUL:                     // figyeljük meg a Lexer:: minősítőt
                  left *= prim(true);          // nem kell minősítő
          // ...
          }
  // ...
}
```

A *Parser* minősítőre itt azért van szükség, hogy kifejezzük, hogy ez a *term()* az, amelyet a *Parser*-ben bevezettünk, és nem valamilyen más globális függvény. Mivel a *term()*

a *Parser* tagja, nem kell minősítenie a *prim()*-et. Ha azonban a *Lexer* minősítőt nem tesszük ki, a fordítóprogram a *curr_tok* változót úgy tekinti, mintha az nem deklarált lenne, mivel a *Lexer* névtér tagjai nem tartoznak a *Parser* névtér hatókörébe.

8.2.2. Using deklarációk

Ha egy név gyakran használatos saját névterén kívül, bosszantó lehet állandóan minősíteni névterének nevével. Vegyük a következőt:

```
double Parser::prim(bool get)           // elemi szimbólumok kezelése
{
  if (get) Lexer::get_token();

  switch (Lexer::curr_tok) {
  case Lexer::NUMBER:                   // lebegőpontos konstans
        Lexer::get_token();
        return Lexer::number_value;
  case Lexer::NAME:
  {     double& v = table[Lexer::string_value];
        if (Lexer::get_token() == Lexer::ASSIGN) v = expr(true);
        return v;
  }
  case Lexer::MINUS:                    // mínusz előjel (egyoperandusú mínusz)
        return -prim(true);
  case Lexer::LP:
  {     double e = expr(true);
        if (Lexer::curr_tok != Lexer::RP) return Error::error(") szükséges");
        Lexer::get_token();             // ')' lenyelése
        return e;
  }
  case Lexer::END:
        return 1;
  default:
        return Error::error("elemi szimbólum szükséges");
  }
}
```

A *Lexer* minősítés ismételgetése igen fárasztó, de ki lehet küszöbölni egy *using* deklarációval, amellyel egy adott helyen kijelentjük, hogy az ebben a hatókörben használt *get_token* a *Lexer get_token*-je:

```
double Parser::prim(bool get)           // elemi szimbólumok kezelése
{
  using Lexer::get_token;               // a Lexer get_token-jének használata
  using Lexer::curr_tok;                // a Lexer curr_tok-jának használata
  using Error::error;                   // az Error error-jának használata
```

```
  if (get) get_token();

  switch (curr_tok) {
  case Lexer::NUMBER:                          // lebegőpontos konstans
          get_token();
          return Lexer::number_value;
  case Lexer::NAME:
  {       double& v = table[Lexer::string_value];
          if (get_token() == Lexer::ASSIGN) v = expr(true);
          return v;
  }
  case Lexer::MINUS:                           // mínusz előjel
          return -prim(true);
  case Lexer::LP:
  {       double e = expr(true);
          if (curr_tok != Lexer::RP) return error("') szükséges");
          get_token();                         // ')' lenyelése
          return e;
  }
  case Lexer::END:
          return 1;
  default:
          return error("elemi szimbólum szükséges");
  }
}
```

A *using* direktíva egy lokális szinonímát vezet be.

A lokális szinonímákat általában célszerű a lehető legszűkebb hatókörrel használni, hogy elkerüljük a tévedéseket. A mi esetünkben azonban az elemző minden függvénye ugyanazokat a neveket használja a többi modulból, így a *using* deklarációkat elhelyezhetjük a *Parser* névtér meghatározásában is:

```
namespace Parser {
  double prim(bool);
  double term(bool);
  double expr(bool);

  using Lexer::get_token;       // a Lexer get_token-jének használata
  using Lexer::curr_tok;        // a Lexer curr_tok-jának használata
  using Error::error;           // az Error error-jának használata
}
```

Így a *Parser* függvényeit majdnem az eredeti változatukhoz (§6.1.1) hasonlóra egyszerűsíthetjük:

```
double Parser::term(bool get)          // szorzás és osztás
{
  double left = prim(get);

  for (;;)
        switch (curr_tok) {
        case Lexer::MUL:
              left *= prim(true);
              break;
        case Lexer::DIV:
              if (double d = prim(true)) {
                    left /= d;
                    break;
              }
              return error("osztás 0-val");
        default:
              return left;
        }
}
```

Azt is megtehetnénk, hogy a lexikai szimbólumok (token, nyelvi egység) neveit a *Parser* névtérbe is bevezetjük. Azért hagyjuk őket minősített alakban, hogy emlékeztessenek, a *Parser* a *Lexer*-re támaszkodik.

8.2.3. Using direktívák

Mit tehetünk, ha célunk az, hogy a *Parser* függvényeit annyira leegyszerűsítsük, hogy pontosan olyanok legyenek, mint eredeti változataik? Egy nagy program esetében ésszerűnek tűnik, hogy egy előző, kevésbé moduláris változatát névtereket használva alakítsuk át.

A *using* direktíva majdnem ugyanúgy teszi elérhetővé egy névtér neveit, mintha azokat a névterükön kívül vezettük volna be (§8.2.8):

```
namespace Parser {
  double prim(bool);
  double term(bool);
  double expr(bool);

  using namespace Lexer;     // a Lexer összes nevét elérhetővé teszi
  using namespace Error;     // az Error összes nevét elérhetővé teszi
}
```

Ez lehetővé teszi számunkra, hogy a *Parser* függvényeit pontosan úgy írjuk meg, ahogy azt eredetileg tettük (§6.1.1):

```
double Parser::term(bool get)                  // szorzás és osztás
{
  double left = prim(get);

  for (;;)
        switch (curr_tok) {                           // a Lexer-beli curr_tok
        case MUL:                              // a Lexer-beli MUL
              left *= prim(true);
              break;
        case DIV:                              // a Lexer-beli DIV
              if (double d = prim(true)) {
                    left /= d;
                    break;
              }
              return error("osztás 0-val");          // az Error-beli error
        default:
              return left;
        }
}
```

A *using* direktívák a névterekben más névterek beépítésére használhatók (§8.2.8), függvényekben jelölésbeli segítségként vehetők biztonságosan igénybe (§8.3.3.1). A globális *using* direktívák a nyelv régebbi változatairól való átállásra szolgálnak (§8.2.9), egyébként jobb, ha kerüljük őket.

8.2.4. Több felület használata

Világos, hogy a *Parser* számára létrehozott névtér nem a felület, amit a *Parser* a felhasználó programelem számára nyújt. Inkább olyan deklarációhalmaznak tekinthetjük, ami az egyes elemző függvények kényelmes megírásához szükséges. A *Parser* felülete a felhasználó elemek számára sokkal egyszerűbb kellene, hogy legyen:

```
namespace Parser {
  double expr(bool);
}
```

Szerencsére a két névtér-meghatározás együttesen létezhet, így mindkettő felhasználható ott, ahol az a legmegfelelőbb. Láthatjuk, hogy a *Parser* névtér két dolgot nyújt:

[1] Közös környezetet az elemzőt megvalósító függvények számára
[2] Külső felületet, amit az elemző a felhasználó programelem rendelkezésére bocsát

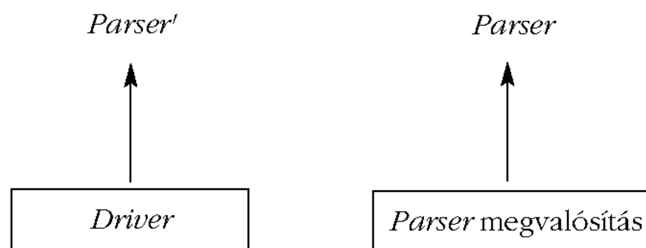
Ennek értelmében a *main()* vezérlőkód csak a következőt kell, hogy lássa:

```
namespace Parser {                    // felhasználói felület
  double expr(bool);
}
```

Bármelyik felületet is találtuk a legjobbnak az elemző függvények közös környezetének ábrázolására, a függvényeknek látniuk kell azt:

```
namespace Parser {              // felület a megvalósításhoz
  double prim(bool);
  double term(bool);
  double expr(bool);

  using Lexer::get_token;       // a Lexer get_token-jének használata
  using Lexer::curr_tok;        // a Lexer curr_tok-jának használata
  using Error::error;           // az Error error-jának használata
}
```

Ábrával:



A nyilak „a … által nyújtott felületen alapul” viszonyokat fejezik ki.

A *Parser'* (Parser prime) a felhasználó programelemek számára nyújtott szűk felület; nem C++ azonosító. Szándékosan választottam, hogy jelöljem, ennek a felületnek nincs külön neve a programban. A külön nevek hiánya nem okozhat zavart, mert a programozók az egyes felületek számára különböző és maguktól értetődő neveket találnak ki, és mert a program fizikai elrendezése (lásd §9.3-at) természetesen különböző (fájl)neveket ad.

A programozói felület nagyobb a felhasználóknak nyújtottnál. Ha ez a felület egy valódi rendszer valóságos méretű moduljának felülete lenne, sokkal gyakrabban változna, mint a felhasználók által látható felület. Fontos, hogy a modulokat használó függvényeket (ebben az esetben a *Parser*-t használó *main()*-t) elkülönítsük az ilyen módosításoktól.

A két felület ábrázolására nem kell önálló névtereket használnunk, de ha akarnánk, megtehetnénk. A felületek megtervezése az egyik legalapvetőbb tevékenység, de kétélű fegyver. Következésképpen érdemes végiggondolni, valójában mit próbálunk megvalósítani, és több megoldást is kipróbálni.

Az itt bemutatott megoldás az általunk megtekintettek közül a legegyszerűbb és gyakran a legjobb. Legfőbb gyengéje, hogy a két felület neve nem különbözik, valamint hogy a fordítóprogram számára nem áll rendelkezésre elegendő információ, hogy ellenőrizze a névtér két definiciójának következetességét. A fordítóprogram azonban rendszerint akkor is megpróbálja ellenőrizni az összefüggéseket, ha erre nincs mindig lehetősége, a szerkesztőprogram pedig észreveszi a legtöbb olyan hibát, amin a fordítóprogram átsiklott.

Az itt bemutatott megoldást használom a fizikai modularitás (§9.3) tárgyalására is, és ezt ajánlom arra az esetre is, amikor nincsenek további logikai megszorítások (lásd még §8.2.7-et).

8.2.4.1. Felülettervezési módszerek

A felületek célja az, hogy a lehetséges mértékig csökkentsék a programok különböző részei között fennálló függőségeket. A kisebb felület könnyebben érthető rendszerhez vezet, melynek adatrejtési tulajdonságai jobbak, könnyebben módosítható és gyorsabban lefordítható.

Amikor a függőségeket nézzük, fontos emlékeznünk arra, hogy a fordítóprogramok és a programozók az alábbi egyszerű hozzáállással viszonyulnak hozzájuk: „ha egy definíció az X pontról látható (a hatókörben van), akkor bármi, ami az X pontban van leírva, bármitől függhet, ami abban a definícióban lett meghatározva”. Persze a helyzet általában nem ennyire rossz, mert a legtöbb definíció a legtöbb kód számára nem bír jelentőséggel. Korábbi definícióinkat adottnak véve vegyük a következőt:

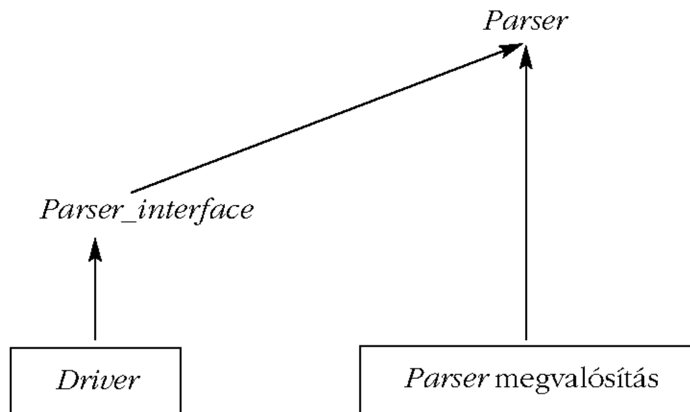
```
namespace Parser {          // felület a megvalósításhoz
  // ...
  double expr(bool);
  // ...
}

int main()
{
  // ...
  Parser::expr(false);
  // ...
}
```

A *main()* függvény csak a *Parser::expr()* függvénytől függ, de időre, gondolkodásra, számolgatásra stb. van szükség ahhoz, hogy erre rájöjjünk. Következésképpen a valóságos méretű programok esetében a programozók és a fordítási rendszerek többnyire „biztosra mennek" és feltételezik, hogy ahol előfordulhat függőség, ott elő is fordul, ami teljesen ésszerű megközelítés. Célunk ezért az, hogy úgy fejezzük ki programunkat, hogy a lehetséges függőségek halmazát a valóban érvényben levő függőségek halmazára szűkítjük.

Először megpróbáljuk a magától értetődőt: a már meglévő megvalósítási felület segítségével az elemző számára felhasználói felületet határozunk meg:

```
namespace Parser {                  // felület a megvalósításhoz
  // ...
  double expr(bool);
  // ...
}

namespace Parser_interface {        // felület a felhasználóknak
  using Parser::expr;
}
```

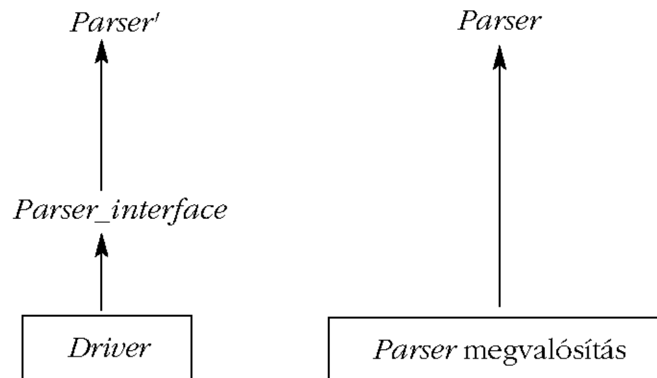
Nyilvánvaló, hogy a *Parser_interface*-t használó programelemek kizárólag – és csupán közvetett módon – a *Parser::expr()* függvénytől függnek. Mégis, ha egy pillantást vetünk a függőségek ábrájára, a következőt látjuk:



Most a *Driver* (a vezérlő) tűnik sebezhetőnek a *Parser* felület változásaival szemben, pedig azt hittük, jól elszigeteltük tőle. Még a függőség ilyen megjelenése sem kívánatos, így megszorítjuk a *Parser_interface* függőségét a *Parser*-től, úgy, hogy a megvalósítási felületnek csak az elemző számára lényeges részét (ezt korábban *Parser'*-nek neveztük) tesszük láthatóvá ott, ahol a *Parser_interface*-t meghatározzuk:

```
namespace Parser {                    // felület a felhasználóknak
  double expr(bool);
}

namespace Parser_interface {          // eltérő nevű felület a felhasználóknak
  using Parser::expr;
}
```

Ábrával:



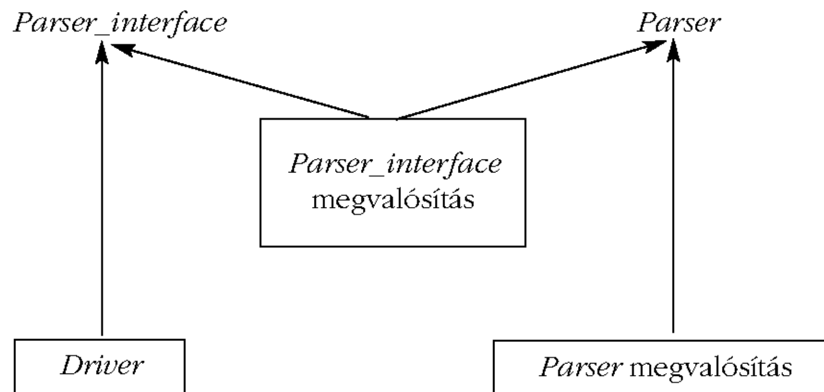
A *Parser* és a *Parser'* egységességét biztosítandó, az egyetlen fordítási egységen dolgozó fordítóprogram helyett ismét a fordítási rendszer egészére támaszkodunk. Ez a megoldás csak abban különbözik a §8.2.4-ben szereplőtől, hogy kiegészül a *Parser_interface* névtérrel. Ha akarnánk, a *Parser_interface*-t egy saját *expr()* függvénnyel konkrétan is ábrázolhatnánk:

```
namespace Parser_interface {
  double expr(bool);
}
```

Most a *Parser*-nek nem kell a hatókörben lennie, hogy meghatározhassuk a *Parser_interface*-t. Csak ott kell „láthatónak" lennie, ahol a *Parser_interface::expr()* függvényt kifejtjük:

```
double Parser_interface::expr(bool get)
{
  return Parser::expr(get);
}
```

Az utóbbi változatot ábrával így szemléltethetjük:



A függőségeket ezzel a lehető legkevesebbre csökkentettünk. Mindent kifejtettünk és megfelelően elneveztünk. Mégis, ezt a megoldást a legtöbb esetben túlzónak találhatjuk.

8.2.5. A névütközések elkerülése

A névterek logikai szerkezetek kifejezésére valók. A legegyszerűbb eset, amikor két személy által írt kódot kell megkülönböztetnünk. Ez gyakran fontos gyakorlati jelentőséggel bír. Ha csak egyetlen globális hatókört használunk, igen nehéz lesz a programot különálló részekből létrehozni. Az a probléma merülhet fel, hogy az önállónak feltételezett részek mindegyike ugyanazokat a neveket használja, így amikor egyetlen programban egyesítjük azokat, a nevek ütközni fognak. Vegyük a következőt:

```
// my.h:
  char f(char);
  int f(int);
  class String { /* ... */ };

// your.h:
  char f(char);
  double f(double);
  class String { /* ... */ };
```

Ha a fentieket meghatározzuk, egy harmadik személy csak nehezen használhatja egyszerre a *my.h*-t és a *your.h*-t is. A kézenfekvő megoldás, hogy mindkét deklarációhalmazt saját, külön névtérbe helyezzük:

```
namespace My {
  char f(char);
  int f(int);
  class String { /* ... */ };
}

namespace Your {
  char f(char);
  double f(double);
  class String { /* ... */ };
}
```

Most már alkalmazhatjuk a *My* és a *Your* deklarációit, ha minősítőket (§8.2.1), *using* deklarációkat (§8.2.2) vagy *using* direktívákat (§8.2.3) használunk.

8.2.5.1. Névtelen névterek

Gyakran hasznos deklarációk halmazát névtérbe helyezni, pusztán azért, hogy védekezzünk a lehetséges névütközésekkel szemben. A célunk az, hogy a kód helyileg maradjon érvényes, nem pedig az, hogy felületet nyújtsunk a felhasználóknak:

```
#include "header.h"
namespace Mine {
  int a;
  void f() { /* ... */ }
  int g() { /* ... */ }
}
```

Mivel nem akarjuk, hogy a *Mine* név „ismert” legyen az adott környezeten kívül is, nem érdemes olyan felesleges globális nevet kitalálni, amely véletlenül ütközhet valaki más nevevel. Ilyen esetben a névteret névtelenül hagyhatjuk:

```
#include "header.h"
namespace {
  int a;
  void f() { /* ... */ }
  int g() { /* ... */ }
}
```

Világos, hogy kell lennie valamilyen módszernek arra is, hogy kívülről férhessünk hozzá egy névtelen névtér (unnamed namespace) tagjaihoz. A névtelen névtérhez tartozik egy rejtett *using* direktíva is. Az előző deklaráció egyenértékű a következővel:

```
namespace $$$ {
  int a;
  void f() { /* ... */ }
  int g() { /* ... */ }
}
using namespace $$$;
```

Itt *$$$* valamilyen név, amely egyedi abban a hatókörben, ahol a névteret meghatároztuk. A különböző fordítási egységekben lévő névtelen névterek mindig különbözőek. Ahogy azt szerettük volna, nincs mód arra, hogy egy névtelen névtér egy tagját egy másik fordítási egységből megnevezhessük.

8.2.6. Nevek keresése

Egy *T* típusú paraméterrel rendelkező függvényt általában a *T*-vel azonos névtérben szokás megadni. Következésképpen ha egy függvényt nem találunk meg használati környezetében, akkor paramétereinek névterében fogjuk keresni:

```
namespace Chrono {
  class Date { /* ... */ };
```

```
  bool operator==(const Date&, const std::string&);

  std::string format(const Date&);      // string ábrázolás
  // ...
}

void f(Chrono::Date d, int i)
{
  std::string s = format(d);            // Chrono::format()
  std::string t = format(i);            // hiba: a hatókörben nincs format()
}
```

Ez a keresési szabály – a minősítők használatával ellentétben – sok gépeléstől kíméli meg a programozót, és nem is „szennyezi” úgy a névteret, mint a *using* direktíva (§8.2.3). Alkalmazása különösen fontos az operátorok operandusai (§11.2.4) és a sablonparaméterek (§C.13.8.4) esetében, ahol a minősítők használata nagyon fárasztó lehet.

Vegyük észre, hogy maga a névtér a hatókörben kell, hogy legyen, a függvényt pedig csak akkor találhatjuk meg és használhatjuk fel, ha előbb bevezettük.

Természetesen egy függvény több névtérből is kaphat paramétereket:

```
void f(Chrono::Date d, std::string s)
{
  if (d == s) {
        // ...
  }
  else if (d == "1914 augusztus 4") {
        // ...
  }
}
```

Az ilyen esetekben a függvényt a fordítóprogram a szokásos módon, a hívás hatókörében, illetve az egyes paraméterek névterében (beleértve a paraméterek osztályát és alaposztályát is) keresi, és minden talált függvényre elvégzi a túlterhelés feloldását (§7.4). Nevezetesen, a fordító a *d==s* hívásnál az *operator==*-t az *f()*-et körülvevő hatókörben, az (==-t *string*-ekre meghatározó) *std* névtérben, és a *Chrono* névtérben keresi. Létezik egy *std::operator==()*, de ennek nincs *Date* paramétere, ezért a *Chrono::operator==()*-t használja, amelynek viszont van. Lásd még §11.2.4-et.

Amikor egy osztálytag meghív egy névvel rendelkező függvényt, az osztály és bázisosztályának tagjai előnyben részesülnek azokkal a függvényekkel szemben, melyeket a fordítóprogram a paraméterek típusa alapján talált. Az operátoroknál más a helyzet (§11.2.1, §11.2.4).

8.2.7. Névtér-álnevek

Ha a felhasználók névtereiknek rövid neveket adnak, a különböző névterek nevei könnyebben ütközhetnek:

```
namespace A {        // rövid név, (előbb-utóbb) ütközni fog
  // ...
}

A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

Valódi kódban viszont általában nem célszerű hosszú névtérneveket használni:

```
namespace American_Telephone_and_Telegraph {        // túl hosszú
  // ...
}

American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

A dilemmát úgy oldhatjuk fel, ha a hosszabb névtérneveknek rövid álneveket (alias) adunk:

```
// használjunk névtér-álneveket a nevek rövidítésére:

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

A névtér-álnevek azt is lehetővé teszik a felhasználónak, hogy „a könyvtárra" hivatkozzon és egyetlen deklarációban határozza meg, valójában melyik könyvtárra gondol:

```
namespace Lib = Foundation_library_v2r11;

// ...

Lib::set s;
Lib::String s5 = "Sibelius";
```

Ez nagymértékben egyszerűsítheti a könyvtárak másik változatra történő cseréjét. Azáltal, hogy közvetlenül *Lib*-et használunk a *Foundation_library_v2r11* helyett, a *Lib* álnév értékének módosításával és a program újrafordításával a „v3r02" változatra frissíthetjük a könyvtárat. Az újrafordítás észre fogja venni a forrásszintű összeférhetetlenségeket. Másrészről, a (bármilyen típusú) álnevek túlzott használata zavart is okozhat.

8.2.8. Névterek összefűzése

Egy felületet gyakran már létező felületekből akarunk létrehozni:

```
namespace His_string {
  class String { /* ... */ };
  String operator+(const String&, const String&);
  String operator+(const String&, const char*);
  void fill(char);
  // ...
}

namespace Her_vector {
  template<class T> class Vector { /* ... */ };
  // ...
}

namespace My_lib {
  using namespace His_string;
  using namespace Her_vector;
  void my_fct(String&);
}
```

Ennek alapján – a *My_lib* névteret használva – már megírhatjuk a programot:

```
void f()
{
  My_lib::String s = "Byron";              // megtalálja a My_lib::His_string::String nevet
  // ...
}

using namespace My_lib;

void g(Vector<String>& vs)
{
  // ...
  my_fct(vs[5]);
  // ...
}
```

Ha az említett névtérben egy explicit módon minősített név (mint a *My_lib::String*) nem bevezetett, a fordító a nevet a *using* direktívákban szereplő névterekben (például *His_string*) fogja keresni.

Egy elem valódi névterét csak akkor kell tudnunk, ha valamit megakarunk határozni:

```
void My_lib::fill(char c)          // hiba: a My_lib-ben nincs megadva fill()
{
  // ...
}

void His_string::fill(char c)      // rendben: fill() szerepel a His_string-ben
{
  // ...
}

void My_lib::my_fct(String& v)     // rendben; a String jelentése My_lib::String, ami
                                   // His_string::String
{
  // ...
}
```

Ideális esetben egy névtér

1. logikailag összetartozó szolgáltatások halmazát fejezi ki,
2. nem ad hozzáférést a nem kapcsolódó szolgáltatásokhoz,
3. és nem ró nagy jelölésbeli terhet a felhasználóra.

Az itt és a következő részekben bemutatott összefűzési, beépítési módszerek –az *#include*-dal (§9.2.1) együtt – komoly támogatást nyújtanak ehhez.

8.2.8.1. Kiválasztás

Alkalmanként előfordul, hogy egy névtérből csak néhány névhez akarunk hozzáférni. Ezt meg tudnánk tenni úgy is, hogy olyan névtér-deklarációt írunk, amely csak azokat a neveket tartalmazza, melyeket szeretnénk. Például megadhatnánk a *His_string* azon változatát, amely csak magát a *String*-et és az összefűző operátort nyújtja:

```
namespace His_string {             // csak egy része a His_string-nek
  class String { /* ... */ };
  String operator+(const String&, const String&);
  String operator+(const String&, const char*);
}
```

Ez azonban könnyen zavarossá válhat, hacsak nem mi vagyunk a *His_string* tervezői vagy „karbantartói". A *His_string* „valódi" meghatározásának módosítása ebben a deklarációban nem fog tükröződni. Az adott névtérben szereplő szolgáltatások kiválasztását jobban ki lehet fejezni *using* deklarációkkal:

```
namespace My_string {
  using His_string::String;
  using His_string::operator+;          // bármelyik His_string-beli + használható
}
```

A *using* deklaráció az adott név minden deklarációját a hatókörbe helyezi, így például egyetlen *using* deklarációval egy túlterhelt függvény összes változatát bevezethetjük.

Így ha a *His_string*-et úgy módosítják, hogy egy tagfüggvényt vagy az összefűző művelet egy túlterhelt változatát adják a *String*-hez, akkor ez a változtatás automatikusan hozzáférhető lesz a *My_string*-et használó elemek számára. Fordítva is igaz: ha a *His_string*-ből eltávolítunk egy szolgáltatást vagy megváltozatjuk a *His_string* felületét, a fordítóprogram fel fogja ismerni a *My_string* minden olyan használatát, amelyre ez hatással van (lásd még §15.2.2).

8.2.8.2. Összefűzés és kiválasztás

A (*using* direktívákkal történő) összefűzés és a (*using* deklarációkkal történő) kiválasztás összekapcsolása azt a rugalmasságot eredményezi, amelyre a legtöbb valódi programban szükségünk van. Ezek révén úgy adhatunk hozzáférést különféle eszközökhöz, hogy feloldjuk az egybeépítésükből adódó névütközéseket és többértelműségeket:

```
namespace His_lib {
  class String { /* ... */ };
  template<class T> class Vector { /* ... */ };
  // ...
}

namespace Her_lib {
  template<class T> class Vector { /* ... */ };
  class String { /* ... */ };
  // ...
}

namespace My_lib {
  using namespace His_lib;              // minden a His_lib-ből
  using namespace Her_lib;              // minden a Her_lib-ből
```

```
    using His_lib::String;              // az esetleges ütközések feloldása a His_lib javára
    using Her_lib::Vector;              // az esetleges ütközések feloldása a Her_lib javára

    template<class T> class List { /* ... */ };          // továbbiak
    // ...
}
```

Amikor megvizsgálunk egy névteret, a névtérben lévő, kifejezetten megadott nevek (beleértve a *using* deklarációkkal megadottakat is) előnyben részesülnek azokkal a nevekkel szemben, melyeket más hatókörökből tettünk hozzáférhetővé a *using* direktívával (lásd még §C.10.1-et). Következésképpen a *My_lib*–et használó elemek számára a *String* és *Vector* nevek ütközését a fordítóprogram a *His_lib::String* és *Her_lib::Vector* javára fogja feloldani. Továbbá a *My_lib::List* lesz használatos alapértelmezés szerint, függetlenül attól, hogy szerepel-e *List* a *His_lib* vagy *Her_lib* névtérben.

Rendszerint jobban szeretem változatlanul hagyni a neveket, amikor új névtérbe teszem azokat. Ily módon nem kell ugyanannak az elemnek két különböző nevére emlékeznem. Néha azonban új névre van szükség, vagy egyszerűen jó, ha van egy új nevünk:

```
namespace Lib2 {
    using namespace His_lib;            // minden a His_lib-ből
    using namespace Her_lib;            // minden a Her_lib-ből

    using His_lib::String;              // az esetleges ütközések feloldása a His_lib javára
    using Her_lib::Vector;              // az esetleges ütközések feloldása a Her_lib javára

    typedef Her_lib::String Her_string;         // átnevezés

    template<class T> class His_vec             // "átnevezés"
            : public His_lib::Vector<T> { /* ... */ };

    template<class T> class List { /* ... */ }; // továbbiak
    // ...
}
```

Az átnevezésre nincs külön nyelvi eljárás. Ehelyett az új elemek meghatározására való általános módszerek használatosak.

8.2.9. Névterek és régi kódok

Sok millió sor C és C++ kód támaszkodik globális nevekre és létező könyvtárakra. Hogyan használhatjuk a névtereket arra, hogy csökkentsük az ilyen kódokban lévő problémákat? A már létező kódok újraírása nem mindig járható út. Szerencsére a C könyvtárakat úgy is

használhatjuk, mintha azokat egy névtérben deklarálták volna. A C++-ban írt könyvtárak esetében ez nem így van (§9.2.4), másrészről viszont a névtereket úgy tervezték, hogy a lehető legcsekélyebb károkozással be lehessen azokat építeni a régebbi C++ programokba is.

8.2.9.1. Névterek és a C

Vegyük a hagyományosan első C programot:

```
#include <stdio.h>

int main()
{
  printf("Helló, világ!\n");
}
```

Ezt a programot nem lenne jó ötlet széttördelni. Az sem ésszerű, ha a szabványos könyvtárakat egyedi megoldásoknak tekintjük. Emiatt a névterekre vonatkozó nyelvi szabályokat úgy határozták meg, hogy viszonylag könnyedén lehessen egy névterek nélkül megírt program szerkezetét névterek használatával világosabban kifejezni. Tulajdonképpen erre példa a számológép program (§6.1). Ennek megvalósításához a kulcs a *using* direktíva. A *stdio.h* C fejállományban lévő szabványos bemeneti/kimeneti szolgáltatások deklarációi például egy névtérbe kerültek, a következőképpen:

```
// stdio.h:

  namespace std {
            // ...
            int printf(const char* ... );
            // ...
  }
  using namespace std;
```

Ez megőrzi a visszirányú kompatibilitást. Azoknak viszont, akik nem akarják, hogy a nevek automatikusan hozzáférhetők legyenek, készítettünk egy új fejállományt is, a *cstdio*-t:

```
// cstdio:

  namespace std {
            // ...
            int printf(const char* ... );
            // ...
  }
```

A C++ standard könyvtárának azon felhasználói, akik aggódnak a deklarációk másolása miatt, a *stdio.h*-t természetesen úgy fogják meghatározni, hogy beleveszik a *cstdio*-t:

```
// stdio.h:

   #include<cstdio>
   using namespace std;
```

A *using* direktívákat elsődlegesen a nyelv régebbi változatairól való átállást segítő eszközöknek tekintem. A legtöbb olyan kódot, amely más névtérben lévő nevekre hivatkozik, sokkal világosabban ki lehet fejezni minősítésekkel és *using* deklarációkkal.

A névterek és az összeszerkesztés közötti kapcsolatot a §9.2.4 részben tárgyaljuk.

8.2.9.2. Névterek és túlterhelés

A túlterhelés (§7.4) névtereken keresztül működik. Ez alapvető ahhoz, hogy a már meglévő könyvtárakat a forráskód lehető legkisebb módosításával fejleszthessük névtereket használóvá. Például:

```
// old A.h:

   void f(int);
   // ...

// old B.h:

   void f(char);
   // ...

// old user.c:

   #include "A.h"
   #include "B.h"

   void g()
   {
             f('a');       // f()-et hívja B.h-ból
   }
```

Ezt a programot anélkül alakíthatjuk névtereket használó változatra, hogy a tényleges programkódot megváltoztatnánk:

```
// new A.h:

  namespace A {
        void f(int);
        // ...
  }

// new B.h:

  namespace B {
        void f(char);
        // ...
  }

// new user.c:

  #include "A.h"
  #include "B.h"

  using namespace A;
  using namespace B;

  void g()
  {
        f('a');       // f()-et hívja B.h-ból
  }
```

Ha teljesen változatlanul akartuk volna hagyni a *user.c*-t, a *using* direktívákat a fejállományokba tettük volna.

8.2.9.3. A névterek nyitottak

A névterek nyitottak; azaz számos névtér deklarációjából adhatunk hozzájuk neveket:

```
namespace A {
  int f();             // most f() az A tagja
}

namespace A {
  int g();             // most A két tagja f() és g()
}
```

Ezáltal úgy hozhatunk létre egyetlen névtéren belül lévő nagy programrészeket, ahogy egy régebbi könyvtár vagy alkalmazás élt az egyetlen globális névtéren belül. Hogy ezt megtehessük, a névtér-meghatározásokat szét kell osztanunk számos fejállomány és forrásfájl kö-

zött. Ahogy azt a számológép példájában (§8.2.4.) mutattuk, a névterek nyitottsága lehetővé teszi számunkra, hogy a különböző programelemeknek különböző felületeket nyújtsunk azáltal, hogy egy adott névtér különböző részeit mutatjuk meg nekik. Ez a nyitottság szintén a nyelv régebbi változatairól való átállást segíti. Például a

```
// saját fejállomány:
  void f(); // saját függvény
  // ...
  #include<stdio.h>
  int g(); // saját függvény
  // ...
```

újraírható anélkül, hogy a deklarációk sorrendjét megváltoztatnánk:

```
// saját fejállomány:

  namespace Mine {
          void f(); // saját függvény
          // ...
  }

  #include<stdio.h>

  namespace Mine {
          int g(); // saját függvény
          // ...
  }
```

Amikor új kódot írok, jobban szeretek sok kisebb névteret használni (lásd §8.2.8), mint igazán nagy programrészeket egyetlen névtérbe rakni. Ez azonban gyakran kivitelezhetetlen, ha nagyobb programrészeket alakítunk át névtereket használó változatra.

Amikor egy névtér előzetesen bevezetett tagját kifejtjük, biztonságosabb a *Mine::* utasításformát használni ahelyett, hogy újra megnyitnánk a *Mine*-t:

```
void Mine::ff()      // hiba: nincs ff() megadva Mine-ban
{
  // ...
}
```

A fordítóprogram ezt a hibát észreveszi. Mivel azonban egy névtéren belül új függvényeket is meghatározhatunk, a fordítóprogram a fentivel azonos jellegű hibát az újra megnyitott névterekben már nem érzékeli:

```
namespace Mine {   // Mine újra megnyitása függvények meghatározásához

  void ff()          // hoppá! nincs ff() megadva Mine-ban; ezzel a definícióval adjuk hozzá
  {
        // ...
  }

  // ...
}
```

A fordítóprogram nem tudhatja, hogy nem egy új *ff()* függvényt akartunk meghatározni.

A meghatározásokban szereplő nevek minősítésére használhatunk névtér-álneveket (§8.2.7), de az adott névtér újbóli megnyitására nem.

8.3. Kivételek

Ha egy program különálló modulokból áll – különösen ha ezek külön fejlesztett könyvtárakból származnak –, a hibakezelést két különálló részre kell szétválasztanunk:

1. az olyan hibaesemények jelzésére, melyeket nem lehet helyben megszüntetni,
2. illetve a máshol észlelt hibák kezelésére.

A könyvtár létrehozója felismerheti a futási idejű hibákat, de általában nem tud mit kezdeni velük. A könyvtárt felhasználó programelem tudhatná, hogyan birkózzon meg a hibákkal, de nem képes észlelni azokat – máskülönben a felhasználó kódjában szerepelnének a hibákat kezelő eljárások és nem a könyvtár találná meg azokat.

A számológép példájában ezt a problémát azzal kerültük ki, hogy a program egészét egyszerre terveztük meg, ezáltal beilleszthettük a hibakezelést a teljes szerkezetbe. Amikor azonban a számológép logikai részeit különböző névterekre bontjuk szét, látjuk, hogy minden névtér függ az *Error* névtértől (§8.2.2), az *Error*-ban lévő hibakezelő pedig arra támaszkodik, hogy minden modul megfelelően viselkedik, miután hiba történt. Tegyük fel, hogy nincs lehetőségünk a számológép egészét megtervezni és nem akarjuk, hogy az *Error* és a többi modul között szoros legyen a kapcsolat. Ehelyett tegyük fel, hogy a elemzőt és a többi részt úgy írták meg, hogy nem tudták, hogyan szeretné a vezérlő kezelni a hibákat.

Bár az *error()* nagyon egyszerű volt, magában foglalt egy hibakezelési módszert:

```
namespace Error {
  int no_of_errors;

  double error(const char* s)
  {
        std::cerr << "hiba: " << s << '\n';
        no_of_errors++;
        return 1;
  }
}
```

Az *error()* függvény egy hibaüzenetet ír ki, olyan alapértelmezett értéket ad, mely lehetővé teszi a hívó számára, hogy folytassa a számolást, és egy egyszerű hibaállapotot követ nyomon. Fontos, hogy a program minden része tudjon az *error()* létezéséről és arról, hogyan lehet meghívni, illetve mit várhat tőle. Ez túl sok feltétel lenne egy olyan program esetében, amit külön fejlesztett könyvtárakból hoztunk létre.

A hibajelzés és a hibakezelés szétválasztására szánt C++ eszköz a *kivétel*. Ebben a részben röviden leírjuk a kivételeket, abban a környezetben, ahogy a számológép példájában lennének használatosak. A 14. fejezet átfogóbban tárgyalja a kivételeket és azok használatát.

8.3.1. „Dobás és elkapás”

A kivételeket (exception) arra találták ki, hogy segítsenek megoldani a hibák jelzését:

```
struct Range_error {
  int i;
  Range_error(int ii) { i = ii; }                    // konstruktor (§2.5.2, §10.2.3)
};

char to_char(int i)
{
  if (i<numeric_limits<char>::min() || numeric_limits<char>::max()<i)    // lásd §22.2
        throw Range_error(i);
  return i;
}
```

A *to_char()* függvény vagy az *i* számértékét adja vissza karakterként, vagy *Range_error* kivételt vált ki. Az alapgondolat az, hogy ha egy függvény olyan problémát talál, amellyel nem képes megbirkózni, kivételt vált ki („kivételt dob”, *throw*), azt remélve, hogy (közvetett vagy közvetlen) meghívója képes kezelni a problémát. Ha egy függvény képes erre, je-

lezheti, hogy el akarja kapni (*catch*) azokat a kivételeket, melyek típusa megegyezik a probléma jelzésére használt típussal. Ahhoz például, hogy meghívjuk a *to_char()*-t és elkapjuk azt a kivételt, amit esetleg kiválthat, a következőt írhatjuk:

```
void g(int i)
{
  try {
          char c = to_char(i);
          // ...
  }
  catch (Range_error) {
          cerr << "hoppá\n";
  }
}
```

A

```
  catch ( /* ... */ ) {
          // ...
  }
```

szerkezetet ***kivételkezelőnek*** (exception handler) nevezzük. Csak közvetlenül olyan blokk után használható, amit a *try* kulcsszó előz meg, vagy közvetlenül egy másik kivételkezelő után. A *catch* szintén kulcsszó. A zárójelek olyan deklarációt tartalmaznak, amely a függvényparaméterek deklarációjához hasonló módon használatos. A deklaráció határozza meg azon objektum típusát, melyet a kezelő elkaphat. Nem kötelező, de megnevezheti az elkapott objektumot is. Ha például meg akarjuk tudni a kiváltott *Range_error* értékét, akkor pontosan úgy adhatunk nevet a *catch* paraméterének, ahogy a függvényparamétereket nevezzük meg:

```
void h(int i)
{
  try {
          char c = to_char(i);
          // ...
  }
  catch (Range_error x) {
          cerr << "hoppá: to_char(" << x.i << ")\n";
  }
}
```

Ha bármilyen *try* blokkban szereplő vagy onnan meghívott kód kivételt vált ki, a *try* blokk kezelőit kell megvizsgálni. Ha a kivétel típusa megegyezik a kezelőnek megadott típussal,

a kezelő végrehajtja a megfelelő műveletet. Ha nem, a kivételkezelőket figyelmen kívül hagyjuk és a *try* blokk úgy viselkedik, mint egy közönséges blokk. Ha a kivételt nem kapja el egyetlen *try* blokk sem, a program befejeződik (§14.7).

A C++ kivételkezelése alapvetően nem más, mint a vezérlés átadása a hívó függvény megfelelő részének. Ahol szükséges, a hibáról információt adhatunk a hívónak. A C programozók úgy gondolhatnak a kivételkezelésre, mint egy olyan, „jól viselkedő" eljárásra, amely a *setjmp/longjmp* (§16.1.2) használatát váltja fel. Az osztályok és a kivételkezelés közötti kölcsönhatást a 14. fejezetben tárgyaljuk.

8.3.2. A kivételek megkülönböztetése

Egy program futásakor általában számos hiba léphet fel, melyeket különböző nevű kivételeknek feleltethetünk meg. Én a kivételkezelés céljára külön típusokat szoktam megadni. Ez a lehető legkisebbre csökkenti a céljukkal kapcsolatos zavart. Beépített típusokat, mint amilyen az *int*, viszont sohasem használok kivételként. Egy nagy programban nem lenne hatékony mód arra, hogy megtaláljam a más célra használt *int* kivételeket, ezért sosem lehetnék biztos abban, hogy az *int* egy efféle eltérő használata nem okoz-e zavart az én kódomban.

Számológépünknek (§6.1) kétfajta futási idejű hibát kell kezelnie: a formai követelmények megsértését és a nullával való osztás kísérletét. A kezelőnek nem kell értéket átadni abból a kódból, amelyik felismerte a nullával való osztás kísérletét, így a nullával való osztást egy egyszerű üres típussal ábrázolhatjuk:

```
struct Zero_divide { };
```

Másrészt a kezelő a nyelvi hibákról bizonyára szeretne jelzést kapni. Itt egy karakterláncot adunk át:

```
struct Syntax_error {
  const char* p;
  Syntax_error(const char* q) { p = q; }
};
```

A kényelmesebb jelölés végett a szerkezethez hozzáadtam egy konstruktort (§2.5.2, §10.2.3).

Az elemzőt használó programrészben megkülönböztethetjük a két kivételt, ha mindkettőjük számára hozzáadunk egy-egy kezelőt a *try* blokkhoz, így szükség esetén a megfelelő kezelőbe léphetünk. Ha az egyik kezelő „alján kiesünk", a végrehajtás a kezelők listájának végétől folytatódik:

```
try {
        // ...
        expr(false);
        // kizárólag akkor jutunk ide, ha expr() nem okozott kivételt
        // ...
}
catch (Syntax_error) {
        // szintaktikus hiba kezelése
}
catch (Zero_divide) {
        // nullával osztás kezelése
}
// akkor jutunk ide, ha expr() nem okozott kivételt vagy ha egy Syntax_error
// vagy Zero_divide kivételt elkaptunk (és kezelőjük nem tért vissza,
// nem váltott ki kivételt, és más módon sem változtatta meg a vezérlést).
```

A kezelők listája némileg egy *switch* utasításhoz hasonlít, de itt nincs szükség *break* utasításokra. E listák formai követelményei részben ezért különböznek a *case*-étől, részben pedig azért, hogy jelöljék, minden kezelő külön hatókört (§4.9.4) alkot.

A függvényeknek nem kell az összes lehetséges kivételt elkapniuk. Az előző *try* blokk például nem próbálta elkapni az elemző bemeneti műveletei által kiváltott kivételeket, azok csupán „keresztülmennek" a függvényen, megfelelő kezelővel rendelkező hívót keresve.

A nyelv szempontjából a kivételeket rögtön kezeltnek tekintjük, amint „belépnek" a kezelőjükbe, ezért a *try* blokkot meghívó programrésznek kell foglalkoznia azokkal a kivételekkel, melyek a kezelő végrehajtása közben lépnek fel. A következő például nem okoz végtelen ciklust:

```
class Input_overflow { /* ... */ };

void f()
{
  try {
        // ...
  }
  catch (Input_overflow) {
        // ...
        throw Input_overflow();
  }
}
```

A kivételkezelők egymásba is ágyazhatók:

```
class XXII { /* ... */ };

void f()
{
  // ...
  try {
          // ...
  }
  catch (XXII) {
          try {
                  // valami bonyolult
          }
          catch (XXII) {
                  // a bonyolult kezelő nem járt sikerrel
          }
  }
  // ...
}
```

Ilyen – gyakran rossz stílusra utaló – egymásba ágyazott kivételkezelőket azonban ritkán írunk.

8.3.3. Kivételek a számológépben

Az alapvető kivételkezelő eljárásokból kiindulva újraírhatjuk a §6.1 részben szereplő számológépet, hogy különválasszuk a futási időben talált hibák kezelését a számológép fő programrészétől. Ez a program olyan elrendezését eredményezi, amely jobban hasonlít a különálló, lazán kapcsolódó részekből létrehozott programokéra.

Először kiküszöbölhetjük az *error()* függvényt. Helyette az elemző függvények csak a hibák jelzésére használatos típusokról fognak tudni:

```
namespace Error {
  struct Zero_divide { };

  struct Syntax_error {
          const char* p;
          Syntax_error(const char* q) { p = q; }
  };
}
```

Az elemző három szintaktikus hibát ismer fel:

```
Lexer::Token_value Lexer::get_token()
{
  using namespace std;        // az input, isalpha(), stb. használata miatt (§6.1.7)

          // ...

          default:            // NAME, NAME =, vagy hiba
                  if (isalpha(ch)) {
                          input->putback(ch);
                          *input >> string_value;
                          return curr_tok=NAME;
                          string_value = ch;
                          while (input->get(ch) && isalnum(ch))
                                  string_value.push_back(ch);
                          input->putback(ch);
                          return curr_tok=NAME;
                  }
                  throw Error::Syntax_error("rossz szimbólum");
  }
}

double Parser::prim(bool get)          // elemi szimbólumok kezelése
{
  // ...

  case Lexer::LP:
  {       double e = expr(true);
          if (curr_tok != Lexer::RP) throw Error::Syntax_error("')' szükséges");
          get_token();                 // ')' lenyelése
          return e;
  }
  case Lexer::END:
          return 1;
  default:
          throw Error::Syntax_error("elemi szimbólum szükséges");
  }
}
```

Ha az elemző ilyen hibát talál, a *throw*-t használja arra, hogy átadja a vezérlést egy kezelőnek, amelyet valamilyen (közvetett vagy közvetlen) hívó függvény határoz meg. A *throw* operátor egy értéket is átad a kezelőnek. Például a

```
throw Syntax_error("elemi szimbólum szükséges");
```

a kezelőnek egy *Syntax_error* objektumot ad át, amely a *primary expected* karakterláncra hivatkozó mutatót tartalmazza.

A nullával való osztás hibájának jelzéséhez nem szükséges semmilyen adatot átadni:

```
double Parser::term(bool get)            // szorzás és osztás
{
  // ...
  case Lexer::DIV:
        if (double d = prim(true)) {
              left /= d;
              break;
        }
        throw Error::Zero_divide();

  // ...
}
```

Most már elkészíthetjük a vezérlőt, hogy az kezelje a *Zero_divide* és *Syntax_error* kivételeket:

```
int main(int argc, char* argv[])
{
  // ...
  while (*input) {
        try {
              Lexer::get_token();
              if (Lexer::curr_tok == Lexer::END) break;
              if (Lexer::curr_tok == Lexer::PRINT) continue;
              cout << Parser::expr(false) << '\n';
        }
        catch(Error::Zero_divide) {
              cerr << "nullával osztás kísérlete\n";
              if (Lexer::curr_tok != Lexer::PRINT) skip();
        }
        catch(Error::Syntax_error e) {
              cerr << "formai hiba:" << e.p << "\n";
              if (Lexer::curr_tok != Lexer::PRINT) skip();
        }
  }

  if (input != &cin) delete input;
  return no_of_errors;
}
```

Ha nem történt hiba a *PRINT* (azaz sorvége vagy pontosvessző) szimbólummal lezárt kifejezés végén, a *main()* meghívja a *skip()* helyreállító függvényt. A *skip()* az elemzőt egy meghatározott állapotba próbálja állítani, azáltal, hogy eldobja a karaktereket addig, amíg sorvégét vagy pontosvesszőt nem talál. A *skip()* függvény, a *no_of_errors* és az *input* kézenfekvő választás a *Driver* névtér számára:

```
namespace Driver {
  int no_of_errors;
  std::istream* input;
  void skip();
}

void Driver::skip()
{
  no_of_errors++;

  while (*input) {                // karakterek elvetése sortörésig vagy pontosvesszőig
        char ch;
        input->get(ch);

        switch (ch) {
        case '\n':
        case ';':
              return;
        }
  }
}
```

A *skip()* kódját szándékosan írtuk az elemző kódjánál alacsonyabb elvonatkoztatási szinten. Így az elemzőben lévő kivételek nem kapják el, miközben éppen az elemző kivételeinek kezelését végzik. Megtartottam azt az ötletet, hogy megszámoljuk a hibákat, és ez a szám lesz a program visszatérési értéke. Gyakran hasznos tudni a hibákról, még akkor is, ha a program képes volt helyreállni a hiba után.

A *main()*-t nem tesszük a *Driver* névtérbe. A globális *main()* a program indító függvénye (§3.2), így a *main()* egy névtéren belül értelmetlen. Egy valóságos méretű programban a *main()* kódjának legnagyobb részét a *Driver* egy külön függvényébe tenném át.

8.3.3.1. Más hibakezelő módszerek

Az eredeti hibakezelő kód rövidebb és elegánsabb volt, mint a kivételeket használó változat. Ezt azonban úgy érte el, hogy a program részeit szorosan összekapcsolta. Ez a megközelítés nem felel meg olyan programok esetében, melyeket külön fejlesztett könyvtárakból hoztak létre. Felvetődhet, hogy a különálló *skip()* hibakezelő függvényt a *main()*-ben, egy állapotváltozó bevezetésével küszöböljük ki:

```
int main(int argc, char* argv[ ])         // rossz stílus
{
  // ...

  bool in_error = false;
```

```
    while (*Driver::input) {
        try {
            Lexer::get_token();
            if (Lexer::curr_tok == Lexer::END) break;
            if (Lexer::curr_tok == Lexer::PRINT) {
                in_error = false;
                continue;
            }
            if (in_error == false) cout << Parser::expr(false) << '\n';
        }
        catch(Error::Zero_divide) {
            cerr << "nullával osztás kísérlete\n";
            ++ Driver::no_of_errors;
            in_error = true;
        }
        catch(Error::Syntax_error e) {
            cerr << "formai hiba:" << e.p << "\n";
            ++ Driver::no_of_errors;
            in_error = true;
        }
    }
    if (Driver::input != &std::cin) delete Driver::input;
    return Driver::no_of_errors;
}
```

Ezt számos okból rossz ötletnek tartom:

1. Az állapotváltozók gyakran zavart okoznak és hibák forrásai lehetnek, különösen akkor, ha lehetőséget adunk rá, hogy elszaporodjanak és hatásuk nagy programrészekre terjedjen ki. Nevezetesen az *in_error*-t használó *main()*-t kevésbé olvashatónak tartom, mint a *skip()* függvényt használó változatot.
2. Általában jobb külön tartani a hibakezelést és a „közönséges" kódot.
3. Veszélyes, ha a hibakezelés elvonatkoztatási szintje megegyezik annak a kódnak az absztrakciós szintjével, ami a hibát okozta; a hibakezelő kód ugyanis megismételheti azt a hibát, amely a hibakezelést először kiváltotta. (A gyakorlatok között szerepel, hogy mi történik, ha a *main()* *in_error*-t használ. §8.5[7]).
4. Több munkával jár az egész kódot módosítani a hibakezelés hozzáadásával, mint külön hibakezelő függvényeket adni a kódhoz.

A kivételkezelés nem helyi problémák megoldására való. Ha egy hiba helyben kezelhető, akkor majdnem mindig ezt is kell tennünk. Például nincs ok arra, hogy kivételt használjunk a „túl sok paraméter" hiba fellépésekor:

```
int main(int argc, char* argv[])
{
  using namespace std;
  using namespace Driver;

  switch (argc) {
  case 1:                                   // olvasás szabványos bemenetről
           input = &cin;
           break;
  case 2:                                   // karakterlánc paraméter beolvasása
           input = new istringstream(argv[1]);
           break;
  default:
           cerr << "túl sok paraméter\n";
           return 1;
  }

  // mint korábban
}
```

A kivételek további tárgyalása a 14. fejezetben történik.

8.4. Tanácsok

[1] Használjunk névtereket a logikai felépítés kifejezésére. §8.2.
[2] A *main()* kivételével minden nem lokális nevet helyezzünk valamilyen névtérbe. §8.2.
[3] A névtereket úgy tervezzük meg, hogy utána kényelmesen használhassuk, anélkül, hogy véletlenül hozzáférhetnénk más, független névterekhez. §8.2.4.
[4] Lehetőleg ne adjunk a névtereknek rövid neveket. §8.2.7.
[5] Ha szükséges, használjunk névtér-álneveket a hosszú névtérnevek rövidítésére. §8.2.7.
[6] Lehetőleg ne rójunk nehéz jelölésbeli terheket névtereink felhasználóira. §8.2.2., §8.2.3.
[7] Használjuk a *Névtér::tag* jelölést, amikor a névtér tagjait meghatározzuk. §8.2.8.
[8] A *using namespace*–t csak a C-ről vagy régebbi C++-változatokról való átálláskor, illetve helyi hatókörben használjuk. §8.2.9.
[9] Használjunk kivételeket arra, hogy a szokásos feldolgozást végző kódrészt elválasszuk attól a résztől, amelyben a hibákkal foglalkozunk. §8.3.2.

[10] Inkább felhasználói típusokat használjunk kivételekként, mint beépített típusokat. §8.3.2.
[11] Ne használjunk kivételeket, amikor a helyi vezérlési szerkezetek is megfelelőek. §8.3.3.1.

8.5. Gyakorlatok

1. (*2,5) Írjunk *string* elemeket tartalmazó kétirányú láncolt lista modult a §2.4-ben található *Stack* modul stílusában. Próbáljuk ki úgy, hogy létrehozunk egy programnyelvekből álló listát. Adjunk erre listára egy *sort()* függvényt és egy olyat, ami megfordítja a listában szereplő karakterláncok sorrendjét.
2. (*2) Vegyünk egy nem túl nagy programot, amely legalább egy olyan könyvtárat használ, ami nem használ névtereket. Módosítsuk úgy, hogy a könyvtár névtereket használjon. Tipp: §8.2.9.
3. (*2) Készítsünk modult a számológép programból névterek felhasználásával a §2.4 pont stílusában. Ne használjunk globális *using* direktívákat. Jegyezzük fel, milyen hibákat vétettünk. Tegyünk javaslatokat arra, miként kerülhetnénk el az ilyen hibákat a jövőben.
4. (*1) Írjunk programot, amelyben egy függvény kivételt „dob”, egy másik pedig elkapja.
5. (*2) Írjunk programot, amely olyan egymást hívó függvényekből áll, ahol a hívás mélysége 10. Minden függvénynek adjunk egy paramétert, amely eldönti, melyik szinten lépett fel a kivétel. A *main()*-nel kapjuk el a kivételeket és írjuk ki, melyiket kaptuk el. Ne felejtsük el azt az esetet, amikor a kivételt a kiváltó függvényben kapjuk el.
6. (*2) Módosítsuk a §8.5[5] programját úgy, hogy megmérjük, van-e különbség a kivételek elkapásának nehézségében attól függően, hogy a *stack* osztályon belül hol jött létre kivétel. Adjunk minden függvényhez egy karakterlánc objektumot és mérjük meg újra a különbséget.
7. (*1) Találjuk meg a hibát a §8.3.3.1-ben szereplő *main()* első változatában.
8. (*2) Írjunk függvényt, amely vagy visszaad egy értéket, vagy egy paraméter alapján eldobja azt. Mérjük meg a két módszer futási idejének különbségét.
9. (*2) Módosítsuk a §8.5[3]-ban lévő számológépet kivételek használatával. Jegyezzük fel, milyen hibákat vétettünk. Tegyünk javaslatokat arra, miként kerülhetnénk el az ilyen hibákat a jövőben.
10. (*2,5) Írjuk meg a *plus()*, *minus()*, *multiply()* és *divide()* függvényeket, amelyek ellenőrzik a túlcsordulást és az alulcsordulást, és kivételeket váltanak ki, ha ilyen hibák történnek.
11. (*2) Módosítsuk a számológépet, hogy a §8.5[10] függvényeit használja.

9

Forrásfájlok és programok

„A formának a rendeltetéshez kell igazodnia."
(Le Corbusier)

Külön fordítás • Összeszerkesztés • Fejállományok • A standard könyvtár fejállományai • Az egyszeri definiálás szabálya • Összeszerkesztés nem C++ kóddal • Az összeszerkesztés és a függvényekre hivatkozó mutatók • Fejállományok használata a modularitás kifejezésére • Egyetlen fejállományos elrendezés • Több fejállományos elrendezés • Állományőrszemek • Programok • Tanácsok • Gyakorlatok

9.1. Külön fordítás

A fájl (az egyes fájlrendszerekben) a tárolás és fordítás hagyományos egysége. Vannak olyan rendszerek, amelyek a C++ programokat nem fájlok halmazaként tárolják és fordítják, és a programok sem fájlok formájában jelennek meg a programozó számára. Ez a leírás azonban csak azokra a rendszerekre összpontosít, amelyek a fájlok hagyományos használatára támaszkodnak.

Egy teljes programot rendszerint lehetetlen egy fájlban tárolni, már csak azért sem, mert a szabványos könyvtárak és az operációs rendszer forráskódja általában nem szerepel a program forrásában. A valóságos méretű alkalmazásokban az sem kényelmes és célszerű, ha a felhasználó saját kódját egyetlen fájl tárolja. A program elrendezési módja segíthet kihangsúlyozni a program logikai felépítését, segítheti az olvasót a program megértésében és segíthet abban is, hogy a fordítóprogram kikényszerítse ezt a logikai szerkezetet. Amikor a fordítási egység a fájl, akkor a teljes fájlt újra kell fordítani, ha (bármilyen kis) változtatást hajtottak végre rajta, vagy egy másik fájlon, amelytől az előző függ. Az újrafordításra használt idő még egy közepes méretű program esetében is jelentősen csökkenthető, ha a programot megfelelő méretű fájlokra bontjuk.

A felhasználó a fordítóprogramnak egy *forrásfájlt* (source file) ad át. Ezután a fájl előfordítása történik: azaz végrehajtódik a makrófeldolgozás (§7.8), az *#include* utasítások pedig beépítik a fejállományokat (§2.4.1, §9.2.1). Az előfeldolgozás eredményét *fordítási egységnek* (translation unit) hívják. A fordítóprogram valójában csak ezekkel dolgozik és a C++ szabályai is ezek formáját írják le. Ebben a könyvben csak ott teszek különbséget a forrásfájl és a fordítási egység között, ahol meg kell különböztetni azt, amit a programozó lát, és amit a fordítóprogram figyelembe vesz. Ahhoz, hogy a programozó lehetővé tegye az elkülönített fordítást, olyan deklarációkat kell megadnia, amelyek biztosítják mindazt az információt, ami ahhoz szükséges, hogy a fordítási egységet a program többi részétől elkülönítve lehessen elemezni. A több fordítási egységből álló programok deklarációinak ugyanúgy következetesnek kell lenniük, mint az egyetlen forrásfájlból álló programokénak. A rendszerünkben vannak olyan eszközök, amelyek segítenek ezt biztosítani; nevezetesen a szerkesztőprogram (*linker*), amely számos következetlenséget képes észrevenni. Ez az a program, ami összekapcsolja a külön fordított részeket. A szerkesztőt néha (zavaró módon) betöltőnek (*loader*) is szokták nevezni. A teljes összeszerkesztést el lehet végezni a program futása előtt. Emellett lehetőség van arra is, hogy később új kódot adjunk a programhoz („dinamikus szerkesztés").

A program *fizikai szerkezetén* általában a forrásfájlokba szervezett programot értik. A program forrásfájlokra való fizikai szétválasztását a program logikai felépítése kell, hogy irányítsa. A programok forrásfájlokba rendezését is ugyanaz a függőségi kapcsolat vezérli, mint azok névterekből való összeállítását. A program logikai és fizikai szerkezetének azonban nem kell megegyeznie. Hasznos lehet például több forrásfájlt használni egyetlen névtér függvényeinek tárolására, névtér-meghatározások egy gyűjteményét egyetlen fájlban tárolni, vagy egy névtér definícióit több fájl között szétosztani (§8.2.4).

Először áttekintünk néhány, az összeszerkesztéshez kapcsolódó részletet és szakkifejezést, majd kétféle módját ismertetjük annak, hogyan lehet fájlokra szétválasztani a számológépet (§6.1, §8.2).

9.2. Összeszerkesztés

A függvények, osztályok, sablonok, változók, névterek, felsorolások és felsorolók neveit következetesen kell használni az összes fordítási egységben, kivéve, ha kifejezetten lokálisként nem határoztuk meg azokat.

A programozó feladata biztosítani, hogy minden névtér, osztály, függvény stb. megfelelően legyen deklarálva minden olyan fordítási egységben, amelyben szerepel, és hogy minden deklaráció, amely ugyanarra az egyedre vonatkozik, egységes legyen. Vegyük például a következő két fájlt:

```
// file1.c:
  int x = 1;
  int f() { /* csinálunk valamit */ }

// file2.c:
  extern int x;
  int f();
  void g() { x = f(); }
```

A *file2.c*-ben lévő *g()* által használt *x* és *f()* meghatározása a *file1.c*-ben szerepel. Az *extern* kulcsszó jelzi, hogy a *file2.c*-ben az *x* deklarációja (csak) deklaráció és nem definíció (§4.9). Ha *x* már rendelkezne kezdőértékkel, a fordítóprogram az *extern* kulcsszót egyszerűen figyelmen kívül hagyná, mert a kezdőértéket is meghatározó deklarációk egyben definíciónak is minősülnek. Egy objektumot a programban csak pontosan egyszer határozhatunk meg. Deklarálni többször is lehet, de a típusoknak pontosan meg kell egyezniük:

```
// file1.c:
  int x = 1;
  int b = 1;
  extern int c;

// file2.c:
  int x;                    // jelentése int x = 0;
  extern double b;
  extern int c;
```

Itt három hiba van: *x*-et kétszer definiáltuk, *b*-t kétszer deklaráltuk különböző típusokkal, *c*-t pedig kétszer deklaráltuk, de egyszer sem definiáltuk. Az effajta hibákat (szerkesztési hiba, linkage error) a fordítóprogram – ami egyszerre csak egy fájlt néz – nem ismeri fel, a szerkesztő azonban a legtöbbet igen. Jegyezzük meg, hogy a globális vagy névtér-hatókör-

ben kezdőérték nélkül megadott változók alapértelmezés szerint kapnak kezdőértéket. Ez nem vonatkozik a lokális változókra (§4.9.5, §10.4.2) vagy a szabad tárban létrehozott objektumokra (§6.2.6).

A következő programrészlet két hibát tartalmaz:

```
// file1.c:
  int x;
  int f() { return x; }

// file2.c:
  int x;
  int g() { return f(); }
```

A *file2.c*-ben az *f()* meghívása hiba, mert *f()*-et a *file2.c* nem deklarálja. Ezenkívül a szerkesztő nem fogja összeszerkeszteni a programot, mert *x*-et kétszer definiáltuk. Jegyezzük meg, hogy az *f()* meghívása a C nyelvben nem lenne hiba (§B.2.2).

Az olyan neveket, amelyeket a nevet meghatározó fordítási egységtől különböző fordítási egységben is használhatunk, ***külső szerkesztésűnek*** (external linkage) nevezzük. Az előző példákban szereplő összes név külső név. Az olyan neveket, amelyekre csak abban a fordítási egységben lehet hivatkozni, ahol meghatározásuk szerepel, ***belső szerkesztésű*** névnek nevezzük.

A helyben kifejtett (inline) függvényeket (§7.1.1, §10.2.9) minden olyan fordítási egységben definiálni kell – azonos módon (§9.2.3) –, amelyben használatosak. Ezért a következő példa nem csak rossz stílusra vall, hanem szabálytalan is:

```
// file1.c:
  inline int f(int i) { return i; }

// file2.c:
  inline int f(int i) { return i+1; }
```

Sajnos, ezt a hibát a C++ egyes változatai nehezen veszik észre, ezért a helyben kifejtett kód és a külső szerkesztés következő – különben teljesen logikus – párosítása tiltott, hogy a fordítóprogram-írók élete könnyebb legyen:

```
// file1.c:
  extern inline int g(int i);
  int h(int i) { return g(i); }   // hiba: g() nincs definiálva ebben a fordítási egységben

// file2.c:
  extern inline int g(int i) { return i+1; }
```

Alapértelmezés szerint a *const*-ok (§5.4) és a *typedef*-ek (§4.9.7) belső szerkesztésűek. Következésképpen ez a példa szabályos (bár zavaró lehet):

```
// file1.c:
  typedef int T;
  const int x = 7;

// file2.c:
  typedef void T;
  const int x = 8;
```

Az olyan globális változók, amelyek egy adott fordítási egységben lokálisnak számítanak, gyakran okoznak zavart, ezért legjobb elkerülni őket. A globális konstansokat és a helyben kifejtett függvényeket rendszerint csak fejállományokba (§9.2.1) szabadna tennünk, hogy biztosítsuk a következetességet. A konstansokat kifejezett utasítással tehetjük külső szerkesztésűvé:

```
// file1.c:
  extern const int a = 77;

// file2.c:
  extern const int a;

  void g()
  {
        cout << a << '\n';
  }
```

Itt *g()* 77-et fog kiírni.

A névtelen névtereket (§8.2.5) arra használhatjuk, hogy a neveket egy adott fordítási egységre nézve lokálissá tegyük. A névtelen névterek és a belső szerkesztés hatása nagyon hasonló:

```
// file 1.c:
  namespace {
        class X { /* ... */ };
        void f();
        int i;
        // ...
  }

// file2.c:
  class X { /* ... */ };
  void f();
  int i;
  // ...
```

A *file1.c*-ben lévő *f()* függvény nem azonos a *file2.c*-ben lévő *f()* függvénnyel. Ha van egy adott fordítási egységre nézve lokális nevünk és ugyanazt a nevet használjuk máshol egy külső szerkesztésű egyed számára is, akkor magunk keressük a bajt.

A C nyelvű és a régebbi C++ programokban a *static* kulcsszót használták (zavaróan) annak a kifejezésére, hogy „használj belső szerkesztést" (§B.2.3). A *static* kulcsszót lehetőleg csak függvényeken (§7.2.1) és osztályokon (§10.2.4) belül használjuk.

9.2.1. Fejállományok

A típusoknak ugyanannak az objektumnak, függvénynek, osztálynak stb. minden deklarációjában egységesnek kell lenniük, következésképpen a fordítónak átadott és később összeszerkesztett forráskódnak is. A különböző fordítási egységekben lévő deklarációk egységességének elérésére nem tökéletes, de egyszerű módszer, hogy a végrehajtható kódot és/vagy adatleírásokat tartalmazó forrásfájlokba beépítjük (*#include*) a felületre vonatkozó információkat tartalmazó fejállományokat (header).

Az *#include* szövegkezelő eszköz, ami arra való, hogy a forráskód-részeket egyetlen egységbe (fájlba) gyűjtsük össze a fordításhoz. Az

```
#include "beépítendő"
```

utasítás a *beépítendő* fájl tartalmára cseréli azt a sort, amelyben az *#include* előfordul. A fájl tartalmának C++ forrásszövegnek kell lennie, mert a fordítóprogram ennek olvasásával halad tovább.

A standard könyvtárbeli fejállományok beépítéséhez a fájl nevét idézőjelek helyett a < > zárójelpárok közé kell foglalni:

```
#include <iostream>          // a szabványos include könyvtárból
#include "myheader.h"        // az aktuális könyvtárból
```

Sajnos a beépítő utasításban a szóközök mind a < >, mind a *" "* belsejében fontosak:

```
#include < iostream >        // nem fogja megtalálni az <iostream>-et
```

Furcsának tűnhet, hogy egy fájlt minden egyes alkalommal újra kell fordítani, ha valahová máshová beépítjük, de a beépített fájlok jellemzően csak deklarációkat tartalmaznak, és nem olyan kódot, amelyet a fordítóprogramnak alaposan elemeznie kellene. Továbbá a leg-

több modern C++-változat valamilyen formában támogatja az előfordított fejállományokat, hogy csökkentse a munkát, amit ugyanannak a fejállománynak az ismételt fordítása jelent.

Alapszabályként fogadjuk el, hogy egy fejállományban a következők szerepelhetnek:

Nevesített névterek	*namespace N { /* … */ }*
Típusdefiníciók	*struct Point { int x, y; };*
Sablondeklarációk	*template<class T> class Z;*
Sablondefiníciók	*template<class T> class V { /* … */ };*
Függvénydeklarációk	*extern int strlen(const char*);*
Helyben kifejtett függvények definíciói	*inline char get(char* p) { return *p++; }*
Adatdeklarációk	*extern int a;*
Konstansdefiníciók	*const float pi = 3.141593;*
Felsorolások	*enum Light { red, yellow, green };*
Névdeklarációk	*class Matrix;*
Beépítő utasítások	*#include <algorithm>*
Makródefiníciók	*#define VERSION 12*
Feltételes fordítási utasítások	*#ifdef __cplusplus*
Megjegyzések	*/* check for end of file */*

Mindez nem nyelvi követelmény, csak ésszerű módja az *#include* használatának a logikai szerkezet kifejezésére. Ezzel ellentétben egy fejállomány sohasem tartalmazhatja a következőket:

Közönséges függvénydefiníciók	*char get(char *p) { return *p++; }*
Adatdefiníciók	*int a;*
Agregátum-definíciók	*short tbl[ ] = { 1, 2, 3 };*
Névtelen névterek	*namespace { /* … */ }*
Exportált sablondefiníciók	*export template<class T>f(T t) { /* … */ }*

A fejállományok hagyomány szerint *.h* kiterjesztésűek, a függvény- és adatdefiníciókat tartalmazó fájlok kiterjesztése pedig *.c*, ezért gyakran hívják ezeket „.h fájlok”-nak és „.c fájlok”-nak. Más szokásos jelöléseket is találhatunk, mint a *.C*, *.cxx*, *.cpp* és *.cc*. Fordítóprogramunk dokumentációja ezt jól meghatározza.

Az egyszerű állandókat ajánlatos fejállományokba tenni. Az agregátumokat azonban nem, mert az egyes C++-változatok nehezen tudják elkerülni, hogy a több fordítási egységben előforduló egyedeiből másodpéldányt készítsenek. Ezenkívül az egyszerű esetek sokkal gyakoribbak, ezért jó kód készítéséhez fontosabbak. Bölcs dolog nem túl okosnak lenni az *#include* használatánál. Azt ajánlom, csak teljes deklarációkat és definíciókat építsünk be és csak globális hatókörben, szerkesztési blokkokban, vagy olyan névtérdefinícióknál, amikor régi kódot alakítunk át (§9.2.2) tegyük ezt. Célszerű elkerülni a makrókkal való ügyeskedést is. Az egyik legkevésbé kedvelt foglalatosságom olyan hibát nyomon követni, amit egy olyan név okoz, amelyet egy közvetetten beépített, számomra teljesen ismeretlen fejállományban szereplő makró helyettesít.

9.2.2. A standard könyvtár fejállományai

A standard könyvtár eszközeit szabványos fejállományok halmazán keresztül mutatjuk be (§16.1.2). A standard könyvtárbeli fejállományokat nem kell utótaggal ellátnunk; tudjuk róluk, hogy fejállományok, mert beillesztésükhöz az *#include* <...> formát használjuk az *#include "..."* helyett. A *.h* kiterjesztés hiánya nem utal semmire a fejállomány tárolásával kapcsolatban. Egy olyan fejállomány, mint a <*map*>, valószínűleg a *map.h* nevű szövegfájlban tárolódik a szokásos könyvtárban. Másfelől a szabványos fejállományokat nem muszáj hagyományos módon tárolni. Az egyes C++-változatok számára megengedett, hogy kihasználják a standard könyvtár definícióinak ismeretét és ezáltal optimalizálják annak megvalósítását, illetve a szabványos fejállományok kezelésének módját. A nyelv adott megvalósítása ismerheti a beépített szabványos matematikai könyvtárat (§22.3) és úgy kezelheti az *#include* <*cmath*> utasítást, mint egy kapcsolót, ami anélkül teszi elérhetővé a szabványos matematikai függvényeket, hogy bármilyen fájlt beolvasnánk. A C standard könyvtárának minden <*X.h*> fejállományához létezik megfelelő szabványos <*cX*> C++ fejállomány. Az *#include* <*cstdio*> például azt nyújtja, amit az *#include* <*stdio.h*>. A *stdio.h* fájl általában valahogy így néz ki:

```
#ifdef __cplusplus              // csak C++ fordítók számára (§9.2.4)
namespace std {                 // a standard könyvtárat az std névtér írja le (§8.2.9)
extern "C" {                    // az stdio függvények C szerkesztésűek (§9.2.4)
#endif
  // ...
  int printf(const char* ...);
  // ...
#ifdef __cplusplus
}
}
using namespace std;            // az stdio elérhetővé tétele a globális névtérben
#endif
```

Azaz a deklarációk (nagy valószínűséggel) közösek, de az összeszerkesztéssel és névterekkel kapcsolatos dolgokra oda kell figyelnünk, hogy lehetővé tegyük, hogy a C és C++ osztozzanak a fejállományon.

9.2.3. Az egyszeri definiálás szabálya

Egy adott osztályt, felsorolást, sablont stb. mindig csak egyszer definiálhatunk egy programban. Gyakorlati szempontból ez azt jelenti, hogy például egy osztálynak, amelyet valahol egy fájlban tárolunk, pontosan egy kifejtéssel kell rendelkeznie. Sajnos, a nyelvi szabály nem lehet ennyire egyszerű. Egy osztály definícióját például össze lehet állítani makrók behelyettesítésével is, de *#include* utasításokkal (§9.2.1) szöveges formában két forrásfájlban is el lehet helyezni. Még ennél is nagyobb baj, hogy a „fájl" fogalma nem része a C és C++ nyelvnek, így vannak olyan változatok, amelyek a programokat nem forrásfájlokban tárolják.

Következésképpen a szabványban lévő szabályt – amely azt mondja, hogy egy osztály, sablon stb. definíciójának egyedinek kell lennie – valamelyest bonyolultabb és ravaszabb módon fogalmaztuk meg. Ezt a szabályt gyakran az „egyszeri definiálás szabályának" (ODR, one-definition rule) nevezik. Azaz egy osztály, sablon, vagy helyben kifejtett függvény kétféle definiálása kizárólag akkor fogadható el ugyanazon egyed két példányaként, ha

1. különböző fordítási egységben szerepelnek és
2. szimbólumról szimbólumra megegyeznek és
3. ezen szimbólumok jelentése mindkét fordítási egységben ugyanaz.

Például:

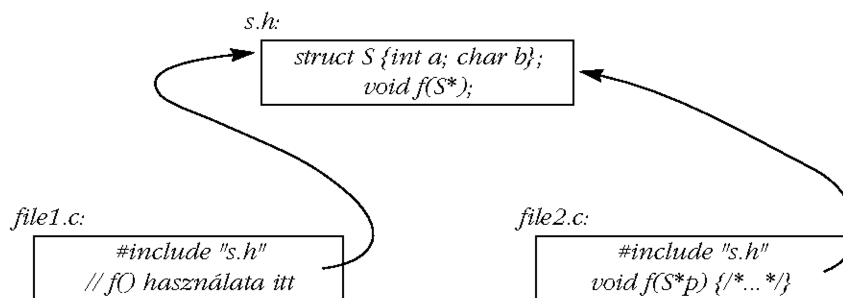
```
// file1.c:
   struct S { int a; char b; };
   void f(S*);

// file2.c:
   struct S { int a; char b; };
   void f(S* p) { /* ... */ }
```

Az ODR értelmében a fenti példa helyes és *S* ugyanarra az osztályra vonatkozik mindkét forrásfájlban. Nem bölcs dolog azonban egy definíciót ilyen módon kétszer leírni. Ha valaki módosítja a *file2.c*-t, azt feltételezheti, hogy az ott szereplő *S* az *S* egyetlen definiálása és szabadon megváltoztathatja azt, ami nehezen felfedezhető hibát okozhat.

Az ODR szándéka az, hogy megengedje egy osztálydefiníció beillesztését különböző forrásfájlokba egy közös forrásfájlból:

```
// file s.h:
  struct S { int a; char b; };
  void f(S*);

// file1.c:
  #include "s.h"
  // f() használata itt

// file2.c:
  #include "s.h"
  void f(S* p) { /* ... */ }
```

Ábrával:



Nézzünk példákat az ODR szabály megsértésének mindhárom módjára:

```
// file1.c:
  struct S1 { int a; char b; };

  struct S1 { int a; char b; };  // hiba: két definíció
```

Ez azért hiba, mert egy *struct*-ot egyetlen fordítási egységben nem lehet kétszer definiálni.

```
// file1.c:
  struct S2 { int a; char b; };

// file2.c:
  struct S2 { int a; char bb; };// hiba
```

Ez azért hiba, mert *S2* olyan osztályokat nevez meg, amelyek egy tag nevében különböznek.

```
// file1.c:
  typedef int X;
  struct S3 { X a; char b; };

// file2.c:
  typedef char X;
  struct S3 { X a; char b; };      // hiba
```

Itt az *S3* két definiciója szimbólumról szimbólumra megegyezik, de a példa hibás, mert az *X* név (trükkös módon) mást jelent a két fájlban.

A legtöbb C++-változat nem képes a különböző fordítási egységekben lévő osztály-definíciók következetességét ellenőrizni, ezért az ODR-t megsértő deklarációk nehezen észrevehető hibákat okozhatnak. Sajnos az a módszer sem képes az ODR utolsóként bemutatott megszegése ellen védelmet nyújtani, amikor a közös definiciókat fejállományokba tesszük és aztán azokat építjük be. A helyi *typedef*-ek és makrók ugyanis módosíthatják a beépített deklarációk jelentését:

```
// file s.h:
  struct S { Point a; char b; };

// file1.c:
  #define Point int
  #include "s.h"
  // ...

// file2.c:
  class Point { /* ... */ };
  #include "s.h"
  // ...
```

Az ilyen kódmódosulás ellen úgy védekezhetünk a legjobban, ha a fejállományokat annyira különállóvá tesszük, amennyire csak lehetséges. Például ha a *Point* osztályt az *s.h* állományban vezettük volna be, a fordítóprogram felismerte volna a hibát.

A sablondefiníciókat több fordítási egységbe is beépíthetjük, amíg ez nem sérti az ODR-t, az exportált sablonokat pedig úgy is használhatjuk, hogy csak a deklarációjukat adjuk meg:

```
// file1.c:
  export template<class T> T twice(T t) { return t+t; }
```

```
// file2.c:
  template<class T> T twice(T t);                    // deklaráció
  int g(int i) { return twice(i); }
```

Az *export* kulcsszó azt jelenti, hogy „más fordítási egységből elérhető" (§13.7).

9.2.4. Összeszerkesztés nem C++ kóddal

A C++ programok általában más nyelven megírt részleteket is tartalmaznak. Hasonlóan gyakori az is, hogy C++ kódrészletet használnak más nyelven megírt programok részeként. Az együttműködés a különböző nyelven megírt programrészek között nem mindig könnyű – sőt még az azonos nyelven írt, de különböző fordítóprogrammal lefordított kódrészletek között sem. A különböző nyelvek és ugyanazon nyelv különböző megvalósításai például különbözőképpen használhatják a gépi regisztereket a paraméterek tárolására, másképpen helyezhetik azokat a verembe, különböző lehet a beépített típusok, például a karakterláncok és egészek szerkezete, illetve azon nevek formája, melyeket a fordítóprogram a szerkesztőnek átad, és a szerkesztőtől megkövetelt típusellenőrzések. Hogy segítsünk, összeszerkesztési szabályt határozhatunk meg az *extern* deklarációkra. A következő példa bevezeti a C és a C++ standard könyvtáraiban levő *strcpy()* függvényt, és meghatározza, hogy a C összeszerkesztési szabályainak megfelelően kell azt hozzászerkeszteni a kódhoz:

```
extern "C" char* strcpy(char*, const char*);
```

A deklaráció hatása a „sima" deklarációkétól csak az *strcpy()* hívására használt összeszerkesztési szabályban tér el.

```
extern char* strcpy(char*, const char*);
```

Az *extern "C"* utasítás különösen fontos a C és a C++ közötti szoros kapcsolat miatt. Jegyezzük meg, hogy az *extern "C"*-ben szereplő *"C"* az összeszerkesztési szabályt, nem pedig a programnyelvet jelöli. Az *extern "C"*-t gyakran használják olyan Fortran vagy assembler eljárásokkal való összeszerkesztéshez, melyek véletlenül éppen megfelelnek a C követelményeinek.

Az *extern "C"* utasítás (csak) az összeszerkesztési szabályt határozza meg, a függvényhívások szerepét nem befolyásolja. Az *extern "C"*-ként megadott függvényekre is a C++ típusellenőrzési és paraméter-átalakítási szabályai vonatkoznak, nem pedig a gyengébb C szabályok.

Például:

```
extern "C" int f();

int g()
{
   return f(1);         // hiba: nem vár paramétert
}
```

Kényelmetlen lehet, ha sok deklarációhoz kell hozzáadnunk az *extern "C"*-t, ezért bevezettünk egy eljárást, mellyel deklarációk egy csoportjának összeszerkesztését határozhatjuk meg:

```
extern "C" {
   char* strcpy(char*, const char*);
   int strcmp(const char*, const char*);
   int strlen(const char*);
   // ...
}
```

Ezt a szerkezetet, melyet gyakran *szerkesztési blokknak* (linkage block) neveznek, úgy is használhatjuk, hogy belefoglalunk egy teljes C fejállományt és így alkalmassá tesszük azt a C++-ban való használatra:

```
extern "C" {
#include <string.h>
}
```

A fenti módszer gyakran használatos arra, hogy C fejállományokból C++ fejállományokat hozzanak létre. Egy másik lehetőség, ha feltételes fordítást (§7.8.1) használunk, hogy közös C és C++ fejállományt készítsünk:

```
#ifdef __cplusplus
extern "C" {
#endif

   char* strcpy(char*, const char*);
   int strcmp(const char*, const char*);
   int strlen(const char*);
   // ...

#ifdef __cplusplus
}
#endif
```

A „készen kapott” *__cplusplus* makró használatával azt biztosíthatjuk, hogy a C++ szerkezetek eltűnjenek, amikor a fájlt C fejállományként használjuk.

Egy szerkesztési blokkon belül bármilyen deklaráció szerepelhet:

```
extern "C" {          // bármilyen deklaráció jöhet ide, pl:
  int g1;             // definíció
  extern int g2;      // deklaráció, nem definíció
}
```

Ez a változók hatókörét és tárolási osztályát nem érinti, így *g1* globális változó marad, és definíciója is lesz, nem csak deklarációja. Ha egy változót csak deklarálni, nem pedig definiálni akarunk, az *extern* kulcsszót közvetlenül a deklaráció előtt kell megadnunk:

```
extern "C" int g3;    // deklaráció, nem definíció
```

Ez első látásra furcsának tűnik, pedig célja csak annyi, hogy a deklaráció jelentése változatlan maradjon, amikor egy *extern* deklarációhoz *"C"*-t adunk (és a fájl jelentése is, amikor majd a szerkesztési blokkba foglaljuk).

Egy C szerkesztésű nevet névtérben is megadhatunk. A névtér azt befolyásolni fogja, hogyan lehet a névhez hozzáférni C++ programokból, de azt nem, hogy a szerkesztő hogyan fogja látni a nevet. Egy jellemző példa erre az *std* névtér *printf()* függvénye:

```
#include<cstdio>

void f()
{
  std::printf("Helló, ");        // rendben
  printf("világ!\n");            // hiba: nincs globális printf()
}
```

Még ha *std::printf()*-nek nevezzük is, ez még mindig ugyanaz a régi C *printf()* (§21.8). Ez lehetővé teszi számunkra, hogy C szerkesztésű könyvtárakat építsünk be egy általunk választott névtérbe, ahelyett, hogy a globális névteret „szennyeznénk". Sajnos ugyanez a rugalmasság nem áll rendelkezésünkre az olyan fejállományok esetében, amelyek C++ szerkesztésű függvényeket határoznak meg a globális névtérben. Ennek az az oka, hogy a C++ egyedek összeszerkesztésénél figyelembe kell venni a névtereket is, így a létrehozott tárgykód (*object* fájl) tükrözni fogja a névterek használatát vagy annak hiányát.

9.2.5. Az összeszerkesztés és a függvényekre hivatkozó mutatók

Ha egy programban a C és C++ kódrészleteket keverjük, előfordulhat, hogy az egyik nyelven megírt függvényekre hivatkozó mutatókat a másik nyelven definiált függvényeknek szeretnénk átadni. Ha a két nyelv adott változatainak összeszerkesztési szabályai, illetve a függvényhívási eljárások közösek, a függvényekre hivatkozó mutatók átadása egyszerű. Ennyi közös tulajdonság azonban általában nem tételezhető fel, így figyelnünk kell arra, hogy biztosítsuk a függvények oly módon történő meghívását, ahogy azt a függvény elvárja. Ha egy deklaráció számára meghatározzuk az összeszerkesztési módot, akkor az minden olyan függvénytípusra, függvénynévre, és változónévre vonatkozni fog, amit a deklaráció(k) bevezet(nek). Ez mindenféle furcsa – de néha alapvető – összeszerkesztési módot lehetővé tesz. Például:

```
typedef int (*FT)(const void*, const void*);                // FT C++ szerkesztésű

extern "C" {
  typedef int (*CFT)(const void*, const void*);             // CFT C szerkesztésű
  void qsort(void* p, size_t n, size_t sz, CFT cmp);        // cmp C szerkesztésű
}

void isort(void* p, size_t n, size_t sz, FT cmp);           // cmp C++ szerkesztésű
void xsort(void* p, size_t n, size_t sz, CFT cmp);          // cmp C szerkesztésű
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp); // cmp C++ szerkesztésű

int compare(const void*, const void*);                      // compare() C++ szerkesztésű
extern "C" int ccmp(const void*, const void*);              // ccmp() C szerkesztésű

void f(char* v, int sz)
{
  qsort(v,sz,1,&compare);   // hiba
  qsort(v,sz,1,&ccmp);      // rendben

  isort(v,sz,1,&compare);   // rendben
  isort(v,sz,1,&ccmp);      // hiba
}
```

Egy olyan nyelvi változat, amelyben a C és C++ ugyanazt a függvényhívási módot használja, nyelvi kiterjesztésként elfogadhatja a hibaként megjelölt eseteket.

9.3. Fejállományok használata

A fejállományok használatának illusztrálására most bemutatjuk a számológép program (§6.1, §8.2) néhány lehetséges fizikai elrendezését.

9.3.1. Egyetlen fejállományos elrendezés

Egy programot úgy bonthatunk a legegyszerűbben több fájlra, hogy a definíciókat megfelelő számú *.c* fájlba, a *.c* fájlok közötti kapcsolatot biztosító típusok deklarációit pedig egyetlen *.h* fájlba tesszük, melyet minden *.c* fájl beépít (*#include*). A számológép program esetében öt *.c* fájlt – *lexer.c*, *parser.c*, *table.c*, *error.c* és *main.c* – használhatnánk a függvények és adatleírások tárolására, és a *dc.h* fejállományban tárolhatnánk azoknak a neveknek a deklarációit, amelyek egynél több fájlban használatosak.

A *dc.h* fejállomány így nézne ki:

```
// dc.h:

namespace Error {
  struct Zero_divide { };

  struct Syntax_error {
            const char* p;
            Syntax_error(const char* q) { p = q; }
  };
}

#include <string>

namespace Lexer {

  enum Token_value {
            NAME,           NUMBER,          END,
            PLUS='+',       MINUS='-',       MUL='*',        DIV='/',
            PRINT=';',      ASSIGN='=',      LP='(',         RP=')'
  };

  extern Token_value curr_tok;
  extern double number_value;
  extern std::string string_value;

  Token_value get_token();
}
```

```
namespace Parser {
    double prim(bool get);      // elemi szimbólumok kezelése
    double term(bool get);      // szorzás és osztás
    double expr(bool get);      // összeadás és kivonás

    using Lexer::get_token;
    using Lexer::curr_tok;
}

#include <map>

extern std::map<std::string,double> table;

namespace Driver {
    extern int no_of_errors;
    extern std::istream* input;
    void skip();
}
```

Minden változódeklarációban az *extern* kulcsszót használjuk annak biztosítására, hogy egy meghatározás ne forduljon elő többször is, amikor a *dc.h*-t a fájlokba beépítjük. Az egyes definíciók a megfelelő *.c* fájlban szerepelnek.

A lényegi kód elhagyásával a *lexer.c* valahogy így néz ki:

```
// lexer.c:

#include "dc.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

A fejállomány ilyen használata biztosítja, hogy a benne lévő minden deklaráció valamilyen ponton be legyen építve abba a fájlba, amely a hozzá tartozó kifejtést tartalmazza. A *lexer.c* fordításakor például a fordítóprogramnak a következő kód adódik át:

```
namespace Lexer {           // a dc.h-ból
    // ...
    Token_value get_token();
}

// ...
Lexer::Token_value Lexer::get_token() { /* ... */ }
```

Így a fordítóprogram biztosan észreveszi, ha egy névhez megadott típusok nem egységesek. Ha a *get_token()*-t például *Token_value* típusú visszatérési értékkel vezettük volna be, de *int* visszatérési értékkel definiáltuk volna, a *lexer.c* fordítása típusütközési vagy „nem megfelelő típus" (type mismatch) hiba miatt nem sikerült volna. Ha egy definíció hiányzik, a szerkesztő fogja észrevenni a problémát, ha egy deklaráció, akkor valamelyik *.c* fájl fordítása hiúsul meg.

A *parser.c* fájl így fog kinézni:

```
// parser.c:

#include "dc.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

A *table.c* pedig így:

```
// table.c:

#include "dc.h"

std::map<std::string,double> table;
```

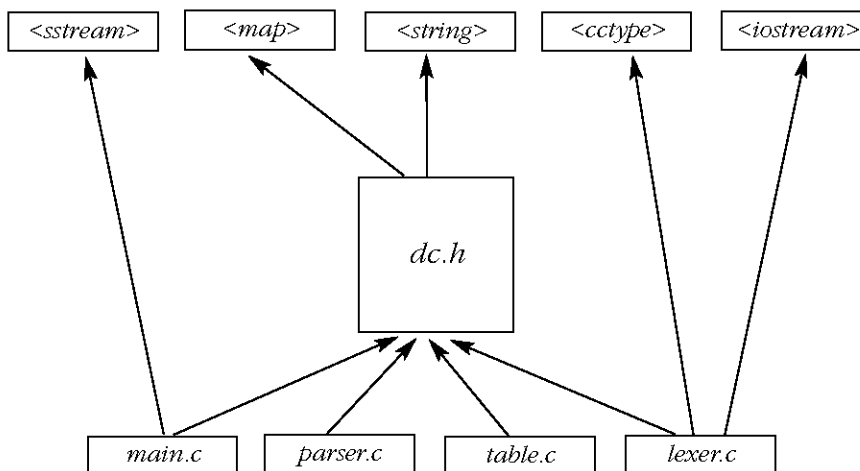
A szimbólumtábla nem más, mint egy standard könyvtárbeli *map* típusú változó. A fenti definíció a *table*-t globálisként határozza meg. Egy valóságos méretű programban a globális névtér efféle kis „szennyeződései" felhalmozódnak és végül problémákat okoznak. Csak azért voltam itt ilyen hanyag, hogy lehetőségem legyen figyelmeztetni rá.

A *main.c* fájl végül így fog kinézni:

```
// main.c:

#include "dc.h"
#include <sstream>

int Driver::no_of_errors = 0;
std::istream* Driver::input = 0;

void Driver::skip() { /* ... */ }

int main(int argc, char* argv[ ]) { /* ... */ }
```

Ahhoz, hogy a program *main()* függvényének ismerjék fel, a *main()*-nek globális függvénynek kell lennie, ezért itt nem használtunk névteret.

A program fizikai elrendezését valahogy így lehet bemutatni:



Észrevehetjük, hogy a felül lévő fejállományok mind a standard könyvtár fájljai. A program elemzésekor ezek a könyvtárak számos esetben kihagyhatók, mert széleskörűen ismertek és stabilak. A kis programoknál az elrendezés egyszerűsíthető, ha minden *#include* utasítást közös fejállományba teszünk.

Az egyetlen fejállományos fizikai részekre bontás akkor a leghasznosabb, ha a program kicsi és részeit nem áll szándékunkban külön használni. Jegyezzük meg, hogy amikor névtereket használunk, a *dc.h*-ban egyben a program logikai felépítését is ábrázoljuk. Ha nem használunk névtereket, a szerkezet homályos lesz, bár ezen a megjegyzések segíthetnek.

A nagyobb programok egyetlen fejállományos elrendezése nem működik a hagyományos, fájl alapú fejlesztőkörnyezetekben. A közös fejállomány módosítása maga után vonja az egész program újrafordítását és nagy a hibalehetőség, ha több programozó is módosítja az egyetlen fejállományt. Hacsak nem fektetnek hangsúlyt a névterekkel és osztályokkal kapcsolatos programozási stílusra, a logikai felépítés a program növekedésével együtt romlani fog.

9.3.2. Több fejállományos elrendezés

Egy másik fizikai elrendezés szerint minden logikai modulnak saját fejállománya lenne, amely leírja a modul által nyújtott szolgáltatásokat. Ekkor minden *.c* fájlhoz tartozik egy megfelelő *.h* fájl, ami meghatározza a *.c* szolgáltatásait (felületét). Minden *.c* fájl beépíti a saját *.h* fájlját, és rendszerint további olyan *.h* fájlokat is, amelyek meghatározzák, mire van szüksége más modulokból ahhoz, hogy megvalósítsa a felületében közzétett szolgáltatásokat. Ez a fizikai elrendezés megegyezik a modul logikai felépítésével. A felhasználóknak szánt felületet a *.h* fájl tartalmazza, a programozói felület egy *_impl.h* végződésű fájlban szerepel, a modul függvény- és változódefiníciói stb. pedig a *.c* fájlokban vannak elhelyezve. Ily módon az elemzőt három fájl képviseli, felhasználói felületét pedig a *parser.h* nyújtja:

```
// parser.h:

namespace Parser {              // felület a felhasználóknak
  double expr(bool get);
}
```

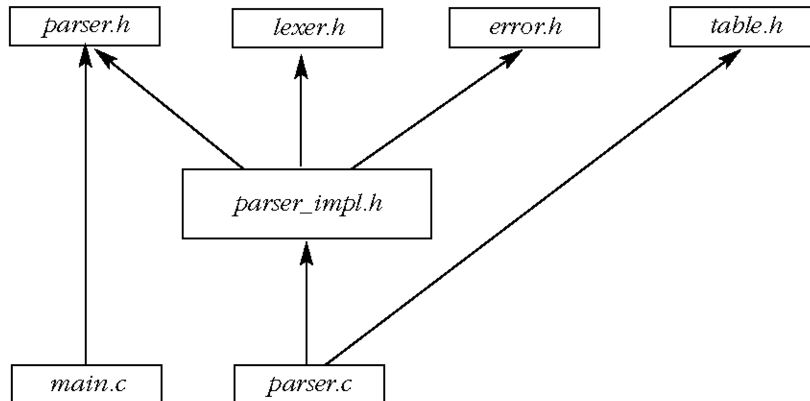
Az elemzőt megvalósító függvények közös környezetét a *parser_impl.h* adja:

```
// parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"

namespace Parser {              // felület a megvalósításhoz
  double prim(bool get);
  double term(bool get);
  double expr(bool get);
  using Lexer::get_token;
  using Lexer::curr_tok;
}
```

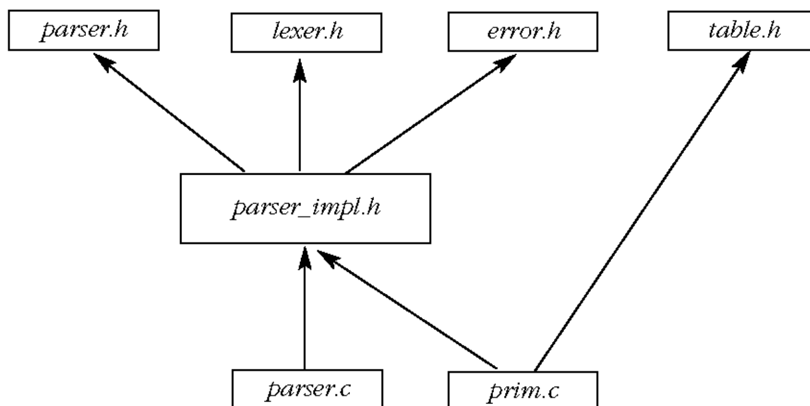
A *parser.h* felhasználói fejállományt azért építjük be, hogy a fordítóprogram ellenőrizhesse a következetességet (§9.3.1). Az elemző függvényeket a *parser.c* fájlban együtt tároljuk azokra a fejállományokra vonatkozó *#include* utasításokkal, melyekre a *Parser* függvényeinek szüksége van:

```
// parser.c:

#include "parser_impl.h"
#include "table.h"
double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

Az elemző és annak a vezérlő általi használata ábrával így mutatható be:



Ahogy vártuk, ez elég jól egyezik a §8.3.3-ban leírt logikai szerkezettel. Ha a *table.h*-t a *parser_impl.h*-ba építettük volna be a *parser.c* helyett, a szerkezetet még tovább egyszerűsíthettük volna. A *table.h* azonban valami olyasmire példa, ami nem szükséges az elemző függvények közös környezetének kifejezéséhez, csak a függvények megvalósításainak van szüksége rá. Tulajdonképpen egyetlen függvény, a *prim()* használja, így ha a függőségeket valóban a lehető legkevesebbre szeretnénk csökkenteni, a *prim()*-et tegyük külön *.c* fájlba és csak oda építsük be a *table.h*-t:



Ilyen alaposságra a nagyobb modulokat kivéve nincs szükség. A valóságos méretű modulok esetében gyakori, hogy további fájlokat építenek be ott, ahol egyes függvények számára azok szükségesek. Továbbá nem ritka, hogy egynél több *_impl.h* fájl van, mivel a modul függvényeinek részhalmazai különböző közös környezetet igényelnek.

Meg kell jegyeznünk, hogy az *_impl.h* használata nem szabványos és még csak nem is gyakori megoldás – én egyszerűen így szeretek elnevezni dolgokat.

Miért törődünk ezzel a bonyolultabb több fejállományos elrendezéssel? Nyilván sokkal kevesebb gondolkodást igényel, ha egyszerűen minden deklarációt bedobunk egy fejállományba, mint ahogy azt a *dc.h*-nál tettük.

A több fejállományos elrendezés olyan modulok és programok esetében hatásos, amelyek nagyságrendekkel nagyobbak, mint a mi apró elemzőnk és számológépünk. Alapvetően azért használtuk ezt az elrendezéstípust, mert jobban azonosítja a kapcsolatokat. Egy nagy program elemzésekor vagy módosításakor alapvető, hogy a programozó viszonylag kis kódrészletre összpontosíthasson. A több fejállományos elrendezés segít, hogy pontosan eldönthessük, mitől függ az elemző kód, és hogy figyelmen kívül hagyhassuk a program többi részét. Az egyetlen fejállományos elrendezés rákényszerít minket, hogy minden olyan deklarációt megnézzünk, amelyet valamelyik modul használ, és eldöntsük, hogy odaillő-e. A lényeg, hogy a kód módosítása mindig hiányos információk és helyi nézőpont alapján történik. A több fejállományos elrendezés megengedi, hogy sikeresen dolgozzunk „belülről kifelé", csak helyi szemszögből. Az egyetlen fejállományos elrendezés – mint minden más elrendezés, ahol egy globális információtár van a középpontban – felülről lefelé haladó megközelítést igényel, így örökké gondolkodnunk kell azon, hogy pontosan mi függ egy másik dologtól.

A jobb összpontosítás azt eredményezi, hogy kevesebb információ kell a modul lefordításához, így az gyorsabban történik. A hatás drámai lehet. Előfordul, hogy a fordítási idő tizedrészére csökken, pusztán azért, mert egy egyszerű függőségelemzés a fejállományok jobb használatához vezet.

9.3.2.1. A számológép egyéb moduljai

A számológép többi modulját az elemzőhöz hasonlóan rendezhetjük el. Ezek a modulok azonban olyan kicsik, hogy nem igényelnek saját *_impl.h* fájlokat. Az ilyen fájlok csak ott kellenek, ahol egy logikai modul sok függvényből áll, amelyeknek közös környezetre van szükségük.

A hibakezelőt a kivételtípusok halmazára egyszerűsítettük, így nincs szükségünk az *error.c*-re:

```
// error.h:

namespace Error {
  struct Zero_divide { };

  struct Syntax_error {
          const char* p;
          Syntax_error(const char* q) { p = q; }
  };
}
```

Az adatbeviteli kód (lexikai elemző, lexer) meglehetősen nagy és rendezetlen felületet nyújt:

```
// lexer.h:

#include <string>

namespace Lexer {

  enum Token_value {
          NAME,          NUMBER,        END,
          PLUS='+',      MINUS='-',     MUL='*',      DIV='/',
          PRINT=';',     ASSIGN='=',    LP='(',       RP=')'
  };

  extern Token_value curr_tok;
  extern double number_value;
  extern std::string string_value;

  Token_value get_token();
}
```

A *lexer.h*-n kívül a lexikai elemző az *error.h*-ra, az <*iostream*>-re és a <*ctype*>-ban megadott, a karakterek fajtáit eldöntő függvényekre támaszkodik:

```
// lexer.c:

#include "lexer.h"
#include "error.h"
#include <iostream>
#include <cctype>
```

```
Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

Az *error.h #include* utasításait külön tehettük volna, a *Lexer*-hez tartozó *_impl.h* fájlba, ez azonban túlzás egy ilyen kis program esetében.

A modul megvalósításában szokásos módon építjük be (*#include*) a modul által nyújtott felületet – ebben az esetben a *lexer.h*-t –, hogy a fordítóprogram ellenőrizhesse a következetességet.

A szimbólumtábla alapvetően önálló, bár a standard könyvtárbeli *<map>* fejállomány használatával számos érdekes dolog kerülhet bele, hogy hatékonyan valósíthassa meg a *map* sablonosztályt:

```
// table.h:

#include <map>
#include <string>

extern std::map<std::string,double> table;
```

Mivel feltesszük, hogy az egyes fejállományok több *.c* fájlba is bele lehetnek építve, a *table* deklarációját külön kell választanunk annak kifejtésétől, még akkor is, ha a *table.c* és a *table.h* közötti különbség csak az *extern* kulcsszó:

```
// table.c:

#include "table.h"

std::map<std::string,double> table;
```

A vezérlő tulajdonképpen mindenre támaszkodik:

```
// main.c:

#include "parser.h"
#include "lexer.h"
#include "error.h"
#include "table.h"
```

```
namespace Driver {
  int no_of_errors;
  std::istream* input;
  void skip();
}

#include <sstream>

int main(int argc, char* argv[ ]) { /* ... */ }
```

Mivel a *Driver* névteret kizárólag a *main()* használja, a *main.c-be* tesszük azt. Külön is szerepelhetne *driver.h* fájlként, amit az *#include* utasítással beépítünk.

A nagyobb programokat rendszerint megéri úgy elrendezni, hogy a vezérlőnek kevesebb közvetlen függősége legyen. Gyakran ésszerű az olyan műveletekből is minél kevesebbet alkalmazni, amit a *main()* tesz, nevezetesen hogy meghív egy külön forrásfájlban lévő vezérlő függvényt. Ez különösen fontos olyan kódok esetében, amelyeket könyvtárként akarunk használni. Ekkor ugyanis nem támaszkodhatunk a *main()*-ben megírt kódra és fel kell készülnünk arra is, hogy különböző függvényekből hívják meg kódunkat (§9.6[8]).

9.3.2.2. A fejállományok használata

A programban használt fejállományok (header file) száma több tényezőtől függ. Ezen tényezők közül sok inkább a rendszer fájlkezeléséből adódik és nem a C++-ból. Például, ha szövegszerkesztő programunk nem képes arra, hogy több fájlt nézzünk vele egyszerre, akkor nem előnyös sok fejállományt használnunk. Hasonlóan, ha 20 darab 50 soros fájl olvasása észrevehetően több időt igényel, mint egyetlen 1000 soros fájlé, akkor kétszer is gondoljuk meg, mielőtt egy kis projektben a több fejállományos stílust használjuk.

Néhány figyelmeztetés: egy tucatnyi fejállomány (természetesen a szokásos fejállományokkal együtt, melyeket gyakran százas nagyságrendben számolhatunk) a program végrehajtási környezete számára rendszerint még kezelhető. Ha azonban egy nagy program deklarációit logikailag a lehető legkisebb fejállományokra bontjuk (például úgy, hogy minden szerkezet deklarációját külön fájlba tesszük), könnyen egy több száz fájlból álló, kezelhetetlen zűrzavar lehet az eredmény.

Nagy projekteknél persze elkerülhetetlen a sok fejállomány. Az ilyeneknél több száz fájl (nem számolva a szokásos fejállományokat) az általános. Az igazi bonyodalom ott kezdődik, amikor elérik az ezres nagyságrendet. A fent tárgyalt alapvető módszerek ekkor is alkalmazhatók, de az állományok kezelése sziszifuszi feladattá válik. Emlékezzünk, hogy

a valódi méretű programoknál az egyetlen fejállományos elrendezést általában nem választhatjuk, mert az ilyen programok rendszerint eleve több fejállományt tartalmaznak. A kétfajta elrendezési módszer között a program alkotórészeinek létrehozásakor kell (néha többször is) választanunk.

Igazán nem is a mi ízlésünkre van bízva, hogy az egyetlen és a több fejállományos elrendezés közül válasszunk. Ezek olyan egymást kiegészítő módszerek, melyeket mindig figyelembe kell vennünk a lényegi modulok tervezésekor, és újra kell gondolnunk azokat, ahogy a rendszer fejlődik. Rendkívül fontos emlékeznünk arra, hogy egy felület nem szolgálhat minden célra ugyanolyan jól. Rendszerint megéri különbséget tenni a fejlesztői és a felhasználói felület között. Ezenkívül sok nagyobb program szerkezete olyan, hogy célszerű a felhasználók többségének egyszerű, a tapasztaltabb felhasználóknak pedig terjedelmesebb felületet nyújtani. A tapasztalt felhasználók felületei (a „teljes felületek") sokkal több szolgáltatást építenek be, mint amennyiről egy átlagos felhasználónak tudnia kell. Valójában az átlagos felhasználó felületét úgy határozhatjuk meg, hogy nem építjük be azokat a fejállományokat, amelyek olyan szolgáltatásokat írnak le, amelyek ismeretlenek lennének az átlagos felhasználó számára. Az „átlagos felhasználó" kifejezés nem lekicsinylő. Ahol nem *muszáj* szakértőnek lennem, jobban szeretek átlagos felhasználó lenni. Így ugyanis kevesebb a veszekedés.

9.3.3. „Állomány-őrszemek"

A több fejállományos megközelítés gondolata az, hogy minden logikai modult következetes, önálló egységként ábrázoljunk. A program egészének szempontjából nézve viszont azon deklarációk többsége, melyek ahhoz kellenek, hogy minden logikai egység teljes legyen, felesleges. Nagyobb programoknál az ilyen fölösleg (redundancia) hibákhoz vezethet, amint egy osztályleírást vagy helyben kifejtett függvényeket tartalmazó fejállományt ugyanabban a fordítási egységben (§9.2.3) kétszer építünk be az *#include*-dal.

Két választásunk lehet.

1. Átszervezhetjük a programunkat, hogy eltávolítsuk a fölösleget, vagy
2. találunk valamilyen módot arra, hogy a fejállományok többszöri beépítése megengedett legyen.

Az első megközelítés – ami a számológép végső változatához vezetett – fárasztó és valóságos méretű programoknál gyakorlatilag kivitelezhetetlen. A fölöslegre azért is szükségünk van, hogy a program egyes egységei elkülönülten is érthetőek legyenek. A fölös *#include*-ok kiszűrése és az ennek eredményeképpen létrejött egyszerűsített program nagyon előnyös lehet mind logikai szempontból, mind azáltal, hogy csökken a fordítási idő. Az összes

előfordulás megtalálása azonban ritkán sikerül, így alkalmaznunk kell valamilyen eszközt, ami megengedi a fölös *#include*-ok jelenlétét. Lehetőleg szisztematikusan kell használnunk, mert nem tudhatjuk, hogy a felhasználó mennyire alapos elemzést tart érdemesnek.

A hagyományos megoldás az, hogy a fejállományokba *„állomány-őrszemeket"* (beépítésfigyelőket, include-guards) illesztünk:

```
// error.h:

#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace Error {
  // ...
}

#endif      // CALC_ERROR_H
```

A fájlnak az *#ifndef* és az *#endif* közötti tartalmát a fordítóprogram nem veszi figyelembe, ha a *CALC_ERROR_H* már definiált. Ezért amikor a fordítóprogram az *error.h*-val először találkozik, beolvassa annak tartalmát, a *CALC_ERROR_H* pedig értéket kap. Ha ismét találkozna vele a fordítás során, másodszor már nem fogja figyelembe venni. Ez makrókkal való ügyeskedés, de működik és mindenütt jelen van a C és C++ világában. A standard könyvtár fejállományainak mindegyike tartalmaz állomány-őrszemeket.

A fejállományokat mindenféle környezetben használják, a makrónevek ütközése ellen pedig nincs névtér védelem. Az állomány-őrszemeknek ezért hosszú és csúnya neveket szoktam választani.

Amint a programozó hozzászokik a fejállományokhoz és az állomány-őrszemekhez, hajlamos közvetlenül vagy közvetve sok fejállományt beépíteni. Ez nem kívánatos, még azoknál a C++-változatoknál sem, melyek optimalizálják a fejállományok feldolgozását. Szükségtelenül hosszú fordítási időt okozhatnak és számos deklarációt és makrót elérhetővé tehetnek, ami kiszámíthatatlanul és kedvezőtlenül befolyásolhatja a program jelentését. Csak akkor építsünk be fejállományokat, amikor tényleg szükség van rá.

9.4. Programok

A program külön fordított egységek gyűjteménye, melyet a szerkesztőprogram egyesít. Minden, ebben a gyűjteményben használt függvénynek, objektumnak, típusnak stb. egyedi meghatározással (definícióval) kell rendelkeznie (§4.9, §9.2.3) és pontosan egy *main()* nevű függvényt kell tartalmaznia (§3.2). A program által végzett fő tevékenység a *main()* meghívásával kezdődik és az abból való visszatéréssel ér véget. A *main()* által visszaadott *int* érték lesz a program visszatérési értéke, amit a *main()*-t meghívó rendszer megkap.

Ezen az egyszerű meghatározáson a globális változókat tartalmazó (§10.4.9) vagy el nem kapott kivételt (§14.7) kiváltó programok esetében finomítanunk kell.

9.4.1. Kezdeti értékadás nem lokális változóknak

Elvileg a függvényeken kívül megadott, nem lokálisnak számító változók (azaz a globális, névtér-, vagy *static* osztályváltozók) a *main()* meghívása előtt, a fordítási egységben definiciójuk sorrendjében kapnak kezdőértéket (§10.4.9). Ha egy ilyen változónak nincs pontosan meghatározott (explicit) kezdőértéke, akkor a típusának megfelelő alapértelmezett értékkel töltődik fel (§10.4.2). A beépített típusok és felsorolások esetében az alapértelmezett kezdőérték a *0*:

```
double x = 2;                    // nem lokális változók
double y;
double sqx = sqrt(x+y);
```

Itt az *x* és az *y* az *sqx* előtt kap kezdőértéket, így az *sqrt(2)* hívódik meg.

A különböző fordítási egységekben lévő globális változók kezdőértékkel való ellátásának sorrendje nem kötött, következésképpen nem bölcs dolog ezeknél a kezdőértékek között sorrendi függőségeket létrehozni. Továbbá nem lehetséges olyan kivételt sem elkapni, amit egy globális változó kezdeti értékadása váltott ki (§14.7). Általában az a legjobb, ha minél kevesebb globális változót használunk; főleg a bonyolult kezdeti értékadást igénylő globális változók használatát kell korlátoznunk.

A különböző fordítási egységekben lévő globális változók kezdeti értékkel való feltöltésének sorrendjét számos módon kényszeríthetjük ki, de nincs köztük olyan, amely egyszerre „hordozható” és hatékony is lenne. Főleg a dinamikus csatolású könyvtárak (DLL) nem képesek

a bonyolult függőségekkel rendelkező globális változókkal boldogan együtt élni. Globális változók helyett gyakran használhatunk referenciát visszaadó függvényeket:

```
int& use_count()
{
  static int uc = 0;
  return uc;
}
```

A *use_count()* hívás most globális változóként működik, kivéve, hogy első használatakor kap kezdőértéket (§5.5):

```
void f()
{
  cout << ++use_count();      // növelés és kiírás
  // ...
}
```

A nem lokális statikus változók kezdeti értékadását bármilyen eljárás vezérelheti, amit az adott nyelvi változat arra használ, hogy elindítsa a C++ programot. Csak akkor garantált, hogy a módszer megfelelően működik, ha a *main()* végrehajtására sor kerül, ezért el kell kerülnünk azon nem lokális változók használatát, melyek futási idejű kezdeti értékadást igényelnek olyan C++ kódban, amit nem C++ program használ.

Jegyezzük meg, hogy a kezdőértéket konstans kifejezésektől kapó változók (§C.5) nem függhetnek más fordítási egységben levő objektumok értékétől és nem igényelnek futási idejű kezdeti értékadást, így minden esetben biztonságosan használhatók.

9.4.1.1. A program befejezése

A programok futása számos módon érhet véget:

- A *main()*-ből való visszatéréssel
- Az *exit()* meghívásával
- Az *abort()* meghívásával
- El nem kapott kivétel kiváltásával

Továbbá többféle hibás felépítés és nyelvi változattól függő módszer létezik arra, hogy egy program összeomoljon. Ha a program befejezésére a standard könyvtárbeli *exit()* függvényt használjuk, akkor meghívódnak a létrehozott statikus objektumok destruktorai (§10.4.9, §10.2.4). Ha azonban a program a standard könyvtár *abort()* függvényét használja,

a destruktorok meghívására nem kerül sor. Jegyezzük meg: ez azt is jelenti, hogy az *exit()* nem fejezi be rögtön a programot; destruktorban való meghívása végtelen ciklust eredményezhet. Az *exit()* függvény típusa

```
void exit(int);
```

A *main()* visszatérési értékéhez (§3.2) hasonlóan az *exit()* paramétere is visszaadódik „a rendszernek" a program visszatérési értékeként. A nulla sikeres befejezést jelent.

Az *exit()* meghívása azt jelenti, hogy a hívó függvény lokális változóinak és az azt hívó függvények hasonló változóinak destruktorai nem hívódnak meg. A lokális objektumok megfelelő megsemmisítését (§14.4.7) egy kivétel „dobása" és elkapása biztosítja. Emellett az *exit()* meghívása úgy fejezi be a programot, hogy az *exit()* hívójának nem ad lehetőséget arra, hogy megoldja a problémát. Ezért gyakran az a legjobb, ha a környezetet egy kivétel kiváltásával elhagyjuk, és megengedjük egy kivételkezelőnek, hogy eldöntse, mi legyen a továbbiakban. A C (és C++) standard könyvtárának *atexit()* függvénye lehetőséget ad arra, hogy kódot hajthassunk végre a program befejeződésekor:

```
void my_cleanup();

void somewhere()
{
  if (atexit(&my_cleanup)==0) {
          // normál programbefejezéskor a my_cleanup hívódik meg
  }
  else {
          // hoppá: túl sok atexit függvény
  }
}
```

Ez nagyban hasonlít a globális változók destruktorainak a program befejeződésekor történő automatikus meghívásához (§10.4.9, §10.2.4). Jegyezzük meg, hogy az *atexit()* paraméterének nem lehet paramétere és nem adhat vissza értéket. Az *atexit* függvények számát az adott nyelvi változat korlátozza; a függvény nem nulla érték visszaadásával jelzi, ha ezt a korlátot elérték. Ezek a korlátozások az *atexit()*-et kevésbé használhatóvá teszik, mint amilyennek első pillantásra látszik.

Az *atexit(f)* meghívása előtt létrehozott objektum destruktora az *f* meghívása után fog meghívódni, az *atexit(f)* meghívása után létrehozott objektum destruktora pedig az *f* meghívása előtt.

Az *exit()*, *abort()*, és *atexit()* függvények deklarációját a *<cstdlib>* fejállomány tartalmazza.

9.5. Tanácsok

[1] Használjuk fejállományokat a felületek ábrázolására és a logikai szerkezet kihangsúlyozására. §9.1, §9.3.2.
[2] Abban a forrásfájlban építsük be őket (*#include*), amelyben függvényeiket kifejtjük. §9.3.1.
[3] Ne adjunk meg globális egyedeket ugyanazzal a névvel és hasonló, de különböző jelentéssel különböző fordítási egységekben. §9.2.
[4] Kerüljük a fejállományokban a nem helyben kifejtendő függvényeket. §9.2.1.
[5] Csak globális hatókörben és névterekben használjuk az *#include*-ot. §9.2.1.
[6] Csak teljes deklarációkat építsünk be. §9.2.1
[7] Használjunk „állomány-őrszemeket". §9.3.3.
[8] A C fejállományokat névterekben építsük be, hogy elkerüljük a globális neveket. §9.3.2.
[9] Tegyük a fejállományokat különállóvá. §9.2.3.
[10] Különböztessük meg a fejlesztői és a felhasználói felületet. §9.3.2.
[11] Különböztessük meg az átlagos és a tapasztalt felhasználók felületét. §9.3.2.
[12] Kerüljük az olyan nem lokális objektumok használatát, amelyek futási idejű kezdeti értékadást igényelnek olyan kódban, amit nem C++ program részeként szándékozunk felhasználni. §9.4.1.

9.6. Gyakorlatok

1. (*2) Találjuk meg, hol tárolja rendszerünk a szabványos fejállományokat. Írassuk ki neveiket. Van-e olyan nem szabványos fejállomány, amely ezekkel együtt tárolódik? Be lehet-e építeni nem szabványos fejállományokat a <> jelölést használva?
2. (*2) Hol tárolódnak a nem szabványos „foundation" könyvtárak fejállományai?
3. (*2,5) Írjunk programot, amely beolvas egy forrásfájlt és kiírja a beépített fájlok neveit. Használjunk behúzást a beépített fájlok által beépített fájlok kiírásakor, a befoglalás mélységének jelölésére. Próbáljuk ki a programot néhány valódi forrásfájlon (hogy elképzelésünk legyen a beépített információ nagyságáról).
4. (*3) Módosítsuk az előbbi programot, hogy minden beépített fájlra kiírja a megjegyzések és a nem megjegyzések sorainak számát, illetve a nem megjegyzésként szereplő, üreshelyekkel elválasztott szavak számát.

5. (*2,5) A külső beépítésfigyelő olyan programelem, amely a megfigyelt fájlon kívül végzi az ellenőrzést, és fordításonként csak egyszer végez beépítést. Készítsünk egy ilyen szerkezeti elemet, tervezzünk módszert a tesztelésére, és fejtsük ki előnyeit és hátrányait a §9.3.3-ban leírt „állomány-őrszemekkel" szemben. Van-e a külső beépítésfigyelőknek bármilyen jelentős futási időbeli előnye a rendszerünkben?
6. (*3) Hogyan valósul meg a dinamikus csatolás (szerkesztés) a rendszerünkben? Milyen megszorítások vonatkoznak a dinamikusan szerkesztett kódra? Milyen követelményeknek kell, hogy megfeleljen a kód, hogy dinamikusan csatolható legyen?
7. (*3) Nyissunk meg és olvassunk be 100 fájlt, melyek mindegyike 1500 karaktert tartalmaz. Nyissunk meg és olvassunk be egy 150 000 karakterből álló fájlt. Tipp: nézzük meg a példát a §21.5.1 pontban. Van-e eltérés a teljesítményben? Hány fájl lehet egyszerre megnyitva rendszerünkben? Válaszoljuk meg ezeket a kérdéseket a beépített fájlok használatával kapcsolatban is.
8. (*2) Módosítsuk a számológépet, hogy meg lehessen hívni a *main()*-ből vagy más függvényből is, egy egyszerű függvényhívással.
9. (*2) Rajzoljuk meg a „modulfüggőségi diagramokat" (§9.3.2) a számológép azon változataira, melyek az *error()*-t használták kivételek helyett. (§8.2.2).

Második rész
Absztrakciós módszerek

Ebben a részben azzal foglalkozunk, milyen lehetőségeket nyújt a C++ nyelv új típusok meghatározására és használatára, illetve bemutatjuk az összefoglaló néven objektumorientált programozásnak és általánosított (generikus) programozásnak nevezett eljárásokat.

Fejezetek

10. Osztályok
11. Operátorok túlterhelése
12. Származtatott osztályok
13. Sablonok
14. Kivételkezelés
15. Osztályhierarchiák

„... nincs nehezebb, kétesebb kimenetelű, veszélyesebb dolog, mint új törvények bevezetéséért síkraszállni. Mert ellenségei azok, akiknek a régi törvények hasznára vannak, azok pedig, akiknek az új rendelkezések szolgálnak hasznukra, pusztán lagymatag védelmezői...”

Niccolo Machiavelli
(A fejedelem (§vi), Lutter Éva fordítása)

10

Osztályok

„Ezek a típusok nem „elvontak";
ugyanannyira valóságosak,
mint az int és a float."
(Doug McIlroy)

Fogalmak és osztályok • Osztálytagok • Az elérhetőség szabályozása • Konstruktorok • Statikus tagok • Alapértelmezett másolás • *const* tagfüggvények • *this* • *struct*-ok • Osztályon belüli függvénydefiníciók • Konkrét osztályok • Tagfüggvények és segédfüggvények • Operátorok túlterhelése • A konkrét osztályok használata • Destruktorok • Alapértelmezett konstruktorok • Lokális változók • Felhasználói másolás • *new* és *delete* • Tagobjektumok • Tömbök • Statikus tárolás • Ideiglenes változók • Uniók • Tanácsok • Gyakorlatok

10.1. Bevezetés

A C++ nyelv osztályai azt a célt szolgálják, hogy a programozó a beépített adattípusokkal azonos kényelmi szinten használható új adattípusokat hozhasson létre. Ezenkívül az öröklődés (12. fejezet) és a sablonok (13. fejezet) segítségével úgy szervezhetjük az egymással kapcsolatban álló osztályokat, hogy kapcsolataikat hatékonyan használhassuk ki.

A *típus* egy fogalom konkrét ábrázolása. A C++ beépített *float* típusa például a +, -, * stb. műveleteivel együtt a valós szám matematikai fogalmának egy megközelítése. Az *osztály* egy felhasználói típus. Azért tervezünk új típust, hogy meghatározzunk egy fogalmat, amelynek nincs közvetlen megfelelője a nyelv beépített típusai között. Lehet például *Trunk_line* típusunk egy telefonos kapcsolatokat kezelő programban, *Explosion* típusunk egy videójáték számára, vagy *list<Paragraph>* típusunk egy szövegszerkesztő programban. Egy programot könnyebb megérteni és módosítani, ha abban az általa kezelt fogalmaknak megfelelő típusok szerepelnek. Ha a programozó alkalmas osztályokat használ, a program tömörebb lesz, ráadásul sokféle kódelemző eljárás használata válik lehetővé. A fordítóprogram például felderítheti az objektumok nem szabályos használatát, amit másként csak egy alapos ellenőrzés során fedezhetnénk fel.

Új típus definiálásakor az alapvető szempont a megvalósítás véletlenszerű, esetleges részleteinek (például a tárolandó adatok elrendezésének) elválasztása a típus helyes használatához alapvetően szükséges tulajdonságoktól, például az adatokat elérő függvények teljes listájától. Ez az elválasztás legjobban úgy fejezhető ki, ha az adott típus adatszerkezetét érintő összes külső használatot és belső rendrakó függvényt csak az adott típusra vonatkozó programozási felületen keresztül tesszük elérhetővé. Ez a fejezet a viszonylag egyszerű, „konkrét" felhasználói típusokkal foglalkozik. Ideális esetben ezek csak létrehozásuk módjukban különböznek a beépített típusoktól, a használat módjában nem.

10.2. Osztályok

Az *osztály* (class) a programozó által meghatározott, más néven felhasználói típus. Az alábbiakban az osztályok meghatározásának, illetve az osztályba tartozó objektumok létrehozásának és használatának főbb eszközeit mutatjuk be.

10.2.1. Tagfüggvények

Vizsgáljuk meg, hogyan ábrázolnánk a „dátum" fogalmát egy *Date* adatszerkezettel (struktúrával, *struct*) és egy sor, ilyen változókat kezelő függvénnyel:

```
struct Date {          // ábrázolás
    int d, m, y;
};
```

```
void init_date(Date& d, int, int, int);        // kezdeti értékadás d-nek
void add_year(Date& d, int n);                 // n évet ad d-hez
void add_month(Date& d, int n);                // n hónapot ad d-hez
void add_day(Date& d, int n);                  // n napot ad d-hez
```

Az adattípus és ezen függvények között nincs kifejezett kapcsolat. Ilyen kapcsolatot azáltal hozhatunk létre, hogy a függvényeket tagfüggvényekként adjuk meg:

```
struct Date {
    int d, m, y;

    void init(int dd, int mm, int yy);         // kezdeti értékadás
    void add_year(int n);                      // n év hozzáadása
    void add_month(int n);                     // n hónap hozzáadása
    void add_day(int n);                       // n nap hozzáadása
};
```

Az osztálydefiníción belül bevezetett függvényeket (a *struct* is osztály, §10.2.8) *tagfüggvényeknek* hívjuk. A tagfüggvényeket az adatszerkezetek tagjainak elérésére vonatkozó szokásos formában alkalmazhatjuk és csak megfelelő típusú objektumra:

```
Date my_birthday;

void f()
{
    Date today;

    today.init(16,10,1996);
    my_birthday.init(30,12,1950);

    Date tomorrow = today;
    tomorrow.add_day(1);
    // ...
}
```

Minthogy a különböző adatszerkezeteknek azonos nevű függvényeik is lehetnek, a tagfüggvények meghatározásakor meg kell adnunk az adatszerkezet nevét is:

```
void Date::init(int dd, int mm, int yy)
{
    d = dd;
    m = mm;
    y = yy;
}
```

A tagfüggvényekben a tagokat az objektum kifejezett megadása nélkül is használhatjuk. Ekkor a név azon objektum megfelelő tagjára vonatkozik, amelyre a tagfüggvényt meghívtuk. Amikor például a *Date::init()* tagfüggvényt alkalmazzuk a *today* változóra, az *m=mm* értékadás a *today.m* változóra vonatkozik. Ha ugyanezt a tagfüggvényt a *my_birthday* változóra alkalmaznánk, az *m=mm* értékadás a *my_birthday.m* változóra vonatkozna. A tagfüggvény mindig „tudja", hogy milyen objektumra hívták meg.

A

```
class X { ... };
```

kifejezést osztálydefiníciónak hívjuk, mert egy új típust határoz meg. Történeti okokból az osztálydefiníciót néha osztálydeklarációként említik. Azon deklarációkhoz hasonlatosan, amelyek nem definíciók, az osztálydefiníciók az *#include* utasítás felhasználásával több forrásállományban is szerepeltethetők, feltéve, hogy nem sértjük meg az egyszeri definiálás szabályát (§9.2.3).

10.2.2. Az elérhetőség szabályozása

A *Date* előző pontbeli deklarációja azon függvények halmazát adja meg, melyekkel a *Date* típusú objektumot kezelhetjük. Ebből azonban nem derül ki, hogy kizárólag ezek lehetnek mindazok a függvények, amelyek közvetlenül függnek a *Date* típus ábrázolásától és közvetlenül elérhetik az ilyen típusú objektumokat. A megszorítást úgy fejezhetjük ki, ha *struct* helyett *class*-t használunk:

```
class Date {
    int d, m, y;
public:
    void init(int dd, int mm, int yy);          // kezdeti értékadás

    void add_year(int n);                       // n év hozzáadása
    void add_month(int n);                      // n hónap hozzáadása
    void add_day(int n);                        // n nap hozzáadása
};
```

A *public* címke két részre osztja az osztály törzsét. Az első, privát (*private*) részbeli neveket csak a tagfüggvények használhatják. A második, nyilvános (*public*) rész az osztály nyilvános felülete. A *struct* szerkezetek egyszerűen olyan osztályok, melyekben a tagok alapértelmezett elérhetősége nyilvános (§10.2.8). A tagfüggvényeket a megszokott módon definiálhatjuk és használhatjuk:

```
inline void Date::add_year(int n)
{
    y += n;
}
```

Mindazonáltal a nem tag függvények a privát tagokat nem használhatják:

```
void timewarp(Date& d)
{
    d.y -= 200;     // hiba: Date::y privát
}
```

Számos előnnyel jár, ha a tagok elérhetőségét egy pontosan megadott lista függvényeire korlátozzuk. Például, ha egy hiba miatt a *Date* érvénytelen értéket kap (mondjuk *1985. december 36.*-át), akkor biztosak lehetünk abban, hogy ez a hiba csak valamelyik tagfüggvényben lehet. Ebből következik, hogy a hibakeresés első szakasza, a hiba helyének behatárolása már azelőtt megtörténik, hogy a program egyáltalán lefutna. Ez egyik esete annak az általános megfigyelésnek, hogy az osztály viselkedésének bármilyen módosítása csakis a tagfüggvények megváltoztatásával érhető el. Például ha megváltoztatjuk egy osztály adatábrázolását, akkor elég a tagfüggvényeket ennek megfelelően módosítanunk. Az osztályt használó kód közvetlenül csak az osztály nyilvános felületétől függ, ezért nem kell újraírni (bár lehet, hogy újra kell fordítani). A másik előny, hogy a leendő felhasználónak elég a tagfüggvények meghatározását tanulmányoznia ahhoz, hogy megtudja, hogyan lehet használni az osztályt.

A privát tagok védelme az osztálytagok név szerinti elérhetőségének korlátozásán múlik, ezért a címek megfelelő kezelésével vagy pontosan meghatározott típuskonverzióval megkerülhető. Ez persze csalás. A C++ a véletlen hibák ellen véd, nem a védelmi rendszer tudatos megkerülése, a csalás ellen. Egy általános célú nyelvben csak hardverszinten lehetne a rosszindulatú használat ellen védekezni, és igazi rendszerekben még ez is nehezen kivitelezhető.

Az *init()* függvényt részben azért vettük fel, mert általában célszerű, ha van egy, az objektumnak értéket adó függvényünk, részben pedig azért, mert az adattagok priváttá tétele miatt erre kényszerültünk.

10.2.3. Konstruktorok

Az *init()*-hez hasonló függvények használata az objektumok kezdeti értékadására nem elegáns és hibák forrása lehet. Minthogy sehol sincs lefektetve, hogy egy objektumnak kezdőértéket kell adni, a programozó elfelejtheti azt – vagy éppen többször is megteheti (mind-

két esetben egyformán végzetes következményekkel). Jobb megoldás, ha lehetővé tesszük a programozónak, hogy megadjon egy olyan függvényt, melynek célja kifejezetten az objektumok előkészítése. Mivel az ilyen függvény létrehozza az adott típusú értékeket, *konstruktornak* (vagyis létrehozónak, constructor) hívjuk. A konstruktort arról ismerjük meg, hogy ugyanaz a neve, mint magának az osztálynak:

```
class Date {
    // ...
    Date(int, int, int);                    // konstruktor
};
```

Ha egy osztály rendelkezik konstruktorral, akkor minden, ebbe az osztályba tartozó objektum kap kezdőértéket. Ha a konstruktornak paraméterekre van szüksége, azokat meg kell adni:

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);                      // rövidített forma
Date my_birthday;                           // hiba: nincs kezdőérték
Date release1_0(10,12);                     // hiba: a harmadik paraméter hiányzik
```

Gyakran célszerű, ha a kezdeti értékadás többféleképpen is lehetséges. Ezt úgy érhetjük el, ha többféle konstruktor áll rendelkezésre:

```
class Date {
    int d, m, y;
public:
    // ...
    Date(int, int, int);                    // nap, hónap, nap
    Date(int, int);                         // nap, hónap, aktuális év
    Date(int);                              // nap, aktuális hónap és év
    Date();                                 // alapértelmezett Date: mai dátum
    Date(const char*);                      // a dátum karakterlánccal ábrázolva
};
```

A konstruktorokra ugyanazok a túlterhelési szabályok vonatkoznak, mint más függvényekre (§7.4). Amíg a konstruktorok kellően különböznek a paraméterek típusaiban, a fordítóprogram ki fogja tudni választani, melyiket kell az egyes meghívásokkor alkalmazni:

```
Date today(4);
Date july4("July 4, 1983");
Date guy("5 Nov");
Date now;               // alapértelmezett kezdeti értékadás az aktuális dátummal
```

A *Date* példa esetében megfigyelhetjük a konstuktorok „elburjánzását”, ami általános jelenség. A programozó egy osztály tervezésekor mindig kísértést érez új és új függvényekkel bővíteni azt, mondván, valakinek úgyis szüksége lesz rájuk. Több gondot igényel ugyanis

mérlegelni, mire van igazán szükség és arra szorítkozni. Ez az odafigyelés ugyanakkor általában kisebb és érthetőbb programokhoz vezet. Az egymással rokon függvények számának csökkentésére az egyik mód az alapértelmezett paraméter-értékek használata (§7.5). A *Date* osztály paramétereinek például egy olyan alapértelmezett értéket adhatunk, melynek jelentése: „vegyük az alapértelmezett *today*-t".

```
class Date {
    int d, m, y;
public:
    Date(int dd =0, int mm =0, int yy =0);
    // ...
};

Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
    y = yy ? yy : today.y;

    // ellenőrizzük, hogy Date érvényes dátum-e
}
```

Ha egy paraméter-értéket az alapértelmezett érték jelzésére használunk, annak kívül kell esnie a lehetséges értékek halmazán. A *day* (nap) és *month* (hónap) paraméterek esetében ez a halmaz világosan meghatározható, a *year* (év) mezőnél azonban a zéró érték nem esik nyilvánvalóan a halmazon kívülre. Szerencsére az európai naptárban nincs 0-dik év; az időszámításunk utáni első év (*year==1*) közvetlenül az időszámításunk előtti első év (*year==-1*) után következik.

10.2.4. Statikus tagok

A *Date* típushoz tartozó kényelmes alapértelmezett értéket egy jelentős rejtett probléma árán hoztuk létre: *Date* osztályunk a *today* nevű globális változótól függ. Így az osztály csak akkor használható, ha a *today* változót definiáltuk és minden kódrészletben megfelelően használjuk. Ez a fajta megszorítás az osztályt az eredeti környezeten kívül használhatatlanná teszi. A felhasználóknak túl sok kellemetlen meglepetésben lesz részük, amikor ilyen környezetfüggő osztályokat próbálnak használni és a kód módosítása is problematikus lesz. Ezzel az egy „kis globális változóval" még talán megbirkózunk, de ez a stílus vezet az eredeti programozón kívül más számára használhatatlan kódhoz. Kerüljük el!

Szerencsére a kívánt célt elérhetjük a nyilvánosan elérhető globális változó jelentette tehertétel nélkül is. Az olyan változókat, melyek egy osztályhoz tartoznak, de annak objektumaihoz nem, *statikus tagnak* nevezzük. A statikus tagokból mindig pontosan egy példány

létezik, nem pedig objektumonként egy, mint a közönséges, nem statikus adattagokból. Ehhez hasonlóan az olyan függvényeket, melyek egy adott osztálytaghoz hozzáférnek, de nem szükséges objektumra meghívni azokat, ***statikus tagfüggvénynek*** hívjuk. Tervezzük át az osztályt úgy, hogy megőrizzük az alapértelmezett konstruktor-értékek szerepét, de közben elkerüljük a globális változó használatának hátrányát:

```
class Date {
    int d, m, y;
    static Date default_date;

public:
    Date(int dd =0, int mm =0, int yy =0);
    // ...
    static void set_default(int, int, int);
};
```

A *Date* konstruktort immár így határozhatjuk meg:

```
Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : default_date.d;
    m = mm ? mm : default_date.m;
    y = yy ? yy : default_date.y;

    // ellenőrizzük, hogy Date érvényes dátum-e
}
```

Amikor szükséges, módosíthatjuk az alapértelmezett értéket. Egy statikus tagra ugyanúgy hivatkozhatunk, mint bármilyen más tagra, sőt, akár egy objektum megnevezése nélkül is; ekkor az osztály nevével minősíthetjük:

```
void f()
{
    Date::set_default(4,5,1945);
}
```

A statikus tagokat – mind az adattagokat, mind a függvényeket – definiálni kell valahol:

```
Date Date::default_date(16,12,1770);

void Date::set_default(int d, int m, int y)
{
    Date::default_date = Date(d,m,y);
}
```

Az alapértelmezett érték itt Beethoven születési dátuma, amíg valaki át nem állítja valami másra. Vegyük észre, hogy a *Date()* jelölés a *Date::default_date* értéket szolgáltatja:

```
Date copy_of_default_date = Date();
```

Következésképpen nincs szükség külön függvényre az alapértelmezett dátum lekérdezéséhez.

10.2.5. Osztály típusú objektumok másolása

Alapértelmezés szerint az osztály típusú objektumok másolhatók és kezdőértékként egy azonos típusú osztály egy objektumának másolatát is kaphatják, még akkor is, ha konstruktorokat is megadtunk:

```
Date d = today;    // kezdeti értékadás másolással
```

Alapértelmezés szerint az osztály objektum másolata minden tag másolatából áll. Ha nem ez a megfelelő viselkedés egy *X* osztály számára, az *X::X(const X&)* másoló konstruktorral megváltoztathatjuk azt. (Erre a §10.4.4.1 pontban részletesebben is visszatérünk.) Ennek megfelelően az osztály objektumokat alapértelmezés szerint értékadással is másolhatjuk:

```
void f(Date& d)
{
    d = today;
}
```

Az alapértelmezett viselkedés itt is a tagonkénti másolás. Ha ez nem megfelelő egy osztály számára, a programozó megadhatja a megfelelő értékadó operátort (§10.4.4.1).

10.2.6. Konstans tagfüggvények

A *Date* osztályhoz eddig olyan tagfüggvényeket adtunk, melyek értéket adnak egy *Date* objektumnak vagy megváltoztatják azt, de az érték lekérdezésére sajnos nem adtunk lehetőséget. Ezen könnyen segíthetünk, ha készítünk néhány függvényt, amelyekkel kiolvashatjuk az évet, a hónapot és a napot:

```
class Date {
    int d, m, y;
public:
    int day() const { return d; }
    int month() const { return m; }
    int year() const;
    // ...
};
```

Vegyük észre a *const* minősítőt a függvénydeklarációkban az (üres) paraméterlista után. Ez azt jelenti, hogy ezek a függvények nem változtatják meg az objektum állapotát. Természetesen a fordítóprogram megakadályozza, hogy véletlenül megszegjük ezt az ígéretet:

```
inline int Date::year() const
{
    return y++;    // hiba: kísérlet tag értékének módosítására konstans függvényben
}
```

Ha egy konstans tagfüggvényt osztályán kívül határozzuk meg, a *const* utótagot ki kell írnunk:

```
inline int Date::year() const   // helyes
{
    return y;
}

inline int Date::year()         // hiba: a const minősítő hiányzik a tagfüggvény típusából
{
    return y;
}
```

Vagyis a *const* minősítés része a *Date::day()* és *Date::year()* függvények típusának.

Egy konstans tagfüggvényt alkalmazhatunk állandó (konstans) és változó (nem konstans) objektumokra is, a nem konstans tagfüggvényeket viszont csak nem konstans objektumokra:

```
void f(Date& d, const Date& cd)
{
    int i = d.year();       // rendben
    d.add_year(1);          // rendben

    int j = cd.year();      // rendben
    cd.add_year(1);         // hiba: cd konstans, értéke nem módosítható
}
```

10.2.7. Önhivatkozás

Az *add_year()*, *add_month()*, és *add_year()* állapotfrissítő függvényeket úgy határoztuk meg, hogy azok nem adnak vissza értéket. Az ilyen, egymással kapcsolatban levő frissítő függvények esetében sokszor hasznos, ha visszaadunk egy, a frissített objektumra mutató referenciát, mert a műveleteket ekkor láncba kapcsolhatjuk („láncolhatjuk”).

Tegyük fel, hogy a következőt szeretnénk írni:

```
void f(Date& d)
{
    // ...
    d.add_day(1).add_month(1).add_year(1);
    // ...
}
```

Ezzel egy napot, egy hónapot és egy évet adunk *d*-hez. Ehhez viszont minden függvényt úgy kell megadnunk, hogy azok egy *Date* típusú referenciát adjanak vissza:

```
class Date {
    // ...

    Date& add_year(int n);            // n év hozzáadása
    Date& add_month(int n);           // n hónap hozzáadása
    Date& add_day(int n);             // n nap hozzáadása
};
```

Minden (nem statikus) tagfüggvény tudja, melyik objektumra hívták meg, így pontosan hivatkozhat rá:

```
Date& Date::add_year(int n)
{
    if (d==29 && m==2 && !leapyear(y+n)) {    // figyeljünk február 29-re!
            d = 1;
            m = 3;
    }
    y += n;
    return *this;
}
```

A **this* kifejezés azt az objektumot jelenti, amelyre a tagfüggvényt meghívták. (Egyenértékű a Simula nyelv *THIS* és a Smalltalk *self* kifejezésével.)

Egy nem statikus tagfüggvényben a *this* kulcsszó egy mutatót jelent arra az objektumra, amelyre a függvényt meghívták. Az *X* osztály egy nem *const* tagfüggvényében a *this* típusa *X**. Mindazonáltal a *this* nem közönséges változó, így nem lehet a címét felhasználni vagy értéket adni neki. Az *X* osztály egy konstans tagfüggvényben a *this* típusa *const X** lesz, hogy ne lehessen megváltoztatni magát az objektumot (lásd még §5.4.1).

A *this* használata legtöbbször automatikus. Például minden nem statikus tagra való hivatkozás tulajdonképpen a *this*-t használja, hogy a megfelelő objektum tagját érje el. Az *add_year* függvényt például egyenértékű, ám fáradságos módon így is megadhattuk volna:

```
Date& Date::add_year(int n)
{
    if (this->d==29 && this->m==2 && !leapyear(this->y+n)) {
            this->d = 1;
            this->m = 3;
    }
    this->y += n;
    return *this;
}
```

A *this*-t meghatározott (explicit) módon gyakran láncolt listák kezelésére használjuk (például §24.3.7.4).

10.2.7.1. Fizikai és logikai konstansok

Esetenként előfordulhat, hogy egy tagfüggvény logikailag állandó, mégis meg kell változtatnia egy tag értékét. A felhasználó számára a függvény nem módosítja az objektum állapotát, de valamilyen, a felhasználó által közvetlenül nem látható részlet megváltozik. Az ilyen helyzetet gyakran hívják logikai konstans mivoltnak. A *Date* osztályt például egy függvény visszatérési értéke egy karakterlánccal ábrázolhatja, melyet a felhasználó a kimenetben felhasználhat. Egy ilyen ábrázolás felépítése időigényes feladat, ezért érdemes egy példányt tárolni belőle, amit az egymást követő lekérdezések mind felhasználhatnak, amíg a *Date* értéke meg nem változik. Ilyen belső gyorsítótár (gyorstár, cache) inkább bonyolultabb adatszerkezeteknél használatos, de nézzük meg, hogyan működhetne ez a *Date* osztály esetében:

```
class Date {
    bool cache_valid;
    string cache;
    void compute_cache_value();                 // gyorstár feltöltése
    // ...
public:
    // ...
    string string_rep() const;                  // ábrázolás karakterlánccal
};
```

A felhasználó szemszögéből a *string_rep* függvény nem változtatja meg az objektum állapotát, ezért világos, hogy konstans tagfüggvénynek kell lennie. Másrészt a gyorsítótárat fel kell tölteni a használat előtt. Ezt elérhetjük típuskényszerítés alkalmazásával is:

```
string Date::string_rep() const
{
    if (cache_valid == false) {
        Date* th = const_cast<Date*>(this);     // konstans elvetése
        th->compute_cache_value();
        th->cache_valid = true;
    }
    return cache;
}
```

Vagyis a *const_cast* operátort (§15.4.2.1) használtuk, hogy egy *Date** típusú mutatót kapjunk a *this*-re. Ez aligha elegáns megoldás, és nem biztos, hogy egy eredetileg is állandóként megadott objektum esetében is működik:

```
Date d1;
const Date d2;
string s1 = d1.string_rep();
string s2 = d2.string_rep();                    // nem meghatározott viselkedés
```

A *d1* változó esetében a *string_rep()* egyszerűen az eredeti típusra alakít vissza, így a dolog működik. Ám *d2*-t konstansként adtuk meg és az adott nyelvi változat esetleg valamilyen memória-védelmet alkalmaz az állandó értékek megőrzésére. Ezért a *d2.string_rep()* hívás nem biztos, hogy pontosan meghatározható, az adott nyelvi változattól független eredménnyel fog járni.

10.2.7.2. A mutable minősítő

Az előbb leírt típuskényszerítés (a *const* minősítő átalakítása) és a vele járó, megvalósítástól függő viselkedés elkerülhető, ha a gyorsítótárba kerülő adatokat „változékony”-ként (*mutable*) adjuk meg:

```
class Date {
    mutable bool cache_valid;
    mutable string cache;
    void compute_cache_value() const;           // (változékony) gyorstár feltöltése
    // ...
public:
    // ...
    string string_rep() const;                  // ábrázolás karakterlánccal
};
```

A *mutable* tárolási minősítés azt jelenti, hogy a tagot úgy kell tárolni, hogy akkor is módosítható legyen, ha konstans objektum. Vagyis a *mutable* azt jelenti, hogy „soha nem állandó". Ezt felhasználva egyszerűsíthetünk a *string_rep()* meghatározásán:

```
string Date::string_rep() const
{
    if (!cache_valid) {
        compute_cache_value();
        cache_valid = true;
    }
    return cache;
}
```

Ezáltal a *string_rep()*-et megfelelően használatba vehetjük:

```
Date d3;
const Date d4;
string s3 = d3.string_rep();
string s4 = d4.string_rep();                    // rendben!
```

A tagok változékonyként való megadása akkor alkalmas leginkább, ha az ábrázolásnak csak egy része változhat. Ha az objektum logikailag változatlan marad, de a tagok többsége módosulhat, jobb a változó adatrészt külön objektumba tenni és közvetett úton elérni. Ezzel a módszerrel a gyorsítótárba helyezett karakterláncot tartalmazó program így írható meg:

```
struct cache {
    bool valid;
    string rep;
};

class Date {
    cache* c;                           // kezdeti értékadás a konstruktorban (§10.4.6)
    void compute_cache_value() const;   // a gyorstár által mutatott elem feltöltése
    // ...
public:
    // ...
    string string_rep() const;          // ábrázolás karakterlánccal
};

string Date::string_rep() const
{
    if (!c->valid) {
        compute_cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

A gyorsítótárat támogató eljárások az ún. „lusta” vagy takaros kiértékelés (lazy evaluation) különféle formáira is átvihetők.

10.2.8. Struktúrák és osztályok

Definíció szerint a struktúra (struct), olyan osztály, melynek tagjai alapértelmezés szerint nyilvánosak. Vagyis a

```
struct s { ...
```

egyszerűen rövidítése az alábbinak:

```
class s { public: ...
```

A *private:* elérhetőségi minősítés annak jelzésére használható, hogy a következő tagok privát elérésűek, a *public:* pedig azt mondja, hogy a következő tagok nyilvánosak. Attól eltekintve, hogy a nevek különböznek, az alábbi deklarációk egyenértékűek:

```
class Date1 {
    int d, m, y;
public:
    Date1(int dd, int mm, int yy);

    void add_year(int n);                  // n év hozzáadása
};

struct Date2 {
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);

    void add_year(int n);                  // n év hozzáadása
};
```

A választott stílust csak a körülmények és az egyéni ízlés határozza meg. Én általában azokat az osztályokat adom meg *struct*-ként, amelyekben minden tag nyilvános. Ezekre az osztályokra úgy gondolok, mint amik nem igazi típusok, csak adatszerkezetek. A konstruktorok és lekérdező függvények nagyon hasznosak lehetnek a struktúrák számára is, de inkább csak jelölésbeli könnyebbséget jelentenek, mintsem a típus tulajdonságait garantálják (mint az invariánsok, lásd §24.3.7.1).

Az osztályokban nem szükséges először az adattagokat megadni, sőt, sokszor jobb azokat a deklaráció végére tenni, hogy kihangsúlyozzuk a nyilvános felhasználói felületet alkotó függvényeket:

```
class Date3 {
public:
    Date3(int dd, int mm, int yy);

    void add_year(int n);                    // n év hozzáadása
private:
    int d, m, y;
};
```

Valódi kódban, ahol általában mind a nyilvános felület, mind a tényleges megvalósítás terjedelmesebb, mint a tankönyvi példákban, rendszerint a *Date3* stílusát részesítem előnyben.

Az elérhetőségi minősítéseket az osztálydeklarációkon belül többször is használhatjuk:

```
class Date4 {
public:
    Date4(int dd, int mm, int yy);
private:
    int d, m, y;
public:
    void add_year(int n);                    // n év hozzáadása
};
```

Ha azonban a deklaráció több nyilvános részt is tartalmaz (mint a *Date4* osztálynál), akkor a kód zavarossá válhat. Több privát rész használata szintén ezt eredményezi. Mindazonáltal a számítógép által elkészített kódok számára kedvező, hogy az elérhetőségi minősítések ismétlődhetnek.

10.2.9. Osztályon belüli függvénydefiníciók

Az osztályon belül definiált (nem csak deklarált) függvények helyben kifejtett (inline) tagfüggvénynek számítanak, azaz a fordítóprogram a függvény meghívása helyett közvetlenül beilleszti a kódot. Vagyis az osztály meghatározásán belüli kifejtés kicsi, de gyakran használt függvények számára hasznos. Ahhoz az osztály-definicióhoz hasonlóan, amelyben szerepel, az osztályon belül kifejtett függvény is szerepelhet több fordítási egységben (az *#include* utasítással beépítve). Persze az osztályhoz hasonlóan jelentésének minden felhasználásakor azonosnak kell lennie (§9.2.3).

Az a stílus, mely szerint az adattagokat az osztály definiciójának végére helyezzük, kisebb gondhoz vezet az adatábrázolást felhasználó nyilvános „inline” függvények tekintetében. Vegyük ezt a példát:

```
class Date {                                // zavaró lehet
public:
    int day() const { return d; }           // return Date::d
    // ...
private:
    int d, m, y;
};
```

Ez szabályos C++-kód, mivel egy osztály egy tagfüggvénye az osztály minden tagjára hivatkozhat, mintha az osztály definiciója már a tagfüggvény-törzsek beolvasása előtt teljes lett volna. A kódot olvasó programozót azonban ez megzavarhatja, ezért én vagy előreveszem az adatokat, vagy az „inline” tagfüggvényeket az osztály után fejtem ki:

```
class Date {
public:
    int day() const;
    // ...
private:
    int d, m, y;
};

inline int Date::day() const { return d; }
```

10.3. Hatékony felhasználói típusok

Az előző *Date* osztály példáján bemutattuk az osztályok meghatározásához szükséges alapvető nyelvi elemeket. Most az egyszerű és hatékony tervezésre helyezzük a hangsúlyt és azt mutatjuk be, hogy az egyes nyelvi elemek hogyan támogatják ezt.

Számos program használ egyszerű, de sűrűn előforduló elvont fogalmakat, konkrét típusokkal ábrázolva: latin vagy kínai karaktereket, lebegőpontos számokat, komplex számokat, pontokat, mutatókat, koordinátákat, (mutató–eltolás (offset)) párokat, dátumokat, időpontokat, értékkészleteket, kapcsolatokat, csomópontokat, (érték–egység) párokat, lemezcímeket, forráskód-helyeket, *BCD* karaktereket, pénznemeket, vonalakat, téglalapokat, rögzített pontos számokat, törtrésszel bíró számokat, karakterláncokat, vektorokat és tömböket. Gyakran előfordul, hogy egy program közvetetten támaszkodik ezen típusok némelyikére és még többre közvetlenül, könyvtárak közvetítésével.

A C++ más programozási nyelvekkel egyetemben közvetlenül támogat néhányat a fenti típusok közül, ám számuk miatt nem lehetséges az összeset közvetlenül támogatni. Egy általános célú programozási nyelv tervezője nem is láthatja előre az egyes alkalmazások igényeit. Tehát szükség van olyan eljárásokra, melyekkel a felhasználó adott célú típusokat adhat meg. Az ilyen típusokat *konkrét típusoknak* vagy *konkrét osztályoknak* hívjuk, hogy megkülönböztessük őket az absztrakt (elvont) osztályoktól (§12.3), illetve az osztályhierarchiák osztályaitól (§12.2.4 és §12.4).

A C++ nyelv egyik kifejezett célja volt, hogy az ilyen felhasználói típusok megadását és hatékony használatát is támogassa, mert ezek az „elegáns” programozás alapkövei. Mint általában, itt is érvényes, hogy az egyszerű és földhözragadt sokkal jelentősebb, mint a bonyolult és körmönfont.

Ennek fényében készítsünk egy jobb dátumosztályt:

```
class Date {
public:                                                    // nyilvános felület
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Bad_date { };                                    // kivételosztály

    Date(int dd =0, Month mm =Month(0), int yy =0);       // 0 jelentése "vedd az
                                                           // alapértelmezettet"

// függvények a Date vizsgálatához
    int day() const;
    Month month() const;
    int year() const;

    string string_rep() const;                   // ábrázolás karakterlánccal
    void char_rep(char s[ ]) const;              // ábrázolás C stílusú karakterlánccal

    static void set_default(int, Month, int);

// függvények a Date módosításához
    Date& add_year(int n);                       // n év hozzáadása
    Date& add_month(int n);                      // n hónap hozzáadása
    Date& add_day(int n);                        // n nap hozzáadása
private:
    int d, m, y;                                 // ábrázolás
    static Date default_date;
};
```

A végezhető műveletek ilyen halmaza meglehetősen jellemző a felhasználói adattípusokra. A következők szerepelnek benne:

1. Egy konstruktor, amely kezdőértéket ad az objektumoknak és változóknak
2. Lekérdező függvények, melyekkel egy *Date*-et megvizsgálhatunk. Ezek *const* minősítése jelzi, hogy nem módosítják annak az objektumnak vagy változónak az állapotát, amelyre meghívták őket.
3. A *Date* objektumokat és változókat kezelő függvények, melyek az ábrázolás vagy a konkrét megvalósítás ismerete, illetve az egyes elemek szerepével való bajlódás nélkül is meghívhatók.
4. Automatikusan definiált műveletek, melyek segítségével a *Date*-ek szabadon másolhatók.
5. A *Bad_date* osztály, mellyel a hibák mint kivételek jelezhetők.

A *Month* (hónap) típust azért vezettem be, hogy kezeljem azt a problémát, amit az okoz, hogy emlékeznünk kell rá: vajon június 7-ét amerikai stílusban *Date(6,7)*-nek vagy európai stílusban *Date(7,6)*-nak kell-e írnunk. Az alapértelmezett paraméter-értékek kezelésére is gondoltam, ezzel külön eljárás foglalkozik.

Gondolkodtam azon, hogy a napok és évek ábrázolására a *Day*-t és a *Year*-t, mint önálló típusokat bevezessem, hogy a *Date(1995,jul,27)* és a *Date(27,jul,1995)* összekeveredésének veszélyét elkerüljem. Ezek a típusok azonban nem lennének annyira hasznosak, mint a *Month*. Majdnem minden ilyen hiba amúgy is kiderül futási időben – nemigen dolgozom olyan dátumokkal, mint a 27-ik év július 26-ika. Az 1800 előtti történelmi dátumok kezelése annyira bonyolult, hogy jobb történész szakértőkre bízni. Ezenkívül pedig egy „valahanyadikát” nem lehet rendesen ellenőrizni a hónap és az év ismerete nélkül. (Egy alkalmas *Year* típus meghatározására nézve lásd: §11.7.1.)

Az alapértelmezett dátumot mint érvényes *Date* objektumot definiálni kell valahol:

```
Date Date::default_date(22,jan,1901);
```

A §10.2.7.1-ben említett gyorsítótáras (cache) módszer egy ilyen egyszerű típusnál felesleges, így kihagytam. Ha mégis szükséges, kiegészíthetjük vele az osztályt, mint a felhasználói felületet nem érintő megvalósítási részlettel.

Íme egy kicsi elméleti példa arra, hogy lehet *Date*-eket használni:

```
void f(Date& d)
{
    Date lvb_day = Date(16,Date::dec,d.year());

    if (d.day()==29 && d.month()==Date::feb) {
        // ...
    }

    if (midnight()) d.add_day(1);

    cout << "A következő nap:" << d+1 << '\n';
}
```

Feltételezzük, hogy a << kimeneti és a + összeadó művelet a *Date*-ekre definiált; (ezt a §10.3.3-ban valóban meg is tesszük).

Figyeljük meg a *Date::feb* jelölést. Az *f()* nem tagfüggvénye *Date*-nek, így meg kell adni, hogy a *Date*-nek és nem valami másnak a *feb*-jéről van szó.

Miért éri meg egy külön típust megadni egy olyan egyszerű dolog számára, mint egy dátum? Végül is beérhetnénk egy egyszerű adatszerkezettel...

```
struct Date {
    int day, month, year;
};
```

...és hagynánk, hogy a programozók döntsék el, mit csinálnak vele. De ha ezt tennénk, akkor minden felhasználónak magának kellene a *Date*-ek összetevőit kezelnie: vagy közvetlenül, vagy külön függvényekben. Ez pedig azzal járna, hogy a dátum fogalma „szétszóródna”, így azt nehezebb lenne megérteni, dokumentálni és módosítani. Ha egy fogalmat egyszerű adatszerkezetként bocsátunk a felhasználók rendelkezésére, az szükségszerűen külön munkát igényel tőlük.

Ezenkívül bár a *Date* típus látszólag egyszerű, mégis gondot igényel úgy megírni, hogy helyesen működjék. Például egy *Date* objektum növeléséhez szökőévekkel kell törődni, azzal a ténnyel, hogy a hónapok különböző hosszúságúak és így tovább (lásd a §10.6[1]-es feladatot). Az év-hónap-nap adatábrázolás ráadásul sok program számára szegényes. Ha viszont úgy döntünk, hogy megváltoztatjuk, csak a kijelölt függvényeket kell módosítanunk. Ha a *Date*-et például az 1970. január elseje utáni vagy előtti napok számával akarnánk ábrázolni, csak a *Date* tagfüggvényeit kellene megváltoztatnunk (§10.6.[2]).

10.3.1. Tagfüggvények

Természetesen minden tagfüggvényt ki kell fejteni valahol. Íme a *Date* konstruktorának definíciója:

```
Date::Date(int dd, Month mm, int yy)
{
    if (yy == 0) yy = default_date.year();
    if (mm == 0) mm = default_date.month();
    if (dd == 0) dd = default_date.day();

    int max;

    switch (mm) {
    case feb:
          max = 28+leapyear(yy);
          break;
    case apr: case jun: case sep: case nov:
          max = 30;
          break;
    case jan: case mar: case may: case jul: case aug: case oct: case dec:
          max = 31;
          break;
    default:
          throw Bad_date();                    // valaki csalt
    }
    if (dd<1 || max<dd) throw Bad_date();

    y = yy;
    m = mm;
    d = dd;
}
```

A konstruktor ellenőrzi, hogy a kapott adatok érvényes dátumot adnak-e. Ha nem, mint például a *Date(30,Date::Feb,1994)* esetében, kivételt vált ki (§8.3, 14. fejezet), amely jelzi, hogy olyan jellegű hiba történt, amit nem lehet figyelmen kívül hagyni. Ha a kapott adatok elfogadhatóak, a kezdeti értékadás megtörténik. Ez meglehetősen jellemző eljárásmód. Másfelől ha a *Date* objektum már létrejött, akkor az további ellenőrzés nélkül felhasználható és másolható. Más szóval a konstruktor felállítja az osztályra jellemző invariánst (ebben az esetben azt, hogy egy érvényes dátumról van szó). A többi tagfüggvény számíthat erre az állapotra és kötelessége fenntartani azt. Ez a tervezési módszer óriási mértékben leegyszerűsítheti a kódot (lásd a 24.3.7.1-es pontot).

A *Month(0)* értéket (amely nem jelent igazi hónapot) a „vegyük az alapértelmezett hónapot" jelzésére használjuk. A *Month* felsorolásban megadhatnánk egy értéket kifejezetten ennek jelzésére, de jobb egy nyilvánvalóan érvénytelen értéket használni erre a célra, mint hogy olyan látszatot keltsünk, hogy 13 hónap van egy évben. Vegyük észre, hogy a *0* értéket azért használhatjuk, mert az a *Month* felsorolás biztosított garantált értéktartományba esik (§4.8).

Gondolkodtam azon, hogy az adatellenőrzést külön, egy *is_date()* függvénybe teszem, de ez olyan kódhoz vezetne, amely bonyolultabb és kevésbé hatékony, mint a kivételek elkapásán alapuló. Tegyük fel például, hogy a >> művelet értelmezett a *Date* osztályra:

```
void fill(vector<Date>& aa)
{
    while (cin) {
            Date d;
            try {
                    cin >> d;
            }
            catch (Date::Bad_date) {
                    // saját hibakezelő
                    continue;
            }
            aa.push_back(d);                    // lásd §3.7.3
    }
}
```

Mint az ilyen egyszerű konkrét osztályok esetében szokásos, a tagfüggvények meghatározása a triviális és a nem túl bonyolult között mozog. Például:

```
inline int Date::day() const
{
    return d;
}

Date& Date::add_month(int n)
{
    if (n==0) return *this;

    if (n>0) {
            int delta_y = n/12;
            int mm = m+n%12;
            if (12 < mm) {                      // megjegyzés: int(dec)==12
                    delta_y++;
                    mm -= 12;
            }
```

```
        // most azok az esetek jönnek, amikor Month(mm)-nek nincs d napja

        y += delta_y;
        m = Month(mm);
        return *this;
    }

    // negatív n kezelése

    return *this;
}
```

10.3.2. Segédfüggvények

Egy osztályhoz általában számos olyan függvény tartozhat, melyeket nem szükséges magában az osztályban tagként megadni, mert nincs szükségük a belső adatábrázolás közvetlen elérésére:

```
int diff(Date a, Date b);       // napok száma az [a,b] vagy [b,a] tartományban
bool leapyear(int y);
Date next_weekday(Date d);
Date next_saturday(Date d);
```

Ha ezeket a függvényeket magában az osztályban fejtenénk ki, az bonyolultabbá tenné az osztály felületét és a belső adatábrázolás esetleges módosításakor több függvényt kellene ellenőrizni.

Hogyan kapcsolódnak az ilyen segédfüggvények a *Date* osztályhoz? Hagyományosan a deklarációjukat az osztály deklarációjával azonos fájlba tennénk, így azon felhasználók számára, akiknek szükségük van a *Date* osztályra, rögtön ezek is rendelkezésre állnának a felületet leíró fejállomány beépítése után (§9.2.1):

```
#include "Date.h"
```

A *Date.h* fejállomány használata mellett vagy helyett a segédfüggvények és az osztály kapcsolatát úgy tehetjük nyilvánvalóvá, hogy az osztályt és segédfüggvényeit egy névtérbe foglaljuk (§8.2):

```
namespace Chrono {          // dátumkezelő szolgáltatások

    class Date { /* ... */};

    int diff(Date a, Date b);
    bool leapyear(int y);
```

```
    Date next_weekday(Date d);
    Date next_saturday(Date d);
    // ...
}
```

A *Chrono* névtér természetesen a többi kapcsolódó osztályt is tartalmazná, például a *Time* (Idő) és *Stopwatch* (Stopper) osztályokat és azok segédfüggvényeit is. Egy egyetlen osztályt tartalmazó névtér használata általában csak túlbonyolított, kényelmetlen kódhoz vezet.

10.3.3. Operátorok túlterhelése

Gyakran hasznos lehet olyan függvényeket felvenni, amelyek a hagyományos jelölésmód használatát biztosítják. Az *operator==* függvény például lehetővé teszi az == egyenlőségi operátor használatát a *Date* objektumokra:

```
inline bool operator==(Date a, Date b)          // egyenlőség
{
    return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();
}
```

Egyéb kézenfekvő jelöltek:

```
bool operator!=(Date, Date);                    // egyenlőtlenség
bool operator<(Date, Date);                     // kisebb
bool operator>(Date, Date);                     // nagyobb
// ...

Date& operator++(Date& d);                      // Date növelése egy nappal
Date& operator--(Date& d);                      // Date csökkentése egy nappal

Date& operator+=(Date& d, int n);               // n nap hozzáadása
Date& operator-=(Date& d, int n);               // n nap kivonása

Date operator+(Date d, int n);                  // n nap hozzáadása
Date operator-(Date d, int n);                  // n nap kivonása

ostream& operator<<(ostream&, Date d);          // d kiírása
istream& operator>>(istream&, Date& d);         // beolvasás d-be
```

A *Date* osztály számára ezen operátorok használhatósága pusztán kényelmi szempontnak tűnik. Ám sok típus – például a komplex számok (§11.3), a vektorok (§3.7.1) és a függvényszerű objektumok (§18.4) – esetében ezek használata annyira beidegződött a felhasználóknál, hogy szinte kötelező megadni őket. Az operátorok túlterhelésével a 11. fejezet foglalkozik.

10.3.4. A konkrét osztályok jelentősége

Azért hívjuk a *Date* és más egyszerű felhasználói típusokat ***konkrét típusoknak***, hogy megkülönböztessem azokat az absztrakt osztályoktól (§2.5.4) és az osztályhierarchiáktól (12.3), illetve hogy hangsúlyozzam az olyan beépített típusokkal való hasonlóságukat, mint az *int* vagy a *float*. Ezeket ***értéktípusoknak*** (value types) is nevezik, használatukat pedig ***értékközpontú programozásnak*** (value-oriented programming). Használati modelljük és mögötte levő „filozófia" nagyon különbözik attól, amit gyakran objektum-orientált programozásnak hívnak (§2.6.2).

A konkrét osztályok dolga az, hogy egyetlen, viszonylag egyszerű dolgot jól és hatékonyan csináljanak. Általában nem cél, hogy a felhasználónak eszközt adjunk a kezébe egy konkrét osztály viselkedésének megváltoztatására. Így a konkrét osztályokat nem szánjuk arra sem, hogy többalakú (polimorf) viselkedést tanúsítsanak (§2.5.5, §12.2.6).

Ha nem tetszik egy konkrét típus viselkedése, akkor írhatunk egy másikat, ami a kívánalmaknak megfelelően működik. Ez az adott típus „újrahasznosításával" is elérhetjük; a típust pontosan úgy használhatjuk fel az új típus megvalósításához, mint egy *int*-et:

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    Date_and_time(Date d, Time t);
    Date_and_time(int d, Date::Month m, int y, Time t);
    // ...
};
```

A 12. fejezetben tárgyalt öröklődési eljárást úgy használhatjuk fel egy új típus meghatározására, hogy csak az eltéréseket kell leírnunk. A *Vec* osztályt például a *vector* alapján készíthetjük el (§3.7.2).

Egy valamirevaló fordítóprogrammal egy, a *Date*-hez hasonló konkrét osztály használata nem jár a szükséges tárolóhely vagy a futási idő rejtett növekedésével. A konkrét osztályok mérete fordítási időben ismert, ezért az objektumok számára helyet foglalhatunk a futási veremben is, azaz a szabad tárat érintő műveletek nélkül. A memóriakiosztás is ismert, így a helyben fordítás egyszerű feladat. A memóriakiosztásnak más nyelvekkel, például a C-vel vagy a Fortrannal való összeegyeztetése is hasonlóan könnyen, külön erőfeszítés nélkül megoldható.

Az ilyen egyszerű típusok megfelelő halmaza teljes programok alapjául szolgálhat. Ha egy alkalmazásban nincsenek meg a megfelelő kicsi, de hatékony típusok, akkor a túl általános és „költséges" osztályok használata komoly futási időbeli és tárfelhasználás-beli pazarláshoz vezethet. A konkrét típusok hiánya másfelől zavaros programokat eredményez, illetve azt, hogy minden programozó megírja az „egyszerű és sűrűn használt" adatszerkezeteket közvetlenül kezelő kódot.

10.4. Objektumok

Objektumok többféleképpen jöhetnek létre: lehetnek automatikus vagy globális változók, osztályok tagjai stb. Az alábbiakban ezeket a lehetőségeket, a rájuk vonatkozó szabályokat, az objektumok kezdőállapotát beállító konstruktorokat és a használatból kikerülő objektumok „eltakarítására" szolgáló destruktorokat tárgyaljuk.

10.4.1. Destruktorok

Az objektumok kezdőállapotát a konstruktorok állítják be, vagyis a konstruktorok hozzák létre azt a környezetet, amelyben a tagfüggvények működnek. Esetenként az ilyen környezet létrehozása valamilyen erőforrás – fájl, zár, memóriaterület – lefoglalásával jár, amit a használat után fel kell szabadítani (§14.4.7). Következésképpen némelyik osztálynak szüksége van egy olyan függvényre, amely biztosan meghívódik, amikor egy objektum megsemmisül, hasonlóan ahhoz, ahogy a konstruktor meghívására is biztosan sor kerül, amikor egy objektum létrejön: ezek a *destruktor* (megsemmisítő, destructor) függvények. Feladatuk általában a rendbetétel és az erőforrások felszabadítása. A destruktorok automatikusan meghívódnak, amikor egy automatikus változót tartalmazó blokk lefut, egy dinamikusan létrehozott objektumot törölnek és így tovább. Nagyon különleges esetben van csak szükség arra, hogy a programozó kifejezetten meghívja a destruktort (§10.4.11).

A destruktor legjellemzőbb feladata, hogy felszabadítsa a konstruktorban lefoglalt memóriaterületet. Vegyünk például egy valamilyen *Name* típusú elemek táblázatát tartalmazó *Table* osztályt. A konstruktornak le kell foglalnia az elemek tárolásához szükséges memóriát. Ha a *Table* objektum bármilyen módon törlődik, a memóriát fel kell szabadítani, hogy máshol fel lehessen majd használni. Ezt úgy érhetjük el, hogy megírjuk a konstruktort kiegészítő függvényt:

```
class Name {
    const char* s;
    // ...
};
```

```
class Table {
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15) { p = new Name[sz = s]; }          // konstruktor

    ~Table() { delete[ ] p; }                               // destruktor

    Name* lookup(const char *);
    bool insert(Name*);
};
```

A destruktort jelentő *~Table()* jelölés a komplemensképzést jelölő ~ szimbólumot használva utal a destruktornak a *Table()* konstruktorhoz való viszonyára. Az összetartozó konstruktor–destruktor pár meghatározása a C++-ban szokásos eljárás változó méretű objektumok megvalósítására. A standard könyvtár tárolói, például a *map*, ennek a módszernek valamelyik változatát használják, hogy az elemeik számára tárolóhelyet biztosítsanak, ezért a programozó a következőkben leírtakra támaszkodik, amikor valamelyik standard könyvtárbeli tárolót használja.(Így viselkedik például a szabványos *string* osztály is.) A leírtak alkalmazhatóak a destruktor nélküli osztályokra is. Ezekre úgy tekinthetünk, mint amelyeknél egy olyan destruktorunk van, amely nem csinál semmit.

10.4.2. Alapértelmezett konstruktorok

Hasonlóképpen a legtöbb típust úgy tekinthetjük, mint amelynek van alapértelmezett konstruktora. Az alapértelmezett konstruktor az, amelyiket paraméter nélkül hívhatjuk meg. Minthogy a fenti példában a *15* mint alapértelmezett érték adott, a *Table::Table(size_t)* függvény alapértelmezett konstruktor. Ha a programozó megadott alapértelmezett konstruktort, akkor a fordítóprogram azt fogja használni, máskülönben szükség esetén megpróbál létrehozni egyet. A fordítóprogram által létrehozott alapértelmezett konstruktor automatikusan meghívja az osztály típusú tagok és a bázisosztályok (§12.2.2) alapértelmezett konstruktorát:

```
struct Tables {
    int i;
    int vi[10];
    Table t1;
    Table vt[10];
};

Tables tt;
```

Itt *tt* kezdőértékkel való feltöltése fordítás közben létrehozott alapértelmezett konstruktor segítségével történik, amely a *Table(15)*-öt hívja meg *tt.t1*-re és *tt.vt* minden egyes elemére. Másrészt *tt.i* és *tt.vi* elemei nem kapnak kezdőértéket, mert ezek az objektumok nem osztály típusúak. Az osztályok és a beépített típusok egymástól eltérő kezelésmódjának a C-vel való egyeztetés és a futási idő növelésétől való tartózkodás az oka.

Mivel a *const*-ok és a referenciák kötelezően kezdőértéket kell, hogy kapjanak (§5.5, §5.4), az ilyeneket tartalmazó tagoknak nem lehet alapértelmezett konstruktora, hacsak a programozó kifejezetten nem gondoskodik konstruktorról (§10.4.6.1):

```
struct X {
    const int a;
    const int& r;
};

X x;        // hiba: nincs alapértelmezett konstruktor X számára
```

Az alapértelmezett konstruktorok közvetlen módon is hívhatók (§10.4.10). A beépített típusoknak szintén van alapértelmezett konstruktoruk (§6.2.8).

10.4.3. Létrehozás és megsemmisítés

Tekintsük át a különböző módokat: hogyan hozhatunk létre objektumot és később az hogyan semmisül meg. Objektum a következő módokon hozható létre:

- §10.4.4 Névvel ellátott automatikus objektumként, amely akkor keletkezik, amikor a program végrehajtása során deklarációja kiértékelődik, és akkor semmisül meg, amikor a program kilép abból a blokkból, amelyen belül a deklaráció szerepelt.
- §10.4.5 Szabad tárbeli objektumként, amely a *new* operátor használatával jön létre és a *delete* operátor használatával semmisül meg.
- §10.4.6 Nem statikus tagobjektumként, amely egy másik osztály objektum tagjaként jön létre és azzal együtt keletkezik, illetve semmisül meg.
- §10.4.7 Tömbelemként, amely akkor keletkezik és semmisül meg, amikor a tömb, melynek eleme.
- §10.4.8 Lokális statikus objektumként, amely akkor jön létre, amikor a program végrehajtása során először találkozik a deklarációjával és egyszer semmisül meg: a program befejezésekor.
- §10.4.9 Globális, névtérbeli vagy statikus osztály-objektumként, amely egyszer, a program indulásakor jön létre és a program befejezésekor semmisül meg.

§10.4.10 Ideiglenes objektumként, amely egy kifejezés kiértékelésekor jön létre és a teljes kifejezés végén, melyben előfordult, semmisül meg.
§10.4.11 Felhasználó által írt függvénnyel végzett, paraméterekkel vezérelt lefoglalási művelet segítségével nyert, a memóriába helyezett objektumként.
§10.4.12 Unió tagjaként, amelynek nem lehet sem konstruktora, sem destruktora.

Ez a felsorolás nagyjából a fontosság sorrendjében készült. A következő alpontokban részletesen elmagyarázzuk az objektumok létrehozásának ezen változatait és használatukat.

10.4.4. Lokális változók

A lokális változók konstruktora minden alkalommal végrehajtódik, valahányszor a vezérlés fonala „keresztülhalad" a változó deklarációján, a destruktor végrehajtására pedig akkor kerül sor, amikor kilépünk a változó blokkjából. A lokális változók destruktorai konstruktoraik sorrendjéhez viszonyítva fordított sorrendben hajtódnak végre:

```
void f(int i)
{
    Table aa;
    Table bb;
    if (i>0) {
            Table cc;
            // ...
    }
    Table dd;
    // ...
}
```

Itt *aa*, *bb* és *dd* ebben a sorrendben keletkeznek az *f()* meghívásakor és a *dd*, *bb*, *aa* sorrendben semmisülnek meg, amikor a vezérlés kilép az *f()*-ből. Ha egy hívásnál *i>0*, a *cc* a *bb* után jön létre, és *dd* létrejötte előtt semmisül meg.

10.4.4.1. Objektumok másolása

Ha *t1* és *t2* a *Table* osztályba tartozó objektumok, *t2=t1* alapértelmezés szerint *t1*-nek tagonkénti átmásolását jelenti *t2*-be (§10.2.5). Ha nem bíráljuk felül ezt az alapértelmezett viselkedést, meglepő (és rendszerint nemkívánatos) hatás léphet fel, ha olyan osztály objektumaira alkalmazzuk, melynek mutató tagjai vannak. A tagonkénti másolás rendszerint nem megfelelő olyan objektumok számára, amelyek egy konstruktor–destruktor pár által kezelt erőforrásokat tartalmaznak:

```
void h()
{
    Table t1;
    Table t2 = t1;          // kezdeti értékadás másolással: problémás
    Table t3;

    t3 = t2;                // értékadás másolással: problémás
}
```

Itt a *Table* alapértelmezett konstruktora kétszer hívódik meg: egyszer *t1*-re és egyszer *t3*-ra. A *t2*-re nem hívódik meg, mert ez a változó a *t1*-ből való másolással kapott kezdőértéket. A *Table* destruktor viszont háromszor hívódik meg: *t1*-re, *t2*-re és *t3*-ra is. Alapértelmezés szerint az értékadás tagonkénti másolást jelent, így a *h()* függvény végén *t1*, *t2* és *t3* mindegyike arra a névtömbre hivatkozó mutatót fogja tartalmazni, amely *t1* létrejöttekor kapott helyet a szabad tárban. A mutató, mely a *t3* létrejöttekor kijelölt névtömbre mutat, nem marad meg, mert a *t3=t2* értékadás következtében felülíródik, így az általa elfoglalt tárterület a program számára örökre elvész, hacsak nincs automatikus szemétgyűjtés (§10.4.5). Másrészt a *t1* részére létrehozott tömb *t1*-ben, *t2*-ben és *t3*-ban egyaránt megjelenik, tehát háromszor is törlődik. Ez nem meghatározott és valószínűleg katasztrofális eredményhez vezet.

Az ilyen anomáliák elkerülhetők, ha megadjuk, mit jelent egy *Table* objektum másolása:

```
class Table {
    // ...
    Table(const Table&);                    // másoló konstruktor
    Table& operator=(const Table&);         // másoló értékadás
};
```

A programozó bármilyen alkalmas jelentést meghatározhat ezen másoló műveletek számára, de az ilyen típusú tárolók esetében a másoló művelet hagyományos feladata az, hogy lemásolja a tartalmazott elemeket (vagy legalábbis a felhasználó számára úgy tesz, mintha ez a másolás megtörtént volna, lásd §11.12):

```
Table::Table(const Table& t)                // másoló konstruktor
{
    p = new Name[sz=t.sz];
    for (int i = 0; i<sz; i++) p[i] = t.p[i];
}

Table& Table::operator=(const Table& t)     // értékadás
{
    if (this != &t) {                       // óvakodjunk az ön-értékadástól: t = t
            delete[ ] p;
```

```
            p = new Name[sz=t.sz];
            for (int i = 0; i<sz; i++) p[i] = t.p[i];
      }
      return *this;
}
```

Mint majdnem mindig, a másoló konstruktor és az értékadó művelet itt is jelentősen eltér. Ennek alapvető oka az, hogy a másoló konstruktor le nem foglalt memóriát készít fel a felhasználásra, míg az értékadó műveletnek egy már létrehozott objektumot kell helyesen kezelnie.

Az értékadást bizonyos esetekben optimalizálni lehet, de az értékadó operátor általános célja egyszerű: védekezni kell a saját magával való értékadás ellen, törölni kell a régi elemeket, előkészíteni és bemásolni az új elemeket. Általában minden nem statikus tagot másolni kell (§10.4.6.3.)

10.4.5. A szabad tár

A dinamikusan kezelt memóriaterületen, a szabad tárban létrehozott objektumok konstruktorát a *new* operátor hívja meg, és ezek az objektumok addig léteznek, amíg a rájuk hivatkozó mutatóra nem alkalmazzuk a *delete* operátort:

```
int main()
{
      Table* p = new Table;
      Table* q = new Table;

      delete p;
      delete p;                  // valószínűleg futási idejű hibát okoz
}
```

A *Table::Table()* konstruktort kétszer hívjuk meg, csakúgy, mint a *Table::~Table()* destruktort. Sajnos azonban ebben a példában a *new*-k és *delete*-ek nem felelnek meg egymásnak: a *p* által hivatkozott objektumot kétszer töröltük, míg a *q* által mutatottat egyszer sem. Nyelvi szempontból egy objektum nem törlése nem hiba, mindössze a memória pazarlása, mindazonáltal egy hosszan futó programnál az ilyen „memóriaszivárgás” vagy „memórialyuk” (memory leak) súlyos és nehezen felderíthető hiba. Szerencsére léteznek az ilyesfajta memóriaszivárgást kereső eszközök is. A *p* által mutatott objektum kétszeri törlése súlyos hiba; a program viselkedése nem meghatározott és nagy valószínűséggel katasztrofális lesz.

Bizonyos C++-változatok automatikusan újrahasznosítják az elérhetetlen objektumok által elfoglalt memóriát (ezek a szemétgyűjtést alkalmazó megvalósítások), de viselkedésük nem szabványosított. Ha van is szemétgyűjtés, a *delete* operátor kétszeri meghívása egyben a destruktor (ha van ilyen) kétszeri meghívását fogja eredményezi, így az objektum kétszer törlődik, ami ilyenkor is súlyos hiba. A legtöbb esetben az objektumok ezen viselkedése csak apróbb kényelmetlenséget jelent. Jelesül, ahol van szemétgyűjtés, ott is a csak memória-felszabadítást végző destruktorokat lehet megtakarítani. Ennek az egyszerűsítésnek a hordozhatóság elvesztése az ára, sőt bizonyos programoknál a futási idő növekedése és a viselkedés megjósolhatatlansága is (§C.9.1).

Miután egy objektumot a *delete* művelettel töröltünk, bármilyen hozzáférési kísérlet az objektumhoz hibának számít. Sajnos az egyes nyelvi változatok nem képesek megbízható módon jelezni az ilyen hibákat.

A programozó megszabhatja, hogyan történjék a *new* használata esetén a memória lefoglalása, illetve annak a *delete*-tel való felszabadítása (§6.2.6.2 és §15.6). Lehetséges a lefoglalás, a konstruktorok és a kivételek együttműködésének a megadása is (§14.4.5 és 19.4.5). A szabad tárban levő tömböket a §10.4.7. tárgyalja.

10.4.6. Osztály típusú tagok

Nézzünk egy osztályt, amely egy kisebb cégről tárolhat adatokat:

```
class Club {
    string name;
    Table members;
    Table officers;
    Date founded;
    // ...
    Club(const string& n, Date fd);
};
```

A *Club* osztály konstruktoránál paraméterként meg kell adni a nevet és az alapítás dátumát. Az osztálytagok konstruktorainak paramétereit a tartalmazó osztály konstruktor-definiciójának tag-kezdőérték listájában (member initializer) adjuk meg:

```
Club::Club(const string& n, Date fd)
    : name(n), members(), officers(), founded(fd)
{
    // ...
}
```

A tagok kezdőérték-listáját kettőspont előzi meg és az egyes tagoknak kezdőértéket adó kifejezéseket vesszők választják el.

A tagok konstruktorainak végrehajtása megelőzi a tartalmazó osztály saját konstruktora törzsének végrehajtását. A konstruktorok a tagoknak az osztály deklarációjában elfoglalt sorrendjében és nem a kezdőértéket adó kifejezéseknek a listában való felsorolási sorrendjében hajtódnak végre. Az esetleges zavarok elkerülése érdekében nyilván célszerű a tagokat a deklarációban elfoglalt sorrendjükben felvenni a kezdőérték-adó kifejezések listájára. A tagok destruktorai a konstruktorok sorrendjével ellenkező sorrendben hívódnak meg.

Ha egy tag konstruktorának nincs szüksége paraméterre, nem szükséges felvenni a listára, így a következő kódrészlet egyenértékű az előző példabelivel:

```
Club::Club(const string& n, Date fd)
    : name(n), founded(fd)
{
    // ...
}
```

A *Table::Table* konstruktor a *Club::officers* tagot mindkét esetben a *15*-tel, mint alapértelmezett paraméterrel hozza létre.

Ha egy osztálynak osztály típusú tagjai vannak, az osztály megsemmisítésekor először saját destruktor függvényének (ha van ilyen) törzse hívódik meg, majd a tagok destruktorai a deklarációval ellentétes sorrendben. A konstruktor alulról felfelé haladva (a tagokat először) építi fel a tagfüggvények végrehajtási környezetét, a destruktor pedig felülről lefelé (a tagokat utoljára) bontja le azt.

10.4.6.1. A tagok szükségszerű kezdeti értékadása

Azon tagok feltöltése kezdőértékkel szükségszerű, amelyeknél a kezdeti értékadás különbözik az egyszerű értékadástól – azaz az alapértelmezett konstruktor nélküli osztályba tartozó, a *const* és a referencia típusú tagoké:

```
class X {
    const int i;
    Club c;
    Club& pc;
    // ...
    X(int ii, const string& n, Date d, Club& c) : i(ii), c(n,d), pc(c) { }
};
```

Ezen tagok kezdeti értékadására nincs egyéb lehetőség, és hiba azt nem megtenni is. A legtöbb típus esetében azonban a programozó választhat a kezdeti és a „sima” értékadás közül. Ilyenkor én általában a tag-kezdőérték listás megoldást választom, hogy egyértelmű legyen a kezdeti értékadás ténye. Ez a módszer ráadásul hatékonyabb is:

```
class Person {
    string name;
    string address;
    // ...
    Person(const Person&);
    Person(const string& n, const string& a);
};

Person::Person(const string& n, const string& a)
    : name(n)
{
    address = a;
}
```

Itt a *name* az *n* egy másolatával kap kezdőértéket. Másfelől az *address* először egy üres karakterlánccal töltődik fel, majd értékül az a egy másolatát kapja.

10.4.6.2. Konstans tagok

Egy statikus, egész típusú konstans tagot lehetséges a deklarációban egy kezdőérték-adó konstans kifejezéssel is feltölteni:

```
class Curious {
public:
    static const int c1 = 7;          // rendben, de ne felejtsük el a meghatározást
    static int c2 = 11;               // hiba: nem állandó
    const int c3 = 13;                // hiba: nem statikus
    static const int c4 = f(17);      // hiba: a kezdőérték-adó nem állandó
    static const float c5 = 7.0;      // hiba: a kezdőérték-adó nem egész értékű
    // ...
};
```

Akkor és csak akkor, ha a kezdőértéket kapott tagot memóriában tárolt objektumként használjuk, szükséges, hogy az ilyen tag (de csak egy helyen) definiált legyen, de ott nem szabad megismételni a kezdőérték-adó kifejezést:

```
const int Curious::c1;     // szükséges, de a kezdőérték-adó nem szerepelhet itt még egyszer

const int* p = &Curious::c1;   // rendben: Curious::c1 meghatározott
```

Másik megoldásként, jelképes állandóként használhatunk felsoroló konstanst (§4.8, §14.4.6, §15.3) is az osztály deklarációján belül, ha szükséges:

```
class X {
    enum { c1 = 7, c2 = 11, c3 = 13, c4 = 17};
    // ...
};
```

Így a programozó nem fog kísértésbe esni, hogy az osztályban változóknak, lebegőpontos számoknak stb. adjon kezdőértéket.

10.4.6.3. Tagok másolása

Az alapértelmezett másoló konstruktor és az alapértelmezett másoló értékadás (§10.4.4.1) az osztály összes tagját másolja. Ha ez nem lehetséges, az ilyen osztályú objektum másolási kísérlete hiba:

```
class Unique_handle {
private:                        // a másoló műveleteket privátta tesszük, megelőzendő az
                                // alapértelmezett másolást (§11.2.2)
    Unique_handle(const Unique_handle&);
    Unique_handle& operator=(const Unique_handle&);
public:
    // ...
};

struct Y {
    // ...
    Unique_handle a;        // explicit kezdőértéket igényel
};

Y y1;
Y y2 = y1;                      // hiba: Y::a nem másolható
```

Ezenkívül az alapértelmezett értékadás nem jöhet létre a fordításkor, ha az osztály egy nem statikus tagja: referencia, konstans, vagy olyan felhasználói típus melynek nincsen másoló értékadása.

Jegyezzük meg, hogy a referencia típusú tagok ugyanarra az objektumra hivatkoznak az eredeti objektumban és a másolatban is. Ez gond lehet, ha a hivatkozott objektumot törölni kell. Ha másoló konstruktort írunk, ügyeljünk arra, hogy minden tagot másoljunk, amelyet szükséges. Alapértelmezés szerint az elemek alapértelmezett módon kapnak kezdőértéket, de sokszor nem erre van szükség egy másoló konstruktorban:

```
Person::Person(const Person& a) : name(a.name) { }       // vigyázat!
```

Itt elfelejtettem az *address* tagot másolni, így az alapértelmezés szerinti üres karakterláncot kapja kezdőértékként. Ha új taggal bővítünk egy osztályt, ne felejtsük el ellenőrizni, hogy vannak-e olyan felhasználó által megadott konstruktorok, amelyeket az új tagok kezdeti értékadására és másolására való tekintettel meg kell változtatni.

10.4.7. Tömbök

Ha egy osztály egy tagjának van alapértelmezett, azaz paraméter nélkül hívható konstruktora, akkor ilyen osztályú objektumok tömbjét is meghatározhatjuk:

```
Table tbl[10];
```

A fenti egy 10 *Table* elemből álló tömböt hoz létre és minden elemet a *Table::Table()* konstruktorral, a *15* értékű alapértelmezett paraméterrel tölt fel.

A kezdőérték-lista (§5.2.1, §18.6.7) alkalmazásán kívül nincs más mód egy tömb elemeinek konstruktorai számára (nem alapértelmezett) paramétereket megadni. Ha feltétlenül szükséges, hogy egy tömb tagjai különböző kezdőértéket kapjanak, írjunk olyan alapértelmezett konstruktort, amely előállítja a kívánt értékeket:

```
class Ibuffer {
    string buf;
public:
    Ibuffer() { cin>>buf; }
    // ...
};

void f()
{
    Ibuffer words[100];     // minden elem a cin-ről kap kezdőértéket
    // ...
}
```

Az ilyen trükköket azonban általában jobb elkerülni.

Amikor egy tömb megsemmisül, az összes elemére meghívódik a destruktor. Ha nem *new* művelettel létrehozott tömbről van szó, akkor ez automatikusan történik. A C nyelvhez hasonlóan a C++ sem különbözteti meg az egyedi elemre és a tömb kezdőelemére hivatkozó mutatót (§5.3), ezért a programozónak meg kell adnia, hogy egyedi elemet vagy tömböt kell-e törölni:

```
void f(int sz)
{
    Table* t1 = new Table;
    Table* t2 = new Table[sz];
    Table* t3 = new Table;
    Table* t4 = new Table[sz];

    delete t1;          // helyes
    delete[ ] t2;       // helyes
    delete[ ] t3;       // helytelen; probléma
    delete t4;          // helytelen; probléma
}
```

A tömbök és egyedi elemek dinamikus tárterületen való elhelyezése az adott nyelvi változattól függ. Ezért a különböző változatok különbözőképpen fognak viselkedni, ha hibásan használjuk a *delete* és *delete[ ]* operátorokat. Egyszerű és érdektelen esetekben, mint az előző példa, a fordító észreveheti a hibát, de általában futtatáskor fog valami csúnya dolog történni.

A kifejezetten tömbök törlésére szolgáló *delete[ ]* logikailag nem szükséges. Elképzelhető lenne, hogy a szabad tártól megköveteljük, hogy minden objektumról tartsa nyilván, hogy egyedi objektum avagy tömb. Ekkor a nyilvántartás terhét levennénk a programozó válláról, de ez a kötelezettség egyes C++-változatokban jelentős memória- és futási idő-többletet jelentene. Ha az olvasó túl nehézkesnek találja a C stílusú tömbök használatát, itt is használhat helyettük olyan osztályokat, mint a *vector* (§3.7.1, §16.3):

```
void g()
{
    vector<Table>* p1 = new vector<Table>(10);
    Table* p2 = new Table;

    delete p1;
    delete p2;
}
```

10.4.8. Lokális statikus adatok

A lokális statikus objektumok (§7.1.2) konstruktora akkor hajtódik végre, amikor a végrehajtási szál először halad keresztül az objektum meghatározásán:

```
void f(int i)
{
    static Table tbl;
    // ...
    if (i) {
        static Table tbl2;
        // ...
    }
}

int main()
{
    f(0);
    f(1);
    f(2);
    // ...
}
```

Itt *tbl* konstruktora *f()* első meghívásakor hívódik meg. Mivel *tbl*-t statikusként adtuk meg, így nem semmisül meg, amikor *f()*-ből visszatér a vezérlés és nem jön újra létre *f()* második meghívásakor. Mivel a *tbl2* változó deklarációját tartalmazó blokk nem hajtódik végre az *f(0)* meghíváskor, *tbl2* is csak *f(1)* végrehajtásakor jön létre, a blokk újbóli végrehajtásakor nem.

A lokális statikus objektumok destruktorai akkor hívódnak meg, amikor a program leáll (§9.4.1.1). Hogy pontosan mikor, az nincs meghatározva.

10.4.9. Nem lokális adatok

A függvényeken kívül meghatározott (azaz globális, névtérbeli és osztályhoz tartozó statikus) változók a *main()* függvény meghívása előtt jönnek létre (és kapnak kezdőértéket), és minden létrehozott objektum destruktora a *main()* függvényből való kilépés után végre fog hajtódni. A dinamikus könyvtárak használata (dinamikus csatolás) kissé bonyolultabbá teszi ezt, hiszen ilyenkor a kezdeti értékadásra akkor kerül sor, amikor a dinamikus kód a futó programhoz kapcsolódik.

A fordítási egységeken belül a nem lokális objektumok konstruktorainak végrehajtása a definiciójuk sorrendjében történik:

```
class X {
    // ...
    static Table memtbl;
};

Table tbl;

Table X::memtbl;

namespace Z {
    Table tbl2;
}
```

A konstruktorok végrehajtási sorrendje a következő: *tbl*, *X::memtbl*, *Z::tbl2*. Vegyük észre, hogy a definíció és nem a deklaráció sorrendje számít. A destruktorok a konstruktorokkal ellentétes sorrendben hajtódnak végre: *Z::tbl2*, *X::memtbl*, *tbl*.

Nincs nyelvi változattól független meghatározása annak, hogy az egyes fordítási egységek nem lokális objektumai milyen sorrendben jönnek létre:

```
// file1.c:
    Table tbl1;

// file2.c:
    Table tbl2;
```

Az, hogy *tbl1* vagy *tbl2* fog előbb létrejönni, a C++ adott változatától függ, de a sorrend azon belül is változhat. Dinamikus csatolás használata vagy akár a fordítási folyamat kis módosítása is megváltoztathatja a sorrendet. A destruktorok végrehajtási sorrendje is hasonlóan változatfüggő.

Könyvtárak tervezésekor szükséges vagy egyszerűen kényelmes lehet egy olyan, konstruktorral és destruktorral bíró típus elkészítése, amely kizárólag a kezdeti értékadás és rendrakás célját szolgálja. Ilyen típusú adatot csak arra célra fogunk használni, hogy egy statikus objektum számára memóriaterületet foglaljunk le azért, hogy lefusson a konstruktora és a destruktora:

```
class Zlib_init {
    Zlib_init();    // Zlib előkészítése használatra
    ~Zlib_init();   // Zlib utáni takarítás
};
```

```
class Zlib {
    static Zlib_init x;
    // ...
};
```

Sajnos egy több fordítási egységből álló program esetében nincs garancia arra, hogy egy ilyen objektum kezdeti értékadása az első használat előtt megtörténik és a destruktor az utolsó használat után fut le. Egyes C++-változatok biztosíthatják ezt, de a legtöbb nem. Programozói szinten azonban lehetséges azt a megoldást alkalmazni, amit a nyelvi változatok általában a lokális statikus objektumokra alkalmaznak: egy-egy, az első használatot figyelő kapcsolót:

```
class Zlib {
    static bool initialized;
    static void initialize() { /* kezdeti értékadás */ initialized = true; }
public:
    // nincs konstruktor

    void f()
    {
        if (initialized == false) initialize();
        // ...
    }
    // ...
};
```

Ha sok függvényben kell lekérdezni az első használatot figyelő kapcsolót, az fárasztó feladat lehet, de megoldható. Ez a módszer azon alapul, hogy a konstruktor nélküli statikus objektumok *0* kezdőértéket kapnak. A dolog akkor válik igazán problematikussá, ha az objektum első használata egy végrehajtási időre érzékeny függvényben történik, ahol az ellenőrzés és szükség esetén a kezdeti értékadás túl sok időt vehet igénybe. Ilyenkor további trükkökre van szükség (§21.5.2).

Egy lehetséges másik megközelítés, hogy az egyes objektumokat függvényekkel helyettesítjük (§9.4.1):

```
int& obj() { static int x = 0; return x; }          // kezdeti értékadás első használatkor
```

Az első használatot figyelő kapcsolók nem kezelnek minden elképzelhető helyzetet. Lehetséges például olyan objektumokat megadni, amelyek a kezdeti értékadás alatt egymásra hivatkoznak – az ilyesmit jobb elkerülni. Ha mégis ilyen objektumokra van szükség, akkor óvatosan, fokozatosan kell létrehozni azokat. Egy másik probléma, hogy az utolsó használatot nem tudjuk egy jelzővel jelezni. Ehelyett lásd §9.4.1.1 és §21.5.2.

10.4.10. Ideiglenes objektumok

Ideiglenes objektumok legtöbbször aritmetikai kifejezésekből jönnek létre. Például az *x*y+z* kifejezés kiértékelése során egy ponton az *x*y* részeredményt valahol tárolni kell. Hacsak nem a program gyorsításán dolgozik (§11.6), a programozó ritkán kell, hogy az ideiglenes objektumokkal törődjék, habár ez is előfordul (§11.6, §22.4.7).

Egy ideiglenes objektum, hacsak nincs referenciához kötve vagy nem egy nevesített objektumnak ad kezdőértéket, törlődik a tartalmazó teljes kifejezés kiértékelése végén. A *teljes kifejezés* olyan kifejezés, amely nem részkifejezése más kifejezésnek.

A szabványos *string* osztály *c_str()* nevű tagfüggvénye egy C stílusú, nullkarakterrel lezárt karaktertömböt ad vissza (§3.5.1, §20.4.1). A + operátor karakterláncok esetében összefűzést jelöl. Ezek nagyon hasznos dolgok a karakterláncok kezelésekor, de együttes használatuk furcsa problémákhoz vezethet:

```
void f(string& s1, string& s2, string& s3)
{
    const char* cs = (s1+s2).c_str();
    cout << cs;
    if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
            // cs használata
    }
}
```

Az olvasó valószínűleg azt mondja erre, hogy „nem kell ilyet csinálni”, és egyetértek vele, de ilyen kódot szoktak írni, így érdemes tudni, hogyan kell azt értelmezni.

Először egy ideiglenes, *string* osztályú objektum jön létre, amely az *s1+s2* művelet eredményét tárolja. Ettől az objektumtól aztán elkérjük a C stílusú karaktertömböt, majd a kifejezés végén az ideiglenes objektum törlődik. Vajon hol foglalt helyet a fordító a C stílusú karaktertömb számára? Valószínűleg az *s1+s2*-t tartalmazó ideiglenes objektumban, és annak megsemmisülése után nem biztos, hogy nem semmisül meg az a terület is, következésképpen *cs* felszabadított memóriaterületre mutat. A *cout << cs* kimeneti művelet működhet a várt módon, de ez puszta szerencse kérdése. A fordítóprogram esetleg felderítheti az ilyen problémát és figyelmeztethet rá.

Az *if* utasításos példa egy kicsit ravaszabb. Maga a feltétel a várakozásnak megfelelően fog működni, mert a teljes kifejezés, amelyben az *s2+s3*-at tartalmazó ideiglenes objektum létrejön, maga az *if* feltétele. Mindazonáltal az ideiglenes objektum a feltételesen végrehajtandó utasítás végrehajtásának megkezdése előtt megsemmisül, így a *cs* változó bármiféle ottani használata nem biztos, hogy működik.

Vegyük észre, hogy ebben az esetben, mint sok más esetben is, az ideiglenes objektumokkal kapcsolatos probléma abból adódik, hogy egy magasabb szintű adatot alacsony szinten használtunk. Egy tisztább programozási stílus nem csak jobban olvasható programrészletet eredményezett volna, de az ideiglenes objektumokkal kapcsolatos problémákat is teljesen elkerülte volna:

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;

    if (s.length()<8 && s[0]=='a') {
        // s használata
    }
}
```

Ideiglenes változót használhatunk konstans referencia vagy nevesített objektum kezdőértékeként is:

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
    const string& s = s1+s2;
    string ss = s1+s2;

    g(s,ss);        // s és ss itt használható
}
```

Ez a kódrészlet jól működik. Az ideiglenes változó megsemmisül, amikor az „ő" hivatkozását vagy nevesített objektumát tartalmazó kódblokk lefut. Emlékezzünk arra, hogy hiba egy lokális változóra mutató referenciát visszaadni egy függvényből (§7.3) és hogy ideiglenes objektumot nem adhatunk egy nem konstans referencia kezdőértékéül (§5.5). Ideiglenes változót létrehozhatunk kifejezett konstruktorhívással is:

```
void f(Shape& s, int x, int y)
{
    s.move(Point(x,y));     // Point létrehozása a Shape::move() számára
    // ...
}
```

Az ilyen módon létrehozott ideiglenes változók is ugyanolyan szabályok szerint semmisülnek meg, mint az automatikusan létrehozottak.

10.4.11. Az objektumok elhelyezése

A *new* operátor alapértelmezés szerint a szabad tárban hozza létre az objektumokat. Mit tegyünk, ha máshol szeretnénk, hogy egy objektum létrejöjjön? Vegyünk példaként egy egyszerű osztályt:

```
class X {
public:
    X(int);
    // ...
};
```

Az objektumokat tetszés szerinti helyre tehetjük, ha megadunk egy memória-lefoglaló függvényt, amelynek további paraméterei vannak, és a *new* operátor használatakor megadjuk ezeket a paramétereket:

```
void* operator new(size_t, void* p) { return p; }     // explicit elhelyező operátor

void* buf = reinterpret_cast<void*>(0xF00F);          // fontos cím
X* p2 = new(buf)X;                                    // X létrehozása a 'buf-ban', az operator
                                                      // new(sizeof(X),buf) meghívásával
```

Ezen használat miatt a *new (buf) X* utasításforma, amely az *operator new*-nak további paramétereket ad, *elhelyező utasításként* (placement syntax) ismert. Jegyezzük meg, hogy minden *new* operátor a méretet várja első paraméterként, és ezt, mint a létrehozandó objektum méretét, automatikusan megkapja (§15.6). Hogy melyik operátort fogja egy adott hívás elérni, azt a szokásos paraméter-egyeztetési szabályok fogják eldönteni (§7.4); minden *new()* operátornak egy *size_t* típusú első paramétere van.

Az „elhelyező” *operator new()* a legegyszerűbb ilyen lefoglaló függvény, és definiciója a *<new>* szabványos fejállományban szerepel.

A *reinterpret_cast* a „legdurvább” és a legnagyobb károkozásra képes a típuskonverziós operátorok közül (§6.2.7). Legtöbbször egyszerűen a paraméterének megfelelő bitsorozatú értéket, mint a kívánt típust adja vissza, így aztán a lényegéből fakadóan nyelvi változattól függő, veszélyes és esetenként feltétlenül szükséges egészek és mutatók közötti átalakításra használható.

Az elhelyező *new* operátor felhasználható arra is, hogy egy bizonyos helyről (*Arena* objektumtól) foglaljunk memóriát:

```
class Arena {
public:
    virtual void* alloc(size_t) =0;
    virtual void free(void*) =0;
    // ...
};

void* operator new(size_t sz, Arena* a)
{
    return a->alloc(sz);
}
```

A különböző *Arena* objektumokban szükség szerint tetszőleges típusú objektumokat hozhatunk létre:

```
extern Arena* Persistent;
extern Arena* Shared;

void g(int i)
{
    X* p = new(Persistent) X(i);          // X állandó tárterületen
    X* q = new(Shared) X(i);              // X megosztott memóriában
    // ...
}
```

Ha egy objektumot olyan helyre helyezünk, amelyet nem (közvetlenül) a szabványos szabadtár-kezelő kezel, némi óvatosságra van szükség annak megsemmisítésekor. Ennek alapvető módja az, hogy közvetlenül meghívjuk a destruktort:

```
void destroy(X* p, Arena* a)
{
    p->~X();            // destruktor meghívása
    a->free(p);         // memória felszabadítása
}
```

Jegyezzük meg, hogy a destruktorok közvetlen meghívását – csakúgy, mint az egyedi igényeket kielégítő globális memória-lefoglalók – használatát inkább kerüljük el, ha lehet. Esetenként mégis alapvető szükségünk van rájuk: például nehéz lenne egy hatékonyan működő általános tárolóosztályt készíteni a standard könyvtár *vector* (§3.7.1, §16.3.8) típusa nyomán, közvetlen destruktorhívás nélkül. Mindazonáltal egy kezdő C++-programozó inkább háromszor gondolja meg, mielőtt közvetlenül

meghívna egy destruktort és akkor is inkább kérje előtte tapasztalt kollégájának tanácsát. Az elhelyező operátor és a kivételkezelés kapcsolatáról lásd a §14.4.4-es pontot.

A tömböknél nincs megfelelője az elhelyező operátornak, de nincs is szükség rá, mert az elhelyező operátort tetszőleges típusokra alkalmazhatjuk. Tömbökre vonatkozóan azonban megadhatunk például egyedi *operator delete()*-et (§19.4.5).

10.4.12. Uniók

Egy nevesített *unió* (union) olyan adatszerkezet (*struct*), amelyben minden tag címe azonos (lásd §C.8.2). Egy uniónak lehetnek tagfüggvényei, de nem lehetnek statikus tagjai.

A fordítóprogram általában nem tudhatja, hogy az unió melyik tagja van használatban, vagyis nem ismert, hogy milyen típusú objektum van az unióban. Ezért egy uniónak nem lehet olyan tagja, amelynek konstruktorral vagy destruktorral rendelkezik, mert akkor nem lehetne a helyes memóriakezelést biztosítani, illetve azt, hogy az unió megsemmisülésével a megfelelő destruktor hívódik meg.

Az uniók felhasználása leginkább alacsony szinten vagy olyan osztályok belsejében történik, amelyek nyilvántartják, hogy mi van az unióban (§10.6[20]).

10.5. Tanácsok

[1] A fogalmakat osztályokra képezzük le. §10.1.
[2] Csak akkor használjunk nyilvános adatokat (*struct*-okat), amikor tényleg csak adatok vannak és nincs rájuk nézve invariánst igénylő feltétel. §10.2.8.
[3] A konkrét típusok a legegyszerűbb osztályok. Hacsak lehet, használjunk inkább konkrét típust, mint bonyolultabb osztályokat vagy egyszerű adatszerkezeteket. §10.3.
[4] Egy függvény csak akkor legyen tagfüggvény, ha közvetlenül kell hozzáférnie az osztály ábrázolásához. §10.3.2.
[5] Használjunk névteret arra, hogy nyilvánvalóvá tegyük egy osztálynak és segédfüggvényeinek összetartozását. §10.3.2.
[6] Egy tagfüggvény, ha nem változatja meg az objektumának az értékét, legyen *const* tagfüggvény. §10.2.6.
[7] Egy függvény, amelynek hozzá kell férnie az osztály ábrázolásához, de nem

szükséges, hogy egy objektumon keresztül hívjuk meg, legyen statikus tagfüggvény. §10.2.4.

[8] Az osztályra állapotbiztosítóit (invariáns) a konstruktorban állítsunk be. §10.3.1.

[9] Ha egy konstruktor lefoglal valamilyen erőforrást, akkor legyen destruktora az osztálynak, amelyik felszabadítja azt. §10.4.1.

[10] Ha egy osztálynak van mutató tagja, akkor legyenek másoló műveletei (másoló konstruktora és másoló értékadása). §10.4.4.1.

[11] Ha egy osztálynak van referencia tagja, valószínűleg szüksége lesz másoló műveletekre (másoló konstruktorra és másoló értékadásra) is. §10.4.6.3.

[12] Ha egy osztálynak szüksége van másoló műveletre vagy destruktorra, valószínűleg szüksége lesz konstruktorra, destruktorra, másoló konstruktorra és másoló értékadásra is. §10.4.4.1.

[13] A másoló értékadásnál ügyeljünk az önmagával való értékadásra. §10.4.4.1.

[14] Másoló konstruktor írásakor ügyeljünk arra, hogy minden szükséges elemet másoljunk (ügyeljünk az alapértelmezett kezdeti értékadásra). §10.4.4.1.

[15] Ha új taggal bővítünk egy osztályt, ellenőrizzük, nincsenek-e felhasználói konstruktorok, amelyekben kezdőértéket kell adni az új tagnak. 10.4.6.3.

[16] Használjunk felsoroló konstansokat, ha egész konstansokra van szükség egy osztály deklarációjában. §10.4.6.2.

[17] Globális vagy névtérhez tartozó objektumok használatakor kerüljük a végrehajtási sorrendtől való függést. §10.4.9.

[18] Használjunk első használatot jelző kapcsolókat, hogy a végrehajtási sorrendtől való függést a lehető legkisebbre csökkentsük. §10.4.9.

[19] Gondoljunk arra, hogy az ideiglenes objektumok annak a teljes kifejezésnek a végén megsemmisülnek, amelyben létrejöttek. §10.4.10.

10.6. Gyakorlatok

1. (*1) Találjuk meg a hibát a §10.2.2-beli *Date::add_year()* függvényben. Aztán találjunk még két további hibát a §10.2.7-beli változatban.
2. (*2.5) Fejezzük be és próbáljuk ki a *Date* osztályt. Írjuk újra úgy, hogy az adatábrázolásra az 1970.01.01. óta eltelt napokat használjuk.
3. (*2) Keressünk egy kereskedelmi használatban levő *Date* osztályt. Elemezzük az általa nyújtott szolgáltatásokat. Ha lehetséges, vitassuk meg az osztályt egy tényleges felhasználóval.
4. (*1) Hogyan érjük el a *Chrono* névtér *Date* osztályának *set_default* függvényét (§10.3.2)? Adjunk meg legalább három változatot.
5. (*2) Határozzuk meg a *Histogram* osztályt, amely a konstruktorában paraméter-

ként megadott időtartományokra vonatkozó gyakoriságokat tartja nyilván. Biztosítsunk műveletet a grafikon kiíratására és kezeljük az értelmezési tartományon kívül eső értékeket is.

6. (*2) Határozzunk meg osztályokat, amelyek bizonyos (például egyenletes vagy exponenciális) eloszlások szerinti véletlen számokat adnak. Mindegyik osztálynak legyen egy konstruktora, amely az eloszlást megadja, és egy *draw* függvénye, amely a következő értéket adja vissza.
7. (*2.5) Készítsük el a *Table* osztályt, amely (név–érték) párokat tárol. Ezután módosítsuk a számológép programot (§6.1), hogy az a *map* helyett a *Table* osztályt használja. Hasonlítsuk össze a két változatot.
8. (*2) Írjuk újra a §7.10[7]-beli *Tnode*-ot, mint olyan osztályt, amelynek konstruktorai, destruktorai stb. vannak. Adjuk meg a *Tnode*-ok egy fáját, mint osztályt (konstruktorokkal és destruktorokkal).
9. (*3) Határozzuk meg, készítsük el és ellenőrizzük az *Intset* osztályt, amely egészek halmazát ábrázolja. Legyen meg az unió, a metszet, és a szimmetrikus differencia művelet is.
10. (*1.5) Módosítsuk az *Intset* osztályt, hogy csomópontok (*Node* objektumok) halmazát jelentse, ahol a *Node* egy meghatározott adatszerkezet.
11. (*3) Hozzunk létre egy olyan osztályt, amely egész konstansokból és a +, -, * és / műveletekből álló egyszerű aritmetikai kifejezéseket képes elemezni, kiértékelni, tárolni és kiírni. A nyilvános felület ilyesmi legyen:

```
class Expr {
        // ...
public:
        Expr(const char*);
        int eval();
        void print();
};
```

Az *Expr::Expr()* konstruktor karakterlánc paramétere a kifejezés. Az *Expr::eval()* függvény visszaadja a kifejezés értékét, az *Expr::print()* pedig ábrázolja azt a *cout*-on. A program így nézhet ki:

```
Expr x("123/4+123*4-3");
cout << "x = " << x.eval() << "\n";
x.print();
```

Határozzuk meg az *Expr* osztályt kétféleképpen: egyszer mint csomópontok láncolt listáját, másszor egy karakterlánccal ábrázolva. Kísérletezzünk a kifejezés különböző kiíratásaival: teljesen zárójelezve, a műveleti jelet utótagként használva, assembly kóddal stb.

12. (*2) Határozzuk meg a *Char_queue* osztályt, hogy a nyilvános felület ne függjön az ábrázolástól. Készítsük el a *Char_queue*-t mint (a) láncolt listát, illetve (b) vektort.
13. (*3) Tervezzünk egy szimbólumtábla és egy szimbólumtábla-elem osztályt valamely nyelv számára. Nézzük meg az adott nyelv egy fordítóprogramjában, hogyan néznek ki ott az igazi szimbólumtáblák.
14. (*2) Módosítsuk a 10.6[11]-beli kifejezésosztályt, hogy változókat is kezelni tudjon, valamint a = értékadó műveletet is. Használjuk a 10.6[131]-beli szimbólumtábla osztályt.
15. (*1) Adott a következő program:

```
#include <iostream>

int main()
{
        std::cout << "Helló, világ!\n";
}
```

Módosítsuk úgy, hogy a következő kimenetet adja:

```
Kezdeti értékadás
Helló, világ!
Takarítás
```

A *main()* függvényt semmilyen módon nem változtathatjuk meg.

16. (*2) Határozzunk meg egy olyan *Calculator* osztályt, amilyet a §6.1-beli függvények nagyrészt megvalósítanak. Hozzunk létre *Calculator* objektumokat és alkalmazzuk azokat a *cin*-ből származó bemenetre, a parancssori paraméterekre és a programban tárolt karakterláncokra. Tegyük lehetővé a kimenetnek a bemenethez hasonló módon többféle helyre való irányítását.
17. (*2) Határozzunk meg két osztályt, mindegyikben egy-egy statikus taggal, úgy, hogy mindegyik létrehozásához a másikra hivatkozunk. Hol fordulhat elő ilyesmi igazi kódban? Hogyan lehet módosítani az osztályokat, hogy kiküszöböljük a végrehajtási sorrendtől való függést?
18. (*2.5) Hasonlítsuk össze a *Date* osztályt (§10.3) az §5.9[13] és a §7.10[19] feladatra adott megoldással. Értékeljük a megtalált hibákat és gondoljuk meg, milyen különbségekkel kell számolni a két osztály módosításakor.
19. (*3) Írjunk olyan függvényt, amely egy *istream*-ből és egy *vector<string>*-ből kiindulva elkészít egy *map<string,vector<int> >* objektumot, amely minden karakterláncot és azok előfordulásának sorszámát tartalmazza. Futtassuk a programot egy olyan szövegfájllal, amely legalább 1000 sort tartalmaz, és legalább 10

11

Operátorok túlterhelése

„Amikor én használok egy
szót, azt értem alatta,
amit én akarok – se többet,
se kevesebbet."
(Humpty Dumpty)

Jelölés • Operátor függvények • Egy- és kétoperandusú műveleti jelek • Az operátorok előre meghatározott jelentése • Az operátorok felhasználói jelentése • Operátorok és névterek • Komplex szám típusok • Tag és nem tag operátorok • Vegyes módú aritmetika • Kezdeti értékadás • Másolás • Konverziók • Literálok • Segédfüggvények • Konverziós operátorok • A többértelműség feloldása • Barát függvények és osztályok • Tagok és barát függvények • Nagy objektumok • Értékadás és kezdeti értékadás • Indexelés • Függvényhívás • Indirekció • Növelés és csökkentés • Egy karakterlánc osztály • Tanácsok • Gyakorlatok

11.1. Bevezetés

Minden műszaki szakterületnek – és a legtöbb nem műszakinak is – kialakultak a maga megszokott rövidítései, amelyek kényelmessé teszik a gyakran használt fogalmak kifejezését, tárgyalását. Az alábbi például

*x+y*z*

világosabb számunkra, mint a

vegyük y-t z-szer és az eredményt adjuk x-hez

Nem lehet eléggé megbecsülni a szokásos műveletek tömör jelölésének fontosságát.

A legtöbb nyelvvel együtt a C++ is támogat egy sor, a beépített típusokra vonatkozó műveletet. A legtöbb fogalomnak, amelyre műveleteket szoktak alkalmazni, azonban nincs megfelelője a beépített típusok között, így felhasználói típussal kell azokat ábrázolni. Például ha komplex számokkal akarunk számolni, ha mátrix-műveletekre, logikai jelölésekre vagy karakterláncokra van szükségünk a C++-ban, osztályokat használunk, hogy ezeket a fogalmakat ábrázoljuk. Ha ezekre az osztályokra vonatkozó műveleteket definiálunk, megszokottabb és kényelmesebb jelölés felhasználásával kezelhetjük az objektumokat, mintha csak az alapvető függvény-jelölést használnánk.

```
class complex {            // nagyon leegyszerűsített complex típus
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) { }
    complex operator+(complex);
    complex operator*(complex);
};
```

Itt például a komplex szám fogalmának egy egyszerű megvalósítását láthatjuk. Egy *complex* értéket egy kétszeres pontosságú lebegőpontos számpár ábrázol, melyet a + és a * műveletek kezelnek. A felhasználó adja meg a *complex::operator+()* és *complex::operator*()* operátorokat, hogy értelmezze a + és * műveleteket. Ha például *b* és *c* *complex* típusúak, akkor a *b+c* a *b.operator(c)*-t jelenti. Ezek után közelítőleg meghatározhatjuk a *complex* számokat tartalmazó kifejezések megszokott jelentését:

```
void f()
{
    complex a = complex(1, 3.1);
    complex b = complex(1.2, 2);
    complex c = b;

    a = b+c;
    b = b+c*a;
    c = a*b+complex(1,2);
}
```

A szokásos kiértékelési szabályok érvényesek, így a második kifejezés azt jelenti, hogy *b=b+(c*a)*, és nem azt, hogy *b=(b+c)*a*.

Az operátorok túlterhelésének legnyilvánvalóbb alkalmazásai közül sok konkrét típusokra vonatkozik (§10.3). Az operátorok túlterhelése azonban nemcsak konkrét típusoknál hasznos. Általános és absztrakt felületek felépítésénél például gyakran használunk olyan operátorokat, mint a ->, a *[ ]* és a *()*.

11.2. Operátor függvények

A következő operátorok (§6.2) jelentését meghatározó függvényeket megadhatjuk:

```
+      -      *      /      %      ^       &
|      ~      !      =      <      >       +=
-=     *=     /=     %=     ^=     &=      |=
<<     >>     >>=    <<=    ==     !=      <=
>=     &&     ||     ++     --     ->*     ,
->     [ ]    ()     new    new[ ] delete  delete[ ]
```

A következőknek viszont nem lehet felhasználói jelentést tulajdonítani:

:: (hatókör-feloldás, §4.9.4, §10.2.4)
. (tagkiválasztás, §5.7)
.* (tagkiválasztás a tagra hivatkozó mutatón keresztül, §15.5)

Ezek olyan operátorok (műveleti jelek), amelyek második operandusként nem értéket, hanem nevet várnak és a tagokra való hivatkozás alapvető módjai. Ha túl lehetne terhelni ezeket – azaz ha a felhasználó határozhatná meg jelentésüket – akkor ez érdekes mellékhatásokkal járhatna [Stroustrup, 1994]. A háromparaméterű feltételes-kifejezés operátor, a *?:* (§6.3.2) sem terhelhető túl, mint ahogy a *sizeof* (§4.6) és a *typeid* (§15.4.4) sem.

Új műveleti jeleket sem adhatunk meg; ehelyett a függvényhívási jelölés használható, ha a rendelkezésre állókon kívül további operátorokra is szükség van. Így például ne **-ot használjunk, hanem azt, hogy *pow()*. Ezek a megszorítások túl szigorúnak tűnhetnek, de rugalmasabb szabályok könnyen az egyértelműség elvesztéséhez vezetnének. Első pillantásra nyilvánvalónak és egyszerűnek tűnhet a ** operátort használni a hatványozásra, de gondoljunk csak meg: a ** műveleti jel balról kössön, mint a Fortranban, vagy jobbról, mint az Algolban? Az *a**p* kifejezést hogyan értelmezzük: mint *a*(*p)*-t vagy mint *(a)**(p)*-t?

Az operátor függvények neve az *operator* kulcsszóból és azt követően magából az operátorból áll; például *operator* <<. Az operátor függvényeket ugyanúgy deklarálhatjuk és hívhatjuk meg, mint a többi függvényt. Az operátorral való jelölés csak az operátor függvény közvetlen meghívásának rövidítése:

```
void f(complex a, complex b)
{
    complex c = a + b;                  // rövid forma
    complex d = a.operator+(b);         // explicit hívás
}
```

A *complex* előző definicióját adottnak véve a fenti két kezdeti értékadás jelentése azonos.

11.2.1. Egy- és kétoperandusú műveletek

Kétoperandusú műveleti jelet egyparaméterű nem statikus tagfüggvényként vagy kétparaméterű nem tag függvényként definiálhatunk. Ha @ kétoperandusú műveletet jelöl, akkor *aa@bb* vagy *aa.operator@(bb)*-t, vagy *operator@(aa,bb)*-t jelöli. Ha mindkettő értelmezett, a túlterhelés-feloldási szabályok (§7.4) döntik el, melyik alkalmazható, illetve hogy egyáltalán bármelyik alkalmazható-e:

```
class X {
public:
    void operator+(int);
    X(int);
};

void operator+(X,X);
void operator+(X,double);

void f(X a)
{
    a+1;          // a.operator+(1)
    1+a;          // ::operator+(X(1),a)
    a+1.0;        // ::operator+(a,1.0)
}
```

Az egyoperandusú (akár elő-, akár utótagként használt) műveleti jelek paraméter nélküli nem statikus tagfüggvényként vagy egyparaméterű nem tag függvényként definiálhatók. Ha @ előtag és egyoperandusú műveletet jelöl, akkor *@aa* vagy *aa.operator@()*-t, vagy *operator@(aa)*-t jelöli. Ha mindkettő értelmezett, a túlterhelés-feloldási szabályok (§7.4) döntik el, melyik alkalmazható, illetve hogy egyáltalán bármelyik alkalmazható-e. Ha @ utótag és egyoperandusú műveletet ad meg, akkor *aa@* vagy *aa.operator@(int)*-et, vagy *operator@(aa,int)*-et jelöli. (Ezt részletesebben a §11.11 pont írja le.) Ha mindkettő értelmezett, ismét csak a túlterhelés-feloldási szabályok (§7.4) döntik el, melyik alkalmazható, illetve hogy egyáltalán bármelyik alkalmazható-e. Operátort csak a nyelvi szabályoknak megfelelően definiálhatunk (§A.5), így nem lehet például egyoperandusú % vagy háromoperandusú + műveletünk:

```
class X {
    // tagok (a 'this' mutató automatikus):

    X* operator&();         // előtagként használt egyoperandusú & (cím)
    X operator&(X);         // kétoperandusú & (és)
    X operator++(int);      // utótagként használt növelő operátor (lásd §11.1)
    X operator&(X,X);       // hiba: háromoperandusú
    X operator/();          // hiba: egyoperandusú /
};

// nem tag függvények :

X operator-(X);             // előtagként használt egyoperandusú mínusz (mínusz előjel)
X operator-(X,X);           // kétoperandusú mínusz (kivonás)
X operator--(X&,int);       // utótagként használt csökkentő operátor
X operator-();              // hiba: nincs operandus
X operator-(X,X,X);         // hiba: háromoperandusú
X operator%(X);             // hiba: egyoperandusú %
```

A *[ ]* operátort a §11.8, a *()* operátort a §11.9, a -> operátort a §11.10, a ++ és -- operátorokat a 11.11, a memóriafoglaló és felszabadító operátorokat a §6.2.6.2, a §10.4.11 és a §15.6 pontokban írjuk le.

11.2.2. Az operátorok előre meghatározott jelentése

A felhasználói operátorok jelentésének csak néhány előírásnak kell megfelelniük. Az *operator=*, *operator[ ]*, *operator()* és az *operator->* nem statikus tagfüggvény kell, hogy legyen; ez biztosítja, hogy első operandusuk balérték (lvalue) lesz (§4.9.6).

Bizonyos beépített operátorok jelentése megegyezik más operátoroknak ugyanazon paraméterre összetetten gyakorolt hatásával. Például ha *a* egy *int*, akkor *++a* jelentése megegyezik *a+=1*-gyel, ami pedig azt jelenti, hogy *a=a+1*. Hacsak a felhasználó nem gondoskodik róla, ilyen összefüggések nem állnak fenn a felhasználói operátorokra, így a fordítóprogram például nem fogja kitalálni a *Z::operator+=()* művelet jelentését pusztán abból, hogy megadtuk a *Z::operator+()* és *Z::operator=()* műveleteket.

Hagyományosan az = (értékadó), a & (címképző) és a , (vessző; §6.2.2) operátorok előre definiáltak, ha osztályba tartozó objektumra alkalmazzuk azokat. Ezeket az előre meghatározott jelentéseket az általános felhasználó elől elrejthetjük, ha privátként adjuk meg azokat:

```
class X {
private:
    void operator=(const X&);
    void operator&();
    void operator,(const X&);
    // ...
};

void f(X a, X b)
{
    a = b;          // hiba: az értékadó operátor privát
    &a;             // hiba: a cím operátor (&) privát
    a,b;            // hiba: a vessző operátor (,) privát
}
```

Alkalmas módon definiálva azonban új jelentés is tulajdonítható nekik.

11.2.3. Operátorok és felhasználói típusok

Az operátoroknak tagfüggvénynek kell lenniük vagy paramétereik között legalább egy felhasználói típusnak kell szerepelnie (kivételek ez alól a *new* és *delete* operátorok jelentését felülbíráló függvények.) Ez a szabály biztosítja, hogy a programozó egy kifejezés értelmét csak akkor módosíthassa, ha legalább egy felhasználói típus előfordul benne. Ebből adódóan nem definiálható olyan operátor, amely kizárólag mutatókkal működik. A C++ tehát bővíthető, de nem változtatható meg, (az osztályba tartozó objektumokra vonatkozó =, & és , operátorokat kivéve).

Az olyan operátorok, melyeket arra szánunk, hogy első paraméterként valamilyen alaptípust fogadjanak el, nem lehetnek tagfüggvények. Vegyük például azt az esetet, amikor egy *complex* változót akarunk a *2* egészhez hozzáadni: az *aa+2* kifejezést alkalmas tagfüggvény

megléte esetén értelmezhetjük *aa.operator+(2)*-ként, de a *2+aa* kifejezést nem, mert nincs *int* osztály, amelynek + olyan tagfüggvénye lehetne, hogy a *2.operator(aa)* eredményre jussunk. De ha lenne is, akkor is két tagfüggvény kellene ahhoz, hogy *2+aa*-val és *aa+2*-vel is megbirkózzunk. Minthogy a fordítóprogram nem ismeri a felhasználói + művelet jelentését, nem tételezheti fel róla a felcserélhetőséget (kommutativitást), hogy annak alapján *2+aa*-t mint *aa+2*-t kezelje. Az ilyesmit rendszerint nem tag függvényekkel kezelhetjük (§11.3.2, §11.5).

A felsorolások felhasználói típusok, így rájuk is értelmezhetünk operátorokat:

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

Day& operator++(Day& d)
{
    return d = (sat==d) ? sun : Day(d+1);
}
```

A fordítóprogram minden kifejezést ellenőriz, hogy nem lép-e fel többértelműség. Ha egy felhasználói operátor is biztosít lehetséges értelmezést, a kifejezés ellenőrzése a §7.4 pontban leírtak szerint történik.

11.2.4. Névterek operátorai

Az operátor mindig valamilyen osztály tagja vagy valamilyen névtérben (esetleg a globálisban) definiált. Vegyük például a standard könyvtár karakterlánc-kiírási műveletének egyszerűsített változatát:

```
namespace std {             // egyszerűsített std

    class ostream {
            // ...
            ostream& operator<<(const char*);
    };

    extern ostream cout;

    class string {
            // ...
    };

    ostream& operator<<(ostream&, const string&);
}
```

```
int main()
{
    char* p = "Helló";
    std::string s = "világ";
    std::cout << p << ", " << s << "!\n";
}
```

Ez természetesen azt írja ki, hogy „Helló, világ!”. De miért? Vegyük észre, hogy nem tettem mindent elérhetővé az *std* névtérből azáltal, hogy azt írtam volna:

```
using namespace std;
```

Ehelyett az *std::* előtagot alkalmaztam a *string* és a *cout* előtt. Vagyis a legrendesebben viselkedve nem „szennyeztem be” a globális névteret és egyéb módon sem vezettem be szükségtelen függéseket.

A C stílusú karakterláncok (*char**) kimeneti művelete az *std::ostream* egy tagja, így

```
std::cout << p
```

jelentése definíció szerint:

```
std::cout.operator<<(p)
```

Mivel azonban az *std::ostream*-nek nincs olyan tagja, amelyet az *std::string*-re alkalmazhatnánk, így

```
std::cout << s
```

jelentése:

```
operator<<(std::cout,s)
```

A névtérben definiált operátorokat ugyanúgy operandusuk típusa szerint találhatjuk meg, mint ahogy a függvényeket paramétereik típusa szerint (§8.2.6). Minthogy a *cout* az *std* névtérben van, így az *std* is szóba kerül, amikor a << számára alkalmas definíciót keresünk. Így aztán a fordítóprogram megtalálja és felhasználja a következő függvényt:

```
std::operator<<(std::ostream&, const std::string&)
```

Jelöljön @ egy kétoperandusú műveletet. Ha *x* az *X* típusba, az *y* pedig az *Y* típusba tartozik, akkor *x@y* feloldása, azaz a paraméterek típusának megfelelő függvény megkeresése a következőképpen történik:

- Ha *X* egy osztály, keresünk egy *operator@*-t, amely az *X* osztálynak vagy valamelyik bázisosztályának tagfüggvénye.
- Keresünk egy *operator@* deklarációt az *x@y* kifejezést körülvevő környezetben.
- Ha *X* az *N* névtér tagja, az *operator@*-t az N névtérben keressük.
- Ha *Y* az *M* névtér tagja, az *operator@*-t az M névtérben keressük.

Ha az *operator@* többféle deklarációját is megtaláltuk, a feloldási szabályokat kell alkalmazni (§7.4), hogy a legjobb egyezést megtaláljuk, ha egyáltalán van ilyen. Ez a keresési eljárás csak akkor alkalmazandó, ha legalább egy felhasználói típus szerepel az operandusok között, ami azt is jelenti, hogy a felhasználói konverziókat (§11.3.2, §11.4) is figyelembe vesszük. (A *typedef*-fel megadott nevek csak szinonimák, nem felhasználói típusok (§4.9.7).) Az egyoperandusú műveletek feloldása hasonlóan történik.

Jegyezzük meg, hogy az operátorok feloldásában a tagfüggvények nem élveznek előnyt a nem tag függvényekkel szemben. Ez eltér a névvel megadott függvények keresésétől (§8.2.6). Az operátorok el nem rejtése biztosítja, hogy a beépített operátorok nem válnak elérhetetlenné, és hogy a felhasználó a meglevő osztálydeklarációk módosítása nélkül adhat meg új jelentéseket. A szabványos *iostream* könyvtár például definiálja a << tagfüggvényeket a beépített típusokra, a felhasználó viszont a felhasználói típusoknak a << művelettel való kimenetre küldését az *ostream* osztály (§21.2.1) módosítása nélkül definiálhatja.

11.3. Komplex szám típusok

A komplex számoknak a bevezetőben említett megvalósítása túl keveset nyújt ahhoz, hogy bárkinek is tessék. Egy matematika tankönyvet olvasva azt várnánk, hogy a következő függvény működik:

```
void f()
{
    complex a = complex(1,2);
    complex b = 3;
    complex c = a+2.3;
    complex d = 2+b;
    complex e = -b-c;
    b = c*2*c;
}
```

Ráadásul elvárnánk, hogy létezzék néhány további művelet is, például a == az összehasonlításra és a << a kimenetre, és még a matematikai függvények (mint a *sin()* és a *sqrt()*) megfelelő készletét is igényelnénk.

A *complex* osztály egy konkrét típus, így felépítése megfelel a §10.3-beli elveknek. Ráadásul a komplex aritmetika felhasználói olyan nagy mértékben építenek az operátorokra, hogy a *complex* osztály definiálása az operátor-túlterhelésre vonatkozó szinte valamennyi szabály alkalmazását igényli.

11.3.1. Tag és nem tag operátorok

Előnyös lenne, ha minél kevesebb függvény férne hozzá közvetlenül egy adott objektum belső adatábrázolásához. Ezt úgy érhetjük el, ha csak azokat az operátorokat adjuk meg magában az osztályban, amelyek értelmüknél fogva módosítják első paraméterüket, mint például a +=. Azokat az operátorokat, amelyek csak egy új értéket állítanak elő paramétereik alapján, mint például a +, az osztályon kívül definiálom és az alapvető operátorok segítségével valósítom meg:

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a);          // hozzá kell férni az ábrázoláshoz
    // ...
};

complex operator+(complex a, complex b)
{
    complex r = a;
    return r += b;       // az ábrázolás elérése a += operátoron keresztül
}
```

Ezen deklarációk alapján már leírhatjuk a következőt:

```
void f(complex x, complex y, complex z)
{
    complex r1 = x+y+z;                  // r1 = operator+(operator+(x,y),z)
    complex r2 = x;                      // r2 = x
    r2 += y;                             // r2.operator+=(y)
    r2 += z;                             // r2.operator+=(z)
}
```

Esetleges hatékonysági különbségektől eltekintve *r1* és *r2* kiszámítása egyenértékű.

Az összetett értékadó operátorokat, például a +=-t és a *=-t általában könnyebb definiálni, mint egyszerű megfelelőiket, a + és * operátorokat. Ez többnyire meglepést kelt, pedig pusztán abból következik, hogy az összeadásnál 3 objektum játszik szerepet (a két összeadandó és az eredmény), míg a += operátornál csak kettő. Az utóbbi esetében hatékonyabb a megvalósítás, ha nem használunk ideiglenes változókat:

```
inline complex& complex::operator+=(complex a)
{
    re += a.re;
    im += a.im;
    return *this;
}
```

A fenti megoldásnál nincs szükség ideiglenes változóra az eredmény tárolására és a fordítóprogram számára is könnyebb feladat a teljes helyben kifejtés.

Egy jó fordítóprogram az optimálishoz közeli kódot készít a sima + operátor használata esetén is. De nincs mindig jó optimalizálónk és nem minden típus olyan „egyszerű”, mint a *complex*, ezért a §11.5 pont tárgyalja, hogyan adhatunk meg olyan operátorokat, amelyek hozzáférhetnek az osztály ábrázolásához.

11.3.2. Vegyes módú aritmetika

Ahhoz, hogy a

```
complex d = 2+b;
```

kódot kezelni tudjuk, olyan + operátorra van szükségünk, amely különböző típusú paramétereket is elfogad. A Fortran kifejezésével élve tehát „vegyes módú aritmetikára” (mixed-mode arithmetic) van szükség. Ezt könnyen megvalósíthatjuk, ha megadjuk az operátor megfelelő változatait:

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a) {
            re += a.re;
            im += a.im;
            return *this;
    }
```

```
    complex& operator+=(double a) {
            re += a;
            return *this;
    }

    // ...
};

complex operator+(complex a, complex b)
{
    complex r = a;
    return r += b;          // complex::operator+=(complex)-et hívja meg
}

complex operator+(complex a, double b)
{
    complex r = a;
    return r += b;          // complex::operator+=(double)-t hívja meg
}

complex operator+(double a, complex b)
{
    complex r = b;
    return r += a;          // complex::operator+=(double)-t hívja meg
}
```

Egy *double* hozzáadása egy komplex számhoz egyszerűbb művelet, mint egy komplex szám hozzáadása; ezt tükrözik a fenti definíciók is. A *double* operandust kezelő műveletek nem érintik a komplex szám képzetes részét, így hatékonyabbak lesznek.

A fenti deklarációk mellett most már leírhatjuk a következőt:

```
void f(complex x, complex y)
{
    complex r1 = x+y;     // operator+(complex,complex)-et hívja meg
    complex r2 = x+2;     // operator+(complex,double)-t hívja meg
    complex r3 = 2+x;     // operator+(double,complex)-et hívja meg
}
```

11.3.3. Kezdeti értékadás

A *complex* változóknak skalárokkal való kezdeti és egyszerű értékadás kezeléséhez szükségünk van a skalárok (egész vagy lebegőpontos (valós) értékek) *complex*-szé átalakítására:

```
complex b = 3;     // b.re=3, b.im=0-t kell jelentenie
```

Az olyan konstruktor, amely egyetlen paramétert vár, konverziót jelent a paraméter típusáról a konstruktor típusára:

```
class complex {
    double re, im;
public:
    complex(double r) : re(r), im(0) { }
    // ...
};
```

Ez a konstruktor a valós számegyenesnek a komplex síkba való szokásos beágyazását jelenti.

Egy konstruktor mindig azt írja elő, hogyan hozhatunk létre egy adott típusú értéket. Ha egy adott típusú értéket kell létrehozni egy (kezdeti vagy egyszerű) értékadó kifejezés értékéből és ebből egy konstruktor létre tudja hozni a kívánt típusú értéket, akkor konstruktort alkalmazunk. Ezért az egyparaméterű konstruktorokat nem kell explicit meghívnunk:

```
complex b = 3;
```

A fenti egyenértékű a következővel:

```
complex b = complex(3);
```

Felhasználói konverzióra csak akkor kerül sor automatikusan, ha az egyértelmű (§7.4). Arra nézve, hogyan adhatunk meg csak explicite meghívható konstruktorokat, lásd a §11.7.1 pontot.

Természetesen szükségünk lesz egy olyan konstruktorra is, amelynek két *double* típusú paramétere van, és a *(0,0)* kezdőértéket adó alapértelmezett konstruktor is hasznos:

```
class complex {
    double re, im;
public:
    complex() : re(0), im(0) { }
    complex(double r) : re(r), im(0) { }
    complex(double r, double i) : re(r), im(i) { }
    // ...
};
```

Alapértelmezett paraméter-értékeket használva így rövidíthetünk:

```
class complex {
    double re, im;
public:
    complex(double r =0, double i =0) : re(r), im(i) { }
    // ...
};
```

Ha egy típusnak van konstruktora, a kezdeti értékadásra nem használhatunk kezdőérték-listát (§5.7, §4.9.5):

```
complex z1 = { 3 };       // hiba: complex rendelkezik konstruktorral
complex z2 = { 3, 4 };    // hiba: complex rendelkezik konstruktorral
```

11.3.4. Másolás

A megadott konstruktorokon kívül a *complex* osztálynak lesz egy alapértelmezett másoló konstruktora (§10.2.5) is. Az alapértelmezett másoló konstruktor egyszerűen lemásolja a tagokat. A működést pontosan így határozhatnánk meg:

```
class complex {
    double re, im;
public:
    complex(const complex& c) : re(c.re), im(c.im) { }
    // ...
};
```

Én előnyben részesítem az alapértelmezett másoló konstruktort azon osztályok esetében, amelyeknél ez megfelelő. Rövidebb lesz a kód, mintha bármi mást írnék, és a kód olvasójáról feltételezem, hogy ismeri az alapértelmezett működést. A fordítóprogram is ismeri és azt is, hogyan lehet azt optimalizálni. Ezenkívül pedig sok tag esetén fárasztó dolog kézzel kiírni a tagonkénti másolást és könnyű közben hibázni (§10.4.6.3).

A másoló konstruktor paramétereként referenciát kell használnom. A másoló konstruktor határozza meg a másolás jelentését – beleértve a paraméter másolásáét is – így a

```
complex::complex(complex c) : re(c.re), im(c.im) { }     // hiba
```

hibás, mert a függvény meghívása végtelen rekurzióhoz vezet.

Más, *complex* paraméterű függvények esetében én érték és nem referencia szerinti paraméter-átadást használok. Mindig az osztály készítője dönt. A felhasználó szemszögéből nézve nincs sok különbség egy *complex* és egy *const complex&* paramétert kapó függvény között. Erről bővebben ír a §11.6 pont.

Elvileg a másoló konstruktort az ilyen egyszerű kezdeti értékadásoknál használjuk:

```
complex x = 2;                // complex(2) létrehozása; ezzel adunk kezdőértéket x-nek
complex y = complex(2,0);     // complex(2,0) létrehozása; ezzel adunk kezdőértéket y-nak
```

A fordítóprogram azonban optimalizál és elhagyja a másoló konstruktor meghívását. Írhattuk volna így is:

```
complex x(2);        // x kezdőértéke 2
complex y(2,0);      // y kezdőértéke (2,0)
```

A *complex*-hez hasonló aritmetikai típusok esetében jobban kedvelem az = jel használatát. Ha a másoló konstruktort privátté tesszük (§11.2.2) vagy ha egy konstruktort *explicit*-ként adunk meg (§11.7.1), az = stílusú értékadás által elfogadható értékek körét a *()* stílusú értékadás által elfogadotthoz képest korlátozhatjuk.

A kezdeti értékadáshoz hasonlóan a két azonos osztályba tartozó objektum közötti értékadás alapértelmezés szerint tagonkénti értékadást jelent (§10.2.5). A *complex* osztálynál erre a célra megadhatnánk kifejezetten a *complex::operator=* műveletet, de ilyen egyszerű osztály esetében erre nincs ok, mert az alapértelmezett működés pont megfelelő.

A másoló konstruktor – akár a fordítóprogram hozta létre, akár a programozó írta – nemcsak a változók kezdőértékének beállítására használatos, hanem paraméter-átadáskor, érték visszaadásakor és a kivételkezeléskor is (lásd §11.7). Ezek szerepét a nyelv a kezdeti értékadáséval azonosként határozza meg (§7.1, §7.3, §14.2.1).

11.3.5. Konstruktorok és konverziók

A négy alapvető aritmetikai műveletnek eddig három-három változatát határoztuk meg:

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);
// ...
```

Ez fárasztóvá válhat, és ami fárasztó, ott könnyen előfordulhatnak hibák. Mi lenne, ha minden paraméter háromféle típusú lehetne? Minden egyparaméterű műveletből három változat kellene, a kétparaméterűekből kilenc, a háromparaméterűekből huszonhét és így tovább. Ezek a változatok gyakran nagyon hasonlóak. Valójában majdnem mindegyik úgy működik, hogy a paramétereket egy közös típusra alakítja, majd egy szabványos algoritmust hajt végre.

Ahelyett, hogy a paraméterek minden lehetséges párosítására megadnánk egy függvényt, típuskonverziókra hagyatkozhatunk. Tegyük fel, hogy *complex* osztályunknak van egy olyan konstruktora, amely egy *double* értéket alakít *complex*-szé, így a *complex* osztály számára elég egyetlen egyenlőség-vizsgáló műveletet megadnunk:

```
bool operator==(complex,complex);

void f(complex x, complex y)
{
    x==y;           // jelentése operator==(x,y)
    x==3;           // jelentése operator==(x,complex(3))
    3==y;           // jelentése operator==(complex(3),y)
}
```

Lehetnek azonban okok, melyek miatt jobb külön függvényeket megadni. Egyes esetekben például a konverzió túl bonyolult művelet lehet, máskor bizonyos paramétertípusokra egyszerűbb algoritmusok alkalmazhatók. Ahol ilyen okok nem lépnek fel jelentős mértékben, ott a függvény legáltalánosabb formáját megadva (esetleg néhány kritikus változattal kiegészítve) és a konverziókra hagyatkozva elkerülhetjük, hogy a vegyes módú aritmetikából adódóan nagyon sokféle függvényt kelljen megírnunk.

Ha egy függvény vagy operátor több változattal rendelkezik, a fordítóprogram feladata a legalkalmasabb változat kiválasztása, a paramétertípusok és a lehetséges (szabványos vagy felhasználói) konverziók alapján. Ha nincs legjobb változat, a kifejezés többértelmű és hibás (lásd §7.4).

Az olyan objektumok, melyeket a konstruktor közvetlen meghívása vagy automatikus használata hozott létre, ideiglenes változónak számítanak és amint lehetséges, megsemmisülnek (lásd §10.4.10). A . és -> operátorok bal oldalán nem történik automatikus felhasználói konverzió. Ez akkor is így van, ha maga a . implicit (a kifejezésbe beleértett):

```
void g(complex z)
{
    3+z;                // rendben: complex(3)+z
    3.operator+=(z);    // hiba: 3 nem egy osztály objektuma
    3+=z;               // hiba: 3 nem egy osztály objektuma
}
```

Ezt kihasználva egy műveletet tagfüggvénnyé téve kifejezhetjük, hogy a művelet bal oldali operandusként balértéket vár.

11.3.6. Literálok

Osztály típusú literálokat nem definiálhatunk abban az értelemben, ahogyan *1.2* és *1.2e3* *double* típusú literálok. Az alapvető típusokba tartozó literálokat viszont gyakran használhatjuk, ha a tagfüggvények fel vannak készítve a kezelésükre. Az egyparaméterű konstruktorok általános eljárást biztosítanak erre a célra. Egyszerű és helyben kifejtett (inline) konstruktorok esetében ésszerű a literál paraméterű konstruktorhívásokra mint literálokra gondolni. A *complex(3)* kifejezést például én úgy tekintem, mint egy *complex* értékű literált, noha a szó „technikai" értelmében véve nem az.

11.3.7. Kiegészítő tagfüggvények

Eddig csak konstruktorokat és aritmetikai műveleteket adtunk a *complex* osztályhoz. A tényleges használathoz ez kevés. A valós és a képzetes rész lekérdezése például sűrűn használatos:

```
class complex {
    double re, im;
public:
    double real() const { return re; }
    double imag() const { return im; }
    // ...
};
```

A *complex* osztály többi tagfüggvényével ellentétben a *real()* és az *imag()* nem változtatja meg egy *complex* objektum értékét, így *const*-ként adható meg.

A *real()* és *imag()* függvények alapján egy sor hasznos függvényt definiálhatunk anélkül, hogy azoknak hozzáférést kellene adnunk a *complex* osztály adatábrázolásához:

```
inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```

Vegyük észre, hogy a valós és a képzetes részt elég olvasnunk, írnunk sokkal ritkábban kell.

Ha „részleges frissítésre” van szükségünk, a következőt írhatjuk:

```
void f(complex& z, double d)
{
    // ...
    z = complex(z.real(),d);        // d hozzárendelése z.im-hez
}
```

Egy jól optimalizáló fordító ebből egyetlen értékadást készít.

11.3.8. Segédfüggvények

Ha mindent összerakunk, *complex* osztályunk így alakul:

```
class complex {
    double re, im;
public:
    complex(double r =0, double i =0) : re(r), im(i) { }

    double real() const { return re; }
    double imag() const { return im; }

    complex& operator+=(complex);
    complex& operator+=(double);
    // -=, *=, és /=
};
```

Kiegészítésként egy sor segédfüggvényt kell biztosítanunk:

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);

// -, *, és /

complex operator-(complex);                     // egyoperandusú mínusz
complex operator+(complex);                     // egyoperandusú plusz

bool operator==(complex,complex);
bool operator!=(complex,complex);

istream& operator>>(istream&,complex&);         // bemenet
ostream& operator<<(ostream&,complex);          // kimenet
```

Vegyük észre, hogy a *real()* és *imag()* függvények szerepe alapvető az összehasonlító függvények definiálásában. A következő segédfüggvények is nagyrészt ezekre építenek.

Megadhatnánk olyan függvényeket is, amelyek a polár-koordinátás jelölést támogatják:

```
complex polar(double rho, double theta);
complex conj(complex);

double abs(complex);
double arg(complex);
double norm(complex);

double real(complex);    // a kényelmesebb jelölésért
double imag(complex);    // a kényelmesebb jelölésért
```

Végül szükségünk lesz a további alapvető matematikai függvényekre:

```
complex acos(complex);
complex asin(complex);
complex atan(complex);
// ...
```

Felhasználói szemszögből nézve az itt bemutatott *complex* osztály szinte azonos a *complex<double>*-lal (lásd a standard könyvtárbeli *<complex>*-et, §22.5).

11.4. Konverziós operátorok

Konstruktorok használata típuskonverzió céljára kényelmes lehet, de nemkívánatos következményei vannak. Egy konstruktor nem tud

1. automatikus átalakítást megadni felhasználói adattípusról beépített adattípusra (mert a beépített adattípusok nem osztályok)
2. átalakítást megadni egy újabban megadott osztályról egy régebbire, a régebbi osztály deklarációjának megváltoztatása nélkül.

Ezeket a feladatokat az átalakítandó osztály konverziós (átalakító) operátorának definiálásával oldhatjuk meg. Ha *T* egy típus neve, akkor az *X::operator T()* függvény hatá-

rozza meg az *X* típus *T*-re való konverzióját. Definiálhatunk például a 6 bites, nem negatív egészeket ábrázoló *Tiny* osztályt, melynek objektumait aritmetikai kifejezésekben szabadon keverhetjük egészekkel:

```
class Tiny {
    char v;
    void assign(int i) { if (i&~077) throw Bad_range(); v=i; }
public:
    class Bad_range { };

    Tiny(int i) { assign(i); }
    Tiny& operator=(int i) { assign(i); return *this; }

    operator int() const { return v; }              // konverzió int típusra
};
```

Amikor egy *Tiny* egy egésztől kap értéket vagy kezdőértéket, ellenőrizzük, hogy az érték a megengedett tartományba esik-e. Minthogy egy *Tiny* másolásakor nincs szükség az értékellenőrzésre, az alapértelmezett másoló konstruktor és értékadás éppen megfelelő.

Ahhoz, hogy a *Tiny* változókra is lehetővé tegyük az egészeknél szokásos műveleteket, határozzuk meg a *Tiny*-ről *int*-re való automatikus konverziót, a *Tiny::operator int()*-et. Jegyezzük meg, hogy a konverzió céltípusa az operátor nevének része és nem szabad kiírni, mint a konverziós függvény visszatérési értékét:

```
Tiny::operator int() const { return v; }          // helyes
int Tiny::operator int() const { return v; }      // hiba
```

Ilyen tekintetben a konverziós operátor a konstruktorra hasonlít.

Ha egy *int* helyén egy *Tiny* szerepel, akkor arra a helyre a megfelelő *int* érték fog kerülni:

```
int main()
{
    Tiny c1 = 2;
    Tiny c2 = 62;
    Tiny c3 = c2-c1;      // c3 = 60
    Tiny c4 = c3;         // nincs tartományellenőrzés (nem szükséges)
    int i = c1+c2;        // i = 64

    c1 = c1+c2;           // tartományhiba: c1 nem lehet 64
    i = c3-64;            // i = -4
    c2 = c3-64;           // tartományhiba: c2 nem lehet -4
    c3 = c4;              // nincs tartományellenőrzés (nem szükséges)
}
```

A konverziós függvények különösen hasznosak olyan adatszerkezetek kezelésekor, amelyeknél az adatoknak (a konverziós operátor által definiált) kiolvasása egyszerű feladat, ellentétben az értékadással és a kezdőérték-adással.

Az *istream* és *ostream* típusok egy konverzió segítségével támogatják az alábbihoz hasonló vezérlési szerkezeteket:

```
while (cin>>x) cout<<x;
```

A *cin>>x* bemeneti művelet egy *istream&* referenciát ad vissza, amely automatikusan a *cin* objektum állapotát tükröző értékre alakítódik. Ezt azután a *while* utasítás ellenőrzi (§21.3.3.). Általában azonban nem jó ötlet adatvesztéssel járó automatikus konverziót meghatározni két típus között.

Célszerű takarékoskodni a konverziós operátorok bevezetésével. Ha túl sok van belőlük, az a kifejezések többértelműségéhez vezethet. A többértelműséget mint hibát jelzi ugyan a fordítóprogram, de kiküszöbölni fáradságos lehet. Talán a legjobb eljárás az, ha kezdetben nevesített függvényekkel végeztetjük az átalakítást (például *X::make_int()*). Ha később valamelyik ilyen függvény annyira népszerű lesz, hogy alkalmazása nem „elegáns" többé, akkor kicserélhetjük az *X::operator int()* konverziós operátorra.

Ha vannak felhasználói konverziók és felhasználói operátorok is, lehetséges, hogy többértelműség lép fel a felhasználói és a beépített operátorok között:

```
int operator+(Tiny,Tiny);

void f(Tiny t, int i)
{
    t+i;            // hiba, többértelmű: operator+(t,Tiny(i)) vagy int(t)+i ?
}
```

Ezért vagy felhasználói konverziókra építsünk, vagy felhasználói operátorokra, de ne mindkettőre.

11.4.1. Többértelműség

Egy *X* osztályú objektum értékadása egy *V* típusú értékkel akkor megengedett, ha van olyan *X::operator=(Z)* értékadó operátor, amely szerint *V* típus egyben *Z* is, vagy ha van egy egyedi konverzió *V*-ről *Z*-re. A kezdeti értékadásnál hasonló a helyzet.

Bizonyos esetekben a kívánt típusú értéket konstruktorok és konverziós operátorok ismételt alkalmazásával állíthatjuk elő. Ezt a helyzetet közvetlen konverzióval kell megoldani; az automatikus felhasználói konverzióknak csak egy szintje megengedett. Néha a kívánt típusú érték többféleképpen is létrehozható, ez pedig hiba:

```
class X { /* ... */ X(int); X(char*); };
class Y { /* ... */ Y(int); };
class Z { /* ... */ Z(X); };

X f(X);
Y f(Y);

Z g(Z);

void k1()
{
    f(1);                 // hiba: többértelmű f(X(1)) vagy f(Y(1))?
    f(X(1));              // rendben
    f(Y(1));              // rendben

    g("Mack");            // hiba: két felhasználói konverzió szükséges; g(Z(X("Mack")))-et
                          // nem próbáltuk
    g(X("Doc"));          // rendben: g(Z(X("Doc")))
    g(Z("Suzy"));         // rendben: g(Z(X("Suzy")))
}
```

A felhasználói konverziókat a fordítóprogram csak akkor veszi figyelembe, ha szükségesek egy hívás feloldásához:

```
class XX { /* ... */ XX(int); };

void h(double);
void h(XX);

void k2()
{
    h(1);        // h(double(1)) vagy h(XX(1))? h(double(1))!
}
```

A *h(1)* hívás a *h(double(1))* hívást jelenti, mert ehhez csak egy szabványos (és nem felhasználói) konverzióra van szükség (§7.4). A konverziós szabályok se nem a legegyszerűbben megvalósítható, se nem a legegyszerűbben leírható, de nem is az elképzelhető legáltalánosabb szabályok. Viszont viszonylag biztonságosak és alkalmazásukkal kevésbé fordulnak elő meglepő eredmények. A programozónak sokkal könnyebb egy többértelműséget feloldani, mint megtalálni egy hibát, amit egy nem sejtett konverzió alkalmazása okoz.

Az elemzés során alkalmazott szigorúan „alulról felfelé” való haladás elvéből az is következik, hogy a visszatérési értéket nem vesszük figyelembe a túlterhelések feloldásakor:

```
class Quad {
public:
    Quad(double);
    // ...
};

Quad operator+(Quad,Quad);

void f(double a1, double a2)
{
    Quad r1 = a1+a2;                    // kétszeres pontosságú összeadás
    Quad r2 = Quad(a1)+a2;              // Quad aritmetika kikényszerítése
}
```

Ezen tervezési mód választásának egyik oka, hogy a szigorú „alulról felfelé” való haladás elve érthetőbb, a másik pedig az, hogy nem a fordítóprogram dolga eldönteni, milyen fokú pontosságot akar a programozó egy összeadásnál.

Ha egy kezdeti vagy egyszerű értékadás mindkét oldalának eldőlt a típusa, akkor az értékadás feloldása ezen típusok figyelembe vételével történik:

```
class Real {
public:
    operator double();
    operator int();
    // ...
};

void g(Real a)
{
    double d = a;       // d = a.double();
    int i = a;          // i = a.int();

    d = a;              // d = a.double();
    i = a;              // i = a.int();
}
```

Az elemzés itt is alulról felfelé történik, egyszerre csak egy operátornak és paramétereinek figyelembe vételével.

11.5. Barát függvények

Amikor egy függvényt egy osztály tagjaként adunk meg, három, logikailag különböző dolgot jelzünk:

1. A függvény hozzáférhet az osztály deklarációjának privát részeihez.
2. A függvény az osztály hatókörébe tartozik.
3. A függvényt az osztály egy objektumára kell meghívni (egy *this* mutató áll a rendelkezésére).

Ha egy tagfüggvényt *static*-ként határozunk meg (§10.2.4), akkor ez csak az első két tulajdonságot jelenti; ha *friend*-ként („barátként"), csak az elsőt.

Definiáljunk például egy *Matrix*-ot egy *Vector*-ral szorzó operátort. Természetesen mind a *Matrix*, mind a *Vector* osztály alkalmazza az adatrejtés elvét, és csak tagfüggvényeiken keresztül kezelhetjük őket. A szorzást megvalósító függvény azonban nem lehet mindkét osztály tagja. Nem is akarunk általános, alacsonyszintű hozzáférést megengedni, hogy minden felhasználó írhassa és olvashassa a *Matrix* és *Vector* osztályok teljes adatábrázolását. Ahhoz, hogy ezt elkerüljük, a * operátort mindkét osztályban *friend* („barát") függvényként határozzuk meg:

```
class Matrix;

class Vector {
    float v[4];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};

class Matrix {
    Vector v[4];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};

Vector operator*(const Matrix& m, const Vector& v)
{
    Vector r;
    for (int i = 0; i<4; i++) {                          // r[i] = m[i] * v;
            r.v[i] = 0;
            for (int j = 0; j<4; j++) r.v[i] += m.v[i].v[j] * v.v[j];
    }
    return r;
}
```

A *friend* deklarációt az osztály privát és nyilvános részébe is tehetjük. A tagfüggvényekhez hasonlóan a barát függvényeket is az osztály deklarációjában adjuk meg, így ugyanolyan mértékben hozzátartoznak az osztály felületéhez, mint a tagfüggvények.

Egy osztály tagfüggvénye lehet egy másik osztály barát függvénye:

```
class List_iterator {
    // ...
    int* next();
};

class List {
    friend int* List_iterator::next();
    // ...
};
```

Nem szokatlan helyzet, hogy egy osztály összes tagfüggvénye egy másik osztály „barátja". Ennek jelzésére egy rövidítés szolgál:

```
class List {
    friend class List_iterator;
    // ...
};
```

E deklaráció hatására a *List_iterator* osztály összes tagfüggvénye a *List* osztály barát függvénye lesz.

Világos, hogy *friend* osztályokat csak szorosan összetartozó fogalmak kifejezésére szabad használnunk. Számos esetben viszont választhatunk, hogy egy osztályt tag (beágyazott osztály) vagy nem tag barátként adunk meg (§24.4).

11.5.1. A barát függvények elérése

A tagfüggvények deklarációjához hasonlóan a *friend* deklarációk sem vezetnek be új nevet a tartalmazó hatókörbe:

```
class Matrix {
    friend class Xform;
    friend Matrix invert(const Matrix&);
    // ...
};

Xform x;                                   // hiba: a hatókörben nincs Xform
Matrix (*p)(const Matrix&) = &invert;      // hiba: a hatókörben nincs invert()
```

Nagy programok és osztályok esetében előnyös, ha egy osztály nem ad hozzá titokban új neveket a tartalmazó hatókörhöz, azoknál a sablon osztályoknál pedig, amelyek több különböző környezetben példányosíthatók (13. fejezet), ez kifejezetten fontos.

A barát (friend) osztályt előzőleg meg kell adnunk a tartalmazó hatókörben vagy ki kell fejtenünk az osztályt közvetlenül tartalmazó nem osztály típusú hatókörben. A közvetlenül tartalmazó névtér hatókörén kívüli neveket nem veszünk figyelembe:

```
class AE { /* ... */ };        // nem "barátja" Y-nak

namespace N {
    class X { /* ... */ };     // Y "barátja"
    class Y {
            friend class X;
            friend class Z;
            friend class AE;
    };
    class Z { /* ... */ };     // Y "barátja"
}
```

A barát függvényeket ugyanúgy megadhatjuk pontosan, mint a barát osztályokat, de elérhetjük paramétereik alapján is (§8.2.6), még akkor is, ha nem a közvetlenül tartalmazó hatókörben adtuk meg:

```
void f(Matrix& m)
{
    invert(m);      // a Matrix "barát" invert()-je
}
```

Ebből következik, hogy egy barát függvényt vagy egy tartalmazó hatókörben kell közvetlenül megadnunk, vagy az osztályának megfelelő paraméterrel kell rendelkeznie, máskülönben nem hívhatjuk meg:

```
// a hatókörben nincs f()

class X {
    friend void f();                          // értelmetlen
    friend void h(const X&);                  // paramétere alapján megtalálható
};

void g(const X& x)
{
    f();                                      // a hatókörben nincs f()
    h(x);    // X h() "barátja"
}
```

11.5.2. Barátok és tagfüggvények

Mikor használjunk barát függvényt és mikor jobb választás egy tagfüggvény egy művelet számára? Az első szempont egy osztálynál az, hogy minél kevesebb függvény érje el közvetlenül az adatábrázolást és hogy az adatlekérdezést a segédfüggvények megfelelő körével támogassuk. Ezért az elsődleges kérdés nem az, hogy „ez a függvény tag legyen, statikus tag vagy barát?”, hanem az, hogy „tényleg szüksége van-e az ábrázolás elérésére?” Általában kevesebb függvénynek van erre ténylegesen szüksége, mint első ránézésre gondolnánk.

Bizonyos műveleteknek tagoknak kell lenniük: például a konstruktoroknak, destruktoroknak és a virtuális függvényeknek (§12.2.6). Sokszor azonban van mérlegelési lehetőség. Mivel egy tagfüggvény neve az osztályra nézve lokálisnak számít, a függvényt inkább tagfüggvényként adjuk meg, hacsak nem szól valamilyen érv amellett, hogy nem tag függvény legyen.

Vegyünk egy *X* osztályt, amely egy művelet különféle módozatait jeleníti meg:

```
class X {
    // ...
    X(int);

    int m1();
    int m2() const;

    friend int f1(X&);
    friend int f2(const X&);
    friend int f3(X);
};
```

A tagfüggvényeket csak az adott osztály objektumaira alkalmazhatjuk; felhasználói átalakítást a fordító nem végez:

```
void g()
{
    99.m1();        // hiba: nem próbáltuk X(99).m1()-et
    99.m2();        // hiba: nem próbáltuk X(99).m2()-őt
}
```

Az *X(int)* átalakítást a fordító nem alkalmazza, hogy a *99*-ből *X* típusú objektumot csináljon.

Az *f1()* globális függvény hasonló tulajdonsággal rendelkezik, mert nem *const* referencia paraméterekre a fordító nem alkalmaz felhasználói átalakítást (§5.5, §11.3.5). Az *f2()* és *f3()* paramétereire azonban alkalmazható ilyen:

```
void h()
{
    f1(99); // hiba: nem próbáltuk f1(X(99))-et
    f2(99); // rendben: f2(X(99));
    f3(99); // rendben: f3(X(99));
}
```

Ezért egy, az objektum állapotát megváltoztató művelet vagy tag legyen, vagy pedig nem *const* referencia (vagy nem *const* mutató) paraméterű globális függvény. Olyan műveletet, amelynek balértékre van szüksége, ha alapvető adattípusra alkalmazzuk (=, *=, ++ stb.), a legtermészetesebb módon felhasználói típus tagfüggvényeként definiálhatunk.

Megfordítva: ha egy művelet összes operandusa automatikusan konvertálható, akkor a megvalósító függvény csak olyan nem tag függvény lehet, amely paraméterként *const* referencia vagy nem referencia típust vár. Ez gyakori eset olyan műveleteket megvalósító függvényeknél, melyeknek nincs szükségük balértékre, ha alapvető adattípusra alkalmazzuk azokat (+, -, || stb.). Az ilyen műveleteknek gyakran az operandus-osztály ábrázolásának elérésére van szükségük, ezért aztán a kétoperandusú operátorok a *friend* függvények leggyakoribb forrásai.

Ha nincs típuskonverzió, akkor nincs kényszerítő ok arra sem, hogy válasszunk a tagfüggvény és a referencia paramétert váró barát függvény közül. Ilyenkor a programozó aszerint dönthet, hogy melyik formát részesíti előnyben. A legtöbb embernek például jobban tetszik az *inv(m)* jelölés, ha egy *m Matrix* inverzéről van szó, mint a másik lehetséges *m.inv()* jelölés. Ha azonban az *inv()* azt a *Matrix*-ot invertálja, amelyikre alkalmaztuk és nem egy új *Matrix*-ként adja vissza az inverzt, akkor persze csak tagfüggvény lehet.

Ha más szempontok nem játszanak közre, válasszunk tagfüggvényt. Nem tudhatjuk, hogy valaki nem ad-e majd meg valamikor egy konverziós operátort, és azt sem láthatjuk előre, hogy egy jövőbeli módosítás nem változtatja-e meg az objektum állapotát. A tagfüggvényhívási forma világossá teszi a felhasználó számára, hogy az objektum állapota megváltozhat; referencia paraméter használata esetén ez sokkal kevésbé nyilvánvaló. Továbbá sokkal rövidebbek a kifejezések egy tagfüggvény törzsében, mint a külső függvénybeli megfelelőik; egy nem tag függvénynek meghatározott paraméterre van szüksége, míg a tagfüggvény automatikusan használhatja a *this* mutatót. Ezenkívül, mivel a tagfüggvények neve az osztályra nézve lokálisnak számít, a külső függvények neve hosszabb szokott lenni.

11.6. Nagy objektumok

A *complex* osztály műveleteinek paramétereit *complex* típusúként határoztuk meg. Ez azt jelenti, hogy a paraméterek minden műveletnél lemásolódnak. Két *double* másolása „költséges” művelet lehet ugyan, de valószínűleg „olcsóbb”, mint egy pár mutatóé. Nem minden osztálynak van azonban kényelmesen kicsi ábrázolása. A nagymérvű másolásokat elkerülendő, megadhatunk referencia típusú paramétereket kezelő függvényeket:

```
class Matrix {
    double m[4][4];
public:
    Matrix();
    friend Matrix operator+(const Matrix&, const Matrix&);
    friend Matrix operator*(const Matrix&, const Matrix&);
};
```

A referenciák alkalmazása nagy objektumokra is lehetővé teszi a szokásos aritmetikai műveletek használatát, nagymérvű másolások nélkül is. Mutatókat nem használhatunk, mert a mutatóra alkalmazott operátorok jelentését nem változtathatjuk meg. Az összeadást így definiálhatnánk:

```
Matrix operator+(const Matrix& arg1, const Matrix& arg2)
{
    Matrix sum;
    for (int i=0; i<4; i++)
            for (int j=0; j<4; j++)
                    sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
    return sum;
}
```

Ez az *operator+()* az operandusokat referenciákon keresztül éri el, de objektum-értéket ad vissza. Referenciát visszaadni hatékonyabbnak tűnhet:

```
class Matrix {
    // ...
    friend Matrix& operator+(const Matrix&, const Matrix&);
    friend Matrix& operator*(const Matrix&, const Matrix&);
};
```

Ez szabályos kód, de egy memória-lefoglalási problémát okoz. Minthogy a függvényből az eredményre vonatkozó referenciát adjuk vissza, az eredmény maga nem lehet automatikus változó (§7.3). Mivel egy műveletet többször is alkalmazhatunk egy kifejezésen belül, az

eredmény nem lehet lokális statikus változó sem. Ezért aztán az eredménynek jellemzően a szabad tárban foglalnánk helyet. A visszatérési érték másolása (végrehajtási időben, kód- és adatméretben mérve) gyakran olcsóbb, mint az objektum szabad tárba helyezése és onnan eltávolítása, és programozni is sokkal egyszerűbb.

Az eredmény másolásának elkerülésére vannak módszerek. A legegyszerűbb ezek közül egy statikus objektumokból álló átmeneti tár használata:

```
const max_matrix_temp = 7;

Matrix& get_matrix_temp()
{
    static int nbuf = 0;
    static Matrix buf[max_matrix_temp];

    if (nbuf == max_matrix_temp) nbuf = 0;
    return buf[nbuf++];
}

Matrix& operator+(const Matrix& arg1, const Matrix& arg2)
{
    Matrix& res = get_matrix_temp();
    // ...
    return res;
}
```

Így egy *Matrix* másolása csak egy kifejezés értékén alapuló értékadáskor történik meg. De az ég legyen irgalmas, ha olyan kifejezést találnánk írni, amelyhez *max_matrix_temp*-nél több ideiglenes érték kell!

Hibákra kevesebb lehetőséget adó módszer, ha a mátrix típust csak a tényleges adatot tároló típus leírójaként (handle, §25.7) határozzuk meg. Így aztán a mátrixleírók úgy képviselhetik az objektumokat, hogy közben a lehető legkevesebb helyfoglalás és másolás történik (§11.12 és §11.14[18]). Ez az eljárás azonban a visszatérési értékként objektumot és nem referenciát vagy mutatót használó operátorokon alapul. Egy másik módszer háromváltozós műveletek meghatározására és azok olyankor automatikusan történő meghívására támaszkodik, amikor olyan kifejezések kiértékelése történik, mint *a=b+c* vagy *a+b*i* (§21.4.6.3 és §22.4.7).

11.7. Alapvető operátorok

Általánosságban, ha *X* egy típus, akkor az *X(const X&)* másoló konstruktor kezeli azt az esetet, amikor egy *X* típusú objektumnak egy ugyanilyen objektumot adunk kezdőértékül. Nem lehet eléggé hangsúlyozni, hogy a kezdeti és az egyszerű értékadás különböző műveletek (§10.4.4.1). Ez különösen fontos akkor, amikor a destruktorral is számolnunk kell. Ha az *X* osztálynak van valamilyen nem magától értetődő feladatot – például a szabad tárban lefoglalt memória felszabadítását – végző destruktora, akkor az osztálynak valószínűleg szüksége lesz az objektum létrehozását, megsemmisítését és másolását végző összes függvényre:

```
class X {
    // ...
    X(Sometype);                    // konstruktor: objektumok létrehozása
    X(const X&);                    // másoló konstruktor
    X& operator=(const X&);         // másoló értékadás: takarítás és másolás
    ~X();                           // destruktor: takarítás
};
```

Ezenkívül még háromféle helyzetben másolódik egy objektum: átadott függvény-paraméterként, függvény visszatérési értékeként, illetve kivételként. Ha paraméterként kerül átadásra, egy addig kezdőérték nélküli változó, a formális paraméter kap kezdőértéket. Ennek szerepe azonos az egyéb kezdeti értékadásokéval. Ugyanez igaz a visszatérési értékre és a kivételre is, még ha kevésbé nyilvánvaló is. Ilyen esetben a másoló konstruktor végzi a munkát:

```
string g(string arg)      // string érték szerint átadva (másoló konstruktor használatával)
{
    return arg;           // string visszaadása (másoló konstruktor használatával)
}

int main ()
{
    string s = "Newton";  // string kezdőértéket kap (másoló konstruktor használatával)
    s = g(s);
}
```

Világos, hogy az *s* változó értékének *"Newton"*-nak kell lennie a *g()* meghívása után. Nem nehéz feladat az *s* értékének egy másolatát az *arg* formális paraméterbe másolni; a *string* osztály másoló konstruktorának hívása ezt megteszi. Amikor *g()* visszaadja a visszatérési értéket, a *string(const string&)* újabb hívása következik, amikor egy olyan ideiglenes

változó kap értéket, amely aztán az *s*-nek ad értéket. Hatékonysági okokból az egyik (de csak az egyik) másolást gyakran elhagyhatjuk. Az ideiglenes változók aztán persze a *string::~string()* destruktor segítségével megsemmisülnek (§10.4.10).

Ha a programozó nem ad meg másoló konstruktort vagy másoló értékadást egy osztály számára, a fordítóprogram hozza létre a hiányzó függvényt vagy függvényeket (§10.2.5). Ez egyben azt is jelenti, hogy a másoló műveletek nem öröklődnek (§12.2.3).

11.7.1. Explicit konstruktorok

Alapértelmezés szerint az egyparaméterű konstruktor egyben automatikus konverziót is jelent. Bizonyos típusok számára ez ideális:

```
complex z = 2;     // z kezdeti értékadása complex(2)-vel
```

Máskor viszont nem kívánatos és hibák forrása lehet:

```
string s = 'a';     // s karakterlánc, int('a') számú elemmel
```

Nagyon valószínűtlen, hogy az *s*-et megadó programozó ezt akarta volna.

Az automatikus konverziókat az *explicit* kulcsszó alkalmazásával akadályozhatjuk meg. Vagyis egy *explicit*-ként megadott konstruktort csak közvetlen módon lehet meghívni. Így ahol elvileg egy másoló konstruktorra van szükség (§11.3.4), ott az *explicit* konstruktor nem hívódik meg automatikusan:

```
class String {
    // ...
    explicit String(int n); // n bájt lefoglalása
    String(const char* p);// a kezdőérték (p) egy C stílusú karakterlánc
};

String s1 = 'a';          // hiba: nincs automatikus char->String átalakítás
String s2(10);            // rendben: String 10 karakternyi hellyel
String s3 = String(10);   // rendben: String 10 karakternyi hellyel
String s4 = "Brian";      // rendben: s4 = String("Brian")
String s5("Fawlty");

void f(String);

String g()
{
```

```
    f(10);              // hiba: nincs automatikus int ->String átalakítás
    f(String(10));
    f("Arthur");        // rendben: f(String("Arthur"))
    f(s1);

    String* p1 = new String("Eric");
    String* p2 = new String(10);

    return 10;          // hiba: nincs automatikus int ->String átalakítás
}
```

A különbség aközött, hogy

```
String s1 = 'a';        // hiba: nincs automatikus char ->String átalakítás
```

és aközött, hogy

```
String s2(10);          // rendben: karakterlánc 10 karakternyi hellyel
```

csekélynek tűnhet, de igazi kódban kevésbé az, mint kitalált példákban.

A *Date* osztályban egy sima *int*-et használtunk az év ábrázolására (§10.3). Ha a *Date* osztály létfontosságú szerepet játszott volna, akkor bevezethettük volna a *Year* osztályt, hogy fordítási időben szigorúbb ellenőrzések történjenek:

```
class Year {
    int y;
public:
    explicit Year(int i) : y(i) { }             // Year létrehozása int-ből
    operator int() const { return y; }          // átalakítás Year-ről int-re
};

class Date {
public:
    Date(int d, Month m, Year y);
    // ...
};

Date d3(1978,feb,21);                           // hiba: a 21 nem Year típusú
Date d4(21,feb,Year(1978));                     // rendben
```

A *Year* egy egyszerű „csomagoló” (beburkoló, wrapper) osztály az *int* körül. Az *operator int()*-nek köszönhetően a *Year* automatikusan mindenhol *int*-té alakul, ahol szükséges. Azáltal, hogy a konstruktort *explicit*-ként adtuk meg, biztosítottuk, hogy az *int*-nek *Year*-ré va-

ló alakítása csak ott történik meg, ahol ezt kérjük és a „véletlen" értékadások a fordításkor kiderülnek. Minthogy a *Year* tagfüggvényeit könnyű helyben kifejtve (inline) fordítani, a futási idő és a szükséges tárhely növekedésétől sem kell tartanunk.

Hasonló módszer tartomány (intervallum) típusokra (§25.6.1) is alkalmazható.

11.8. Indexelés

Osztály típusú objektumoknak az *operator [ ]* (subscripting) függvény segítségével adhatunk „sorszámot" (indexet). Az *operator[ ]* második paramétere (az index) bármilyen típusú lehet, így aztán vektorokat, asszociatív tömböket stb. is definiálhatunk.

Példaként írjuk most újra a §5.5-beli példát, amelyben egy asszociatív tömb segítségével írtunk egy fájlban a szavak előfordulását megszámoló kis programot. Akkor egy függvényt használtunk, most egy asszociatív tömb típust:

```
class Assoc {
    struct Pair {
            string name;
            double val;
            Pair(string n ="", double v =0) :name(n), val(v) { }
    };
    vector<Pair> vec;

    Assoc(const Assoc&);                        // a másolást megakadályozandó privát
    Assoc& operator=(const Assoc&);             // a másolást megakadályozandó privát
public:
    Assoc() {}
    const double& operator[ ](const string&);
    double& operator[ ](string&);
    void print_all() const;
};
```

Az *Assoc* típusú objektumok *Pair*-ek vektorát tartalmazzák. A megvalósításban ugyanazt az egyszerű és nem túl hatékony keresési módszert használjuk, mint az §5.5 pontban:

```
double& Assoc::operator[ ](string& s)
    // megkeressük s-t; ha megtaláltuk, visszaadjuk az értékét; ha nem, új Pair-t hozunk
    // létre és az alapértelmezett 0 értéket adjuk vissza
{
```

```
    for (vector<Pair>::iterator p = vec.begin(); p!=vec.end(); ++p)
        if (s == p->name) return p->val;

    vec.push_back(Pair(s,0));                    // kezdőérték: 0

    return vec.back().val;                       // az utolsó elem visszaadása (§16.3.3)
}
```

Minthogy az *Assoc* objektum ábrázolása kívülről nem érhető el, szükség van egy kimeneti függvényre:

```
void Assoc::print_all() const
{
    for (vector<Pair>::const_iterator p = vec.begin(); p!=vec.end(); ++p)
        cout << p->name << ": " << p->val << '\n';
}
```

Végül megírhatjuk a főprogram egyszerű változatát:

```
int main()      // szavak előfordulásának megszámlálása a bemeneten
{
    string buf;
    Assoc vec;
    while (cin>>buf) vec[buf]++;
    vec.print_all();
}
```

Az asszociatív tömb ötletét továbbfejleszti a §17.4.1 pont.

Az *operator[ ]()* függvényeknek tagfüggvénynek kell lenniük.

11.9. Függvényhívás

A függvényhívás (function call), vagyis a *kifejezés(kifejezés-lista)* jelölés úgy tekinthető, mint egy kétoperandusú művelet, ahol a kifejezés a bal oldali, a kifejezés-lista pedig a jobb oldali operandus. A *()* hívó operátor a többi operátorhoz hasonló módon túlterhelhető. Az *operator()()* paraméterlistájának kiértékelése és ellenőrzése a szokásos paraméter-átadási szabályok szerint történik. A függvényhívó operátor túlterhelése elsősorban olyan típusok létrehozásakor hasznos, amelyeknek csak egy műveletük van vagy általában csak egy műveletük használatos.

A *()* hívó művelet legnyilvánvalóbb és talán legfontosabb alkalmazása az, hogy a valamiképpen függvényként viselkedő objektumokat függvényként hívhassuk meg. Egy függvényként viselkedő objektumot *függvényszerű* vagy egyszerűen *függvényobjektumnak* hívunk (§18.4). Az ilyen függvényobjektumok fontosak, mert lehetővé teszik, hogy olyan kódot írjunk, amelyben valamilyen nem magától értetődő műveletet paraméterként adunk át. A standard könyvtárban például sok olyan algoritmus található, melyek egy függvényt hívnak meg egy tároló minden elemére. Vegyük az alábbi példát:

```
void negate(complex& c) { c = -c; }

void f(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(),aa.end(),negate);        // a vektor összes elemének negálása
    for_each(ll.begin(),ll.end(),negate);        // a lista összes elemének negálása
}
```

Ez a vektor és a lista minden elemét negálja.

Mi lenne, ha a lista minden eleméhez *complex(2,3)*-at akarnánk hozzáadni? Ezt könnyen megtehetjük:

```
void add23(complex& c)
{
    c += complex(2,3);
}

void g(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(),aa.end(),add23);
    for_each(ll.begin(),ll.end(),add23);
}
```

Hogyan tudnánk egy olyan függvényt írni, melyet többször meghívva egy-egy tetszőleges értéket adhatunk az elemekhez? Olyasmire van szükségünk, aminek megadhatjuk a kívánt értéket és utána ezt az értéket használja fel minden hívásnál. Ez a függvényeknek nem természetes tulajdonsága. Jellemző megoldásként valahova a függvényt körülvevő környezetbe helyezve adjuk át az értéket, ami nem „tiszta" megoldás. Viszont írhatunk egy osztályt, amely a megfelelő módon működik:

```
class Add {
    complex val;
public:
    Add(complex c) { val = c; }                  // az érték mentése
    Add(double r, double i) { val = complex(r,i); }

    void operator()(complex& c) const { c += val; } // a paraméter növelése az értékkel
};
```

Egy *Add* osztályú objektum kezdőértékének egy komplex számot adunk, majd a *()* műveletet végrehajtatva ezt a számot hozzáadjuk a paraméterhez:

```
void h(vector<complex>& aa, list<complex>& ll, complex z)
{
    for_each(aa.begin(),aa.end(),Add(2,3));
    for_each(ll.begin(),ll.end(),Add(z));
}
```

Ez a tömb minden eleméhez *complex(2,3)*-at fog adni, a lista elemeihez pedig *z*-t. Vegyük észre, hogy *Add(z)* egy olyan objektumot hoz létre, amelyet aztán a *for_each* ismételten felhasznál. Nem egyszerűen egy egyszer vagy többször meghívott függvényről van szó. A többször meghívott függvény az *Add(z) operator()()* függvénye.

Mindez azért működik, mert a *for_each* egy sablon (template), amely a *()* műveletet alkalmazza a harmadik paraméterére, anélkül, hogy törődne vele, mi is igazából a harmadik paraméter:

```
template<class Iter, class Fct> Fct for_each(Iter b, Iter e, Fct f)
{
    while (b != e) f(*b++);
    return f;
}
```

Első pillantásra ez a módszer furcsának tűnhet, de egyszerű, hatékony, és nagyon hasznos (lásd §3.8.5, §18.4).

Az *operator()()* további népszerű alkalmazásai a részláncok képzésére vagy több dimenziós tömbök indexelésére (§22.4.5) való használat.

Az *operator()()*-nak tagfüggvénynek kell lennie.

11.10. Indirekció

A -> indirekció („hivatkozástalanítás”, dereferencing) operátort egyparaméterű, utótagként használt operátorként definiálhatjuk. Legyen adott egy osztály:

```
class Ptr {
    // ...
    X* operator->();
};
```

Ekkor a *Ptr* osztályú objektumokat az *X* osztály tagjainak elérésére használhatjuk, a mutatókhoz nagyon hasonló módon:

```
void f(Ptr p)
{
    p->m = 7;            // (p.operator->())->m = 7
}
```

A *p* objektumnak a *p.operator->()* mutatóvá való átalakítása nem függ attól, hogy milyen *m* tagra mutat. Az *operator->()* ebben az értelemben egyoperandusú utótag-operátor, formai követelményei viszont nem újak, így a tagnevet ki kell írni utána:

```
void g(Ptr p)
{
    X* q1 = p->;                    // szintaktikus hiba
    X* q2 = p.operator->();         // rendben
}
```

A *->()* operátor túlterhelésének fő alkalmazása az okos vagy intelligens mutató (smart pointer) típusok létrehozása, azaz olyan objektumoké, amelyek mutatóként viselkednek, de ráadásul valamilyen tennivalót végeznek, valahányszor egy objektumot érnek el rajtuk keresztül. Például létrehozhatunk egy *Rec_ptr* osztályt, amellyel a lemezen tárolt *Rec* osztályú objektumok érhetőek el. A *Rec_ptr* konstruktora egy nevet vár, melynek segítségével a keresett objektum a lemezen megkereshető, a *Rec_ptr::operator->()* függvény a memóriába tölti az objektumot, amikor azt a *Rec_ptr*-en keresztül el akarjuk érni, a *Rec_ptr* destruktora pedig szükség esetén a megváltozott objektumot a lemezre írja:

```
class Rec_ptr {
    const char* identifier;
    Rec* in_core_address;
    // ...
public:
    Rec_ptr(const char* p) : identifier(p), in_core_address(0) { }
    ~Rec_ptr() { write_to_disk(in_core_address,identifier); }
    Rec* operator->();
};

Rec* Rec_ptr::operator->()
{
    if (in_core_address == 0) in_core_address = read_from_disk(identifier);
    return in_core_address;
}
```

A *Rec_ptr*-t így használhatjuk:

```
struct Rec {                                  // a Rec típus, amire Rec_ptr mutat
    string name;
    // ...
};

void update(const char* s)
{
    Rec_ptr p(s);              // Rec_ptr előállítása s-ből

    p->name = "Roscoe";        // s módosítása; ha szükséges, először beolvassa a lemezről
    // ...
}
```

Természetesen az igazi *Rec_ptr* egy sablon lenne, a *Rec* típus pedig paraméter. Egy valóságos program hibakezelést is tartalmazna és kevésbé naív módon kezelné a lemezt.

Közönséges mutatók esetében a *->* használata egyenértékű az egyváltozós *** és *[ ]* használatával. Ha adott egy típus:

```
Y* p;
```

akkor teljesül a következő:

```
p->m == (*p).m == p[0].m
```

Ahogy már megszokhattuk, a felhasználói operátorokra nézve ez nem biztosított. Szükség esetén persze gondoskodhatunk erről:

```
class Ptr_to_Y {
    Y* p;
public:
    Y* operator->() { return p; }
    Y& operator*() { return *p; }
    Y& operator[ ](int i) { return p[i]; }
};
```

Ha egy osztályban több ilyen operátort határozunk meg, akkor tanácsos lehet ezt úgy tenni, hogy a fenti egyenértékűség teljesüljön, ugyanúgy, mint ahogy *++x* és *x+=1* is jó, ha *x=x+1*-gyel azonos hatással jár, ha *x* egy olyan osztályú változó, amelyben a ++, +=, + műveletek értelmezettek.

A -> operátor túlterhelhetősége nem csak kis különlegesség, hanem érdekes programok egy osztálya számára fontos is, azon oknál fogva, hogy az indirekció (dereferencing) kulcsfogalom, a -> operátor túlterhelése pedig tiszta, közvetlen és hatékony módja annak egy programban való megjelenítésére. A bejárók (iterátorok) (19. fejezet) jellemző és lényegi példát adnak erre. A -> operátor másik haszna, hogy korlátozott, de hasznos módon lehetővé teszi a C++ nyelvben a delegációt (§24.2.4).

Az *operator->* tagfüggvény kell, hogy legyen. Csak úgy használható, ha mutatót vagy olyan típust ad vissza, amelyre a -> alkalmazható. Ha egy sablon osztály számára adjuk meg, sokszor előfordul, hogy nem is kerül tényleges felhasználásra, ezért ésszerű e megszorítás ellenőrzését a tényleges használatig elhalasztani.

11.11. Növelés és csökkentés

Amint a programozó kitalál egy „intelligens mutatót", sokszor dönt úgy, hogy ehhez a ++ növelő (increment) és -- csökkentő (decrement) művelet is hozzátartozik, a beépített típusokra értelmezett növelés és csökkentés mintájára. Ez különösen nyilvánvaló és szükséges olyankor, amikor a cél egy közönséges mutatónak egy okosra való kicserélése, amely azonos jelentés mellett csak némi futási idejű hiba-ellenőrzéssel van kiegészítve. Vegyük például az alábbi – egyébként problematikus – hagyományos programot:

```
void f1(T a)              // hagyományos használat
{
    T v[200];
    T* p = &v[0];
    p--;
    *p = a;               // hoppá: 'p' tartományon kívüli és nem kaptuk el
    ++p;
    *p = a;               // rendben
}
```

A *p* mutatót ki szeretnénk cserélni valamilyen *Ptr_to_T* osztályú objektumra, amelyre csak akkor tudjuk alkalmazni az indirekció operátort, ha tényleg egy objektumra mutat. Azt is el szeretnénk érni, hogy *p*-t csak úgy lehessen növelni vagy csökkenteni, ha tömbön belüli objektumra mutat, még a növelés vagy csökkentés hatására is. Valami ilyesmit szeretnénk tehát:

```
class Ptr_to_T {
    // ...
};
```

```
void f2(T a)                // ellenőrzött
{
    T v[200];
    Ptr_to_T p(&v[0],v,200);
    p--;
    *p = a;          // futási idejű hiba: 'p' tartományon kívüli
    ++p;
    *p = a;          // rendben
}
```

A növelő és csökkentő operátorok az egyetlenek a C++ nyelv operátorai között, amelyek előtagként (prefix) és utótagként (postfix) egyaránt használhatók. Ezért a *Ptr_to_T* típus számára mindkét fajta növelő és csökkentő operátort definiálnunk kell:

```
class Ptr_to_T {
    T* p;
    T* array;
    int size;
public:

    Ptr_to_T(T* p, T* v, int s);  // csatolás s méretű v tömbhöz, a kezdőérték p
    Ptr_to_T(T* p);               // csatolás önálló objektumhoz, a kezdőérték p

    Ptr_to_T& operator++();                 // előtag
    Ptr_to_T operator++(int);               // utótag

    Ptr_to_T& operator--();                 // előtag
    Ptr_to_T operator--(int);               // utótag

    T& operator*();                         // előtag
};
```

Az *int* paraméterrel jelezzük a ++ utótagként való alkalmazását. Magát az *int*-et nem használjuk, csak ál-paraméter, amely az elő- és utótagként való használat között tesz különbséget. Könnyen megjegyezhetjük, melyik melyik, ha arra gondolunk, hogy a többi (aritmetikai és logikai) egy paraméterű operátorhoz hasonlóan az ál-paraméter nélküli ++ és -- az előtagként, a paraméteres változat a „furcsa" utótagként való működéshez kell.

A *Prt_to_T* osztályt használva a példa egyenértékű az alábbival:

```
void f3(T a)                // ellenőrzött
{
    T v[200];
    Ptr_to_T p(&v[0],v,200);
    p.operator--(0);
```

```
    p.operator*() = a;      // futási idejű hiba: 'p' tartományon kívüli
    p.operator++();
    p.operator*() = a;      // rendben
}
```

A *Prt_to_T* osztály kiegészítése gyakorlatnak marad (§11.14[19]). Átdolgozása olyan sablonná, amely kivételeket is használ a futási időben fellépő hibák jelzésére, egy másik gyakorlat (§14.12[12]). A §13.6.3 egy mutatósablont mutat be, amely öröklődés használata mellett is jól működik.

11.12. Egy karakterlánc osztály

Íme a *String* osztály egy valóságosabb változata, amely a céljainknak még éppen megfelel. Ez a karakterlánc-osztály támogatja az érték szerinti működést (value semantics, érték-szemantika), a karakteríró és -olvasó műveleteket, az ellenőrzött és ellenőrizetlen elérést, az adatfolyam ki- és bemenetet, a karakterliterálokat, az egyenlőségvizsgáló és összefűző műveleteket. A karakterláncokat C stílusú, nullával lezárt karaktertömbként tárolja, a másolások számának csökkentésére pedig hivatkozásszámlálót használ. Egy többet tudó és/vagy több szolgáltatást nyújtó *string* osztály írása jó gyakorlat (§11.14[7-12]). Ha megvagyunk vele, eldobhatjuk a gyakorlatainkat és használhatjuk a standard könyvtárbeli *string*-et (20. fejezet).

Az én majdnem valóságos *String* osztályom három segédosztályt használ: az *Srep*-et, hogy több azonos értékű *String* is használhassa ugyanazt az eltárolt adatot, ha azonos az értékük; a *Range*-et az értéktartomány-megsértési hibákat jelző kivételek kiváltásához; és a *Cref*-et, hogy egy írás és olvasás között különbséget tevő index-operátort támogasson:

```
class String {
    struct Srep;                    // adatábrázolás
    Srep *rep;
public:
    class Cref;                     // referencia char-ra

    class Range { };                // kivételkezeléshez

    // ...
};
```

A többi taghoz hasonlóan a *tagosztályokat* (member class, amit gyakran hívnak *beágyazott osztálynak*, nested class-nak is) deklarálhatjuk az osztályban, majd később kifejthetjük:

```
struct String::Srep {
    char* s;              // mutató az elemekre
    int sz;               // karakterek száma
    int n;                // hivatkozásszámláló

    Srep(int nsz, const char* p)
    {
            n = 1;
            sz = nsz;
            s = new char[sz+1];                   // hely a lezáró nulla számára is
            strcpy(s,p);
    }

    ~Srep() { delete[ ] s; }

    Srep* get_own_copy()                          // másolás, ha szükséges
    {
            if (n==1) return this;
            n--;
            return new Srep(sz,s);
    }

    void assign(int nsz, const char* p)
    {
            if (sz != nsz) {
                    delete[ ] s;
                    sz = nsz;
                    s = new char[sz+1];
            }
            strcpy(s,p);
    }
private:                                          // a másolás megakadályozása
    Srep(const Srep&);
    Srep& operator=(const Srep&);
};
```

A *String* osztálynak megvannak a szokásos konstruktorai, destruktora és értékadó műveletei is:

```
class String {
    // ...
```

```
    String();                               // x = ""
    String(const char*);                    // x = "abc"
    String(const String&);                  // x = másik_karakterlánc
    String& operator=(const char *);
    String& operator=(const String&);
    ~String();

    // ...
};
```

A *String* osztály érték szerint működik, azaz egy *s1=s2* értékadás után *s1* és *s2* két teljesen különböző karakterlánc lesz, vagyis ha később az egyiket módosítjuk, akkor annak nem lesz hatása a másikra. A másik megoldás az lenne, ha a *String* osztály mutatókkal dolgozna. Ekkor az *s1=s2* értékadás után *s2* megváltoztatása *s1*-et is érintené. Ha egy osztálynak megvannak a hagyományos aritmetikai műveletei, mint a komplex számokkal, vektorokkal, mátrixokkal, karakterláncokkal végzettek, én előnyben részesítem az érték szerinti működést. Ahhoz viszont, hogy ennek támogatása ne kerüljön túl sokba, a *String*-et leíróként ábrázolom, amely az adatábrázolásra mutat, amit csak szükség esetén kell másolni:

```
String::String()            // az alapértelmezett érték egy üres karakterlánc
{
    rep = new Srep(0,"");
}

String::String(const String& x)              // másoló konstruktor
{
    x.rep->n++;
    rep = x.rep;                              // az ábrázolás megosztása
}

String::~String()
{
    if (--rep->n == 0) delete rep;
}

String& String::operator=(const String& x)   // másoló értékadás
{
    x.rep->n++;                               // védelem az "st = st" ellen
    if (--rep->n == 0) delete rep;
    rep = x.rep;                              // az ábrázolás megosztása
    return *this;
}
```

A *const char** paraméterű ál-másoló műveletek bevezetésével a karakterliterálokat is megengedjük:

```
String::String(const char* s)
{
    rep = new Srep(strlen(s),s);
}

String& String::operator=(const char* s)
{
    if (rep->n == 1)                              // Srep újrahasznosítása
            rep->assign(strlen(s),s);
    else {                                        // új Srep használata
            rep->n--;
            rep = new Srep(strlen(s),s);
    }
    return *this;
}
```

Az egyes karakterláncokat elérő operátorok megtervezése nehéz, mert az ideális megoldás az lenne, ha ezek a szokásos jelölést (azaz a *[ ]*-t) használnák, a lehető leghatékonyabbak lennének és a paraméter értékét is ellenőriznék. Sajnos, ez a három követelmény nem teljesíthető egyszerre. Én úgy készítettem el az osztályt, hogy hatékony ellenőrizetlen műveleteket adtam meg (egy kicsit kényelmetlenebb jelöléssel), illetve kevésbé hatékony ellenőrzött eljárásokat (a hagyományos jelöléssel):

```
class String {
    // ...

    void check(int i) const { if (i<0 || rep->sz<=i) throw Range(); }

    char read(int i) const { return rep->s[i]; }
    void write(int i, char c) { rep=rep->get_own_copy(); rep->s[i]=c; }

    Cref operator[ ](int i) { check(i); return Cref(*this,i); }
    char operator[ ](int i) const { check(i); return rep->s[i]; }

    int size() const { return rep->sz; }

    // ...
};
```

Az ötlet az, hogy a hagyományos *[ ]* jelöléssel az ellenőrzött elérés legyen biztosított a közönséges felhasználás számára, de a felhasználónak legyen módja egyszerre végignézni a teljes tartományt és a gyorsabb, ellenőrizetlen elérést használni:

```
int hash(const String& s)
{
    int h = s.read(0);
    const int max = s.size();
    for (int i = 1; i<max; i++) h ^= s.read(i)>>1;   // ellenőrzés nélküli hozzáférés s-hez
    return h;
}
```

Nehéz dolog egy operátort, például a *[ ]*-t úgy meghatározni, hogy az az író és az olvasó jellegű hozzáférést is támogassa, ha nem fogadható el az a megoldás, hogy egyszerűen egy referenciát adunk vissza, amit aztán a felhasználó kedve szerint felhasználhat. Itt például ez nem lehetséges, mert a *String*-et úgy határoztam meg, hogy az egyes értékadással, paraméter-átadással stb. megadott értékű *String*-ek ugyanazt a belső ábrázolást használják, míg az egyik *String*-et ténylegesen nem írják: az érték másolása csak ekkor történik meg. Ezt a módszert általában ***íráskori másolásnak*** vagy „másolás íráskor"-nak (copy-on-write) hívják. A tényleges másolást a *String::get_own_copy()* végzi.

Abból a célból, hogy az elérő függvényeket helyben kifejtve (inline) lehessen fordíttatni, olyan helyre kell elhelyezni definiciójukat, ahonnan az *Srep* osztályé elérhető. Tehát vagy az *Srep*-et kell a *String* osztályon belül megadni, vagy pedig az elérő függvényeket kell *inline*-ként meghatározni a *String*-en kívül és az *String::Srep* után (§11.14[2]).

Megkülönböztetendő az írást és az olvasást, a *String::operator[ ]()* egy *Cref*-et ad vissza, ha nem *const* objektumra hívták meg. A *Cref* úgy viselkedik, mint a *char&*, azzal a különbséggel, hogy írásakor meghívja a *String::Sref::get_own_copy()*-t:

```
class String::Cref {                                   // hivatkozás s[i]-re
friend class String;
    String& s;
    int i;
    Cref(String& ss, int ii) : s(ss), i(ii) { }
public:
    operator char() const { return s.read(i); }        // érték kijelölése
    void operator=(char c) { s.write(i,c); }           // érték módosítása
};
```

Például:

```
void f(String s, const String& r)
{
    char c1 = s[1];          // c1 = s.operator[ ](1).operator char()
    s[1] = 'c';              // s.operator[ ](1).operator=('c')
```

```
    char c2 = r[1];          // c2 = r.operator[ ](1)
    r[1] = 'd';              // hiba: char értékadás, r.operator[ ](1) = 'd'
}
```

Vegyük észre, hogy egy nem *const* objektumra az *s.operator[ ](1)* értéke *Cref(s,1)* lesz.

Ahhoz, hogy teljessé tegyük a *String* osztályt, meghatározunk még egy sor hasznos függvényt:

```
class String {
    // ...

    String& operator+=(const String&);
    String& operator+=(const char*);

    friend ostream& operator<<(ostream&, const String&);
    friend istream& operator>>(istream&, String&);

    friend bool operator==(const String& x, const char* s)
            { return strcmp(x.rep->s, s) == 0; }

    friend bool operator==(const String& x, const String& y)
            { return strcmp(x.rep->s, y.rep->s) == 0; }

    friend bool operator!=(const String& x, const char* s)
            { return strcmp(x.rep->s, s) != 0; }

    friend bool operator!=(const String& x, const String& y)
            { return strcmp(x.rep->s, y.rep->s) != 0; }
};

String operator+(const String&, const String&);
String operator+(const String&, const char*);
```

Hely-megtakarítás céljából a ki- és bemeneti operátorokat, illetve az összefűzést meghagytam gyakorlatnak.

A főprogram pusztán egy kissé megdolgoztatja a *String* operátorokat:

```
String f(String a, String b)
{
    a[2] = 'x';
    char c = b[3];
    cout << "Az f-ben: " << a << ' ' << b << ' ' << c << '\n';
    return b;
}
```

```
int main()
{
    String x, y;
    cout << "Adjon meg két karakterláncot!\n";
    cin >> x >> y;
    cout << "Bemenet: " << x << ' ' << y << '\n';
    String z = x;
    y = f(x,y);
    if (x != z) cout << "Az x sérült!\n";
    x[0] = '!';
    if (x == z) cout << "Az írás nem sikerült!\n";
    cout << "Kilépés: " << x << ' ' << y << ' ' << z << '\n';
}
```

Ebből a *String* osztályból még hiányoznak fontosnak vagy akár alapvetőnek tekinthető dolgok, például nincs az értéket C stílusú adatként visszaadó művelet (11.14[10], 20. fejezet).

11.13. Tanácsok

[1] Operátorokat elsősorban azért adjunk meg, hogy a szokásos használati módot támogassuk. §11.1.
[2] Nagy operandusok esetében használjunk *const* referencia paramétereket. §11.6.
[3] Nagy értékeket adó eredményeknél fontoljuk meg az eredmény-visszaadás optimalizálását. 11.6.
[4] Hagyatkozzunk az alapértelmezett másoló műveletekre, ha azok megfelelőek az osztályunk számára. 11.3.4.
[5] Bíráljuk felül vagy tiltsuk meg az alapértelmezett másolást, ha az egy adott típus számára nem megfelelő. 11.2.2.
[6] Ha egy függvénynek szüksége van az adatábrázolás elérésére, inkább tagfüggvény legyen. 11.5.2.
[7] Ha egy függvénynek nincs szüksége az adatábrázolás elérésére, inkább ne legyen tagfüggvény. 11.5.2.
[8] Használjunk névteret, hogy a segédfüggvényeket osztályukhoz rendeljük. §11.2.4.
[9] Szimmetrikus műveletekre használjunk nem tag függvényeket. §11.3.2.
[10] Többdimenziós tömb indexelésére használjunk *()*-t. §11.9.
[11] Az egyetlen „méret” paraméterű konstruktorok legyenek *explicit*-ek. §11.7.1.

[12] Általános célra használjuk a szabványos *string*-et (20. fejezet), ne a gyakorlatok megoldása révén kapott saját változatot. §11.12.
[13] Legyünk óvatosak az automatikus konverziók bevezetésekor. §11.4.
[14] Bal oldali operandusként balértéket váró műveleteket tagfüggvényekkel valósítsunk meg. §11.3.5.

11.14. Gyakorlatok

1. (*2) Milyen konverziókat használunk az alábbi program egyes kifejezéseiben:

```
struct X {
        int i;
        X X(int);
        operator+(int);
};

struct Y {
        int i;
        Y(X);
        Y operator+(X);
        operator int();
};

extern X operator*(X, Y);
extern int f(X);

X x = 1;
Y y = x;
int i = 2;

int main()
{
        i + 10;      y + 10;      y + 10 * y;
        x + y + i;   x * x + i;   f(7);
        f(y);        y + y;       106 + y;
}
```

   Módosítsuk a programot úgy, hogy futás közben kiírjon minden megengedett kifejezést.
2. (*2) Fejezzük be és teszteljük a §11.12-beli *String* osztályt.

3. (*2) Definiáljuk az *INT* osztályt, amely pontosan úgy viselkedik, mint egy *int*. (Segítség: definiáljuk az *INT::operator int()*-et).
4. (*1) Definiáljuk a *RINT* osztályt, amely pontosan úgy viselkedik, mint egy *int*, azzal a különbséggel, hogy csak a következő műveletek engedélyezettek: (egy és két paraméterű) +, (egy és két paraméterű) -, *, / és %. (Segítség: ne definiáljuk a *RINT::operator int()*-et).
5. (*3) Definiáljuk a *LINT* osztályt, amely pontosan úgy viselkedik, mint a *RINT*, de legalább 64 bites pontosságú.
6. (*4) Definiáljunk egy tetszőleges pontosságú aritmetikát támogató osztályt. Teszteljük az *1000* faktoriálisának kiszámíttatásával. (Segítség: a *String* osztályéhoz hasonló társzervezésre lesz szükség.)
7. (*2) Határozzunk meg a *String* osztály számára egy külső bejárót (iterátort):

```
class String_iter {
        // hivatkozás karakterláncra és annak elemére
public:
        String_iter(String& s);             // bejáró s számára
        char& next();                       // hivatkozás a következő elemre

        // igény szerint további műveletek
};
```

   Hasonlítsuk ezt össze hasznosság, stílus és hatékonyság szempontjából a *String* egy belső bejárójával. (Vagyis adott egy kurrens elemünk a *String*-ben, és a vele kapcsolatos műveletek.)
8. (*1.5) A *()* operátor túlterhelésével határozzunk meg egy részlánc-műveletet egy karakterlánc-osztály számára. Milyen más műveletet szeretnénk egy karakterláncon végezni?
9. (*3) Tervezzük meg úgy a *String* osztályt, hogy a részlánc-operátort egy értékadás bal oldalán is fel lehessen használni. Először egy olyan változatot írjunk, amelyiknél egy részláncot egy azonos hosszúságú teljes karakterláncra lehet cserélni, aztán egy olyat, amelyiknél eltérőre.
10. (*2) Definiáljuk a *String* osztály számára az értéket C stílusú adatként visszaadó műveletet. Vitassuk meg annak előnyeit és hátrányait, hogy ez egy konverziós operátor. Vitassuk meg a C stílusú adat számára szükséges memória lefoglalásának különféle módjait.
11. (*2.5) Tervezzünk és készítsünk egy egyszerű reguláris kifejezés illesztő eszközt a *String* osztály számára.
12. (*1.5) Módosítsuk úgy a §11.14[11]-beli eszközt, hogy az működjön a standard könyvtárbeli *string*-gel. A *string* definícióját nem változtathatjuk meg.

13. (*2) Írjunk egy programot, amit operátor-túlterheléssel és a makrók használatával olvashatatlanná teszünk. Egy ötlet: határozzuk meg + -t úgy az *INT*-ekre, hogy - -t jelentsen és fordítva. Ezután egy makróval határozzuk meg úgy az *int*-et, hogy *INT*-et jelentsen. Bíráljuk felül a „népszerű” függvényeket referencia típusú paramétereket használó függvényekként. Néhány félrevezető megjegyzéssel is nagy zavart lehet kelteni.
14. (*3) Cseréljük ki egy barátunkkal a §11.14[13] feladatra adott megoldásunkat, és próbáljuk meg futtatása nélkül kideríteni, mit csinál. A gyakorlat végére meg fogjuk tanulni, hogy mit ne tegyünk többé.
15. (*2) Definiáljuk a *Vec4* típust, mint négy *float*-ból álló vektort. Definiáljuk a *[ ]* műveletet. Adjuk meg a +, -, *, /, =, +, +=, -=, *= és /= műveleteket a vektorok és *float*-ok együttes használatára.
16. (*3) Definiáljuk a *Mat4* típust, mint négy *Vec4*-ből álló vektort. Definiáljuk a *[ ]* műveletet, mint ami *Vec4*-et ad vissza, ha a *Mat4*-re alkalmazzuk. Adjuk meg a szokásos mátrix-műveleteket. Készítsünk függvényt, amely a Gauss-féle kiküszöbölés módszerét alkalmazza egy *Mat4*-re.
17. (*2) Definiáljunk egy *Vector* típust, amely a *Vec4*-hez hasonlít, de *Vector::Vector(int)* konstruktorának paraméterként megadhatjuk a méretet.
18. (*3) Definiáljunk egy *Matrix* típust, amely a *Mat4*-hez hasonlít, de *Matrix::Matrix(int,int)* konstruktorának paraméterként megadhatjuk a dimenziókat.
19. (*2) Fejezzük be a §11.11 pont *Ptr_to_T* osztályát és ellenőrizzük. Legyenek meg legalább a *, ->, =, ++ és -- műveletek. Ne okozzunk futási idejű hibát, csak ha egy hibás mutatót ténylegesen felhasználnak.
20. (*1) Adott két struktúra:

```
struct S { int x, y; };
struct T { char* p; char* q; };
```

Írjunk egy C osztályt, melynek segítségével majdnem úgy használhatjuk valamely *S* és *T* *x*-ét és *p*-jét, mint ha *x* és *p* C tagja lenne.
21. (*1.5) Definiáljuk az *Index* osztályt a *mypow(double,Index)* hatványozó függvény kitevője számára. Találjunk módot arra, hogy *2**I* meghívja *mypow(2,I)*-t.
22. (*2) Definiáljuk az *Imaginary* osztályt képzetes számok ábrázolására. Definiáljuk a *Complex* osztályt ennek alapján. Készítsük el az alapvető aritmetikai műveleteket.

12

Származtatott osztályok

„Ne szaporítsuk az
objektumokat,
ha nem szükséges."
(W. Occam)

Fogalmak és osztályok • Származtatott osztályok • Tagfüggvények • Létrehozás és megsemmisítés • Osztályhierarchiák • Típusmezők • Virtuális függvények • Absztrakt osztályok • Hagyományos osztályhierarchiák • Absztrakt osztályok mint felületek • Az objektumok létrehozásának adott helyre korlátozása • Absztrakt osztályok és osztályhierarchiák • Tanácsok • Gyakorlatok

12.1. Bevezetés

A C++ a Simula nyelvtől kölcsönözte az osztály, mint felhasználói típus, illetve az osztályhierarchia fogalmát, valamint a rendszertervezés azon elvét, hogy a programban használt fogalmak modellezésére osztályokat használjon. A C++ nyújtotta nyelvi szerkezetek közvetlenül támogatják ezeket a tervezési elveket. Megfordítva is igaz: akkor használjuk a C++ nyelvet hatékonyan, ha a nyelvi lehetőségeket a tervezési elvek támogatására használjuk. Aki a nyelvi elemeket csak a hagyományosabb programozás jelölésbeli alátámasztására használja, a C++ valódi erősségének használatáról mond le.

Egy fogalom sohasem önmagában létezik, hanem más fogalmakkal együtt és erejének egy részét is a rokon fogalmakkal való kapcsolatból meríti. Próbáljuk csak megmagyarázni, mi az, hogy autó. Hamarosan bevezetjük a következő fogalmakat: kerék, motor, vezető, gyalogos, teherautó, mentők, utak, olaj, gyorshajtás, bírság, motel stb. Minthogy a fogalmakat osztályokként ábrázoljuk, felmerül a kérdés: hogyan ábrázoljuk a fogalmak közötti kapcsolatokat? Persze tetszőleges kapcsolatot egy programozási nyelvben nem tudunk közvetlenül kifejezni. De ha tudnánk, akkor sem akarnánk, hiszen osztályainkat a mindennapi életben használt fogalmaknál szűkebben és precízebben akarjuk meghatározni. A származtatott osztály fogalma és a vele kapcsolatos nyelvi eljárások célja a viszonyok kifejezése, azaz hogy kifejezzék, mi a közös az osztályokban. A kör és a háromszög fogalmában például közös, hogy mindkettő síkgörbe-alakzat, így a *Circle* (Kör) és *Triangle* (Háromszög) osztályokat úgy írhatjuk le, hogy pontosan meghatározott (explicit) módon megmondjuk, hogy a *Shape* (Alakzat) osztály a közös bennük. Ha egy programban úgy szerepeltetünk köröket és háromszögeket, hogy nem vonjuk be a síkidom fogalmát, akkor valami lényegeset mulasztunk el. Ez a fejezet azt feszegeti, mi következik ebből az egyszerű elvből – ami valójában az általában objektumorientáltnak nevezett programozási elv alapja.

A nyelvi lehetőségek és módszerek bemutatása az egyszerűtől és konkréttól a bonyolultabb, kifinomultabb, elvontabb felé halad. Sokak számára ez a megszokottól a kevésbé ismert felé való haladást is fogja jelenti. De ez nem csupán egyszerű utazás a „régi, rossz módszerektől" az „egyedüli igaz út" felé. Amikor rámutatok egy megközelítés korlátaira, hogy a programozót az új felé tereljem, mindig adott problémák kapcsán teszem; más problémák kapcsán vagy más összefüggésben lehetséges, hogy a korábbi módszer alkalmazása a jobb választás. Használható programokat az itt tárgyalt módszerek mindegyikének felhasználásával írtak már. A cél az, hogy az olvasó megismerje az összes eljárást, hogy aztán okos és kiegyensúlyozott módon tudjon majd választani közülük, amikor igazi feladatokat kell megoldania.

A fejezetben először az objektumorientált programozást támogató alapvető nyelvi eszközöket mutatom be, majd egy hosszabb példa kapcsán azt tárgyalom, hogyan lehet ezek alkalmazásával jól szerkesztett programot írni. Az objektumorientált programozást támogató további nyelvi eszközöket, például a többszörös öröklődést vagy a futási idejű típusazonosítást a 15. fejezet tárgyalja.

12.2. Származtatott osztályok

Vegyünk egy programot, amely egy cég dolgozóit kezeli. Egy efféle programban lehet egy ilyen adatszerkezet:

```
struct Employee {                        // alkalmazott
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};
```

Ezután „meghatározhatunk" egy főnököt is:

```
struct Manager {                         // főnök
    Employee emp;                        // a főnök mint alkalmazott
    set<Employee*> group;                // beosztottak
    short level;
    // ...
};
```

A főnök egyben alkalmazott is, ezért az *Employee* (Alkalmazott) adatokat a *Manager* (Főnök, vezető) objektum *emp* tagjában tároljuk. Ez nyilvánvaló lehet a programozó (különösen egy figyelmes programozó) számára, de a fordítóprogram és az egyéb eszközök sehonnan nem fogják tudni, hogy a *Manager* egyben *Employee* is. Egy *Manager** nem *Employee** is egyben, így a kettő nem cserélhető fel. Egy *Employee*-ket tartalmazó listára nem vehetünk fel egy *Manager*-t a megfelelő kód megírása nélkül. Vagy típuskényszerítést kellene alkalmaznunk a *Manager**-ra, vagy az *emp* tag címét kellene az alkalmazottak listájára tennünk. Egyik megoldás sem elegáns és zavaró is lehet. A helyes megközelítés az, hogy kifejezetten megmondjuk, a *Manager*-ek egyben *Employee*-k is, csak további adatokat is tartalmaznak:

```
struct Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
};
```

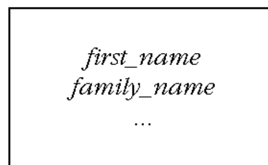
A *Manager* az *Employee*-ból származik, és fordítva, az *Employee* a *Manager* bázisosztálya. A *Manager* osztálynak megvannak azok a tagjai, amelyek az *Employee*-nek is (*first_name*, *department* stb.) és ezekhez jönnek hozzá a saját tagok (*group*, *level* stb.).

A származtatást gyakran úgy ábrázolják grafikusan, hogy a származtatott osztályból egy nyilat rajzolnak a bázisosztály felé, jelezve, hogy a származtatott osztály a bázisosztályra hivatkozik (és nem fordítva):

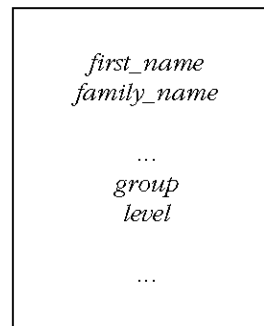
Általában úgy mondják, a származtatott osztály tulajdonságokat örököl a bázisosztálytól. Ennek alapján ezt a kapcsolatot *öröklődésnek* (öröklés, inheritance) is hívják. Az angol kifejezések a *bázisposztályra* és a *származtatott osztályra*: base class (vagy superclass), illetve derived class (subclass). Az utóbbi szóhasználat (superclass – főosztály, subclass – alosztály) azonban zavaró lehet, hiszen a származtatott osztály bizonyos értelemben „szélesebb" a bázisosztálynál, mivel annál több adatot tárol és több függvényt biztosít.

A származtatás népszerű és hatékony megvalósítása a származtatott osztályú objektumot a bázisosztály olyan objektumaként ábrázolja, amely kiegészül a származtatott osztályra egyedileg jellemző adatokkal is:

Emplyee:



Manager:



A *Manager* osztálynak az *Employee*-ból ilyen módon való származtatása a *Manager* típust az *Employee* altípusává teszi, így tehát mindenhol, ahol *Employee* objektum használható, egy *Manager* is megfelel. Most már készíthetünk egy listát az alkalmazottakról (*Employee*), akiknek egy része vezető beosztású (*Manager*):

```
void f(Manager m1, Employee e1)
{
    list<Employee*> elist;

    elist.push_front(&m1);
    elist.push_front(&e1);
    // ...
}
```

Ne feledjük, egy *Manager* egyben *Employee* is, így egy *Manager**-ot használhatunk *Employee**-ként is. Az *Employee* viszont nem feltétlenül *Manager*, így *Employee**-ot nem használhatunk *Manager**-ként. Általában, ha egy *Derived* (származtatott) osztálynak egy *Base* (bázis) osztály nyilvános bázisosztálya (§15.3), akkor egy *Base** változó típuskényszerítés nélkül kaphat *Derived** típusú értéket. A másik irányban (*Base**-ról *Derived**-ra) explicit konverzió szükséges:

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;              // rendben: minden Manager egyben Employee is
    Manager* pm = &ee;               // hiba: nem minden Employee főnök

    pm->level = 2;                   // katasztrófa: ee nem rendelkezik 'level' taggal

    pm = static_cast<Manager*>(pe);  // "nyers erővel": működik, mert pe
                                     // a Manager típusú mm-re mutat

    pm->level = 2;                   // ez is jó: pm a Manager típusú mm-re mutat,
                                     // amelynek van 'level' tagja
}
```

Vagyis mutatók és referenciák használatakor a származtatott osztály objektumait úgy kezelhetjük, mint a bázisosztály objektumait. A másik irányban ez nem áll. A *static_cast* és a *dynamic_cast* használatát a §15.4.2 pont írja le.

Egy osztály bázisosztályként való használata egyenértékű az osztály egy (névtelen) objektumának deklarálásával. Következésképpen egy osztályt csak akkor használhatunk bázisosztályként, ha definiáltuk (§5.7):

```
class Employee;                      // csak deklaráció, nem definíció

class Manager : public Employee {    // hiba: Employee nem definiált
     // ...
};
```

12.2.1. Tagfüggvények

Az egyszerű adatszerkezetek – mint a *Manager* és az *Employee* – nem túl érdekesek és sokszor nem is különösebben hasznosak. Az információt megfelelő típusként kell megadnunk, melyhez az elvégezhető műveletek is hozzátartoznak, és ezt úgy kell megtennünk, hogy közben nem kötődünk az adott ábrázoláshoz:

```
class Employee {
     string first_name, family_name;
     char middle_initial;
     // ...
public:
     void print() const;
     string full_name() const
          { return first_name + ' ' + middle_initial + ' ' + family_name; }
     // ...
};

class Manager : public Employee {
     // ...
public:
     void print() const;
     // ...
};
```

A származtatott osztályok tagfüggvényei ugyanúgy elérhetik a bázisosztály nyilvános (public) – és védett (protected, §15.3) – tagjait, mintha maguk vezették volna be azokat:

```
void Manager::print() const
{
     cout << "A keresett név " << full_name() << '\n';
     // ...
}
```

A származtatott osztály azonban nem éri el a bázisosztály privát (private) tagjait:

```
void Manager::print() const
{
     cout << "A keresett név " << family_name << '\n';     // hiba!
     // ...
}
```

A *Manager::print()* második változatát a fordító nem fogja lefordítani. A származtatott osztálynak nincs különleges engedélye a bázisosztály privát tagjainak elérésére, így a *Manager::print()* számára a *family_name* nem érhető el.

Némelyek meglepődnek ezen, de gondoljuk el, mi lenne fordított esetben: ha a származtatott osztály tagfüggvénye elérhetné a bázisosztály privát tagjait. A privát tag fogalma értelmetlenné válna azáltal, hogy a programozó hozzáférhetne az osztály privát részéhez egy osztályt származtatva belőle. Továbbá nem lehetne többé egy privát tag használatát a tag- és barát (friend) függvények átnézésével megkeresni. Az egész program minden forrásállományát át kéne nézni: a származtatott osztályokat és azok függvényeit keresni, majd a származtatott osztályokból származtatott további osztályokat és azok függvényeit és így tovább. Ez legjobb esetben is fárasztó és sokszor kivitelezhetetlen is. Ott, ahol ez elfogadható, inkább védett (protected) és ne privát (private) tagokat használjunk. Egy védett tag a származtatott osztályok számára olyan, mint egy nyilvános (public), a többiek számára azonban privátnak minősül (§15.3).

A legtisztább megoldás általában az, ha a származtatott osztály a bázisosztálynak csak a nyilvános tagjait használja:

```
void Manager::print() const
{
    Employee::print();      // alkalmazottak adatainak kiírása

    cout << level;          // a főnökökre vonatkozó adatok kiírása
    // ...
}
```

Vegyük észre, hogy a *::* hatókör operátort kellett használni, mert a *print()* függvényt a *Manager* osztály újradefiniálja. A függvénynevek ilyen módon való újrafelhasználása igen általános. Ha óvatlanul ilyet írunk:

```
void Manager::print() const
{
    print();      // hoppá!

    // a főnökökre vonatkozó adatok kiírása
}
```

a program váratlan módon újra és újra meg fogja hívni önmagát.

12.2.2. Konstruktorok és destruktorok

Egyes származtatott osztályoknak konstruktorokra van szükségük. Ha a bázisosztálynak vannak ilyen függvényei, akkor az egyiket meg is kell hívni. Az alapértelmezett konstruktorokat automatikusan is meghívhatjuk, de ha a bázisosztály minden konstruktora paramétert igényel, akkor a megfelelő konstruktort csak explicit módon lehet meghívni. Vegyük a következő példát:

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& n, int d);
    // ...
};

class Manager : public Employee {
    set<Employee*> group;          // beosztottak
    short level;
    // ...
public:
    Manager(const string& n, int d, int lvl);
    // ...
};
```

A bázisosztály konstruktora a származtatott osztály konstruktorának definiciójában kap paramétereket. Ebből a szempontból a bázisosztály konstruktora úgy viselkedik, mintha a származtatott osztály tagja lenne (§10.4.6):

```
Employee::Employee(const string& n, int d)
    : family_name(n), department(d)          // tagok kezdeti értékadása
{
    // ...
}

Manager::Manager(const string& n, int d, int lvl)
    : Employee(n,d),                         // a bázisosztály kezdeti értékadása
      level(lvl)                             // tagok kezdeti értékadása
{
    // ...
}
```

Egy származtatott osztály konstruktora csak a saját tagjai és a bázisosztály konstruktora számára adhat meg kezdőértéket; a bázisosztály tagjainak nem:

```
Manager::Manager(const string& n, int d, int lvl)
    : family_name(n),     // hiba: family_name nem deklarált a Manager osztályban
      department(d),      // hiba: department nem deklarált a Manager osztályban
      level(lvl)
{
    // ...
}
```

A fenti definíció három hibát tartalmaz: nem hívja meg az *Employee* konstruktorát, és két ízben is megpróbál közvetlenül kezdőértéket adni az *Employee* tagjainak.

Az osztályba tartozó objektumok „alulról felfelé" épülnek fel; először a bázisosztály, aztán a tagok, végül maga a származtatott osztály. A megsemmisítés fordított sorrendben történik: először a származtatott osztály, aztán a tagok, végül a bázisosztály. A tagok a deklaráció sorrendjében jönnek létre és fordított sorrendben semmisülnek meg (§10.4.6 és §15.2.4.1).

12.2.3. Másolás

Egy osztályba tartozó objektum másolását a másoló konstruktor és az értékadások határozzák meg (§10.4.4.1):

```
class Employee {
    // ...
    Employee& operator=(const Employee&);
    Employee(const Employee&);
};

void f(const Manager& m)
{
    Employee e = m;      // e létrehozása m Employee részéből
    e = m;               // m Employee részének másolása e-be
}
```

Minthogy az *Employee* osztály másoló függvényei nem tudnak a *Manager* osztályról, a *Manager* objektumnak csak az *Employee* része fog lemásolódni. Az objektumnak ez a „felszeletelődése" (*slicing*), azaz a tény, hogy ekkor az objektumnak csak egy szelete másolódik le, meglepő lehet és hibákhoz vezethet. A felszeletelődés megakadályozása az egyik oka annak, hogy osztályhierarchiába tartozó objektumok esetében célszerűbb mutatókat és referenciákat használnunk. A hatékonysági megfontolások mellett további ok, hogy megőrizzük a többalakú (polimorfikus) viselkedést (§2.5.4 és §12.2.6).

Jegyezzük meg, hogy ha nem határozunk meg másoló értékadó operátort, akkor a fordítóprogram fog létrehozni egyet. Ebből következik, hogy az értékadó operátorok nem öröklődnek. (A konstruktorok soha.)

12.2.4. Osztályhierarchiák

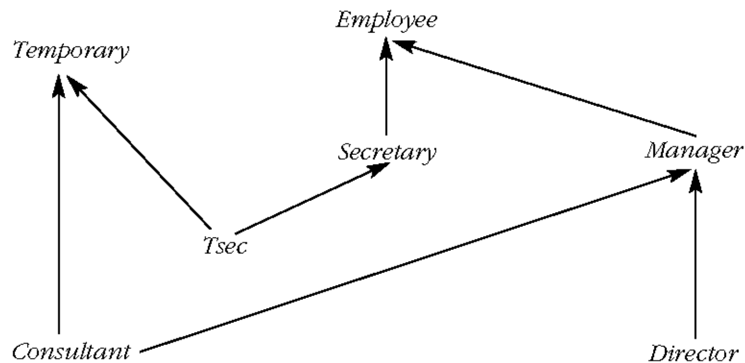
Egy származtatott osztály lehet maga is bázisosztály:

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

Az egymással kapcsolatban álló osztályok ilyen halmazát hagyományosan *osztályhierarchiának* hívjuk. A hierarchia leggyakrabban fa szokott lenni, de lehet ennél általánosabb gráf is.:

```
class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };
class Tsec : public Temporary, public Secretary { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

Ábrával:



Vagyis ahogy a §15.2 pont részletesen elmagyarázza, a C++ nyelv képes az osztályoknak egy irányított körmentes gráfját kifejezni.

12.2.5. Típusmezők

Ha a deklarációkban alkalmazott kényelmes rövidítésnél többre akarjuk használni az osztályok származtatását, meg kell válaszolnunk a következő kérdést: ha adott egy *Base** mutató, akkor milyen származtatott osztályba tartozik az objektum, amelyre mutat? A problémának négy alapvető megoldása van:

1. Érjük el, hogy csak egyféle objektum jöhessen szóba (§2.7, 13. fejezet).
2. Helyezzünk egy típusmezőt a bázisosztályba és a függvények ezt kérdezzék le.
3. Használjuk a *dynamic_cast*-ot (dinamikus típuskonverzió, §15.4.2, §15.4.5).
4. Használjunk virtuális függvényeket (§2.5.5, §12.2.6).

Bázisosztályra hivatkozó mutatókat gyakran használunk olyan *tároló osztályokban* (container class), mint a halmaz (*set*), a vektor (*vector*) és a lista (*list*). Ekkor az első megoldás homogén listákat, azaz azonos típusú objektumokat eredményez. A többi megoldás lehetővé tesz heterogén listákat is, azaz olyanokat, ahol különböző típusú objektumok (vagy ilyenekre hivatkozó mutatók) lehetnek. A 3. megoldás a 2. megoldásnak a nyelv által támogatott változata, a 4. pedig a 2.-nak egyedi, típusbiztos átalakítása. Az 1. és a 4. megoldás párosításai különösen érdekesek és erősek; majdnem mindig világosabb kódot eredményeznek, mint a 2. vagy a 3. megoldás.

Nézzünk meg először egy típusmezős megoldást, hogy lássuk, legtöbbször miért kerülendő. A „főnök – alkalmazott” példát így írhatnánk át:

```
struct Employee {
    enum Empl_type { M, E };
    Empl_type type;

    Employee() : type(E) { }

    string first_name, family_name;
    char middle_initial;

    Date hiring_date;
    short department;
    // ...
};
struct Manager : public Employee {
    Manager() { type = M; }

    set<Employee*> group;           // beosztottak
    short level;
    // ...
};
```

Most már írhatunk egy függvényt, amely minden *Employee*-ről ki tudja írni az információt:

```
void print_employee(const Employee* e)
{
    switch (e->type) {
    case Employee::E:
            cout << e->family_name << '\t' << e->department << '\n';
            // ...
            break;
    case Employee::M:
    {       cout << e->family_name << '\t' << e->department << '\n';
            // ...
            const Manager* p = static_cast<const Manager*>(e);
            cout << " szint " << p->level << '\n';
            // ...
            break;
    }
    }
}
```

Az alkalmazottak listája így írható ki:

```
void print_list(const list<Employee*>& elist)
{
    for (list<Employee*>::const_iterator p = elist.begin(); p!=elist.end(); ++p)
            print_employee(*p);
}
```

Ez jól működik, különösen az egyetlen programozó által fenntartott kis programokban. Alapvető gyengéje, hogy az a feltétele, hogy a programozó a megfelelő (és a fordítóprogram által nem ellenőrizhető módon) kell, hogy kezelje a típusokat. A problémát általában súlyosbítja, hogy az olyan függvények, mint a *print_employee*, a szóba jöhető osztályok közös vonásait használják ki:

```
void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    // ...
    if (e->type == Employee::M) {
            const Manager* p = static_cast<const Manager*>(e);
            cout << " szint " << p->level << '\n';
            // ...
    }
}
```

Egy nagy függvényben, ahol sok származtatott típust kell kezelni, nehéz lehet az összes ilyen típusmező-ellenőrzést megtalálni. De ha megtaláltuk is, nehéz lehet megérteni, hogy mi is történik. Továbbá, ha a rendszer új *Employee*-vel bővül, az összes fontos függvényt módosítani kell – vagyis az összes olyat, amelyik ellenőrzi a típusmezőt. Minden olyan függvényt meg kell vizsgálni, amelyiknek egy változtatás után szüksége lehet a típusmező ellenőrzésére. Ez azt jelenti, hogy hozzá kell férni a kritikus forráskódhoz és külön munkát jelent a változtatás utáni teszt is. A típuskényszerítés árulkodó jele annak, hogy javítani lehetne a kódon.

Vagyis a típusmezős megoldás hibákra ad lehetőséget és nehezen módosítható kódot eredményez. Ahogy a rendszer mérete nő, a probléma súlyosbodik, mert a típusmező alkalmazása ellentmond a modularitás és az adatrejtés elvének. Minden típusmezőt használó függvénynek ismernie kell az összes olyan osztály ábrázolását (és megvalósításuk egyéb részleteit), amely a típusmezős osztály leszármazottja.

Ezenkívül ha van egy olyan adat (például egy típusmező), amely minden származtatott osztályból elérhető, akkor a programozó hajlamos arra, hogy további ilyen mezőket hozzon létre. A közös bázisosztály így mindenféle „hasznos információk" gyűjteményévé válik. Ez aztán a bázisosztály és a származtatott osztályok megvalósításának legkevésbé kívánatos összefonódásához vezet. A tisztább felépítés és könnyebb módosíthatóság kedvéért a különálló dolgokat kezeljük külön, a kölcsönös függéseket pedig kerüljük el.

12.2.6. Virtuális függvények

A virtuális függvények azáltal kerülik el a típusmezős megoldás problémáit, hogy segítségükkel a programozó olyan függvényeket deklarálhat a bázisosztályban, amelyeket a származtatott osztályok felülbírálhatnak. A fordító- és betöltőprogram gondoskodik az objektumtípusok és az alkalmazott függvények összhangjáról:

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& name, int dept);
    virtual void print() const;
    // ...
};
```

A *virtual* kulcsszó azt jelenti, hogy a *print()* függvény felületként szolgál az ebben az osztályban definiált *print()* függvényhez, illetve a származtatott osztályokban definiált *print()* függvényekhez. Ha a származtatott osztályokban szerepelnek ilyenek, a fordítóprogram mindig az adott *Employee* objektumnak megfelelő függvényt fogja meghívni.

Egy virtuális függvény akkor szolgálhat felületként a származtatott osztályokban definiált függvényekhez, ha azoknak ugyanolyan típusú paramétereik vannak, mint a bázisosztálybelinek, és a visszatérési érték is csak nagyon csekély mértékben különbözik (§15.6.2). A virtuális tagfüggvényeket néha *metódusoknak* (method) is hívják.

A virtuális függvényt mindig definiálnunk kell abban az osztályban, amelyben először deklaráltuk, hacsak nem *tisztán virtuális* (pure virtual) függvényként adtuk meg (§12.3):

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    // ...
}
```

Virtuális függvényt akkor is használhatunk, ha osztályából nem is származtatunk további osztályt; ha pedig származtatunk, annak a függvényből nem kell feltétlenül saját változat. Osztály származtatásakor csak akkor írjunk egy megfelelő változatot a függvényből, ha valóban szükséges:

```
class Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    // ...
};

void Manager::print() const
{
    Employee::print();
    cout << "\tszint " << level << '\n';
    // ...
}
```

A származtatott osztály azonos nevű és azonos típusú paraméterekkel bíró függvénye *felülírja* vagy *felülbírálja* (override) a virtuális függvény bázisosztálybeli változatát. Hacsak közvetlen módon meg nem mondjuk, hogy a virtuális függvény melyik változatát akarjuk használni – mint az *Employee::print()* hívásnál –, az objektumhoz leginkább illő felülíró függvény lesz meghívva.

A globális *print_employee()* függvény (§12.2.5) szükségtelen, mert a helyébe a *print()* tagfüggvények léptek. Az alkalmazottak (*Employee*) listáját így írathatjuk ki:

```
void print_list(const list<Employee*>& s)
{
    for (list<Employee*>::const_iterator p = s.begin(); p!=s.end(); ++p)       // lásd §2.7.2
          (*p)->print();
}
```

De akár így is:

```
void print_list(const list<Employee*>& s)
{
    for_each(s.begin(),s.end(),mem_fun(&Employee::print));                    // lásd §3.8.5
}
```

Minden *Employee* a típusának megfelelően íródik ki. A

```
int main()
{
    Employee e("Brown",1234);
    Manager m("Smith",1234,2);
    list<Employee*> empl;
    empl.push_front(&e);              // §2.5.4
    empl.push_front(&m);
    print_list(empl);
}
```

például az alábbi kimenetet eredményezi:

```
Smith 1234
    szint 2
Brown 1234
```

Ez akkor is működik, ha a *print_list()* függvényt azelőtt írtuk meg és fordítottuk le, mielőtt a *Manager* osztályt egyáltalán kitaláltuk volna! Ez az osztályoknak egy kulcsfontosságú tulajdonsága. Ha helyesen alkalmazzuk, az objektumorientált tervezés sarokköve lesz és a programok fejlesztésénél bizonyos fokú stabilitást ad.

Azt, hogy az *Employee* függvényei attól függetlenül „helyesen" viselkednek, hogy pontosan milyen fajta *Employee*-re hívtuk meg azokat, ***többalakúságnak*** (polimorfizmus, polymorphism) nevezzük. A virtuális függvényekkel bíró típus neve ***többalakú típus*** (polimorfikus típus). A C++ nyelvben a többalakú viselkedést virtuális függvények használatával vagy az objektumoknak mutatókon vagy referenciákon át való kezelésével érhetjük el. Ha közvetlenül kezelünk egy objektumot és nem mutató vagy referencia segítségével, a fordítóprogram felismeri annak pontos típusát, így a futási idejű többalakúságra nincs szükség.

Világos, hogy a többalakúság támogatása érdekében a fordítóprogramnak minden *Employee* típusú objektumban valamilyen, a típusra vonatkozó információt (***típusinformációt***) kell nyilvántartania, melynek segítségével képes a megfelelő *print()* függvényt meghívni. Ehhez rendszerint egyetlen mutatónyi hely is elég, és erre is csak azon osztályokban van szükség, amelyeknek van virtuális függvényük; tehát nem minden osztályban és még csak nem is minden származtatott osztályban. A típusmezős megoldás választása esetén ehhez képest jelentős mennyiségű tárterületet kellett volna a típusmező számára biztosítanunk.

Ha egy függvényt (miként a *Manager::print()*-et is) a *::* hatókör-feloldó operátor segítségével hívunk meg, akkor ezáltal kikapcsoljuk a virtualitást. Máskülönben a *Manager::print()* végtelen rekurziót idézne elő. A minősített név használatának van még egy előnye: ha egy virtuális függvény *inline* (ami elő szokott fordulni), akkor a fordítóprogram a *::* minősítővel jelzett hívásokat képes helyben kifejteni. Ennek segítségével a programozó hatékonyan képes azokat az egyedi eseteket kezelni, amikor mondjuk egy virtuális függvény ugyanarra az objektumra egy másik függvényt is meghív. A *Manager::print()* függvény ennek példája. Minthogy a *Manager::print()* meghívásakor meghatározzuk az objektum típusát, az *Employee::print()* ezt követő meghívásakor a típusról már nem kell újra dönteni.

Érdemes megemlíteni, hogy a virtuális függvényhívás hagyományos és nyilvánvaló megvalósítása az egyszerű közvetett függvényhívás (indirekció) (§2.5.5), így a hatékonyság elvesztésétől való félelem ne riasszon vissza senkit a virtuális függvények használatától ott, ahol egy közönséges függvényhívás elfogadhatóan hatékony.

12.3. Absztrakt osztályok

Sok osztály hasonlít az *Employee* osztályra annyiban, hogy önmagában és származtatott osztályok bázisosztályaként is hasznos. Az ilyen osztályok számára elegendőek az előző pontban bemutatott módszerek. De nem minden osztály ilyen. Bizonyos osztályok, például a *Shape* (Alakzat), olyan elvont fogalmakat jelenítenek meg, amelyekhez nem létezhetnek objektumok. A *Shape*-nek csak mint bázisosztálynak van értelme. Ez abból is látható, hogy nem tudunk hozzá virtuális függvényeket értelmesen definiálni:

```
class Shape {
public:
    virtual void rotate(int) { error("Shape::rotate"); }        // nem "elegáns"
    virtual void draw() { error("Shape::draw"); }
    // ...
};
```

Egy ilyen meghatározatlan alakzatot meg tudunk ugyan adni (a nyelv megengedi), de nem sok értelme van:

```
Shape s;    // butaság: "alak nélküli alakzat"
```

A dolog azért értelmetlen, mert az *s* minden művelete hibát fog eredményezni.

Jobb megoldás, ha a *Shape* osztály virtuális függvényeit *tisztán virtuális* (pure virtual) függvényként deklaráljuk. A virtuális függvények az *=0* „kezdeti értékadástól" lesznek tisztán virtuálisak:

```
class Shape {                       // absztrakt osztály
public:
    virtual void rotate(int) = 0;       // tisztán virtuális függvény
    virtual void draw() = 0;            // tisztán virtuális függvény
    virtual bool is_closed() = 0;       // tisztán virtuális függvény
    // ...
};
```

Ha egy osztály legalább egy tisztán virtuális függvénnyel rendelkezik, akkor *absztrakt osztálynak* (elvont osztály, abstract class) hívjuk, ilyen osztályba tartozó objektumot pedig nem hozhatunk létre:

```
Shape s;    // hiba: s az absztrakt Shape osztály változója lenne
```

Az absztrakt osztályokat csak felületként (interface), illetve más osztályok bázisosztályaként használhatjuk:

```
class Point { /* ... */ };

class Circle : public Shape {
public:
    void rotate(int) { }                  // a Shape::rotate felülírása
    void draw();                          // a Shape::draw felülírása
    bool is_closed() { return true; }     // a Shape::is_closed felülírása

    Circle(Point p, int r);
private:
    Point center;
    int radius;
};
```

Ha egy tisztán virtuális függvényt a származtatott osztályban nem definiálunk, akkor az tisztán virtuális függvény marad, sőt, a származtatott osztály is absztrakt osztály lesz. Ez a megvalósítás lépcsőzetes felépítését teszi lehetővé:

```
class Polygon : public Shape {            // absztrakt osztály
public:
    bool is_closed() { return true; }     // a Shape::is_closed felülírása
    // ... a draw és a rotate nincs felülírva ...
};

Polygon b; // hiba: a Polygon osztálynak nem lehet objektuma

class Irregular_polygon : public Polygon {
    list<Point> lp;
public:
    void draw();                          // a Shape::draw felülírása
    void rotate(int);                     // a Shape::rotate felülírása
    // ...
};

Irregular_polygon poly(some_points);      // jó (megfelelő konstrukort feltételezve)
```

Az absztrakt osztályok fontos képessége, hogy segítségükkel a megvalósítás egyéb részeinek elérhetővé tétele nélkül biztosíthatunk felületet. Egy operációs rendszer például egy absztrakt osztály mögé rejtheti eszközmeghajtóinak tulajdonságait:

```
class Character_device {
public:
    virtual int open(int opt) = 0;
    virtual int close(int opt) = 0;
```

```
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
    virtual int ioctl(int ...) = 0;
    virtual ~Character_device() { }       // virtuális destruktor
};
```

Az egyes eszközmeghajtókat a *Character_device*-ból származtatott osztályként definiálhatjuk, és sokféle eszközmeghajtót kezelhetünk ezen felületen keresztül. A virtuális destruktorok fontosságát a §12.4.2 pont magyarázza el.

Az absztrakt osztályok bevezetésével immár minden eszköz a kezünkben van arra, hogy moduláris módon, építőkövekként osztályokat használva egy teljes programot írjunk.

12.4. Osztályhierarchiák tervezése

Vegyük a következő egyszerű tervezési problémát: hogyan lehet egy program számára lehetővé tenni egy egész érték bekérését a felhasználói felületről? Zavarbaejtően sokféle módon. Ahhoz, hogy elszigeteljük programunkat ettől a sokféleségtől és felderíthessük a különböző tervezési módokat, kezdjük a munkát ezen egyszerű adatbeviteli művelet modelljének felállításával. A tényleges felhasználói felület elkészítésének részleteit későbbre halasztjuk.

Az alapötlet az, hogy lesz egy *Ival_box* (értékmező) osztályunk, amely tudja, hogy milyen értékeket fogadhat el. A program elkérheti egy *Ival_box* objektum értékét és felszólíthatja arra is, hogy kérje be ezt az értéket a felhasználótól, ha még nem áll rendelkezésre. Azt is megkérdezheti, hogy az érték megváltozott-e a legutóbbi kérés óta.

Minthogy ez az alapötlet sokféleképpen megvalósítható, abból kell kiindulnunk, hogy sokféle különböző *Ival_box* lesz: csúszkák, szöveges adatbeviteli mezők, ahová a felhasználó beírhatja az értéket, számtárcsák, hanggal vezérelhető eszközök.

Azt az általános megközelítést alkalmazzuk, hogy egy virtuális felhasználói felületet" bocsátunk az alkalmazás rendelkezésére, amely a létező felhasználói felületek szolgáltatásainak egy részét biztosítja. E felület számos rendszeren elkészíthető, így kódja „hordozható" lesz. Természetesen vannak más módok is arra, hogy egy alkalmazást elválasszunk a felhasználói felülettől. Azért választottam ezt, mert általános, mert a kapcsán egy sor eljárást és tervezési szempontot lehet bemutatni, mert ezeket a módszereket alkalmazzák a valódi felhasz-

nálói felületeteket kezelő rendszerekben, és végül – a leglényegesebb ok –, mert ezek a módszerek a felhasználói felületetek szűk tartományánál jóval szélesebb körben is alkalmazhatók.

12.4.1. Hagyományos osztályhierarchia

Első megoldásunk egy hagyományos osztályhierarchia; ilyennel a Simula, Smalltalk és régebbi C++-programokban találkozhatunk.

Az *Ival_box* osztály az összes *Ival_box* által használatos felületet írja le és egy olyan alapértelmezett megvalósítást ad, melyet az egyes *Ival_box*-ok sajátjaikkal felülbírálhatnak. Ezenkívül megadjuk az alapmegoldáshoz szükséges adatokat is:

```
class Ival_box {
protected:
    int val;
    int low, high;
    bool changed;
public:
    Ival_box(int ll, int hh) { changed = false; val = low = ll; high = hh; }

    virtual int get_value() { changed = false; return val; }
    virtual void set_value(int i) { changed = true; val = i; }          // felhasználók számára
    virtual void reset_value(int i) { changed = false; val = i; }       // alkalmazások számára
    virtual void prompt() { }
    virtual bool was_changed() const { return changed; }
};
```

A függvények alapértelmezett változatai meglehetősen vázlatosak, „pongyolák"; céljuk leginkább az, hogy illusztrálják a megközelítést. (Egy „valódi" osztály például értékellenőrzést is végezne.)

Az „*ival* osztályokat" egy programozó így használhatná fel:

```
void interact(Ival_box* pb)
{
    pb->prompt();                    // jelzés a felhasználónak
    // ...
    int i = pb->get_value();
    if (pb->was_changed()) {
                // új érték; valamit csinálunk vele
    }
```

```
    else {
        // a régi érték jó volt; ezt is felhasználjuk valahogy
    }
    // ...
}

void some_fct()
{
    Ival_box* p1 = new Ival_slider(0,5);    // az Ival_slider az Ival_box osztályból származik
    interact(p1);

    Ival_box* p2 = new Ival_dial(1,12);
    interact(p2);
}
```

A programkód legnagyobb része az *interact()* függvény stílusában íródna, és egyszerű *Ival_box*-okat, illetve azokra hivatkozó mutatókat használna. Így a programnak nem kellene tudnia az esetleg nagy számú különböző *Ival_box*-változatokról, csak annak a viszonylag kis számú függvénynek kellene ismernie azokat, amelyek ilyen objektumokat létrehoznak. Ez a felhasználókat elszigeteli a származtatott osztályok esetleges módosításaitól. A kód legnagyobb részének még arról sem kell tudnia, hogy egyáltalán különböző *Ival_box*-ok léteznek.

Hogy egyszerűsítsem a tárgyalást, eltekintek attól a kérdéstől, hogyan vár a program bemenetre. Lehetséges megoldás, hogy a program a *get_value()* függvényben ténylegesen vár a felhasználói bemenetre, megoldható úgy is, hogy az *Ival_box*-ot egy eseményhez kapcsoljuk és egy visszahívás (callback) segítségével válaszolunk, esetleg a programmal külön végrehajtási szálat indíttatunk el az *Ival_box* számára, majd a szál állapotát kérdezzük le. Az ilyen döntések alapvető fontosságúak a felhasználói felületet kezelő rendszerek tervezésekor, de ha itt a valóságot akár csak megközelítő részletességgel tárgyalnánk ezeket, elvonnánk a figyelmet a programozási eljárások és nyelvi eszközök tárgyalásától. Az itt bemutatott tervezési módszerek és az azokat támogató nyelvi eszközök nem kötődnek adott felhasználói felülethez; jóval szélesebb körben is alkalmazhatók.

A különböző *Ival_box*-okat az *Ival_box*-ból származtatott osztályokként határozhatjuk meg:

```
class Ival_slider : public Ival_box {
    // a csúszka kinézetét, viselkedését meghatározó grafikai elemek
public:
    Ival_slider(int, int);

    int get_value();
    void prompt();
};
```

Az *Ival_box* adattagjait *védettként* (protected) vezettük be, hogy a származtatott osztályok-

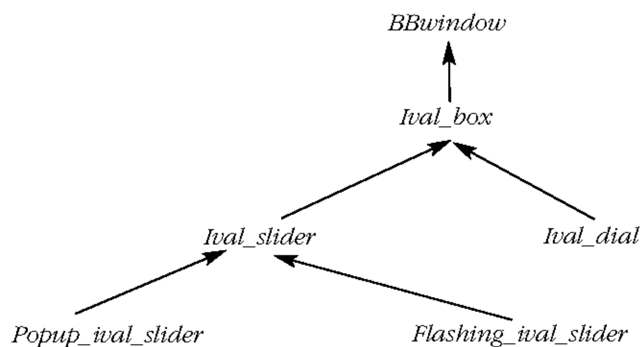
ból elérhetőek legyenek. Így aztán az *Ival_slider::get_value()* függvény elhelyezheti az értéket az *Ival_box::val* adattagban. A védett tagok elérhetők az osztály és a származtatott osztályok függvényei számára is, de az általános felhasználó számára nem (§15.3).

Az *Ival_box*-ból az *Ival_slider* mellett más változatokat is származtathatunk. Ezek között ott lehet az *Ival_dial*, amelynél egy gomb forgatásával adhatunk meg egy értéket, a *Flashing_ival_slider*, amely felvillan, ha a *prompt()* függvénnyel erre kérjük, és a *Popup_ival_slider*, amely a *prompt()* hatására valamilyen feltűnő helyen jelenik meg, a felhasználótól szinte kikövetelve egy érték megadását.

De vajon honnan vegyük a grafikus elemeket? A legtöbb felhasználói felületet kezelő rendszer biztosít egy osztályt, amely leírja a képernyőn levő objektumok alapvető tulajdonságait. Ha például a „Big Bucks Inc." („Sok Pénz Rt.") rendszerét használjuk, akkor az *Ival_slider*, az *Ival_dial* stb. osztályok mindegyike egy-egy fajta *BBwindow* (Big Bucks window) kell, hogy legyen. Ezt a legegyszerűbben úgy érhetjük el, ha *Ival_box*-unkat úgy írjuk át, hogy a *BBwindow*-ból származtatott osztály legyen. Így aztán az összes osztályunk a *BBwindow*-ból származtatott lesz, tehát elhelyezhető lesz a képernyőn, megjelenése igazodik majd a rendszer többi grafikus elemének megjelenéséhez, átméretezhető, áthelyezhető lesz stb., a BBwindow rendszer szabályainak megfelelően. Osztályhierarchiánk tehát így fog kinézni:

```
class Ival_box : public BBwindow { /* ... */ };     // újraírva a BBwindow használatára
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

Ábrával:



12.4.1.1. Bírálat

Ez így sok tekintetben jól fog működni és az ilyesfajta osztályfelépítés számos problémára jó megoldás. Ám van néhány hátulütője, melyek miatt más tervezési lehetőségek után fogunk nézni.

A *BBwindow* osztályt utólag tettük az *Ival_box* bázisosztályává, ami nem egészen helyes. A *BBwindow* osztály nem alapvető része az *Ival_box*-ra épített rendszernek, megléte csupán részletkérdés. Az, hogy az *Ival_box* a *BBwindow* osztály leszármazottja, ezt a részletkérdést elsőrendű tervezési döntéssé emeli. Ez abban az esetben helyes, ha cégünk kulcsfontosságú üzleti döntése, hogy a „Big Bucks Inc." által biztosított környezetet használjuk. De mi történik, ha *Ival_box*-unkat olyan rendszerekre is át szeretnénk ültetni, melyek az „Imperial Bananas", a „Liberated Software" vagy a „Compiler Whizzles"-től származnak? Ekkor programunkból négy változatot kellene készítenünk:

```
class Ival_box : public BBwindow { /* ... */ };      // BB változat
class Ival_box : public CWwindow { /* ... */ };      // CW változat
class Ival_box : public IBwindow { /* ... */ };      // IB változat
class Ival_box : public LSwindow { /* ... */ };      // LS változat
```

Ha ennyi változatunk van, módosításuk, változatkövetésük rémálommá válhat.

Egy másik probléma, hogy az *Ival_box*-ban deklarált adatok minden származtatott osztály rendelkezésére állnak. Ezek az adatok megint csak egy „apró" részletet jelentenek, mégis bekerültek az *Ival_box* felületbe. Ez gyakorlati szempontból azt is jelenti, hogy nem biztosított, hogy mindig a megfelelő adatot kapjuk. Az *Ival_slider* esetében például nem szükséges az adat külön tárolása, minthogy ez a csúszka állásából meghatározható, valahányszor végrehajtják a *get_value()*-t. Általában is problematikus két rokon, de eltérő adathalmaz tárolása. Előbb-utóbb valaki eléri, hogy ne legyenek többé összhangban. A tapasztalat is azt mutatja, hogy kezdő programozók szükségtelen és nehezebb módosíthatóságot eredményező módon szeretnek a védett (protected) adattagokkal ügyeskedni. Jobb, ha az adattagok privátok, mert így a származtatott osztályok írói nem zavarhatják össze azokat. Még jobb, ha az adatok a származtatott osztályokban vannak, mert akkor pontosan meg tudnak felelni a követelményeknek és nem keseríthetik meg az egymással nem rokon származtatott osztályok életét. A védett felület szinte mindig csak függvényeket, típusokat és konstansokat tartalmazzon.

Ha az *Ival_box* a *BBwindow*-ból származik, ez azzal az előnnyel jár, hogy az *Ival_box* felhasználói a *BBwindow* minden szolgáltatását igénybe vehetik, ami sajnos azt is jelenti, hogy a *BBwindow* osztály változásakor az *Ival_box* felhasználóinak újra kell fordítaniuk, esetleg újra kell írniuk a kódjukat. A legtöbb C++-változat úgy működik, hogy ha egy bázisosztály mérete megváltozik, akkor az összes származtatott osztályt újra kell fordítani.

Végül lehetséges, hogy programunknak olyan vegyes környezetben kell futnia, ahol különböző felhasználói felületek ablakai léteznek egyidejűleg. Vagy azért, mert valahogy ezek egy képernyőn tudnak osztozni, vagy mert programunknak különböző rendszerek felhasználóival kell kapcsolatot tartania. Ehhez pedig nem elég rugalmas megoldás, ha a felhasználói felületet az egyetlen *Ival_box* felületünk bázisosztályaként „bedrótozzuk".

12.4.2. Absztrakt osztályok

Nos, kezdjük újra a tervezést, hogy megoldjuk a hagyományos felépítés bírálatában felvetett problémákat:

1. A felhasználói felület valóban olyan részletkérdés legyen, amelyről nem kell tudomást venniük azon felhasználóknak, akiket nem érdekel.
2. Az *Ival_box* osztály ne tartalmazzon adatokat.
3. Ha megváltozik a felhasználói felületet kezelő rendszer, ne legyen szükséges az *Ival_box* családot felhasználó kód újrafordítása.
4. Különböző felhasználói felületekhez tartozó *Ival_box*-ok tudjanak egyszerre létezni a programban.

Többféle megoldás kínálkozik erre; most egy olyat mutatok be, amely tisztán megvalósítható a C++ nyelvvel.

Először is, az *Ival_box* osztályt puszta felületként (pure interface) határozzuk meg:

```
class Ival_box {
public:
    virtual int get_value() = 0;
    virtual void set_value(int i) = 0;
    virtual void reset_value(int i) = 0;
    virtual void prompt() = 0;
    virtual bool was_changed() const = 0;
    virtual ~Ival_box() { }
};
```

Ez sokkal világosabb, mint az *Ival_box* osztály eredeti deklarációja volt. Elhagytuk az adattagokat és a tagfüggvények egyszerűsített kifejtését is. Elmaradt a konstruktor is, mivel nincs kezdőértékre váró adat. Ehelyett egy virtuális destruktorunk van, amely biztosítja az öröklő osztályok adatainak helyes „eltakarítását".

Az *Ival_slider* definíciója így alakulhat:

```
class Ival_slider : public Ival_box, protected BBwindow {
public:
     Ival_slider(int,int);
     ~Ival_slider();

     int get_value();
     void set_value(int i);
     // ...
protected:
     // a BBwindow virtuális függvényeit felülíró függvények
     // pl. BBwindow::draw(), BBwindow::mouse1hit()
private:
     // a csúszka adatai
};
```

Mivel az *Ival_slider* osztály az absztrakt *Ival_box* osztályból származik, meg kell valósítania annak tisztán virtuális (pure virtual) függvényeit. A *BBwindow* osztályból is származik, ezért onnan valók az eszközei, melyekkel ezt megteheti. Az *Ival_box* adja a származtatott osztály felületét, ezért nyilvános (public) módon származik onnan. Mivel a *BBwindow* osztályból való származása mindössze segítséget nyújt a megvalósításhoz, onnan védett (protected) módon származik (§15.3.2). Ebből következik, hogy az *Ival_slider*-t felhasználó programozó nem használhatja közvetlenül a *BBwindow* által nyújtott eszközöket. Az *Ival_slider* felülete az *Ival_box*-tól örökölt részből áll, illetve abból, amit maga az *Ival_slider* kifejezetten deklarál. Azért használunk védett származtatást a szigorúbb megkötést jelentő (és általában biztonságosabb) privát helyett, hogy az *Ival_slider*-ből származtatott osztályok számára a *BBwindow*-t elérhetővé tegyük.

A több osztályból való közvetlen öröklődést általában *többszörös öröklődésnek* (multiple inheritance) hívják (§15.2). Vegyük észre, hogy *Ival_slider*-nek mind az *Ival_box*, mind a *BBwindow* függvényei közül felül kell írnia néhányat, ezért közvetve vagy közvetlenül mindkét osztályból származnia kell. Mint a §12.4.1.1 pontban láttuk, lehetséges ugyan az *Ival_slider* közvetett származtatása a *BBwindow*-ból (azáltal, hogy az *Ival_box* a *BBwindow*-ból származik), de ez nemkívánatos mellékhatásokkal jár. Hasonlóan, az az út, hogy a *BBwindow* „megvalósítási osztály" tagja legyen az *Ival_box*-nak, nem járható, mert egy osztály nem írhatja felül tagjainak virtuális függvényeit (§24.3.4). Az ablaknak az *Ival_box* osztály egy *BBwindow** típusú tagjaként való ábrázolása teljesen eltérő szerkezethez vezet, melynek megvannak a maga előnyei és hátrányai (§12.7[14], §25.7).

Érdekes módon az *Ival_slider* ilyen módon való deklarálása esetén ugyanolyan kódot írhatunk, mint azelőtt. Csak azért változtattunk, hogy a szerkezet logikusabb módon tükrözze a megvalósítást.

Számos osztálynak szüksége van valamilyen „rendrakásra”, mielőtt egy objektuma megsemmisül. Mivel az absztrakt *Ival_box* osztály nem tudhatja, hogy egy származtatott osztálynak nincs-e szüksége erre, fel kell tételeznie, hogy igenis szükség van rá. A rendrakást úgy biztosítjuk, hogy a bázisosztályban definiáljuk az *Ival_box::~Ival_box()* virtuális destruktort és a származtatott osztályokban megfelelő módon felülírjuk azt:

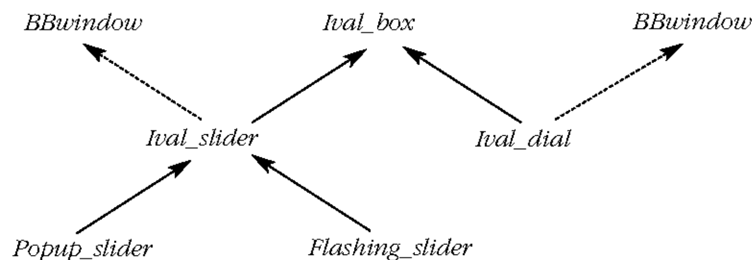
```
void f(Ival_box* p)
{
    // ...
    delete p;
}
```

A *delete* operátor megsemmisíti az objektumot, amelyre *p* mutat. Nem tudhatjuk, hogy pontosan milyen osztályú objektumról van szó, de mivel az *Ival_box*-nak virtuális destruktora van, a megfelelő destruktor fog meghívódni, (ha az adott osztálynak van ilyen).

Az *Ival_box* hierarchiát most így írhatjuk le:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class Ival_dial : public Ival_box, protected BBwindow { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

Egyszerű rövidítésekkel pedig így ábrázolhatjuk:



A szaggatott nyilak a védett (protected) módú öröklődést jelölik. Az általános felhasználók számára ezek csak részletkérdések.

12.4.3. Egyéb megvalósítások

Ez a szerkezet tisztább és könnyebben módosítható, mint a hagyományos, de nem kevésbé hatékony. A változatkövetési problémát azonban nem oldja meg:

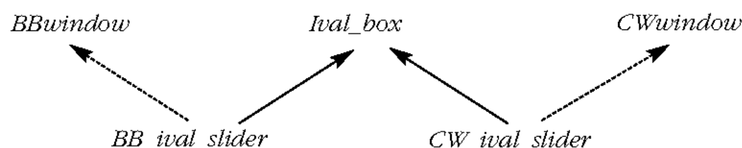
```
class Ival_box { /* ... */ };                                          // közös
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ }; // BB
class Ival_slider : public Ival_box, protected CWwindow { /* ... */ }; // CW
// ...
```

Ráadásul a *BBwindow*-hoz és a *CWwindow*-hoz írt *Ival_slider*-ek nem létezhetnek együtt, még akkor sem, ha egyébként maguk a *BBwindow* és *CWwindow* felhasználói felületek igen.

A nyilvánvaló megoldás az, hogy különböző nevű *Ival_slider* osztályokat hozunk létre:

```
class Ival_box { /* ... */ };
class BB_ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_box, protected CWwindow { /* ... */ };
// ...
```

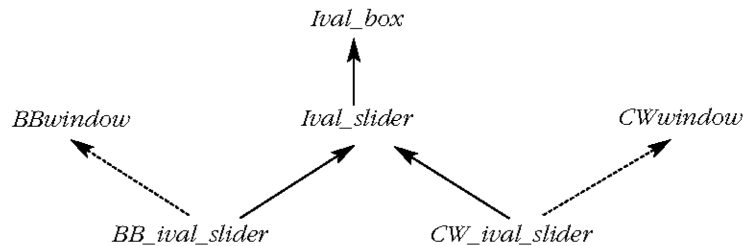
Ábrával:



Hogy programunk *Ival_box* osztályait jobban elszigeteljük a megvalósítás egyéb részleteitől, származtathatunk egy absztrakt *Ival_slider* osztályt az *Ival_box*-ból, majd ebből örököltethetjük az egyes rendszerfüggő *Ival_slider*-eket:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class BB_ival_slider : public Ival_slider, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWwindow { /* ... */ };
// ...
```

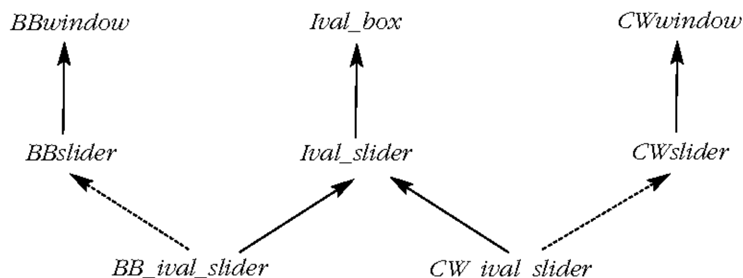
Ábrával:



Általában még ennél is jobban járunk, ha a hierarchiában egyedibb osztályokat használunk. Ha például a „Big Bucks Inc." rendszerében van egy csúszka (slider) osztály, akkor a mi *Ival_slider*-ünket közvetlenül a *BBslider*-ből származtathatjuk:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
```

Ábrával:



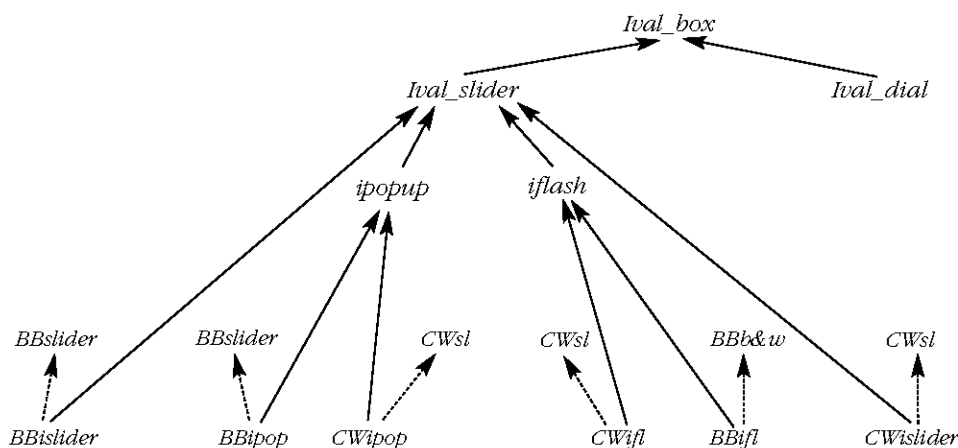
Ez a javítás jelentős lehet abban a (sűrűn előforduló) esetben, ha a mi fogalmaink nem esnek távol a megvalósítás céljából felhasznált rendszer fogalmaitól. Ekkor a programozás tulajdonképpen a rokon fogalmak közötti leképezésre egyszerűsödik, és a *BBwindow*-hoz hasonló általános bázisosztályokból való öröklődés ritkán fordul elő. A teljes hierarchia egyrészt az eredeti, alkalmazásközpontú rendszer származtatott osztályokként megvalósított felületeinek viszonyrendszeréből fog állni:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

Illetve a hierarchiát – szintén az öröklődés segítségével – többféle grafikus felhasználói felületre leképező származtatott osztályokból:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class BB_flashing_ival_slider : public Flashing_ival_slider,
            protected BBwindow_with_bells_and_whistles { /* ... */ };
class BB_popup_ival_slider : public Popup_ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
// ...
```

A kapott felépítményt egyszerű rövidítések segítségével így ábrázolhatjuk:



Az eredeti *Ival_box* hierarchia változatlan marad, csak a konkrét megvalósítást végző osztályok veszik körül.

12.4.3.1. Bírálat

Az absztrakt osztályokat használó osztályszerkezet rugalmas és majdnem ugyanolyan egyszerűen kezelhető, mint a konkrét felhasználói felületet bázisosztályként szerepeltető. Az utóbbiban a fa gyökere a megfelelő ablakosztály, az előbbiben viszont változatlanul az alkalmazás osztályhierarchiája marad a tényleges megvalósítást végző osztályok alapja. A program szempontjából ezek a szerkezetek egyenértékűek abban az értelemben, hogy majdnem az egész kód változtatás nélkül és ugyanúgy működik mindkét esetben, és mindkettőnél az alkalmazott felhasználói felülettől függő elemekre való tekintet nélkül vizsgálhatjuk az *Ival_box* család osztályait. A §12.4.1-beli *interact()* függvényt például nem kell újraírnunk, ha az egyik szerkezetről a másikra váltunk.

Mindkét esetben újra kell írnunk az egyes *Ival_box* osztályokat, ha a felhasználói felület nyilvános felülete megváltozik, de az absztrakt osztályokat használó szerkezet esetében szinte az egész kód védett a megvalósítás változásától és egy ilyen változás után nem kell újrafordítani. Ez különösen akkor fontos, ha a megvalósítást végző elemek készítője egy új, „majdnem kompatibilis" változatot bocsát ki. Ráadásul az absztrakt osztályos megoldást választók a klasszikus hierarchia híveinél kevésbé vannak kitéve az egyedi, máshol nem használható megvalósítás csapdájába való bezáródás veszélyének. Az elvont *Ival_box* osztályokra épített programot választva nem használhatjuk „véletlenül" a megvalósító osztályok nyújtotta lehetőségeket, mert csak az *Ival_box* hierarchiában kifejezetten megadott lehetőségek érhetők el, semmi sem öröklődik automatikusan egy rendszerfüggő bázisosztálytól.

12.4.4. Az objektumok létrehozásának adott helyre korlátozása

A program legnagyobb része megírható az *Ival_box* felület felhasználásával. Ha a származtatott felületek továbbfejlődnek és több szolgáltatást nyújtanak, mint a sima *Ival_box*, akkor nagyrészt használhatjuk az *Ival_box*, *Ival_slider* stb. felületeket. Az objektumokat azonban az adott rendszerre jellemző nevek (például *CW_ival_dial* és *BB_flashing_ival_slider*) felhasználásával kell létrehozni. Jó lenne, ha az ilyen rendszerfüggő nevek minél kevesebb helyen fordulnának elő, mivel az objektumok létrehozása nehezen köthető helyhez, hacsak nem szisztematikusan járunk el.

Szokás szerint az indirekció (közvetett hivatkozás) bevezetése a megoldás. Ezt többféleképpen is megtehetjük. Egyszerű megoldás lehet például egy olyan osztály bevezetése, amely az objektumokat létrehozó műveletekért felelős:

```
class Ival_maker {
public:
     virtual Ival_dial* dial(int, int) =0;                  // tárcsa (dial) készítése
     virtual Popup_ival_slider* popup_slider(int, int) =0;  // előugró csúszka (popup slider)
                                                            // készítése
     // ...
};
```

Az *Ival_maker* osztály az *Ival_box* hierarchia minden olyan felülete számára rendelkezik az adott típusú objektumot létrehozó függvénnyel, mely felületről a felhasználók tudhatnak. Az ilyen osztályokat *gyárnak* (factory) hívják, függvényeiket pedig – némiképp félrevezető módon – *virtuális konstruktoroknak* (§15.6.2).

Az egyes különböző felhasználói felületeket kezelő rendszereket most az *Ival_maker* osztályból származtatott osztályokként ábrázoljuk:

```
class BB_maker : public Ival_maker {            // BB-változatok készítése
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};

class LS_maker : public Ival_maker {            // LS-változatok készítése
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};
```

Minden függvény a kívánt felületű és megvalósítási típusú objektumot hozza létre:

```
Ival_dial* BB_maker::dial(int a, int b)
{
    return new BB_ival_dial(a,b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
    return new LS_ival_dial(a,b);
}
```

Ha adott egy mutató egy *Ival_maker* objektumra, akkor a programozó ennek segítségével úgy hozhat létre objektumokat, hogy nem kell tudnia, pontosan milyen rendszerű felhasználói felület van használatban:

```
void user(Ival_maker* pim)
{
    Ival_box* pb = pim->dial(0,99);     // megfelelő tárcsa létrehozása
    // ...
}
BB_maker BB_impl;                       // BB-felhasználóknak
LS_maker LS_impl;                       // LS-felhasználóknak

void driver()
{
    user(&BB_impl);                     // BB használata
    user(&LS_impl);                     // LS használata
}
```

12.5. Osztályhierarchiák és absztrakt osztályok

Az absztrakt osztályok felületek (interface). Az osztályhierarchia annak eszköze, hogy fokozatosan építsünk fel osztályokat. Természetesen minden osztály ad egy felületet a programozó számára, némely absztrakt osztály pedig jelentős szolgáltatásokat kínál, amelyekre építhetünk, de „felület" és „építőkő" szerepük alapvetően az absztrakt osztályoknak és az osztályhierarchiáknak van.

Klasszikus hierarchiának azt a felépítést nevezzük, amelynek egyes osztályai hasznos szolgáltatásokat kínálnak a felhasználóknak, illetve egyben a fejlettebb vagy egyedi feladatot végző osztályok számára építőkőül szolgálnak. Az ilyen felépítés ideálisan támogatja a lépésenkénti finomítással való fejlesztést, illetve az új osztályok létrehozását, amennyiben ezek megfelelően illeszkednek a hierarchiába.

A klasszikus felépítés a tényleges megvalósítást sokszor összekapcsolja a felhasználóknak nyújtott felülettel. Ez ügyben az absztrakt osztályok segíthetnek. Az absztrakt osztályok segítségével felépített rendszer tisztább és hatékonyabb módot ad a fogalmak kifejezésére, anélkül hogy a megvalósítás részleteivel keveredne vagy jelentősen növelné a program futási idejét. A virtuális függvények meghívása egyszerű és független attól, hogy miféle elvonatkoztatási réteg határát lépi át. Egy absztrakt osztály virtuális függvényét meghívni semmivel sem kerül többe, mint bármely más virtuális függvényt.

A fentiekből adódó végkövetkeztetés az, hogy egy rendszert a felhasználók felé mindig absztrakt osztályok hierarchiájaként mutassunk, de klasszikus hierarchiaként építsünk fel.

12.6. Tanácsok

[1] Kerüljük a típusmezők alkalmazását. §12.2.5.
[2] Az objektumok felszeletelődését (slicing) elkerülendő használjunk mutatókat és referenciákat. §12.2.3.
[3] Használjunk absztrakt osztályokat, hogy a világos felületek elkészítésére összpontosíthassunk. §12.3.
[4] Használjunk absztrakt osztályokat, hogy minél kisebb felületeket használhassunk. §12.4.2.
[5] Használjunk absztrakt osztályokat, hogy a felületeket elválasszuk a megvalósítási részletektől. §12.4.2.

[6] Használjunk virtuális függvényeket, hogy később új megvalósítást készíthessünk a meglevő felhasználói kód befolyásolása nélkül. §12.4.1.
[7] Használjunk absztrakt osztályokat, hogy minél kevesebbszer kelljen a felhasználói kódot újrafordítani. §12.4.2.
[8] Használjunk absztrakt osztályokat, hogy a program többféle rendszeren is működjön. §12.4.3.
[9] Ha egy osztálynak van virtuális függvénye, akkor legyen virtuális destruktora is. §12.4.2.
[10] Az absztrakt osztályoknak általában nincs szükségük konstruktorra. 12.4.2.
[11] Az önálló fogalmakat külön ábrázoljuk. §12.4.1.1.

12.7. Gyakorlatok

1. (*1) Ha adott a következő:

```
class Base {
public:
        virtual void iam() { cout << "Bázisosztály\n"; }
};
```

   Származtassunk két osztályt a *Base*-ből, mindegyiknek legyen egy *iam()* függvénye, amely kiírja az osztály nevét. Hozzunk létre egy-egy ilyen osztályú objektumot és hívjuk meg rájuk az *iam()* függvényt. Rendeljünk a származtatott osztályok objektumaira hivatkozó mutatókat *Base** típusú mutatókhoz és hívjuk meg ezeken keresztül az *iam()* függvényt.
2. (*3.5) Készítsünk egy egyszerű grafikus rendszert a rendelkezésünk álló grafikus felhasználói felület felett. (Ha nincs ilyen vagy nincs tapasztalatunk ilyesmivel, akkor készíthetünk egy egyszerű, ASCII karakterekből felépített megvalósítást, ahol egy pont egy karakterpozíciónak felel meg, és az írás a megfelelő karakter, mondjuk a * megfelelő pozícióra való helyezését jelenti.)
   Ebben a feladatban és a továbbiakban a következő osztályokat használjuk: *Window* (Ablak), *Point* (Pont), *Line* (Vonal), *Dot* (Képernyőpont), *Rectangle* (Téglalap), *Circle* (Kör), *Shape* (Alakzat, Idom), *Square* (Négyzet) és *Triangle* (Háromszög).
   A *Window(n,m)* hozzon létre egy *n*-szer *m* méretű területet a képernyőn.
   A képernyő pontjait az *(x,y)* derékszögű (descartes-i) koordináták segítségével címezzük meg. A *Window* osztályba tartozó *w* aktuális helye – *w.current()* –

kezdetben *Point(0,0)*. A pozíciót a *w.current(p)* hívással állíthatjuk be, ahol *p* egy *Point*. A *Point* objektumokat egy koordináta-pár adja meg: *Point(x,y)*; a *Line* objektumokat egy *Point* pár – *Line(w.current(),p2)* –; a *Shape* osztály a *Dot*, a *Line*, a *Rectangle*, a *Circle* stb. közös felülete. Egy *Point* nem *Shape* is egyben. A *Dot(p)*-ként létrehozott *Dot* egy *Point p*-t jelent a képernyőn. A *Shape*-ek nem láthatók, amíg a *draw()* függvényt meg nem hívjuk; például: *w.draw(Circle(w.current(),10))*. Minden *Shape*-nek 9 érintkezési pontja van: *e* (east – kelet), *w* (west – nyugat), *n* (north – észak), *s* (south – dél), *ne* (északkelet), *nw* (északnyugat), *se* (délkelet), *sw* (délnyugat) és *c* (center – középpont). A *Line(x.c(),y.nw())* például egy vonalat húz az *x* közepétől az *y* bal felső sarkához. Ha egy *Shape*-re alkalmaztuk a *draw()* függvényt, az aktuális pozíció a *Shape se()*-je lesz. Egy *Rectangle*-t a bal alsó és a jobb felső csúcsával adunk meg; *Rectangle(w.current(),Point(10,10))*. Egyszerű tesztként jelenítsünk meg egy gyermekrajzot, amely egy házat ábrázol tetővel, két ablakkal, és egy ajtóval.

3. (*2) Egy *Shape* fontos részei szakaszokként jelennek meg a képernyőn. Adjunk meg olyan műveleteket, amelyek segítségével meg tudjuk változtatni ezen szakaszok kinézetét. Az *s.thickness(n)* a *0,1,2,3* értékek valamelyikére állítsa be a vonalszélességet, ahol a *2* az alapértelmezett érték és a *0* érték azt jelenti, hogy a vonal láthatatlan. A vonal lehessen tömör, szaggatott vagy pontokból álló is. Ezt a *Shape::outline()* függvény állítsa be.
4. (*2.5) Írjuk meg a *Line::arrowhead()* függvényt, amely egy vonal végére egy nyilat rajzol. Minthogy egy vonalnak két vége van és a nyíl a vonalhoz képest kétféle irányba mutathat, így az *arrowhead()* függvény paramétere vagy paraméterei ki kell, hogy tudják fejezni ezt a négyféle lehetőséget.
5. (*3.5) Gondoskodjunk arról, hogy azon pontok és vonalszakaszok, amelyek kívül esnek egy *Window*-n, ne jelenjenek meg. Ezt a jelenséget gyakran hívják „levágásnak” (clipping). E célból – gyakorlatként – ne hagyatkozzunk a felhasznált grafikus felhasználói felületre.
6. (*2.5) Egészítsük ki grafikai rendszerünket a *Text* típussal. A *Text* legyen egy téglalap alakú *Shape*, amely karaktereket tud megjeleníteni. Alapértelmezés szerint egy karakter a koordináta-tengelyen minden irányban egy egységnyi helyet foglaljon el.
7. (*2) Határozzunk meg egy függvényt, amely megtalálja két *Shape* egymáshoz legközelebbi pontjait és összeköti azokat.
8. (*3) Vezessük be grafikai rendszerünkbe a szín fogalmát. Háromféle dolog lehet színes: a háttér, egy zárt *Shape* belseje és egy *Shape* határa.

9. (*2) Vegyük az alábbi osztályt:

```
class Char_vec {
        int sz;
        char element[1];
public:
        static Char_vec* new_char_vec(int s);
        char& operator[ ](int i) { return element[i]; }
        // ...
};
```

   Definiáljuk a *new_char_vec()*-t, hogy egybefüggő memóriaterületet foglalhassunk le egy *Char_vec* objektum számára, így elemeit az *element()* függvénnyel indexelhetjük. Milyen körülmények között okoz ez a trükk komoly problémákat?
10. (*2.5) Ha adottak a *Shape* osztályból származó *Circle*, *Square* és *Triangle* osztályok, határozzuk meg az *intersect()* függvényt, amely két *Shape** paramétert vesz és a megfelelő függvények meghívásával megállapítja, hogy a két *Shape* átfedő-e, metszi-e egymást. Ehhez szükséges lesz az osztályok megfelelő (virtuális) függvényekkel való bővítése. Ne írjuk meg az átfedés tényleges megállapítására szolgáló kódot, csak arra ügyeljünk, hogy a megfelelő függvényeket hívjuk meg. Ezt az eljárást angolul általában „double dispatch" vagy „multi-method" néven emlegetik.
11. (*5) Tervezzünk és írjunk meg egy eseményvezérelt szimulációkat végző könyvtárat. Segítség: nézzük meg a *<task.h>* fejállományt. Ez egy régi program, az olvasó jobbat tud írni. Legyen egy *task* nevű osztály. A *task* osztályú objektumok legyenek képesek állapotuk mentésére és visszaállítására (mondjuk a *task::save()* és a *task::restore()* függvényekkel), hogy kiegészítő eljárásként (co-routine) működhessenek. Az egyes elvégzendő feladatokat a *task* osztályból öröklő osztályok objektumaiként adhassuk meg. A *task*-ok által végrehajtandó programokat virtuális függvényekkel határozzuk meg. Egy új *task* számára legyen lehetséges paramétereket megadni konstruktora(i)nak paramétereként. Legyen egy ütemező, amely megvalósítja a virtuális idő fogalmát. Legyen egy *task::delay(long)* függvény, amely „fogyasztja" ezt a virtuális időt. Az, hogy ez az ütemező a *task* része vagy önálló osztály lesz-e, a fő tervezési döntések egyike. A *task*-oknak kapcsolatot kell tartaniuk egymással. Erre a célra tervezzünk egy *queue* osztályt. Egy *task*-nak legyen lehetősége több forrás felől érkező bemenetre várakozni. Kezeljük a futási idejű hibákat azonos módon. Hogyan lehetne egy ilyen könyvtárat használó programban hibakeresést végezni?
12. (*2) Határozzuk meg egy kalandjáték számára a *Warrior* (harcos), *Monster* (szörny) és *Object* (tárgy; olyasmi, amit fel lehet kapni, el lehet dobni, használni lehet stb.) osztályok felületét.

13. (*1.5) Miért van a §12.7[2]-ben *Point* és *Dot* osztály is? Milyen körülmények között lenne jó ötlet a *Shape* osztályokat a kulcsosztályok, például a *Line* konkrét változataival bővíteni?
14. (*3) Vázoljuk az *Ival_box* példa (§12.4) egy eltérő megvalósítási módját: minden, a program által elérhető osztály egyszerűen egy mutatót tartalmazzon a megvalósító osztályra. Ilyen módon minden „felületosztály" egy megvalósító osztály leírója (handle) lesz, és két hierarchiával fogunk rendelkezni: egy felület- és egy megvalósítási hierarchiával. Írjunk olyan részkódokat, amelyek elég részletesek ahhoz, hogy bemutassák a típuskonverziókból adódó lehetséges problémákat. Gondoljuk át a következő szempontokat: a használat könnyűsége; a programozás könnyűsége; mennyire könnyű a megvalósító osztályok és a felületek újrahasznosítása, ha új fogalmat vezetünk be a hierarchiába; mennyire könnyű változtatásokat eszközölni a felületekben vagy a megvalósításban; és szükség van-e újrafordításra, ha változott a rendszerfüggő elemek megvalósítása.

13

Sablonok

„Az Olvasó idézetének helye."
(B. Stroustrup)

Sablonok • Egy karakterlánc sablon • Példányosítás • Sablonparaméterek • Típusellenőrzés • Függvénysablonok • Sablonparaméterek levezetése • Sablonparaméterek meghatározása • Függvénysablonok túlterhelése • Eljárásmód megadása sablonparaméterekkel • Alapértelmezett sablonparaméterek • Specializáció • Öröklődés és sablonok • Tag sablonok • Konverziók • A forráskód szerkezete • Tanácsok • Gyakorlatok

13.1. Bevezetés

Független fogalmakat függetlenül ábrázoljunk és csak szükség esetén használjunk együtt. Ha megsértjük ezt az elvet, akkor vagy nem rokon fogalmakat kapcsolunk össze vagy szükségtelen függéseket teremtünk, így kevésbé rugalmas részekből vagy összetevőkből kell majd a programokat összeállítanunk. A *sablonok* (template) egyszerű módot adnak arra, hogy általános fogalmak széles körét ábrázoljuk és egyszerű módon használjuk együtt. Az így létrejövő osztályok futási idő és tárigény tekintetében felveszik a versenyt a kézzel írott és egyedibb feladatot végző kóddal.

A sablonok közvetlenül támogatják az általánosított (generikus) programozást (§2.7.), azaz a típusoknak paraméterként való használatát. A C++ sablonjai lehetővé teszik, hogy egy osztály vagy függvény definiálásakor egy típust paraméterként adjunk meg. A sablon a felhasznált típusnak csak azon tulajdonságaitól függ, amelyeket ténylegesen ki is használ. A sablon által felhasznált paramétertípusok nem kell, hogy rokonságban álljanak egymással, így nem szükséges az sem, hogy egyazon öröklődési hierarchia tagjai legyenek.

Ebben a fejezetben a sablonokat úgy mutatjuk be, hogy az elsődleges hangsúly a standard könyvtár tervezéséhez, megvalósításához és használatához szükséges módszerekre esik. A standard könyvtár nagyobb mértékű általánosságot, rugalmasságot és hatékonyságot követel, mint a legtöbb program. Következésképpen a tárgyalandó eljárások széles körben használhatóak és igen sokféle probléma megoldásához biztosítanak hatékony segítséget. Lehetővé teszik, hogy egyszerű felületek mögé kifinomult megvalósításokat rejtsünk és csak akkor „mutassuk be" a bonyolult részleteket a felhasználónak, ha valóban szüksége van rájuk. A *sort(v)* például sokféle tároló objektum tartalmazta sokféle típusú elemnek sokféle rendező algoritmusához adhat felületet. Egy adott *v*-hez a fordítóprogram automatikusan választja ki a legalkalmasabb rendező függvényt.

A standard könyvtár minden főbb fogalmat egy-egy sablonként ábrázol (például *string*, *ostream*, *complex*, *list* és *map*), de a legfőbb műveleteket is, például a karakterláncok (*string*-ek) összehasonlítását, a << kimeneti műveletet, a komplex számok (*complex*) összeadását, egy lista (*list*) következő elemének vételét, vagy a rendezést (*sort()*). Ezért aztán e könyvnek az említett könyvtárral foglalkozó fejezetei (a III. rész) gazdag forrásai a sablonokra és az azokra építő programozási módszerekre vonatkozó példáknak. Következésképpen ez a fejezet a sablonok fogalmát járja körül és csupán a használatuk alapvető módjait bemutató kisebb példákra összpontosít:

§13.2 Az osztálysablonok létrehozására és használatára szolgáló alapvető eljárások
§13.3 Függvénysablonok, függvények túlterhelése, típusok levezetése
§13.4 Általánosított algoritmusok eljárásmódjának megadása sablonparaméterekkel
§13.5 Sablon többféle megvalósítása különböző definiciókkal
§13.6 Öröklődés és sablonok (futási és fordítási idejű többalakúság)
§13.7 A forráskód szerkezete

A *sablon* (template) fogalmát a §2.7.1 és a §3.8 pont vezette be. A sablonnevek feloldására, illetve a sablonok formai követelményeire vonatkozó részletes szabályok a §C.13 pontban vannak.

13.2. Egy egyszerű karakterlánc sablon

Vegyünk egy karakterláncot. A *string* (karakterlánc) olyan osztály, amely karaktereket tárol és olyan indexelési, összefűzési és összehasonlítási műveleteket nyújt, amelyeket rendesen a „karakterlánc" fogalmához kötünk. Ezeket különféle karakterkészletek számára szeretnénk biztosítani. Például az előjeles és előjel nélküli, kínai vagy görög stb. karakterekből álló láncok számos összefüggésben hasznosak lehetnek. Ezért úgy szeretnénk a karakterlánc fogalmát ábrázolni, hogy minél kevésbé függjünk egy adott karakterkészlettől. A karakterlánc definiciója arra épít, hogy egy karaktert le lehet másolni, ezen kívül nem sok egyébre. Ezért ha a §11.2-beli *char*-okból felépülő *string* osztályban a karakterek típusát paraméterré tesszük, általánosabb karakterlánc-osztályt kapunk:

```
template<class C> class String {
    struct Srep;
    Srep *rep;
public:
    String();
    String(const C*);
    String(const String&);

    C read(int i) const;
    // ...
};
```

A *template<class C>* előtag azt jelenti, hogy egy sablon deklarációja következik és abban a *C* típusparamétert fogjuk használni. Bevezetése után a *C*-t ugyanúgy használhatjuk, mint bármely más típusnevet. A *C* hatóköre a *template<class C>* előtaggal bevezetett deklaráció végéig terjed. Jegyezzük meg, hogy a *template<class C>* előtag azt jelenti, hogy *C* egy típusnév; nem feltétlenül kell osztálynévnek lennie. Az osztálysablon neve a <> jelpár közé írott típusnévvel együtt egy, a sablon által meghatározott osztály nevét adja és ugyanúgy használható, mint bármely más osztálynév:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar {
    // japán karakter
};

String<Jchar> js;
```

A névre vonatkozó sajátos formai követelményektől eltekintve a *String<char>* pontosan ugyanúgy működik, mintha a §11.12-beli *String*-definícióval definiáltuk volna. A *String* sablonná tétele lehetővé teszi, hogy a *char*-okból álló karakterláncok szolgáltatásait más típusú karakterekből álló *String*-ek számára is elérhetővé tegyük. Például ha a standard könyvtárbeli *map* és *String* sablonokat használjuk, a §11.8 pont szószámláló példája így írható át:

```
int main()          // szavak előfordulásának megszámlálása a bemeneten
{
    String<char> buf;
    map<String<char>,int> m;
    while (cin>>buf) m[buf]++;
    // eredmény kiírása
}
```

A *Jchar* japán karaktereket használó változat ez lenne:

```
int main()          // szavak előfordulásának megszámlálása a bemeneten
{
    String<Jchar> buf;
    map<String<Jchar>,int> m;
    while (cin>>buf) m[buf]++;
    // eredmény kiírása
}
```

A standard könyvtárban szerepel a sablonná alakított *String*-hez hasonló *basic_string* sablon is (§11.12, §20.3). A standard könyvtárban a *string* mint a *basic_string<char>* szinonimája szerepel:

```
typedef basic_string<char> string;
```

Ez lehetővé teszi, hogy a szószámláló példát így írjuk át:

```
int main()          // szavak előfordulásának megszámlálása a bemeneten
{
    string buf;
    map<string,int> m;
    while (cin>>buf) m[buf]++;
    // eredmény kiírása
}
```

A *typedef*-ek általában is hasznosak a sablonokból létrehozott osztályok hosszú neveinek lerövidítésére. Ráadásul, ha nem érdekel bennünket egy típus pontos definíciója, akkor egy *typedef* elrejti előlünk, hogy sablonból létrehozott típusról van szó.

13.2.1. Sablonok meghatározása

A sablonból létrehozott osztályok teljesen közönséges osztályok, ezért a sablonok használata semmivel sem igényel hosszabb futási időt, mint egy egyenértékű „kézzel írott" osztályé, de nem feltétlenül jelenti a létrehozott kód mennyiségének csökkenését sem.

Általában jó ötlet hibakereséssel ellenőrizni egy osztályt, például a *String*-et, mielőtt sablont készítünk belőle (*String<C>*). Ezáltal számos tervezési hibát, a kódhibáknak pedig a legtöbbjét egy adott osztály összefüggésében kezelhetünk. Ezt a fajta hibakeresést (debugging) a legtöbb programozó jól ismeri, és a legtöbben jobban boldogulnak egy konkrét példával, mint egy elvonttal. Később aztán anélkül foglalkozhatunk a típus általánosításából esetleg adódó problémákkal, hogy a hagyományosabb hibák elvonnák a figyelmünket. Hasonlóan, ha meg akarunk érteni egy sablont, akkor hasznos annak viselkedését először egy konkrét típusú paraméterrel (például a *char*-ral) elképzelni, mielőtt megpróbáljuk a viselkedését teljes általánosságában megérteni.

Egy sablon osztály (template class) tagjait ugyanúgy deklaráljuk és definiáljuk, mint a közönséges osztályokét. Egy tagot nem szükséges magában az osztályban definiálni; valahol máshol is elég, ugyanúgy, mint egy nem sablon osztálytag esetében (§C.13.7.). A sablon osztályok tagjai maguk is sablonok, paramétereik pedig ugyanazok, mint a sablon osztályéi. Ha egy ilyen tagot az osztályán kívül írunk le, kifejezetten sablonként kell megadnunk:

```
template<class C> struct String<C>::Srep {
     C* s;          // mutató az elemekre
     int sz;        // elemek száma
     int n;         // hivatkozásszámláló
     // ...
};

template<class C> C String<C>::read(int i) const { return rep->s[i]; }

template<class C> String<C>::String()
{
     rep = new Srep(0,C());
}
```

A sablonparaméterek – mint a *C* – inkább paraméterek, mint a sablonon kívül definiált típusok, de ez nem érinti azt a módot, ahogyan az azokat használó sablonkódot írjuk. A *String<C>* hatókörén belül a *<C>*-vel való minősítés felesleges, hiszen a sablon neve már tartalmazza azt, így a konstruktor neve *String<C>::String* lesz. De ha jobban tetszik, meg is adhatjuk a minősítést:

```
template<class C> String<C>::String<C>()
{
    rep = new Srep(0,C());
}
```

Egy programban egy tagfüggvényt csak egyetlen függvény definiálhat. Ugyanígy a sablon osztályok tagfüggvényeit is csak egy függvénysablon definiálhatja. De amíg a függvényeket csak túlterhelni lehet (§13.3.2), addig a specializációk (§13.5) használata lehetővé teszi, hogy egy sablonnak több változatát is elkészíthessük.

Az osztálysablonok neve nem terhelhető túl, így ha egy hatókörben már megadtunk egy osztálysablont, ott nem lehet ugyanolyan néven másik egyedet bevezetni (lásd még §13.5):

```
template<class T> class String { /* ... */ };

class String { /* ... */ };    // hiba: két meghatározás
```

A sablonparaméterként használt típusnak biztosítania kell a sablon által várt felületet. A *String* sablon paramétereként használt típusnak például támogatnia kell a szokásos másoló műveleteket (§10.4.4.1, §20.2.1). Jegyezzük meg: az nem követelmény, hogy egy sablon különböző paraméterei öröklődési viszonyban álljanak egymással.

13.2.2. Sablonok példányosítása

Az eljárást, melynek során egy sablon osztályból és egy sablonparaméterből egy osztálydeklaráció keletkezik, gyakran *sablon-példányosításnak* (template instantiation) hívják (§C.13.7.). Ha függvényt hozunk létre egy sablon függvényből és egy sablonparaméterből, az a függvény-példányosítás. A sablon adott paramétertípus számára megadott változatát *specializációnak* (specialization) nevezzük.

Általában az adott C++ fordító és nem a programozó dolga, hogy minden felhasznált paramétertípus számára létrehozza a megfelelő sablon függvényt (§C.13.7):

```
String<char> cs;

void f()
{
    String<Jchar> js;

    cs = "Az adott nyelvi változat feladata, hogy kitalálja, milyen kódot kell létrehozni.";
}
```

A fenti esetben a *String<char>* és a *String<Jchar>*, a megfelelő *Srep* típusok, a destruktorok és az alapértelmezett konstruktorok, illetve a *String<char>::operator=(char *)* deklarációit a fordító hozza létre. Más tagfüggvényeket nem használunk, így ilyeneket nem kell készítenie (remélhetőleg nem is teszi). A létrehozott osztályok közönséges osztályok, így az osztályokra vonatkozó szokásos szabályok érvényesek rájuk. Ugyanígy a létrehozott függvények is közönséges függvények és a függvényekre vonatkozó szokásos szabályok szerint viselkednek.

Nyilvánvaló, hogy a sablonok hatékony eszközt adnak arra, hogy viszonylag rövid definíciókból hozzunk létre kódot. Ezért aztán nem árt némi óvatosság, hogy elkerüljük a memóriának csaknem azonos függvény-definiciókkal való elárasztását (§13.5).

13.2.3. Sablonparaméterek

A sablonoknak lehetnek típust meghatározó, közönséges típusú (pl. *int*), és sablon típusú paraméterei (§C.13.3). Természetesen egy sablonnak több paramétere is lehet:

```
template<class T, T def_val> class Cont { /* ... */ };
```

Ahogy a példa mutatja, egy sablonparamétert felhasználhatunk a további sablonparaméterek meghatározásában is.

Az egész típusú paraméterek méretek és korlátok megadásánál hasznosak:

```
template<class T, int i> class Buffer {
     T v[i];
     int sz;
public:
     Buffer() : sz(i) {}
     // ...
};

Buffer<char,127> cbuf;
Buffer<Record,8> rbuf;
```

A *Buffer*-hez hasonló egyszerű és korlátozott tárolók ott lehetnek fontosak, ahol a futási idejű hatékonyság és a program tömörsége elsődleges szempont, és ahol ezért nem lehet az általánosabb *string*-et vagy *vector*-t használni. Mivel a sablon a méretet paraméterként megkapja, a kifejtésben el lehet kerülni a szabad tár használatát. Egy másik példa erre a *Range* osztály a §25.6.1-ben.

A sablon paramétere lehet konstans kifejezés (§C.5), külső szerkesztésű objektum vagy függvény címe (§9.2), illetve egy tagra hivatkozó, túl nem terhelt mutató (§15.5). A mutató, ha sablon paramétereként akarjuk használni, *&of* alakú kell, hogy legyen, ahol *of* egy objektum vagy függvény neve, illetve *f* alakú, ahol *f* egy függvény neve. A tagra hivatkozó mutatókat *&X::of* alakban kell megadni, ahol *of* a tag neve. Karakterlánc literált nem használhatunk sablonparaméterként.

Az egész típusú paramétereknek konstansnak kell lenniük:

```
void f(int i)
{
    Buffer<int,i> bx;     // hiba: konstans kifejezés szükséges
}
```

Megfordítva, a nem típusba tartozó paraméterek a sablonon belül állandók, így a paraméter értékének módosítására tett kísérlet hibának számít.

13.2.4. Típusok egyenértékűsége

Ha adott egy sablon, akkor különféle paramétertípusok megadásával különféle típusokat hozhatunk létre belőle:

```
String<char> s1;
String<unsigned char> s2;
String<int> s3;

typedef unsigned char Uchar;
String<Uchar> s4;
String<char> s5;

Buffer<String<char>,10> b1;
Buffer<char,10> b2;
Buffer<char,20-10> b3;
```

Ha azonos paraméterekkel adunk meg sablonokat, azok ugyanarra a létrehozott típusra fognak hivatkozni. De mit is jelent itt az, hogy „azonos”? Szokás szerint, a *typedef*-ek nem vezetnek be új típust, így a *String<Uchar>* ugyanaz, mint a *String<unsigned char>*. Megfordítva, mivel a *char* és az *unsigned char* különböző típusok (§4.3), a *String<char>* és a *String<unsigned char>* is különbözőek lesznek.

A fordítóprogram ki tudja értékelni a konstans kifejezéseket is (§C.5), így a *Buffer<char,20-10>*-ről felismeri, hogy a *Buffer<char,10>*-zel azonos típus.

13.2.5. Típusellenőrzés

A sablonokat paraméterekkel definiáljuk és később így is használjuk. A sablondefinícióban a fordítóprogram ellenőrzi a formai hibákat, illetve az olyanokat, amelyek a konkrét paraméterek ismerete nélkül felderíthetőek:

```
template<class T> class List {
    struct Link {
            Link* pre;
            Link* suc;
            T val;
            Link(Link* p, Link* s,const T& v) : pre(p), suc(s), val(v) { }
    }                                   // szintaktikus hiba: hiányzik a pontosvessző
    Link* head;
public:
    List() : head(7) { }                // hiba: kezdeti értékadás mutatónak int-tel
    List(const T& t) : head(new Link(0,o,t)) { }  // hiba: 'o' nem definiált azonosító
    // ...
    void print_all() const { for (Link* p = head; p; p=p->suc) cout << p->val << '\n'; }
};
```

A fordítóprogram az egyszerű nyelvi hibákat már a definíciónál kiszűrheti, néha azonban csak később, a használatnál. A felhasználók jobban szeretik, ha a hibák hamar kiderülnek, de nem minden „egyszerű" hibát könnyű felderíteni. Ebben a példában három hibát vétettem (szándékosan). A sablon paraméterétől függetlenül egy *T** típusú mutatónak nem adhatjuk a *7* kezdőértéket. Hasonlóan, az *o* változó (amely persze egy hibásan írt nulla) nem lehet a *List<T>::Link* konstruktor paramétere, mert ilyen név az adott pontról nem elérhető.

A sablon definiciójában használt névnek vagy ismertnek kell lennie, vagy valamilyen ésszerű és nyilvánvaló módon kell függnie valamelyik sablonparamétertől (§C.13.8.1). A *T* sablonparamétertől való legközönségesebb és legkézenfekvőbb függés egy *T* típusú tag vagy *T* típusú paraméter használata. A *List<T>::print_all()* példában a *cout << p->val* kifejezés használata némileg „kifinomultabb" példa.

A sablonparaméterek használatával összefüggő hibák csak a sablon használatának helyén deríthetők fel:

```
class Rec { /* ... */ };

void f(const List<int>& li, const List<Rec>& lr)
{
    li.print_all();
    lr.print_all();
}
```

Itt a *li.print_all()* rendben van, de a *lr.print_all()* típushibás, mert a *Rec* típusnak nincs << kimeneti művelete. A legelső pont, ahol a sablonparaméterek használatával összefüggő hiba kiderülhet, a sablonnak az adott paraméterrel való első használata. Ezt a pontot rendszerint *első példányosítási pontnak* (first point of instantiation) vagy egyszerűen *példányosítási pontnak* hívják (§C.13.7). Az adott C++-változat – megengedett módon – ezt az ellenőrzést a program összeszerkesztéséig elhalaszthatja. Ha ebben a fordítási egységben a *print_all()*-nak csak a deklarációja és nem a definíciója ismert, lehetséges, hogy az adott fordítónak el is *kell* halasztania a típusellenőrzést a program összeszerkesztéséig (§13.7). A típusellenőrzés azonos szabályok szerint történik, függetlenül attól, hogy melyik ponton megy végbe. A felhasználók itt is a minél korábbi ellenőrzést szeretik. A sablonparaméterekre vonatkozó megszorításokat a tagfüggvények segítségével is kifejezhetjük (§13.9[16]).

13.3. Függvénysablonok

A legtöbb programozó számára a sablonok első számú és legnyilvánvalóbb felhasználása olyasféle tároló osztályok létrehozása és használata, mint a *basic_string* (§20.3), a *vector* (§16.3), a *list* (§17.2.2) vagy a *map* (§17.4.1). Később azonban felmerül a sablonként használt függvények szükségessége. Nézzük például egy tömb rendezését:

```
template<class T> void sort(vector<T>&);          // deklaráció

void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi);                                     // sort(vector<int>&);
    sort(vs);                                     // sort(vector<string>&);
}
```

A sablon függvények (template function) meghívásakor a függvény paraméterei határozzák meg, hogy a sablon melyik példányát használjuk, vagyis a sablonparamétereket a függvényparaméterekből vezetjük le (deduce) (§13.3.1).

Természetesen a sablon függvényt valahol definiálnunk kell (§C.13.7):

```
template<class T> void sort(vector<T>& v)         // definíció
                                                  // Shell rendezés (Knuth, III. kötet, 84. o.2)
{
    const size_t n = v.size();
```

[2] Magyarul: D. E. Knuth: A számítógép-programozás művészete III. kötet, Keresés és rendezés; Műszaki könyvkiadó, Budapest, 1988; 95. oldal

```
    for (int gap=n/2; 0<gap; gap/=2)
          for (int i=gap; i<n; i++)
                for (int j=i-gap; 0<=j; j-=gap)
                      if (v[j+gap]<v[j]) {             // v[j] és v[j+gap] felcserélése
                            T temp = v[j];
                            v[j] = v[j+gap];
                            v[j+gap] = temp;
                      }
}
```

Hasonlítsuk össze a *sort()* ezen definicióját a §7.7-belivel. Ez a sablonná alakított változat világosabb és rövidebb, mert a rendezendő elemek típusára vonatkozóan több információra támaszkodhat. Valószínűleg gyorsabb is, mert nincs szüksége az összehasonlító függvényre hivatkozó mutatóra. Ebből következik, hogy nincs szükség közvetett függvényhívásra, a < összehasonlítást pedig könnyen lehet helyben kifejtve (inline) fordítani.

További egyszerűsítést jelenthet a standard könyvtárbeli *swap()* sablon használata (§18.6.8), mellyel az értékcserét természetes formára alakíthatjuk:

```
if (v[j+gap]<v[j]) swap(v[j],v[j+gap]);
```

Ez a kód hatékonyságát semmilyen módon nem rontja. Ebben a példában a < műveletet használtuk összehasonlításra. Nem minden típusnak van azonban < operátora, ami korlátozza a *sort()* ezen változatának használhatóságát; de ez a korlátozás könnyen megkerülhető (§13.4).

13.3.1. A függvénysablonok paraméterei

A függvénysablonok alapvető fontosságúak a tároló típusok (§2.7.2, §3.8, 18. fejezet) széles körére alkalmazható általános algoritmusok írásához. Alapvető jelentőségű, hogy egy függvényhíváskor a sablonparamétereket le lehet vezetni, ki lehet következtetni (deduce) a függvény paramétereiből.

A fordítóprogram akkor tudja levezetni egy hívás típusos és nem típusba tartozó paramétereit, ha a függvény paraméterlistája egyértelműen azonosítja a sablonparaméterek halmazát (§C.13.4):

```
template<class T, int i> T& lookup(Buffer<T,i>& b, const char* p);

class Record {
    const char[12];
    // ...
};
```

```
Record& f(Buffer<Record,128>& buf, const char* p)
{
    return lookup(buf,p);          // lookup() használata, ahol T egy Record és i értéke 128
}
```

Itt *T*-ről azt állapítja meg a fordítóprogram, hogy *Record*, az *i*-ről pedig azt, hogy értéke *128*.

Megjegyzendő, hogy a fordítóprogram az osztálysablonok paramétereit soha nem vezeti le (§C.13.4). Ennek az az oka, hogy az osztályok többféle konstruktora nyújtotta rugalmasság ezt sok esetben megakadályozná, esetleg áttekinthetetlenné tenné. Egy osztály különféle változatai közötti választásra a specializált változatok használata ad eszközt (§13.5). Ha egy levezett típusú objektumot kell létrehoznunk, ezt sokszor megtehetjük úgy, hogy a létrehozást egy függvény meghívásával hajtatjuk végre (lásd a §17.4.1.2 pontbeli *make_pair()*-t).

Ha egy paramétert nem lehet levezetni a sablon függvény paramétereiből (§C.13.4), akkor közvetlenül meg kell adnunk. Ezt ugyanúgy tehetjük meg, mint ahogy egy sablon osztály számára közvetlenül megadjuk a sablonparamétereket:

```
template<class T> class vector { /* ... */ };
template<class T> T* create();        // T létrehozása és rá hivatkozó mutató visszaadása

void f()
{
    vector<int> v;                  // osztály, sablonparamétere 'int'
    int* p = create<int>();         // függvény, sablonparamétere 'int'
}
```

A közvetlen meghatározás (explicit specification) egyik szokásos használata a sablon függvény visszatérésiérték-típusának megadása:

```
template<class T, class U> T implicit_cast(U u) { return u; }

void g(int i)
{
    implicit_cast(i);                       // hiba: T nem vezethető le
    implicit_cast<double>(i);               // T típusa double, U típusa int

    implicit_cast<char,double>(i);          // T típusa char, U típusa double
    implicit_cast<char*,int>(i);            // T típusa char*, U típusa int; hiba: int
                                            // nem alakítható char*-ra
}
```

Az alapértelmezett függvényparaméter-értékekhez hasonlóan (§7.5), az explicit megadott sablonparaméterek közül is csak az utolsókat lehet elhagyni.

A sablonparaméterek közvetlen megadása átalakító függvénycsaládok és objektum-létrehozó függvények definicióját teszi lehetővé (§13.3.2, §C.13.1, §C.13.5). Az automatikus (implicit) konverziók (§C.6) explicit változatai, például az *implicit_cast()* időnként igen hasznosak lehetnek. A *dynamic_cast*, *static_cast* stb. formai követelményei megfelelnek az explicit minősítésű sablon függvényekéinek. A beépített típuskonverziós operátorok azonban olyan műveleteket támogatnak, amelyeket nem fejezhetünk ki más nyelvi elemmel.

13.3.2. Függvénysablonok túlterhelése

Azonos néven több függvénysablon is szerepelhet, sőt ugyanolyan néven több közönséges függvény is. A túlterhelt (vagyis azonos névvel mást és mást jelentő) függvények meghívásakor a megfelelő meghívandó függvény vagy függvénysablon kiválasztásához a túlterhelés (overloading) feloldása szükséges:

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
    sqrt(2);        // sqrt<int>(int)
    sqrt(2.0);      // sqrt(double)
    sqrt(z);        // sqrt<double>(complex<double>)
}
```

Ahogy a sablon függvény fogalma a függvény fogalmának általánosítása, ugyanúgy a sablon függvényekre alkalmazandó túlterhelés-feloldási szabályok is a függvényekre alkalmazandó túlterhelés-feloldási szabályok általánosításai. A módszer alapvetően a következő: megkeressük minden sablonhoz azt a specializált változatot, amelyik a paramétereknek a legjobban megfelel. Ezután ezekre a példányokra és az összes közönséges függvényre is a szokásos túlterhelés-feloldási szabályokat alkalmazzuk:

1. Meg kell keresni azokat a specializált sablon függvény változatokat (§13.2.2), amelyek részt fognak venni a túlterhelés feloldásában. Ehhez az összes függvénysablont megvizsgáljuk, hogy ha más ugyanilyen nevű függvény vagy sablon függvény nem lenne elérhető, akkor lehetne-e valamilyen sablonparaméterrel alkalmazni. Az *sqrt(z)* hívás esetében például a következő jelöltek adódnak: *sqrt<double>(complex<double>)* és *sqrt<complex<double>>(complex<double>)*.
2. Ha két sablon függvény is meghívható lenne és az egyik specializáltabb a másiknál (§13.5.1), akkor a következő lépésekben csak azt vesszük figyelembe.

Az *sqrt(z)* hívás esetében az *sqrt<double>(complex<double>)*-t választjuk az *sqrt<complex<double>>(complex<double>)* helyett: minden hívás, ami megfelel *sqrt<T>(complex<T>)*-nek, megfelel *sqrt<T>(T)*-nek is.

3. Ezek után végezzük el a közönséges túlterhelés-feloldást ezen függvényekre és a közönséges függvényekre (§7.4.). Ha egy sablon függvény paraméterét a sablonparaméterekből vezettük le (§13.3.1), akkor arra nem alkalmazhatunk kiterjesztést (promotion), illetve szabványos vagy felhasználói konverziót. Az *sqrt(2)* hívás pontosan megfelel az *sqrt<int>(int)*-nek, így azt választjuk a *sqrt(double)* helyett.
4. Ha egy függvény és egy specializált változata ugyanolyan mértékben megfelelő, akkor a függvényt választjuk. Emiatt a *sqrt(2.0)*-hoz a *sqrt(double)*-t választjuk, és nem a *sqrt<double>(double)*-t.
5. Ha nem találunk megfelelő függvényt, akkor a hívás hibás. Ha több ugyanolyan mértékben megfelelő függvényt is találunk, akkor a hívás többértelmű és ezért hibás:

```
template<class T> T max(T,T);

const int s = 7;

void k()
{
    max(1,2);           // max<int>(1,2)
    max('a','b');       // max<char>('a','b')
    max(2.7,4.9);       // max<double>(2.7,4.9)
    max(s,7);           // max<int>(int(s),7) (egyszerű konverzió)

    max('a',1);         // hiba: többértelmű (nincs szabványos konverzió)
    max(2.7,4);         // hiba: többértelmű (nincs szabványos konverzió)
}
```

A fenti példa két nem egyértelmű hívását explicit minősítéssel oldhatjuk fel:

```
void f()
{
    max<int>('a',1);            // max<int>(int('a'),1)
    max<double>(2.7,4);         // max<double>(2.7,double(4))
}
```

Vagy megfelelő deklarációk alkalmazásával:

```
inline int max(int i, int j) { return max<int>(i,j); }
inline double max(int i, double d) { return max<double>(i,d); }
inline double max(double d, int i) { return max<double>(d,i); }
inline double max(double d1, double d2) { return max<double>(d1,d2); }
```

```
void g()
{
    max('a',1);    // max(int('a'),1)
    max(2.7,4);    // max(2.7,double(4))
}
```

Közönséges függvényekre a közönséges túlterhelés-feloldási szabályok érvényesek (§7.4), és a helyben kifejtés (inline) biztosítja, hogy a hívás nem jár külön „költséggel".

A *max()* függvény igen egyszerű, így explicit módon is írhattuk volna, de a sablon specializált használata könnyű és általánosan használható módja az ilyen túlterhelés-feloldó függvények írásának. A túlterhelés-feloldási szabályok biztosítják, hogy a sablon függvények helyesen működnek együtt az öröklődéssel:

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
    f(pb);  // f<int>(pb)
    f(pd);  // f<int>(static_cast<B<int>*>(pd)); szabványos átalakítás D<int>*-ról B<int>*-ra
}
```

Ebben a példában az *f()* sablon függvény minden *T* típusra elfogadja *B<T>**-ot. Egy *D<int>** típusú paraméterünk van, így a fordítóprogram könnyen jut arra a következtetésre, hogy *T*-t *int*-nek véve a hívás egyértelműen feloldható, *f(B<int>*)*-ként. Az olyan függvényparamétereket, amelyek nem vesznek részt a sablonparaméter levezetésében, pontosan úgy kezelhetjük, mint egy nem sablon függvény paraméterét, így a szokásos átkonverziók megengedettek:

```
template<class T, class C> T get_nth(C& p, int n);  // az n-edik elem
```

Ez a függvény feltételezhetően a *C* típusú tároló *n*-edik elemét adja vissza. Minthogy *C*-t a hívás aktuális paraméteréből kell levezetni, az első paraméterre nem alkalmazható konverzió, a második paraméter azonban teljesen közönséges, így a szokásos konverziók mindegyike tekintetbe vehető:

```
class Index {
public:
    operator int();
    // ...
};
```

```
void f(vector<int>& v, short s, Index i)
{
    int i1 = get_nth<int>(v,2);        // pontos illeszkedés
    int i2 = get_nth<int>(v,s);        // szabványos konverzió: short-ról int-re
    int i3 = get_nth<int>(v,i);        // felhasználói konverzió: Index-ről int-re
}
```

13.4. Eljárásmód megadása sablonparaméterekkel

Gondoljuk meg, hogyan rendezhetjük a karakterláncokat. Három dolog játszik szerepet: a karakterlánc, az elemek típusa és a lánc elemeinek összehasonlításakor alkalmazott szempont.

Nem „betonozhatjuk be" a rendezési elvet a tárolóba, mert az általában nem szabhatja meg, mire van szüksége az elemek típusával kapcsolatban, de az elemek típusába sem, mert az elemeket sokféle módon rendezhetjük. Ehelyett a megfelelő művelet végrehajtásakor kell megadni az alkalmazandó feltételeket. Milyen rendezési elvet alkalmazzunk, ha például svéd neveket tartalmazó karakterláncokat akarunk rendezni? A svéd nevek rendezése számára a karakterek két különböző numerikus megfeleltetési módja (collating sequence) használatos. Természetesen sem egy általános *string* típus, sem egy általános rendező algoritmus nem tudhat a nevek rendezésének „svéd szokásairól", ezért bármely általános megoldás megköveteli, hogy a rendező eljárást ne csak egy adott típusra adhassuk meg, hanem adott típusra való adott alkalmazáskor is. Általánosítsuk például a C standard könyvtárának *strcmp()* függvényét tetszőleges *T* típusból álló *String*-ekre (§13.2):

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i< str2.length(); i++)
        if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;
    return str1.length()-str2.length();
}
```

Ha valaki azt szeretné, hogy a *compare()* eltekintsen a kis- és nagybetűk közötti különbségtől vagy figyelembe vegye a program nyelvi környezetét (locale, helyi sajátosságok), akkor ezt a *C::eq()* és a *C::lt()* függvények megfelelő definiálásával teheti meg. Ezzel minden (összehasonlító, rendező stb.) eljárást leírhatunk, ha az a tároló és a „C-műveletek" nyelvén megfogalmazható:

```
template<class T> class Cmp {          // szokásos, alapértelmezett összehasonlítás
public:
    static int eq(T a, T b) { return a==b; }
    static int lt(T a, T b) { return a<b; }
};

class Literate {                       // svéd nevek összehasonlítása
public:
    static int eq(char a, char b) { return a==b; }
    static int lt(char,char);          // kikeresés táblázatból karakterérték alapján (§13.9[14])
};
```

A sablonparaméterek megadásakor most már pontosan megadhatjuk az összehasonlítási szabályokat:

```
void f(String<char> swede1, String<char> swede2)
{
    compare< char,Cmp<char> >(swede1,swede2);
    compare< char,Literate >(swede1,swede2);
}
```

Az összehasonlító műveletek sablonparaméterként való megadásának két jelentős előnye van az egyéb lehetőségekhez, például a függvénymutatók alkalmazásához képest. Egyrészt több művelet megadható egyetlen paraméterként, a futási idő növekedése nélkül. Másrészt, az *eq()* és az *lt()* összehasonlító műveleteket könnyű helyben kifejtve (inline) fordítani, míg egy függvénymutatón keresztüli hívás ilyen módú fordítása különleges mértékű figyelmet követel a fordítóprogramtól.

Természetesen összehasonlító műveleteket nemcsak a beépített, hanem a felhasználói típusokra is megadhatunk. Ez alapvető fontosságú feltétele annak, hogy általános algoritmusokat olyan típusokra alkalmazhassunk, amelyeknek nem maguktól értetődő összehasonlítási feltételeik vannak (§18.4).

Minden osztálysablonból létrehozott osztály saját példányokat kap a sablon statikus változóiból (§C.13.1).

13.4.1. Alapértelmezett sablonparaméterek

Fáradságos dolog minden egyes hívásnál közvetlenül meghatározni az összehasonlítási feltételeket. Túlterheléssel szerencsére könnyen megadhatunk olyan alapértelmezést, hogy csak a szokásostól eltérő összehasonlítási szempontot kelljen megadni:

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2);        // összehasonlítás C
                                                                  // használatával

template<class T>
int compare(const String<T>& str1, const String<T>& str2);        // összehasonlítás
                                                                  // Cmp<T> használatával
```

De a szokásos rendezést megadhatjuk, mint alapértelmezett sablonparaméter-értéket is:

```
template<class T, class C = Cmp<T> >
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i< str2.length(); i++)
            if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;
    return str1.length()-str2.length();
}
```

Így már leírhatjuk a következőt:

```
void f(String<char> swede1, String<char> swede2)
{
    compare(swede1,swede2);                          // Cmp<char> használata
    compare<char,Literate>(swede1,swede2);           // Literate használata
}
```

Egy (nem svédek számára) kevésbé ezoterikus példa a kis- és nagybetűk közötti különbséget figyelembe vevő, illetve elhanyagoló rendezés:

```
class No_case { /* ... */ };

void f(String<char> s1, String<char> s2)
{
    compare(s1,s2);                        // kisbetű-nagybetű különbözik
    compare<char,No_case>(s1,s2);          // kisbetű-nagybetű nem különbözik
}
```

A standard könyvtár széles körben alkalmazza azt a módszert, hogy az alkalmazandó *eljárásmódot* (policy) egy sablonparaméter adja meg, és ennek a legáltalánosabb eljárásmód az alapértelmezett értéke (például §18.4). Eléggé furcsa módon azonban a *basic_string* (§13.2, 20. fejezet) összehasonlításaira ez nem áll. Az eljárásmódot kifejező sablonparamétereket gyakran nevezik „jellemvonásoknak" (traits) is. Például a standard könyvtárbeli *string* a *char_traits*-re épül (§20.2.1), a szabványos algoritmusok a bejárók (iterátorok) jellemvonásait (§19.2.2), a standard könyvtárbeli tárolók pedig a memóriafoglalókét (allokátor, §19.4.) használják fel.

Egy alapértelmezett sablonparaméter értelmi ellenőrzése ott és csak akkor történik meg, ahol és amikor az alapértelmezett paramétert ténylegesen felhasználjuk. Így ha nem használjuk fel az alapértelmezett *Cmp<T>* paramétert, akkor olyan *X* típusokra is használhatjuk a *compare()*-t, amelyekre a fordító nem fordítaná le a *Cmp<X>*-et, mert mondjuk az *X*-re a < nem értelmezett. Ez döntő jelentőségű a szabványos tárolók tervezésénél, hiszen ezek sablonparamétert használnak az alapértelmezett értékek megadására (§16.3.4).

13.5. Specializáció

Alapértelmezés szerint egy sablon (template) egyetlen definíciót ad a felhasználható által elképzelhető összes paraméterérték (vagy paraméterértékek) számára. Ez azonban nem minden sablon írásakor kedvező. Előfordulhat, hogy olyasmit szeretnénk kifejezni, hogy „ha a sablonparaméter egy mutató, használd ezt, ha nem, használd azt”, vagy hogy „hiba, ha a sablonparaméter nem a *My_base* osztály egy leszármazottjára hivatkozó mutató”. Sok hasonló tervezési szempontot figyelembe lehet úgy venni, hogy a sablonnak többféle definíciót adunk és a fordítóprogram az alkalmazott paramétertípusok szerint választ közülük. A sablon ilyenféle többszörös meghatározását *specializációnak* (specialization, egyedi célú felhasználói változatok használata, szakosítás) hívjuk.

Vegyük egy *Vector* sablon valószínű felhasználásait:

```
template<class T> class Vector {        // általános vektortípus
     T* v;
     int sz;
public:
     Vector();
     Vector(int);

     T& elem(int i) { return v[i]; }
     T& operator[](int i);

     void swap(Vector&);
     // ...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

A legtöbb *Vector* valamilyen mutatótípus *Vector*-a lesz. Több okból is, de főleg azért, mert a többalakú (polymorph) viselkedés megőrzése céljából mutatókat kell használnunk (§2.5.4, §12.2.6). Ezért aki objektumorientált programozást folytat és típusbiztos, például standard könyvtárbeli tárolókat használ, az biztosan számos mutatót tartalmazó tárolót fog használni.

A legtöbb C++-változat alapértelmezés szerint lemásolja a sablon függvények kódját. Ez jó a végrehajtási sebesség szempontjából, de kritikus esetekben (mint az iménti *Vector*-nál) a kód „felfúvódásával” jár.

Szerencsére létezik egyszerű megoldás. A mutatókat tartalmazó tárolóknak elég egyetlen megvalósítás. Ezt specializált változat készítésével érhetjük el. Először definiáljuk a *Vector*-nak a *void* mutatókra vonatkozó változatát („specializációját”):

```
template<> class Vector<void*> {
    void** p;
    // ...
    void*& operator[](int i);
};
```

Ezt a változatot aztán az összes, mutatót tartalmazó vektor közös megvalósításaként használhatjuk. A *template<>* előtag azt jelenti, hogy ennél a specializált változatnál nem kell sablonparamétert megadnunk. Azt, hogy milyen típusú sablonparaméterre használjuk, a név utáni <> jelpár között adjuk meg: vagyis a *<void*>* azt jelenti, hogy ezt a definíciót kell minden olyan *Vector* esetében használni, amelyikre a *T* típusa *void**.

A *Vector<void*>* egy teljes specializáció, azaz ezen változat használatakor nincs sablonparaméter, amit meg kellene adni vagy le kellene vezetni; a *Vector<void*>*-ot a következő módon deklarált *Vector*-ok számára használjuk:

```
Vector<void*> vpv;
```

Ha olyan változatot akarunk megadni, ami mutatókat tartalmazó *Vector*-ok, és csak azok esetén használandó, részleges specializációra van szükségünk:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;

    Vector() : Base() {}
    explicit Vector(int i) : Base(i) {}
```

```
    T*& elem(int i) { return reinterpret_cast<T*&>(Base::elem(i)); }
    T*& operator[ ](int i) { return reinterpret_cast<T*&>(Base::operator[ ](i)); }

    // ...
};
```

A név utáni *<T*>* specializáló minta azt jelzi, hogy ezt a változatot kell minden mutatótípus esetében használni; azaz minden olyan sablonparaméternél, ami *T** alakba írható:

```
Vector<Shape*> vps;     // <T*> most <Shape*>, így T is Shape
Vector<int**> vppi;     // <T*> most <int**>, így T is int*
```

Jegyezzük meg, hogy részleges specializáció használata esetén a sablonparaméter a specializációra használt mintából adódik; a sablonparaméter nem egyszerűen az aktuális sablonparaméter. Így például a *Vector<Shape*>* esetében *T* típusa *Shape* és nem *Shape**.

Ha adott a *Vector* ezen részlegesen specializált változata, akkor ez az összes mutatótípusra vonatkozó *Vector* közös megvalósítása. A *Vector<T*>* osztály egyszerűen egy felület a *void**-os változathoz, melyet kizárólag az öröklődés és a helyben kifejtés eszközével valósítottuk meg.

Fontos, hogy a *Vector* ezen finomítása a felhasználói felület megváltoztatása nélkül történt. A specializáció a közös felület többféle meghatározásának eszköze. Természetesen az általános *Vector*-t és a mutatókra vonatkozó változatot hívhattuk volna különbözőképpen is. Amikor ezt kipróbáltam, kiderült, hogy sok felhasználó, akinek tudnia kellett volna róla, mégsem a mutatós változatot használta és a kapott kód a vártnál sokkal nagyobb lett. Ebben az esetben sokkal jobb a fontos részleteket egy közös felület mögé rejteni.

Ez a módszer a gyakorlatban a kód felfúvódásának megakadályozásában volt sikeres. Akik nem alkalmaznak ilyen módszereket (akár a C++-ban, akár egyéb típus-paraméterezési lehetőségeket tartalmazó nyelvekben), könnyen azt vehetik észre, hogy az ismétlődő kód közepes méretű programok esetében is megabájtokra rúghat. A vektorműveletek különböző változatainak lefordításához szükséges idő megtakarításával ez a módszer a fordítási és szerkesztési időt is drámai módon csökkenti. Az összes mutatót tartalmazó lista egyetlen specializált változattal való megvalósítása jó példa arra, hogyan lehet a kódfelfúvódást megakadályozni úgy, hogy a közös kódot a lehető legjobban növeljük.

Az általános sablont az összes specializált változat előtt kell megadni:

```
template<class T> class List<T*> { /* ... */ };

template<class T> class List { /* ... */ };        // hiba: általános sablon a specializált után
```

Az általános sablon által adott létfontosságú információ az, hogy milyen paramétereket kell a felhasználás vagy a specializáció során megadni. Ezért elég az általános sablont a specializált változat deklarációja vagy definiciója előtt megadni:

```
template<class T> class List;

template<class T> class List<T*> { /* ... */ };
```

Ha használjuk is, akkor az általános sablont valahol definiálnunk kell (§13.7).

Ha valahol szerepel egy felhasználói specializált változat, akkor annak deklarációja a specializált használat minden helyéről elérhető kell, hogy legyen:

```
template<class T> class List { /* ... */ };

List<int*> li;

template<class T> class List<T*> { /* ... */ };          // hiba
```

Itt a *List*-et az *int**-ra a *List<int*>* használata után specializálunk.

Egy sablon minden specializált változatát ugyanabban a névtérben kell megadni, mint magát a sablont. Ha használják, akkor egy explicit deklarált (azaz nem egy általánosabból létrehozott) sablonnak valahol szintén explicit definiáltnak kell lennie (§13.7). Vagyis a specializált változat explicit megadásából következik, hogy számára a fordító nem hoz létre definíciót.

13.5.1. Specializációk sorrendje

Az egyik változat specializáltabb egy másiknál, ha minden olyan paraméterlista, amely illeszkedik az egyikre, illeszkedik a másikra is. Ez fordítva nem áll fenn:

```
template<class T> class Vector;              // általános
template<class T> class Vector<T*>;          // mutatókhoz
template<> class Vector<void*>;              // void*-okhoz
```

Minden típust használhatunk a legáltalánosabb *Vector* paramétereként, de a *Vector<T*>* paramétereként csak mutatót, a *Vector<void*>* paramétereként pedig csak *void** mutatót. Az objektumok és mutatók stb. (§13.5) deklarációjában, illetve a túlterhelés feloldásakor (§13.3.2) a leginkább specializált változat részesül előnyben.

A specializáló minta megadásánál a sablonparaméterek levezetésénél (§13.3.1) használt típusokból összeállított típusok használhatók fel.

13.5.2. Függvénysablon specializáció

A specializáció természetesen a sablon függvényeknél (template function) is hasznos. Vegyük a §7.7 és §13.3 példabeli Shell rendezést. Ez az elemeket a < segítségével hasonlítja össze és a részletezett kód segítségével cseréli fel. Jobb definíció lenne a következő:

```
template<class T> bool less(T a, T b) { return a<b; }

template<class T> void sort(Vector<T>& v)
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (less(v[j+gap],v[j])) swap(v[j],v[j+gap]);
}
```

Ez nem javítja magát az algoritmust, de lehetőséget nyújt a megvalósítás javítására. Eredeti formájában egy *Vector<char*>*-ot nem rendez jól, mert két *char**-ot a < segítségével hasonlít össze, azaz az első karakter címét fogja az összehasonlítás alapjául venni. Ehelyett a mutatott karakterek szerinti összehasonlítást szeretnénk. Erre a *less()*-nek egy egyszerű, *const char**-ra vonatkozó specializációja fog ügyelni:

```
template<> bool less<const char*>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

Mint az osztályoknál is (§13.5), a <> sablon-előtag azt jelzi, hogy ez egy sablonparaméter megadása nélküli specializált változat. A sablon függvény neve utáni *<const char*>* azt jelenti, hogy a függvényt azokban az esetekben kell alkalmazni, amikor a sablonparaméter *const char**. Minthogy a sablonparaméter közvetlenül levezethető a függvény paraméterlistájából, nem kell explicit megadnunk. Így egyszerűsíthetünk a specializáció megadásán:

```
template<> bool less<>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

Minthogy adott a *template<>* előtag, a második, üres <> felesleges, hiszen nem hordoz új információt. Így aztán általában így írnánk:

```
template<> bool less(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

Én jobban kedvelem ezt a rövidebb deklarációs formát.

Vegyük a *swap()* kézenfekvő meghatározását:

```
template<class T> void swap(T& x, T& y)
{
    T t = x;        // x másolása az ideiglenes változóba
    x = y;          // y másolása x-be
    y = t;          // ideiglenes változó másolása y-ba
}
```

Ez nem túl hatékony, ha *Vector*-ok vektoraira hívjuk meg; az összes elem másolásával cseréli fel a *Vector*-okat. De ezt is megoldhatjuk megfelelő specializációval. Maga a *Vector* objektum csak annyi adatot tartalmaz, hogy az elemeihez való közvetett hozzáférést támogassa (mint a *string* §11.12, §13.2). Így aztán a cserét a megfelelő ábrázoló adatok cseréjével lehet megoldani. Hogy lehetővé tegyük az ábrázoló adatok kezelését, adjunk meg egy *swap()* függvényt a *Vector* osztály számára (§13.5):

```
template<class T> void Vector<T>::swap(Vector & a)      // ábrázolások cseréje
{
    swap(v,a.v);
    swap(sz,a.sz);

}
```

A *swap()* tagot felhasználhatjuk az általános *swap()* specializált definiciójában:

```
template<class T> void swap(Vector<T>& a, Vector<T>& b)
{
    a.swap(b);
}
```

A *less()* és a *swap()* ezen változatait használja a standard könyvtár (§16.3.9, §20.3.16) is. Ráadásul ezek a példák széles körben használt módszereket mutatnak be. A specializáció akkor hasznos, ha a sablonparaméterek egy adott halmazára egy általános algoritmusnál létezik hatékonyabb megvalósítás (itt a *swap()*). Ezenkívül akkor is hasznos, ha a paramétertípus valamilyen szabálytalansága miatt a szabványos algoritmus valamilyen nem kívánt módon működne (mint a *less()* esetében). Ezek a „szabálytalan” típusok többnyire a beépített mutató- és tömbtípusok.

13.6. Öröklődés és sablonok

Az öröklődés és a sablonok olyan eszközök, amelyekkel meglevők alapján új típusok építhetők, és amelyekkel általánosságban a közös vonások különféle kihasználásával hasznos kód írható. Mint a §3.7.1, §3.8.5 és §13.5 pontokban láttuk, ezen eszközök párosítása sok hasznos módszer alapja.

Egy sablon osztálynak (template class) egy nem sablon osztályból való származtatása módot nyújt arra, hogy sablonok egy halmaza közös megvalósításon osztozzék. A §13.5 pontbeli vektor jó példa erre:

```
template<class T> class List<T*> : private List<void*> { /* ... */ };
```

Másként nézve a példa azt mutatja, hogy a sablon elegáns és típusbiztos felületet ad egy másként nem biztonságosan és nem elegánsan használható eszközhöz.

Természetesen a sablon osztályok közötti öröklődés is hasznos. A bázisosztály egyik haszna az, hogy további osztályok számára építőkőül szolgál. Ha a bázisosztály adatai vagy műveletei valamely származtatott osztály sablonparaméterétől függenek, akkor magának a bázisosztálynak is paramétereket kell adni (lásd például a §3.7.2 pontbeli *Vec* osztályt):

```
template<class T> class vector { /* ... */ };
template<class T> class Vec : public vector<T> { /* ... */ };
```

A sablon függvények túlterhelés-feloldási szabályai biztosítják a függvények helyes működését az ilyen származtatott típusokra (§13.3.2).

Az az eset a leggyakoribb, amikor ugyanaz a sablonparaméter szerepel a bázis- és a származtatott osztályban is, de ez nem szükségszerű. Érdekes, bár ritkábban alkalmazott módszerek alapulnak azon, hogy magát a származtatott osztályt adjuk át a bázisosztálynak:

```
template <class C> class Basic_ops {            // tárolók alapműveletei
public:
    bool operator==(const C&) const;            // minden elem összehasonlítása
    bool operator!=(const C&) const;
    // ...
    // hozzáférés biztosítása C műveleteihez
    const C& derived() const { return static_cast<const C&>(*this); }
};
```

```
template<class T> class Math_container : public Basic_ops< Math_container<T> > {
public:
    size_t size() const;
    T& operator[ ](size_t);
    const T& operator[ ](size_t) const;
    // ...
};
```

Ezáltal lehetővé válik a tárolók alapvető műveleteinek az egyes tárolóktól elkülönített, egyszeri meghatározása. Ám mivel az olyan műveletek definíciójához, mint az == és a != mind a tárolónak, mind annak elemeinek típusára szükség van, a bázisosztályt át kell adni a tárolósablonnak.

Ha feltesszük, hogy a *Math_container* egy hagyományos vektorra hasonlít, akkor a *Basic_ops* egy tagja valahogy így nézhet ki:

```
template <class C> bool Basic_ops<C>::operator==(const C& a) const
{
    if (derived().size() != a.size()) return false;
    for (int i = 0; i<derived().size(); ++i)
            if (derived()[i] != a[i]) return false;
    return true;
}
```

A tárolók és a műveletek elválasztására szolgáló másik módszer öröklődés helyett sablonparaméterekkel kapcsolja össze azokat:

```
template<class T, class C> class Mcontainer {
    C elements;

public:
    T& operator[ ](size_t i) { return elements[i]; }
..  // ...

    friend bool operator==(const Mcontainer&, const Mcontainer&);    // elemek
                                                                      // összehasonlítása
    friend bool operator!=(const Mcontainer&, const Mcontainer&);
    // ...
};

template<class T> class My_array { /* ... */ };

Mcontainer< double,My_array<double> > mc;
```

Egy sablonból létrehozott osztály teljesen közönséges osztály, ezért lehetnek barát (friend) függvényei (§C.13.2) is. Ebben a példában azért használtam barátokat, hogy a == és a != számára a két paraméter szokásos felcserélhetőségét biztosítsam (§11.3.2). Ilyen esetekben meg lehetne fontolni azt is, hogy a *C* paraméterként tároló helyett sablont használjunk.

13.6.1. Paraméterezés és öröklődés

A sablonok egy típust vagy függvényt egy másik típussal paramétereznek. A sablon kódja minden paramétertípus esetében azonos, csakúgy, mint a sablont használó legtöbb kódrész. Az absztrakt osztályok felületeket írnak le, különféle megvalósításaik kódrészeit az osztályhierarchiák közösen használhatják, és az absztrakt osztályt használó kód legnagyobb része nem is függ a megvalósítástól. A tervezés szempontjából a két megközelítés annyira közeli, hogy közös nevet érdemel. Minthogy mindkettő azt teszi lehetővé, hogy egy algoritmust egyszer írjunk le és aztán különféle típusokra alkalmazzuk, néha mindkettőt ***többalakúságnak*** (polimorfizmus, polymorphism) hívják. Hogy megkülönböztessük őket, a virtuális függvények által biztosítottat ***futási idejű*** (run-time) ***többalakúságnak***, a sablonok által nyújtottat pedig ***fordítási idejű*** (compile-time) ***többalakúságnak*** vagy ***paraméteres*** (parametric) ***többalakúságnak*** hívják.

Végül is mikor válasszunk absztrakt osztályt és mikor alkalmazzunk sablont? Mindkét esetben olyan objektumokat kezelünk, amelyeknek azonos műveleteik vannak. Ha nem szükséges egymással alá- és fölérendeltségi viszonyban állniuk, akkor legyenek sablonparaméterek. Ha az objektumok aktuális típusa a fordítási időben nem ismert, akkor legjobban egy közös absztrakt osztályból öröklődő osztályként ábrázolhatjuk azokat. Ha a futási idő nagyon fontos, azaz a műveletek helyben kifejtve történő fordíthatósága alapvető szempont, használjunk sablont. Erről részletesebben a §24.4.1 pont ír.

13.6.2. Tag sablonok

Egy osztálynak vagy osztálysablonnak lehetnek olyan tagjai is, amelyek maguk is sablonok:

```
template<class Scalar> class complex {
    Scalar re, im;
public:
    template<class T>
        complex(const complex<T>& c) : re(c.real()), im(c.imag()) { }
    // ...
};
```

```
complex<float> cf(0,0);
complex<double> cd = cf;          // rendben: float-ról double-ra alakítás használata
class Quad {
                                  // nincs átalakítás int-re
};

complex<Quad> cq;
complex<int> ci = cq;             // hiba: nincs átalakítás Quad-ról int-re
```

Vagyis kizárólag akkor tudunk *complex<T2>*-ből *complex<T1>*-et építeni, ha *T1*-nek kezdőértékül adhatjuk *T2*-t, ami ésszerű megszorításnak tűnik.

Sajnos azonban a C++ nyelv elfogad a beépített típusok közötti bizonyos ésszerűtlen konverziókat is, például *double*-ról *int*-re. A végrehajtási időben a csonkításból eredő hibákat el lehetne kapni, *implicit_cast* (§13.3.1) vagy *checked* (§C.6.2.6.2) stílusú ellenőrzött konverzióval:

```
template<class Scalar> class complex {
    Scalar re, im;
public:
    complex() : re(0), im(0) { }
    complex(const complex<Scalar>& c) : re(c.real()), im(c.imag()) { }

    template<class T2> complex(const complex<T2>& c)
        : re(checked_cast<Scalar>(c.real())), im(checked_cast<Scalar>(c.imag())) { }
    // ...
};
```

A teljesség kedvéért alapértelmezett és másoló konstruktort is megadtam. Elég különös módon egy sablon konstruktorból a fordító soha nem hoz létre másoló konstruktort, így közvetlenül deklarált másoló konstruktor híján a fordító alapértelmezett konstruktort hozott volna létre, ami ebben az esetben azonos lett volna az általam megadottal.

Egy sablon tag nem lehet virtuális:

```
class Shape {
    // ...
    template<class T> virtual bool intersect(const T&) const =0;    // hiba: virtuális sablon
};
```

Ez szabálytalan. Ha megengedett lenne, akkor a virtuális függvények megvalósításának hagyományos virtuálisfüggvény-táblás módja (§2.5.5) nem lenne alkalmazható. A szerkesztőnek az *intersect()* függvény minden új paramétertípussal történő meghívása esetén egy újabb elemmel kellene bővítenie a *Shape* osztály virtuálisfüggvény-tábláját.

13.6.3. Öröklődési viszonyok

Általában úgy gondolnunk egy sablonra, mint új típusok létrehozását segítő általánosításra. Más szóval a sablon egy olyan eszköz, amely szükség esetén a felhasználó előírásai szerinti típusokat hoz létre. Ezért az osztálysablonokat néha *típuskészítőknek* vagy típusgenerátoroknak hívják.

A C++ nyelv szabályai szerint két, azonos osztálysablonból létrehozott osztály nem áll rokonságban egymással:

```
class Shape { /* ... */ };
class Circle : public Shape { /* ... */ };
```

Ilyenkor egyesek megpróbálják a set<Circle*>-ot set<Shape*>-ként kezelni, ami hibás érvelésen nyugvó súlyos logikai hiba: az „egy *Circle* egyben *Shape* is, így a *Circle*-ök halmaza egyben *Shape*-ek halmaza is; tehát a *Circle*-ök halmazát *Shape*-ek halmazaként is kezelhetem" érvelés „tehát" pontja hibás. Oka, hogy a *Circle*-ök halmaza kizárólag *Circle* típusú objektumokat tartalmaz, míg a *Shape*-ek halmaza esetében ez egyáltalán nem biztos:

```
class Triangle : public Shape { /* ... */ };

void f(set<Shape*>& s)
{
    // ...
    s.insert(new Triangle());
    // ...
}

void g(set<Circle*>& s)
{
    f(s);    // típushiba: s típusa set<Circle*>, nem set<Shape*>
}
```

A fenti példát a fordítóprogram nem fogja lefordítani, mert nincs beépített konverzió set<Circle*>&-ről set<Shape*>&-re. Nagyon helyesen. Az a garancia, hogy a set<Circle*> elemei *Circle*-ök, lehetővé teszi, hogy az elemekre biztonságosan és hatékonyan végezzünk *Circle*-ökre jellemző műveleteket, például a sugár lekérdezését. Ha megengednénk a set<Circle*>-öknek set<Shape*>-ként való kezelését, akkor ez már nem lenne biztosított. Például az *f()* függvény egy *Triangle** elemet tesz set<Shape*> paraméterébe; ha a set<Shape*> egy set<Circle*> lehetne, akkor nem lenne többé igaz, hogy egy set<Circle*> csak *Circle**-okat tartalmaz.

13.6.3.1. Sablonok konverziója

Az előző példa azt mutatta be, miért nem lehet semmilyen alapértelmezett kapcsolat két, azonos osztálysablonból létrehozott osztály között. Egyes osztályoknál azonban mégiscsak szeretnénk ilyen kapcsolatot kifejezni. Például ha egy mutató sablont készítünk, szeretnénk tükrözni a mutatott objektumok közötti öröklődési viszonyokat. Tag sablonok (§13.6.2) igény esetén sokféle hasonló kapcsolatot kifejezhetnek:

```
template<class T> class Ptr {                           // mutató T-re
     T* p;
public:
     Ptr(T*);
     template<class T2> operator Ptr<T2> ();            // Ptr<T> konverziója Ptr<T2>-re
     // ...
};
```

Ezután szeretnénk a konverziós operátorokat úgy definiálni, hogy *Ptr*-jeinkre fennálljanak a beépített mutatóknál megszokott öröklődési kapcsolatok:

```
void f(Ptr<Circle> pc)
{
     Ptr<Shape> ps = pc;              // működnie kell
     Ptr<Circle> pc2 = ps;            // elvileg hibát eredményez
}
```

Azt szeretnénk, hogy az első kezdőérték-adás csak akkor legyen engedélyezett, ha a *Shape* valóban közvetett vagy közvetlen nyilvános bázisosztálya a *Circle*-nek. Általában úgy szeretnénk a konverziós operátorokat megadni, hogy a *Ptr<T>*-ről *Ptr<T2>*-re történő konverzió csak akkor legyen engedélyezett, ha egy *T2** típusú mutató értékül kaphat egy *T** típusú mutatót. Ezt így tehetjük meg:

```
template<class T>
     template<class T2>
          Ptr<T>::operator Ptr<T2> () { return Ptr<T2>(p); }
```

A ***return*** utasítást a fordító csak akkor fogadja el, ha *p* (ami *T** típusú) a *Ptr<T2>(T2*)* konstruktor paramétere lehet. Ezért ha a *T** automatikusan *T2**-ra konvertálható, a *Ptr<T>*-ről *Ptr<T2>*-re történő konverzió működni fog:

```
void f(Ptr<Circle> pc)
{
     Ptr<Shape> ps = pc;              // rendben: Circle* átalakítható Shape*-ra
     Ptr<Circle> pc2 = ps;            // hiba: Shape* nem alakítható Circle*-gá
}
```

Legyünk óvatosak és csak logikailag értelmes konverziókat definiáljunk.

Jegyezzük meg, hogy nem lehetséges egy sablonnak, illetve annak szintén sablon tagjának sablonparaméter-listáit egyesíteni:

```
template<class T, class T2>            // hiba
    Ptr<T>::operator Ptr<T2> () { return Ptr<T2>(p); }
```

13.7. A forráskód szerkezete

A sablonokat használó kód szerkezete alapvetően kétféle lehet:

1. A sablonok definícióját beépítjük (*#include*) egy fordítási egységbe, mielőtt használnánk azokat.
2. A használat előtt a sablonoknak csak a deklarációját emeljük be, definiciójukat külön fordítjuk.

Ezenkívül lehetséges, hogy a sablonfüggvényeket egy fordítási egységen belül először csak deklaráljuk, majd használjuk és csak végül definiáljuk.

Hogy lássuk a kétféle megközelítés közötti különbséget, vegyünk egy egyszerű példát:

```
#include<iostream>

template<class T> void out(const T& t) { std::cerr << t; }
```

Hívjuk ezt a fájlt *out.c*-nek és építsük be, valahányszor szükségünk van az *out()*-ra:

```
// user1.c:
    #include "out.c"
    // out() használata

// user2.c:
    #include "out.c"
    // out() használata
```

Vagyis az *out()*-ot és a hozzá szükséges valamennyi deklarációt több különböző fordítási egységbe is beépítjük. A fordítóprogram dolga, hogy csak akkor hozzon létre kódot, ha szükséges, és hogy a felesleges információk feldolgozását optimalizálja, ami azt is jelenti, hogy a sablon függvényeket ugyanúgy kezeli, mint a helyben kifejtett (inline) függvényeket.

Ezzel az a nyilvánvaló gond, hogy minden olyan információ, amelyre az *out()*-nak szüksége van, az *out()*-ot felhasználó valamennyi fordítási egységbe belekerül, és ilyen módon megnő a fordítóprogram által feldolgozandó adat mennyisége. Egy másik gond, hogy a felhasználók esetleg véletlenül „rákapnak" az eredetileg csak az *out()* definíciójához szükséges deklarációk használatára. Ezt a veszélyt névterek használatával, a makrók használatának elkerülésével és általában a beépítendő információ mennyiségének csökkentésével háríthatjuk el.

E gondolatmenet logikus folyománya a külön fordítás: ha a sablon nem épül be a felhasználói kódba, azok az elemek, melyektől függ, nem is befolyásolhatják azt. Így az eredeti *out.c* állományt két részre bontjuk:

```
// out.h:
    template<class T> void out(const T& t);

// out.c:
    #include<iostream>
    #include "out.h"

    export template<class T> void out(const T& t) { std::cerr << t; }
```

Az *out.c* fájl most az *out()* definiálásához szükséges összes információt tartalmazza, az *out.h* csak a meghívásához szükségeset. A felhasználó csak a deklarációt (vagyis a felületet) építi be:

```
// user1.c:
    #include "out.h"
    // out() használata

// user2.c:
    #include "out.h"
    // out() használata
```

Ezen a módon a sablon függvényeket a nem helyben kifejtett függvényekhez hasonlóan kezeljük. Az *out.c*-beli definíciót külön fordítjuk, és az adott fordító dolga szükség esetén az *out()* leírásának megkeresése. Ez némi terhet ró a fordítóra, hiszen a sablon-meghatározás fölös példányainak kiszűrése helyett szükség esetén meg kell találnia az egyetlen definíciót.

Jegyezzük meg, hogy a sablon definíciója csak akkor érhető el más fordítási egységből, ha kifejezetten *export*-ként adjuk meg (§9.2.3). (Ez úgy történik, hogy a definícióhoz vagy egy megelőző deklarációhoz hozzáadjuk az *export* szót.) Ha nem így teszünk, a definíciónak minden használat helyéről elérhetőnek kell lennie.

A fordító- és szerkesztőprogramtól, a fejlesztett program fajtájától, illetve a fejlesztés külső feltételeitől függ, hogy melyik módszer vagy azok milyen párosítása a legjobb. Általában a helyben kifejtett függvényeket és az egyéb, alapvetően más sablon függvényeket hívó függvényeket érdemes minden olyan fordítási egységben elhelyezni, ahol felhasználják azokat. Egy, a sablon-példányosítás terén átlagos támogatást nyújtó szerkesztőprogram esetében ez meggyorsítja a fordítást és pontosabb hibaüzenetekhez vezet.

Ha a definíciót beépítjük, sebezhetővé tesszük azt, mert így értelmét a beépítés helyén érvényes makrók és deklarációk befolyásolhatják. Ezért a nagyobb vagy bonyolultabb függéseket igénylő sablonokat jobb külön fordíttatni, de akkor is ez az eljárás követendő, ha a sablon definiciója sok deklarációt igényel, mert ezek nem kívánatos mellékhatásokkal járhatnak a sablon felhasználásának helyén.

Én azt tekintem ideális megoldásnak, ha a sablondefiníciókat külön fordítjuk, a felhasználói kódban pedig csak deklarációjukat szerepeltetjük. Ezen elvek alkalmazását azonban mindig az adott helyzethez kell igazítanunk, a sablonok külön fordítása pedig egyes nyelvi változatok esetében költséges mulatság lehet.

Bármelyik megközelítést válasszuk is, a nem helyben kifejtett statikus tagoknak (§C.13.1) csak egyetlen definiciója lehet, valamelyik fordítási egységben. Ebből következően ilyen tagokat lehetőleg ne használjunk olyan sablonoknál, amelyek sok fordítási egységben szerepelnek.

Fontos cél, hogy a kód ugyanúgy működjék, akár egyetlen egységben szerepel, akár külön fordított egységekbe elosztva. Ezt inkább úgy érhetjük el, hogy csökkentjük a definíció függését a környezetétől, nem pedig úgy, hogy a környezetből minél többet átviszünk a példányosítás folyamatába.

13.8. Tanácsok

[1] Olyan algoritmusok leírására, amelyek sokféle paramétertípusra alkalmazhatók, sablont használjunk. §13.3.
[2] A tárolókat sablonként készítsük el. §13.2.
[3] A tárolókat specializáljuk mutatótípusokra, hogy csökkentsük a kód méretét. §13.5.
[4] A specializáció előtt mindig adjuk meg a sablon általános formáját. §13.5.

[5] A specializációt deklaráljuk, mielőtt használnánk. §13.5.
[6] A sablonok függését a példányosítás módjától csökkentsük a lehető legkisebbre. §13.2.5, §C.13.8.
[7] Definiáljunk minden deklarált specializációt. §13.5.
[8] Gondoljuk meg, hogy a sablonnak nincs-e szüksége C stílusú karakterláncokra és tömbökre vonatkozó specializációkra. §13.5.2.
[9] Paraméterezzünk eljárásmód objektummal. §13.4.
[10] Specializáció és túlterhelés segítségével adjunk azonos felületet ugyanazon fogalom különböző típusokra vonatkozó megvalósításának. §13.5.
[11] Egyszerű esetekre vonatkozóan egyszerű felületet adjunk; a ritkább eseteket túlterheléssel és alapértelmezett paraméter-értékekkel kezeljük. §13.5, §13.4.
[12] Mielőtt sablonná általánosítanánk valamit, végezzünk hibakeresést egy konkrét példán. §13.2.1.
[13] Ne felejtsük el kitenni az *export* kulcsszót azoknál a definícióknál, amelyeket más fordítási egységből is el kell érni. §13.7.
[14] A nagy vagy nem magától értetődő környezeti függőségű sablonokat külön fordítási egységben helyezzük el. §13.7.
[15] Konverziókat sablonokkal fejezzünk ki, de ezeket nagyon óvatosan definiáljuk. §13.6.3.1.
[16] Szükség esetén egy *constraint()* függvény segítségével korlátozzuk, milyen paraméterei lehetnek a sablonnak. §13.9[16], §C.13.10.
[17] A fordítási és összeszerkesztési idővel való takarékoskodás céljából explicit példányosítást használjunk. §C.13.10.
[18] Ha a futási idő döntő szempont, öröklődés helyett használjunk sablonokat. §13.6.1.
[19] Ha fontos szempont, hogy új változatokat újrafordítás nélkül vezethessünk be, sablonok helyett használjunk öröklődést. §13.6.1.
[20] Ha nem lehet közös bázisosztályt megadni, öröklődés helyett használjunk sablonokat. §13.6.1.
[21] Ha olyan beépített típusokat és adatszerkezeteket kell használnunk, amelyeknek a korábbi változatokkal összeegyeztethetőnek kell maradniuk, öröklődés helyett használjunk sablonokat. §13.6.1.

13.9. Gyakorlatok

1. (*2) Javítsuk ki a hibákat a *List* §13.2.5 pontbeli definiciójában és írjuk meg a fordítóprogram által az *f()* függvény és a *List* számára létrehozott kóddal egyenértékű kódot. Futtassunk le egy egyszerű programot a saját, illetve a sablonból létrehozott változat ellenőrzésére. Amennyiben módunk van rá, hasonlítsuk össze a kétféle kódot.
2. (*3) Írjunk osztálysablont egy egyszeresen láncolt lista számára, amely egy *Link* típusból származtatott típusú elemeket tud tárolni. A *Link* típus tartalmazza az elemek összekapcsolásához szükséges információkat. Az ilyen listát „tolakodó" (intrusive) listának nevezik. Ezen lista felhasználásával írjunk egy bármilyen típusú elemeket tartalmazni képes (azaz nem tolakodó) egyszeresen láncolt listát. Hasonlítsuk össze a két lista hatékonyságát, előnyeiket és hátrányaikat.
3. (*2.5) Írjunk tolakodó és nem tolakodó kétszeresen láncolt listákat. Milyen műveletek szükségesek az egyszeresen láncolt lista műveletein felül?
4. (*2) Fejezzük be a §13.2 pontbeli *String* sablont a §11.12 pontbeli *String* osztály alapján.
5. (*2) Határozzunk meg egy olyan *sort()*-ot, amely az összehasonlítási feltételt sablonparaméterként veszi át. Határozzunk meg egy *Record* osztályt, melynek két tagja van, a *count* és a *price*. Rendezzünk egy *set<Record>* halmazt mindkét tag szerint.
6. (*2) Készítsünk egy *qsort()* sablont.
7. (*2) Írjunk egy programot, amely (*key,value*) párokat olvas be és az egyes kulcsokhoz (key) tartozó értékek (value) összegét írja ki. Adjuk meg, milyen típusok lehetnek kulcsok, illetve értékek.
8. (*2.5) A §11.8 pontbeli *Assoc* osztály alapján készítsünk egy egyszerű *Map* osztályt. A *Map* működjék helyesen kulcsként használt C stílusú karakterláncokkal és *string*-ekkel is, valamint akkor is, ha az alkalmazott típusnak van alapértelmezett konstruktora, és akkor is, ha nincs.
9. (*3) Hasonlítsuk össze a §11.8 pontbeli szószámláló program hatékonyságát egy asszociatív tömböt nem használó programéval. Azonos stílusú be- és kimeneti műveleteket használjunk mindkét esetben.
10. (*3) Írjuk át a §13.9[8]-beli *Map*-et valamilyen alkalmasabb adatszerkezet – például vörös-fekete fa (red-black tree) vagy S-fa (splay tree) – felhasználásával.
11. (*2.5) Használjuk fel a *Map*-et topologikus rendezést megvalósító függvény írására. (A topologikus rendezést a [Knuth,1987] első kötete írja le, a 280. oldaltól kezdődően.)
12. (*1.5) A §13.9[7]-beli összegző programot javítsuk ki, hogy szóközöket is tartalmazó nevekre is helyesen működjön.

13. (*2) Írjunk *readline()* sablonokat különféle sorfajták számára (például (cikk, szám, ár)).
14. (*2) Használjuk a §13.4 pontbeli *Literate*-ben vázolt módszert karakterláncok fordított ábécésorrendű rendezésére. Gondoskodjunk róla, hogy a módszer olyan C++-változatokkal is működjön, ahol a *char* előjeles (*signed*), és ott is, ahol *unsigned*. Adjunk egy, a kis- és nagybetűk közötti különbséget elhanyagoló rendezést támogató változatot is.
15. (*1.5) Szerkesszünk egy példát, amely legalább háromféle eltérést mutat be egy függvénysablon és egy makró között (a formai követelmények különbségén felül).
16. (*2) Tervezzünk egy módszert, amellyel biztosítható, hogy a fordítóprogram minden olyan sablon minden paraméterére ellenőrizzen bizonyos megszorításokat, amelyeknek megfelelő típusú objektum létrejön a programban. Csak „a *T* paraméternek egy *My_base*-ből származtatott osztálynak kell lennie" típusú megszorítások ellenőrzése nem elég!

14

Kivételkezelés

„Ne szóljon közbe,
amikor éppen közbeszólok.”
(Winston S. Churchill)

Hibakezelés • A kivételek csoportosítása • A kivételek elkapása • Minden kivétel elkapása • Továbbdobás • Az erőforrások kezelése • *auto_ptr* • A kivételek és a *new* operátor • Az erőforrások kimerülése • Kivételek konstruktorokban • Kivételek destruktorokban • Olyan kivételek, amelyek nem hibák • Kivételek specifikációja • Váratlan kivételek • El nem kapott kivételek • A kivételek és a hatékonyság • A hibakezelés egyéb módjai • Szabványos kivételek • Tanácsok • Gyakorlatok

14.1. Hibakezelés

Ahogy a §8.3 pontban rámutattunk, egy könyvtár szerzője felfedezheti a futási idejű hibákat, de általában fogalma sincs róla, mit kezdjen velük. A könyvtár felhasználója tudhatja, hogyan kell az ilyen hibákat kezelni, de nem tudja felderíteni azokat – máskülönben a felhasználói kódban kezelné és nem a könyvtárnak kellene megtalálnia őket. A *kivételek* (exception) az ilyen problémák kezelését segítik. Az alapötlet az, hogy ha egy függvény olyan hibát talál, amelyet nem tud kezelni, akkor egy kivételt vált ki („kivételt dob”, throw)

abban a reményben, hogy a (közvetett vagy közvetlen) hívó képes kezelni a problémát. Az adott problémát kezelni tudó függvények jelezhetik, hogy el akarják kapni (catch) a kivételt (§2.4.2, §8.3).

A hibakezelés ezen módja felveszi a versenyt a hagyományosabb módszerekkel. Tekintsük át a többi lehetőséget: ha a program észreveszi, hogy olyan probléma lépett fel, amelyet nem lehet helyben megoldani, akkor

1. befejezheti a program futását,
2. egy „hiba" jelentésű értéket adhat vissza,
3. egy normális értéket adhat vissza és a programot „szabálytalan" állapotban hagyhatja,
4. meghívhat egy, „hiba" esetén meghívandó függvényt.

Alapértelmezés szerint az első eset, azaz a program futásának befejezése történik, ha olyan kivétel lép fel, amelyet nem kap el a program. A legtöbb hiba kezelésére ennél jobb megoldás szükséges és lehetséges. Azok a könyvtárak, amelyek nem ismerik a befoglaló program célját vagy annak általános működését, nem hajthatnak végre egyszerűen egy *abort()*-ot vagy *exit()*-et. Olyan könyvtárat, amely feltétel nélkül befejezi a program futását, nem használhatunk olyan programban, amelynek nem szabad „elszállnia". A kivételek szerepét tehát úgy is megfogalmazhatnánk, hogy lehetőséget adnak arra, hogy a vezérlés visszakerüljön a hívóhoz, ha a megfelelő művelet helyben nem végezhető el.

A második eset („hiba jelentésű érték visszaadása") nem mindig kivitelezhető, mert nem mindig létezik elfogadható „hibát jelentő érték". Ha egy függvény például *int* típussal tér vissza, akkor minden *int* érték hihető visszatérési érték lehet. De ha alkalmazható is ez a módszer, sokszor akkor is kényelmetlen, mert minden hívást ellenőrizni kell, ami a program méretét akár kétszeresére is növelheti (§14.8). Ezért aztán ezt a módszert ritkán alkalmazzák annyira következetesen, hogy minden hibát észleljenek vele.

A harmadik módszer („normális érték visszaadása és a program szabálytalan állapotban való hagyása") azzal a gonddal jár, hogy a hívó esetleg nem veszi észre, hogy a program nem megengedett állapotba került. A C standard könyvtárának számos függvénye például az *errno* globális változót állítja be, hogy hibát jelezzen (§14.8), a programok azonban jellemzően elmulasztják az *errno* változó kellően következetes vizsgálatát, ami megakadályozná a hibás hívások halmozódását. Ezenkívül az egyidejű hozzáférésnél (konkurrencia, concurrency) a globális változók hibajelzésre való használata nem működik jól.

A kivételkezelés nem az olyan esetek kezelésére szolgál, mint amelyekre a negyedik, a „hibakezelő függvény meghívása" mód alkalmas, kivételek híján viszont a hibakezelő függvénynek csak a többi három lehetősége van a hiba kezelésére. A hibakezelő függvények és a kivételek témáját a §14.4.5 pont tárgyalja.

A kivételkezelést akkor célszerű használnunk a hagyományos módszerek helyett, ha azok nem elégségesek, nem „elegánsak" vagy hibákat okozhatnak. A kivételek használata lehetővé teszi a hibakezelő kódnak a „közönséges" kódtól való elválasztását, ezáltal a programot olvashatóbbá és a kódelemző eszközök számára kezelhetőbbé teszi. A kivételkezelő eljárás szabályosabb hibakezelést tesz lehetővé és megkönnyíti a külön megírt kódrészek közötti együttműködést.

A C++ kivételkezelésének a Pascal- és C-programozók számára új vonása, hogy a hibák (különösen a könyvtárakban fellépett hibák) alapértelmezett kezelése a program leállítása. A hagyományos kezelés az volt, hogy valahogy átevickéltünk a hibán, aztán reménykedtünk. A kivételkezelés törékenyebbé teszi a programot abban az értelemben, hogy több gondot és figyelmet kell fordítani arra, hogy a program elfogadhatóan fusson. Mindazonáltal ez előnyösebbnek tűnik, mintha később a fejlesztés során lépnének fel hibás eredmények, vagy akár a fejlesztés lezárulta után, amikor a program már a mit sem sejtő felhasználók kezében van. Ha a leállás az adott programnál elfogadhatatlan, akkor el lehet kapni az összes kivételt (§14.3.2) vagy az összes adott fajtájút (§14.6.2), így a kivétel csak akkor állítja le a programot, ha a programozó ezt hagyja. Ez pedig jobb, mint ha a hiba hagyományos módon történt „nem teljes" kezelését követően végzetes hiba, majd feltétel nélküli leállás történne.

Egyesek a hibákon való „átevickélés" nem vonzó tulajdonságait hibaüzenetek kiíratásával, a felhasználó segítségét kérő ablakokkal stb. próbálták enyhíteni. Az ilyesmi főleg a program hibakeresésénél (debugging, „a program belövése") hasznos, amikor a felhasználó a program szerkezetét ismerő programozó. Nem fejlesztők számára az esetleg jelen sem levő felhasználó/kezelő segítségét kérő könyvtár elfogadhatatlan. Ezenkívül sokszor nincs is hová értesítést küldeni a hibáról (például ha a program olyan környezetben fut, ahol a *cerr* nem vezet a felhasználó által elérhető helyre), és a hibaüzenetek a végfelhasználó számára úgysem mondanának semmit. Az a legkevesebb, hogy a hibaüzenet esetleg nem is a megfelelő nyelven jelenne meg, mondjuk finnül egy angol felhasználó számára. Ennél rosszabb, hogy a hibaüzenet jellemzően a könyvtár fogalmaival lenne megfogalmazva, mondjuk egy grafikus felhasználói felületről érkezett rossz adat hatására „bad argument to atan2". Egy jó könyvtár nem „halandzsázik" így. A kivételek lehetővé teszik az adott kódrészlet számára, hogy a hibát, amit nem tud kezelni, a kód olyan része számára továbbítsa, amely talán boldogul vele. A programnak csak olyan része lehet képes értelmes hibaüzenet küldésére, amelynek van fogalma a program összefüggéseiről, környezetéről.

A kivételkezelésre úgy is tekinthetünk, mint a fordítási idejű típusellenőrzés és többértelműség-kiszűrés futási idejű megfelelőjére. A tervezési folyamatra nagyobb hangsúlyt helyez és megnövelheti egy kezdeti (még hibás) változat előállításához szükséges munka mennyiségét. Ám az eredmény egy olyan kód, amelynek sokkal nagyobb az esélye arra, hogy az elvárt módon fusson, hogy egy nagyobb program része lehessen, hogy más programozók számára is érthető legyen, hogy eszközökkel lehessen kezelni. Ennek megfelelően a kivételek nyelvileg támogatott kezelése kifejezetten támogatja a „jó stílusú" programozást, mint ahogy a C++ nyelv más eszközei támogatják azt. Más nyelveken (C vagy Pascal) a jó stílus csak egyes szabályok „megkerülésével" és nem is tökéletesen érhető el.

Meg kell azonban jegyeznünk, hogy a hibakezelés ezután is nehéz feladat marad és a kivételkezelő eljárás – jóllehet rendszerezettebb, mint azok a módszerek, amelyeket helyettesít – a vezérlést kizárólag helyben szabályozó nyelvi elemekhez képest kevésbé hatékony. A C++ nyelv kivételkezelése a programozónak a hibák azon helyen való kezelésére ad lehetőséget, ahol ez a rendszer szerkezetéből adódóan a legtermészetesebb. A kivételek nyilvánvalóvá teszik a hibakezelés bonyolultságát, de vigyázzunk, hogy a rossz hírért ne annak hozóját hibáztassuk. Ezen a ponton célszerű újraolvasni a §8.3 pontot, amely a kivételkezelés alapvető vonásait mutatja be.

14.1.1. A kivételek más megközelítései

A „kivétel" (exception) azon szavak egyike, amelyek különböző emberek számára különböző jelentéssel bírnak. A C++ nyelv kivételkezelő rendszerét úgy tervezték, hogy hibák és más kivételes jelenségek kezelését támogassa – innen a neve – és hogy támogassa a hibák kezelését függetlenül fejlesztett összetevőkből álló programokban.

A kivételkezelés csak a szinkron kivételek, például a tömbindex-hibák vagy ki- és bemeneti hibák kezelésére szolgál. Az aszinkron események, mint a billentyűzet felől érkező megszakítások vagy bizonyos aritmetikai hibák nem feltétlenül kivételek és nem közvetlenül ezzel az eljárással kezelendők. Az aszinkron események világos és hatékony kezeléséhez a kivételkezelés itt leírt módjától alapvetően különböző eljárásokra van szükség. Sok rendszernek vannak az aszinkronitás kezelésére szolgáló eljárásai (például szignálok (jelzések, signal) használata), de mivel ezek rendszerfüggőek szoktak lenni, leírásuk itt nem szerepel.

A kivételkezelés egy nem lokális, a (végrehajtási) verem „visszatekerésén" (stack unwinding, §14.4) alapuló vezérlési szerkezet, amelyet alternatív visszatérési eljárásként is tekinthetünk. Ezért kivételeket olyan esetekben is szabályosan alkalmazhatunk, amelyeknek semmi közük a hibákhoz (§14.5). A kivételkezelésnek azonban a hibakezelés és a hibatűrő viselkedés támogatása az elsődleges célja és ez a fejezet is ezekre összpontosít.

A szabványos C++ nem ismeri a *végrehajtási szál* (thread) és a *folyamat* (process, processz) fogalmát, ezért az egyidejű hozzáféréssel összefüggő kivételes helyzeteket itt nem tárgyaljuk. A használt rendszer dokumentációja leírja az ezeket kezelő eszközöket; itt csak azt jegyzem meg, hogy a C++ nyelv kivételkezelő rendszerét úgy tervezték, hogy konkurens programban is hatékony legyen, feltéve, hogy a programozó vagy a rendszer betart bizonyos alapvető szabályokat, például azt, hogy szabályosan zárolja (lock) a megosztott adatszerkezeteket a használat előtt.

A C++ kivételkezelő eljárásainak a hibák és kivételes események jelentése és kezelése a céljuk, de a programozónak kell eldöntenie, hogy egy adott programban mi számít kivételesnek. Ez nem mindig könnyű (§14.5). Tekintsünk-e kivételesnek egy olyan eseményt, amely a program legtöbb futásakor fellép? Lehet-e egy tervezett és kezelt esemény hiba? Mindkét kérdésre igen a válasz. A „kivételes" nem azt jelenti, hogy „szinte soha nem történhet meg" vagy hogy „végzetes". Jobb úgy értelmezni a kivételt, hogy „a rendszer valamely része nem tudta megtenni, amire kérték". Szokás szerint ilyenkor valami mással próbálkozunk. Kivételek kiváltása („dobása") a függvényhívásokhoz képest ritkán forduljon elő, különben a rendszer szerkezete áttekinthetetlen lesz. A legtöbb nagy program normális és sikeres futtatása során azonban néhány kivétel kiváltása és elkapása biztosan elő fog fordulni.

14.2. A kivételek csoportosítása

A kivétel olyan objektum, melynek osztálya valamilyen kivétel előfordulását írja le. A hibát észlelő kód (általában egy könyvtár) „eldobja" (*throw*) az objektumot (§8.3). A hibát kezelni képes kódrészlet beavatkozási szándékát egy „elkapó" (*catch*) záradékkal jelzi. A kivétel „eldobása" „visszatekeri" a végrehajtási vermet, egészen addig, amíg egy megfelelő (vagyis a kivételt kiváltó függvényt közvetve vagy közvetlenül meghívó) függvényben *catch*-et nem találunk.

A kivételek gyakran természetes módon családokra oszthatók. Ebből következőleg a kivételek csoportosításához és kezeléséhez az öröklődés hasznos segítséget nyújthat. Egy matematikai könyvtár kivételeit például így csoportosíthatjuk:

```
class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
// ...
```

Ez a szerkezet lehetővé teszi, hogy a pontos típusra való tekintet nélkül kezeljünk bármilyen *Matherr*-t:

```
void f()
{
    try {
        // ...
    }
    catch (Overflow) {
        // az Overflow (túlcsordulás) vagy más onnan származó hiba kezelése
    }
    catch (Matherr) {
        // nem Overflow matematikai (Matherr) hibák kezelése
    }
}
```

Itt az *Overflow*-t külön kezeltük. Az összes többi *Matherr* kivételt az általános záradék fogja kezelni. A kivételeknek hierarchiába való szervezése fontos lehet a kód tömörsége céljából. Gondoljuk meg például, hogyan lehetne a matematikai könyvtár összes kivételét kezelni, ha nem volnának csoportosítva. Az összes kivételt fel kellene sorolni:

```
void g()
{
    try {
        // ...
    }
    catch (Overflow) { /* ... */ }
    catch (Underflow) { /* ... */ }
    catch (Zerodivide) { /* ... */ }
}
```

Ez nemcsak fárasztó, de egy kivétel könnyen ki is maradhat. Gondoljuk meg, mi lenne, ha nem csoportosítanánk a matematikai kivételeket. Ha a matematikai könyvtár egy új kivétellel bővülne, minden, az összes kivételt kezelni kívánó kódrészletet módosítani kellene. Általában a könyvtár első változatának kibocsátása után ilyen teljes körű módosításokra nincs mód. De ha van is, nem biztos, hogy minden kód a rendelkezésünkre áll, vagy ha igen, nem biztos, hogy tényleg hajlandóak vagyunk átdolgozni. Ezek az újrafordítási és módosíthatósági szempontok ahhoz az irányelvhez vezetnének, hogy az első változat kibocsátása után a könyvtár nem bővülhetne további kivételekkel; ez pedig a legtöbb könyvtár számára elfogadhatatlan. Ezért tehát a kivételeket célszerű könyvtáranként vagy alrendszerenként csoportokban megadni (§14.6.2).

Jegyezzük meg, hogy sem a beépített matematikai műveletek, sem a (C-vel közös) alapvető matematikai könyvtár nem kivételek formájában jelzik az aritmetikai hibákat. Ennek egyik oka, hogy számos utasításcsövet alkalmazó (pipelined) rendszer aszinkron módon észlel bizonyos aritmetikai hibákat, például a nullával való osztást. A *Matherr* hierarchia ezért itt csak illusztrációul szolgál. A standard könyvtárbeli kivételeket a §14.10 írja le.

14.2.1. Származtatott kivételek

Az osztályhierarchiák kivételkezelésre való használata természetesen vezet olyan kivételkezelőkhöz, amelyeket a kivételek hordozta információnak csak egy része érdekel. Vagyis egy kivételt általában egy bázisosztályának a kezelője kap el, nem saját osztályának kezelője. A kivétel elkapásának és megnevezésének működése a paraméteres függvényekével azonos. Vagyis a formális paraméter a paraméter-értékkel kap kezdőértéket (§7.2). Ebből következik, hogy a kivételnek csak az elkapott osztálynak megfelelő része másolódik le (felszeletelődés, slicing, §12.2.3):

```
class Matherr {
    // ...
    virtual void debug_print() const { cerr << "Matematikai hiba"; }
};

class Int_overflow: public Matherr {
    const char* op;
    int a1, a2;
public:
    Int_overflow(const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
    virtual void debug_print() const { cerr << op << '(' << a1 << ',' << a2 << ')'; }
    // ...
};

void f()
{
    try {
            g();
    }
    catch (Matherr m) {
            // ...
    }
}
```

A *Matherr* kezelőbe való belépéskor az *m* egy *Matherr* objektum, még ha *g()* egy *Int_overflow*-t váltott is ki. Következésképpen az *Int_overflow*-ban levő többlet információ elérhetetlen.

Mint mindig, mutatók vagy referenciák használatával megakadályozhatjuk az információ-vesztést:

```
int add(int x, int y)
{
    if ((x>0 && y>0 && x>INT_MAX-y) || (x<0 && y<0 && x<INT_MIN-y))
            throw Int_overflow("+",x,y);

    return x+y;     // x+y nem fog túlcsordulni
}

void f()
{
    try {
            int i1 = add(1,2);
            int i2 = add(INT_MAX,-2);
            int i3 = add(INT_MAX,2);                  // ez az!
    }
    catch (Matherr& m) {
            // ...
            m.debug_print();
    }
}
```

Az utolsó *add()* hívás kiváltotta kivétel hatására *Int_overflow::debug_print()* fog végrehajtódni. Ha a kivételt érték és nem referencia szerint kaptuk volna el, akkor ehelyett *Matherr::debug_print()*-re került volna sor.

14.2.2. Összetett kivételek

Nem minden kivételcsoport fa szerkezetű. Egy kivétel gyakran két csoportba is tartozik:

```
class Netfile_err : public Network_err, public File_system_err { /* ... */ };
```

Egy ilyen *Netfile_err* -t el lehet kapni a hálózati kivételekkel törődő függvényekben:

```
void f()
{
    try {
            // valami
    }
    catch(Network_err& e) {
            // ...
    }
}
```

De akár a fájlrendszer-kivételekkel foglalkozó függvényekben is:

```
void g()
{
    try {
        // valami más
    }
    catch(File_system_err& e) {
        // ...
    }
}
```

A hibakezelés ilyen nem hierarchikus szervezése akkor fontos, amikor a szolgáltatások (például a hálózati szolgáltatások) a felhasználó számára láthatatlanok. Ebben a példában a *g()* írója esetleg nem is tudott arról, hogy hálózat is szerepet játszik (lásd még §14.6).

14.3. A kivételek elkapása

Vegyük az alábbi kódrészletet:

```
void f()
{
    try {
        throw E();
    }
    catch(H) {
        // mikor jutunk ide?
    }
}
```

A vezérlés akkor kerül a kezelőhöz, ha

1. H ugyanaz a típus, mint E,
2. H egyértelmű bázisosztálya E-nek,
3. H és E mutatótípusok és a mutatott típusokra teljesül 1. vagy 2.,
4. H referencia és típusára teljesül 1. vagy 2.

Ezenkívül egy kivétel elkapásánál ugyanúgy alkalmazhatjuk a *const* minősítőt, mint egy függvény paraméterénél. Ettől az elkapható kivételek típusa nem változik meg, csak megakadályozza, hogy az elkapott kivételt módosíthassuk.

Az az elv, hogy a kivételről kiváltásakor másolat készül, a kezelő pedig az eredeti kivétel másolatát kapja meg. Lehetséges, hogy egy kivételről elkapása előtt több másolat is készül, ezért nem lehet olyan kivételt kiváltani, ami nem másolható. Az adott nyelvi változat nagyon sokféle módszert alkalmazhat a kivételek tárolására és továbbítására, de az biztosított, hogy a memória elfogyásakor szabványos módon kiváltandó kivétel, a *bad_alloc* (§14.4.5) számára van hely.

14.3.1. A kivételek továbbdobása

Gyakran előfordul, hogy miután egy kezelő elkapott egy kivételt, úgy dönt, hogy nem tudja teljes egészében kezelni azt. Ilyenkor a kezelő általában megteszi helyben, amit lehet, majd továbbdobja a kivételt. Így a hibát végül is a legalkalmasabb helyen lehet kezelni. Ez akkor is igaz, ha a kivétel kezeléséhez szükséges információ nem egyetlen helyen áll rendelkezésre és a hiba következményeit legjobban több kezelő között elosztva küszöbölhetjük ki:

```
void h()
{
    try {
                    // esetleg matematikai hibákat kiváltó kód
    }
    catch (Matherr) {
        if (teljesen_le_tudjuk_kezelni) {
                    // Matherr kezelése

            return;
        }
        else {
                    // megtesszük, amit lehet

            throw; // a kivétel továbbdobása
        }
    }
}
```

A kivétel továbbdobását az operandus nélküli *throw* jelzi. Ha akkor próbálunk meg kivételt továbbdobni, ha nincs is kivétel, *terminate()* hívás fog bekövetkezni (§14.7). A fordítóprogram az ilyen esetek egy részét – de nem mindet – felderítheti és figyelmeztethet rájuk.

A továbbdobás az eredeti kivételre vonatkozik, nem csak annak elkapott részére, ami *Matherr*-ként rendelkezésre áll. Más szóval, ha a program *Int_overflow*-t váltott ki, amelyet *h()* egy *Matherr*-ként kapott el és úgy döntött, hogy továbbdobja, akkor a *h()* hívója is *Int_overflow*-t kap.

14.3.2. Minden kivétel elkapása

Az elkapási és továbbdobási módszer egy végletes esetével is érdemes megismerkednünk. Mint ahogy függvényekre a ... tetszőleges paramétert jelent (§7.6), a *catch(...)* jelentése is a tetszőleges kivétel elkapása:

```
void m()
{
    try {
          // valami
    }
    catch (...) {          // minden kivételt elkapunk
          // takarítás
          throw;
    }
}
```

Így ha *m()* fő részének végrehajtása során bármilyen kivétel lép fel, sor kerül a kezelő függvény rendrakó tevékenységére. Amint a helyi rendrakás megtörtént, az arra okot adó kivétel továbbdobódik a további hibakezelés kiváltása céljából. A §14.6.3.2 pont ír arról, hogyan lehet információhoz jutni a ... kezelő által elkapott kivételről.

A hibakezelésnek általában (a kivételkezelésnek pedig különösen) fontos szempontja a program által feltételezett állapot érvényességének megőrzése (invariáns, §24.3.7.1). Például ha *m()* bizonyos mutatókat olyan állapotban hagy hátra, ahogy azokat találta, akkor a kivételkezelőbe beírhatjuk a nekik elfogadható értéket adó kódot. Így a „minden kivételt elkapó” kezelő tetszőleges invariánsok kezelésére alkalmas hely lehet. Sok fontos esetben azonban egy ilyen kivételkezelő nem a legelegánsabb megoldás (lásd §14.4).

14.3.2.1. A kezelők sorrendje

Mivel egy származtatott osztályú kivételt több típusú kivétel kezelője is elkaphat, a *try* utasításban a kezelők sorrendje lényeges. A kezelők kipróbálására ebben a sorrendben kerül sor:

```
void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // i/o adatfolyam hibák kezelése (§14.10)
    }
    catch (std::exception& e) {
        // standard könyvtárbeli kivételek kezelése (§14.10)
    }
    catch (...) {
        // egyéb kivételek kezelése (§14.3.2)
    }
}
```

Mivel a fordítóprogram ismeri az osztályhierarchiát, sok logikai hibát kiszűrhet:

```
void g()
{
    try {
        // ...
    }
    catch (...) {
        // minden kivétel kezelése (§14.3.2)
    }
    catch (std::exception& e) {
        // standard könyvtárbeli kivételek kezelése (§14.10)
    }
    catch (std::bad_cast) {
        // dynamic_cast hibák kezelése (§15.4.2)
    }
}
```

Itt az *exception* soha nem jut szerephez. Még ha el is távolítjuk a mindent elkapó kezelőt, a *bad_cast* akkor sem kerül szóba, mert az *exception* leszármazottja.

14.4. Erőforrások kezelése

Amikor egy függvény lefoglal valamilyen erőforrást – például megnyit egy fájlt, lefoglal valamennyi memóriát a szabad tárban, zárol valamit stb. –, gyakran fontos feltétel a rendszer további működése szempontjából, hogy az erőforrást rendben fel is szabadítsa a hívóhoz való visszatérés előtt:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"r");

    // f használata

    fclose(f);
}
```

Ez működőképesnek tűnik, amíg észre nem vesszük, hogy ha valami hiba történik az *fopen()* meghívása után, de az *fclose()* meghívása előtt, akkor egy kivétel miatt a *use_file()* függvény az *fclose()* végrehajtása nélkül térhet vissza. Ugyanez a probléma kivételkezelést nem támogató nyelvek esetében is felléphet. Például a C standard könyvtárának *longjmp()* függvénye ugyanilyen problémát okozhat. Még egy közönséges *return* utasítás miatt is visszatérhet a *use_file()* az *fclose()* végrehajtása nélkül.

Egy első kísérlet a *use_file()* hibatűrővé tételére így nézhet ki:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"r");
    try {
        // f használata
    }
    catch (...) {
        fclose(f);
        throw;
    }
    fclose(f);
}
```

A fájlt használó kód egy *try* blokkban van, amely minden hibát elkap, bezárja a fájlt, majd továbbdobja a kivételt.

Ezzel a megoldással az a gond, hogy feleslegesen hosszú, fáradságos és esetleg lassú is lehet. Ráadásul minden túl hosszú és fáradságos megoldás nyomában ott járnak a hibák, hiszen a programozók elfáradnak. Szerencsére van jobb megoldás. A megoldandó helyzet általános formájában így néz ki:

```
void acquire()
{
    // első erőforrás lekötése
    // ...
    // n-edik erőforrás lekötése

    // erőforrások felhasználása

    // n-edik erőforrás felszabadítása
    // ...
    // első erőforrás felszabadítása
}
```

Általában fontos szempont, hogy az erőforrásokat megszerzésükkel ellentétes sorrendben szabadítsuk fel. Ez erőteljesen emlékeztet a konstruktorok által felépített és destruktorok által megsemmisített helyi objektumok viselkedésére. Így aztán az ilyen erőforrás-lefoglalási és -felszabadítási problémákat konstruktorokkal és destruktorokkal bíró osztályok objektumainak használatával kezelhetjük. Például megadhatjuk a *File_ptr* osztályt, amely egy *FILE**-hoz hasonlóan viselkedik:

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a) { p = fopen(n,a); }
    File_ptr(FILE* pp) { p = pp; }
    ~File_ptr() { fclose(p); }

    operator FILE*() { return p; }
};
```

Egy *File_ptr* objektumot vagy egy *FILE** mutatóval, vagy az *fopen()*-hez szükséges paraméterekkel hozhatunk létre; bármelyik esetben élettartama végén a *File_ptr* objektum megsemmisül, destruktora pedig bezárja a fájlt. Programunk mérete így minimálisra csökken:

```
void use_file(const char* fn)
{
    File_ptr f(fn,"r");
    // f használata
}
```

A destruktor attól függetlenül meghívódik, hogy a függvényből „normálisan” vagy kivétel kiváltása folytán léptünk-e ki. Így a kivételkezelő eljárás lehetővé teszi, hogy a hibakezelést eltávolítsuk a fő algoritmusból. Az így adódó kód egyszerűbb és kevesebb hibát okoz, mint hagyományos megfelelője.

Azt a műveletet, amikor a kivételek kezelésekor a hívási láncban megkeressük a megfelelő kezelőt, rendszerint a „verem visszatekerésének” hívják. Ahogy a vermet „visszatekerik”, úgy hívják meg a létrehozott lokális objektumok destruktorait.

14.4.1. Konstruktorok és destruktorok használata

Az erőforrások lokális objektumok használatával való kezelését szokás szerint úgy hívják, hogy „kezdeti értékadás az erőforrás megszerzésével” (resource acquisition is initialization). Ez az általános eljárás a konstruktorok és destruktorok tulajdonságain, valamint a kivételkezelő rendszerrel való együttműködésükön alapul.

Egy objektumot addig nem tekintünk létezőnek, amíg konstruktora le nem fut. Destruktora csak ebben az esetben fog a verem visszatekerésekor meghívódni. Egy részobjektumokból álló objektumot olyan mértékben tekintünk létrehozottnak, amilyen mértékben részobjektumainak konstruktorai lefutottak, egy tömböt pedig addig az eleméig, amelynek konstruktora már lefutott. A destruktor a verem visszatekerésekor csak a teljes egészében létrehozott elemekre fut le.

A konstruktor azt próbálja elérni, hogy az objektum teljesen és szabályosan létrejöjjön. Ha ez nem érhető el, egy jól megírt konstruktor – amennyire csak lehetséges – olyan állapotban hagyja a rendszert, mint meghívása előtt volt. Ideális esetben a „naív módon” megírt konstruktorok teljesítik ezt és nem hagyják az objektumot valamilyen „félig létrehozott” állapotban. Ezt a „kezdeti értékadás az erőforrás megszerzésével” módszernek a tagokra való alkalmazásával érhetjük el.

Vegyünk egy X osztályt, amelynek konstruktora két erőforrást igényel: egy *x* állományt és egy *y* zárolást. Ezek megszerzése nem biztos, hogy sikerül, és kivételt válthat ki. Az *X* osztály konstruktorának semmilyen körülmények között nem szabad úgy visszatérnie, hogy az állományt lefoglalta, de a zárat nem. Ezenkívül ezt anélkül kell elérni, hogy a dolog bonyolultságával a programozónak törődnie kelljen. A megszerzett erőforrások ábrázolására két osztályt használunk, a *File_ptr*-t és a *Lock_ptr*-t. Az adott erőforrás megszerzését az azt jelölő lokális objektum kezdeti értékadása ábrázolja:

```
class X {
    File_ptr aa;
    Lock_ptr bb;
public:
    X(const char* x, const char* y)
            : aa(x,"rw"),                  // 'x' lefoglalása
              bb(y)                        // 'y' lefoglalása
    { }
    // ...
};
```

Csakúgy mint a lokális objektumos példában, itt is az adott fordító dolga a szükséges nyilvántartások vezetése. A felhasználónak nem kell ezzel törődnie. Például ha az *aa* létrehozása után, de a *bb*-é előtt kivétel váltódik ki, akkor az *aa* destruktorának meghívására sor kerül, de a *bb*-ére nem.

Ebből következőleg ahol az erőforrások megszerzése ezen egyszerű modell szerint történik, ott a konstruktor írójának nem kell kifejezett kivételkezelő kódot írnia.

Az alkalmi módon kezelt erőforrások között a leggyakoribb a memória:

```
class Y {
    int* p;
    void init();
public:
    Y(int s) { p = new int[s]; init(); }
    ~Y() { delete[ ] p; }
    // ...
};
```

A fenti módszer használata általános gyakorlat – és a „memória elszivárgásához" (memory leak) vezethet. Ha az *init()*-ben kivétel keletkezik, a lefoglalt memória nem fog felszabadulni, mivel az objektum nem jött létre teljesen, így destruktora nem fut le. Íme egy biztonságos változat:

```
class Z {
    vector<int> p;
    void init();
public:
    Z(int s) : p(s) { init(); }
    // ...
};
```

A *p* által használt memóriát a *vector* kezeli. Ha az *init()* kivételt vált ki, a lefoglalt memóriát *p* (automatikusan meghívott) destruktora fogja felszabadítani.

14.4.2. Auto_ptr

A standard könyvtár *auto_ptr* sablon osztálya támogatja a „kezdeti értékadás az erőforrás megszerzésével" módszert. Egy *auto_ptr*-nek alapvetően egy mutatóval adhatunk kezdőértéket és ugyanúgy hivatkozhatunk a mutatott objektumra, mint egy hagyományos mutatónál. Ezenkívül amikor az *auto_ptr* megsemmisül, az általa mutatott objektum is automatikusan törlődik:

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb)       // kilépéskor ne felejtsük
                                                                  // el törölni pb-t
{
    auto_ptr<Shape> p(new Rectangle(p1,p2));     // p téglalapra mutat
    auto_ptr<Shape> pbox(pb);

    p->rotate(45);          // az auto_ptr<Shape> pont úgy használható, mint a Shape*
    // ...
    if (in_a_mess) throw Mess();
    // ...
}
```

Itt a *Rectangle*, a *pb* által mutatott *Shape* és a *pc* által mutatott *Circle* egyaránt törlődni fog, akár váltódott ki kivétel, akár nem.

Ezen *tulajdonos szerinti kezelés* (vagy *destruktív másolás*) támogatása céljából az *auto_ptr*-eknek a közönséges mutatóktól gyökeresen eltérő másolási módszerük van: egy *auto_ptr*-nek egy másikba való másolása után a forrás nem mutat semmire. Minthogy az *auto_ptr*-eket a másolás megváltoztatja, konstans (*const*) *auto_ptr*-eket nem lehet másolni.

Az *auto_ptr* sablont a *<memory>* fejállomány deklarálja. Íme egy lehetséges kifejtése:

```
template<class X> class std::auto_ptr {
    template <class Y> struct auto_ptr_ref { /* ... */ };      // segédosztály
    X* ptr;
public:
    typedef X element_type;

    explicit auto_ptr(X* p =0) throw() { ptr=p; }     // throw() jelentése "nem vált ki kivételt",
                                                       // lásd §14.6
    ~auto_ptr() throw() { delete ptr; }
```

```
    // figyeljük meg: másolás és értékadás nem konstans paraméterekkel
    auto_ptr(auto_ptr& a) throw();                              // másol, majd a.ptr=0
    template<class Y> auto_ptr(auto_ptr<Y>& a) throw();         // másol, majd a.ptr=0
    auto_ptr& operator=(auto_ptr& a) throw();                   // másol, majd a.ptr=0
    template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw(); // másol, majd a.ptr=0

    X& operator*() const throw() { return *ptr; }
    X* operator->() const throw() { return ptr; }
    X* get() const throw() { return ptr; }                        // mutató kinyerése
    X* release() throw() { X* t = ptr; ptr=0; return t; }         // tulajdonosváltás
    void reset(X* p =0) throw() { if (p!=ptr) { delete ptr; ptr=p; } }

    auto_ptr(auto_ptr_ref<X>) throw();                     // másolás auto_ptr_ref-ből
    template<class Y> operator auto_ptr_ref<Y>() throw(); // másolás auto_ptr_ref-be
    template<class Y> operator auto_ptr<Y>() throw();      // destruktív másolás auto_ptr-ből
};
```

Az *auto_ptr_ref* célja, hogy megvalósítsa a közönséges *auto_ptr* számára a destruktív másolást, ugyanakkor megakadályozza egy konstans *auto_ptr* másolását. Ha egy *D**-ot *B**-gá lehet alakítani, akkor a sablonkonstruktor és a sablon-értékadás (meghatározott módon vagy automatikusan) egy *auto_ptr<D>*-t *auto_ptr<B>*-vé tud alakítani:

```
void g(Circle* pc)
{
    auto_ptr<Circle> p2 = pc;           // most p2 felel a törlésért
    auto_ptr<Circle> p3 = p2;           // most p3 felel a törlésért (p2 már nem)
    p2->m = 7;                          // programozói hiba: p2.get()==0
    Shape* ps = p3.get();               // mutató kinyerése auto_ptr-ből
    auto_ptr<Shape> aps = p3;           // tulajdonosváltás és típuskonverzió
    auto_ptr<Circle> p4 = pc;           // programozói hiba: most p4 is felel a
                                        // törlésért
}
```

Az nem meghatározott, mi történik, ha több *auto_ptr* is mutat egy objektumra, de az a legvalószínűbb, hogy az objektum kétszer törlődik – ami hiba.

Jegyezzük meg, hogy az *auto_ptr* destruktív másolási módszere miatt nem teljesíti a szabványos tárolók elemeire vagy a *sort()*-hoz hasonló szabványos eljárásokra vonatkozó követelményeket:

```
vector< auto_ptr<Shape> >& v;    // veszélyes: auto_ptr használata tárolóban
// ...
sort(v.begin(),v.end());          // Ezt ne tegyük: a rendezéstől v sérülhet
```

Világos, hogy az *auto_ptr* nem egy általános „okos” vagy „intelligens” mutató (smart pointer). De amire tervezték – az automatikus mutatók kivételbiztos kezelésére – arra lényeges „költség” nélkül megfelel.

14.4.3. Figyelmeztetés

Nem minden programnak kell mindenfajta hibával szemben immunisnak lennie és nem minden erőforrás annyira létfontosságú, hogy a védelme megérje a „kezdeti értékadás az erőforrás megszerzésével” módszernek, az *auto_ptr*-nek, illetve a *catch(...)* alkalmazásának fáradságát. Például sok, egyszerűen a bemenetet olvasó és azzal le is futó programnál a súlyosabb futási idejű hibákat úgy is kezelhetjük, hogy egy alkalmas hibaüzenet kiadása után a programot leállítjuk. Ezzel a rendszerre bízzuk a program által lefoglalt összes erőforrás felszabadítását, így a felhasználó újrafuttathatja a programot egy jobb bemenettel. Az itt leírt módszer olyan alkalmazások számára hasznos, amelyeknél a hibák ilyesféle egyszerűsített kezelése elfogadhatatlan. Egy könyvtár tervezője például általában nem élhet feltevésekkel a könyvtárat használó program hibatűrési követelményeit illetően, így aztán el kell kerülnie minden feltétel nélküli futási idejű hibát és minden erőforrást fel kell szabadítania, mielőtt egy könyvtári függvény visszatér. A „kezdeti értékadás az erőforrás megszerzésével” módszer a kivételeknek a hibák jelzésére való felhasználásával együtt sok ilyen könyvtár számára megfelelő.

14.4.4. A kivételek és a new operátor

Vegyük a következőt:

```
void f(Arena& a, X* buffer)
{
    X* p1 = new X;
    X* p2 = new X[10];

    X* p3 = new(buffer[10]) X;          // X átmeneti tárba helyezése (nem
                                        // szükséges felszabadítás)
    X* p4 = new(buffer[11]) X[10];

    X* p5 = new(a) X;                   // tárfoglalás az 'a' Arena-tól (a-t kell
                                        // felszabadítani)
    X* p6 = new(a) X[10];
}
```

Mi történik, ha *X* konstruktora kivételt vált ki? Felszabadul-e a *new()* operátor által lefoglalt memória? Közönséges esetben a válasz igen, így *p1* és *p2* kezdeti értékadása nem okoz memória-elszivárgást.

Ha az elhelyező utasítást (placement syntax, §10.4.11) használjuk, a válasz nem ennyire egyszerű. Az elhelyező utasítás némely felhasználása lefoglal némi memóriát, amit aztán fel kellene szabadítani, de némelyik nem. Ezenkívül az elhelyező utasítás alkalmazásának éppen a memória nem szabványos lefoglalása a lényege, így aztán jellemzően nem szabványos módon is kell felszabadítani azt. Következésképpen a továbbiak a felhasznált memóriafoglalótól (allokátortól) függnek. Ha egy *Z::operator new()* memóriafoglaló szerepelt, akkor a rendszer meghívja a *Z::operator delete()*-et, feltéve, hogy van ilyen; más módon nem próbál felszabadítást végezni. A tömbök kezelése ezzel azonos módon történik (§15.6.1). Ez az eljárás helyesen kezeli a standard könyvtárbeli elhelyező *new* operátort (§10.4.11), csakúgy, mint minden olyan esetet, amikor a programozó összeillő lefoglaló és felszabadító függvénypárt írt.

14.4.5. Az erőforrások kimerülése

Visszatérő programozási dilemma, hogy mi történjék, ha nem sikerül egy erőforrás lefoglalási kísérlete, például mert korábban meggondolatlanul nyitogattunk fájlokat (az *fopen()*-nel) és foglaltunk le memóriát a szabad tárban (a *new* operátorral), anélkül, hogy törődtünk volna vele, mi van, ha nincs meg a fájl vagy hogy kifogytunk-e a szabad tárból. Ilyen problémával szembesülve a programozók kétféle megoldást szoktak alkalmazni:

1. Újrakezdés: kérjünk segítséget valamelyik hívó függvénytől és folytassuk a program futását.
2. Befejezés: hagyjuk abba a számítást és térjünk vissza a hívóhoz.

Az első esetben a hívónak fel kell készülnie arra, hogy segítséget adjon egy ismeretlen kódrészletnek annak erőforrás-lefoglaló problémájában. A második esetben a hívónak arra kell felkészülnie, hogy kezelje az erőforrás-lefoglalás sikertelenségéből adódó problémát. A második eset a legtöbbször sokkal egyszerűbb és a rendszer elvonatkoztatási szintjeinek jobb szétválasztását teszi lehetővé. Jegyezzük meg, hogy a „befejezés” választásakor nem a program futása fejeződik be, hanem csak az adott számítás. A „befejezés” (termination) azon eljárás hagyományos neve, amely egy „sikertelen” számításból a hívó által adott hibakezelőbe tér vissza (ami aztán újra megpróbálhatja a számítást elvégeztetni), ahelyett, hogy megpróbálná maga orvosolni a helyzetet és a hiba felléptének helyéről folytatni a számítást.

A C++-ban az újrakezdő modellt a függvényhívási eljárás, a befejező modellt a kivételkezelő eljárás támogatja. Mindkettőt szemlélteti a standard könyvtárbeli *new()* operátor egy egyszerű megvalósítása és felhasználása:

```
void* operator new(size_t size)
{
    for (;;) {
        if (void* p = malloc(size)) return p;          // megpróbálunk memóriát találni
        if (_new_handler == 0) throw bad_alloc();     // nincs kezelő: feladjuk
        _new_handler();                               // segítséget kérünk
    }
}
```

Itt a C standard könyvtárának *malloc()* függvényét használtam a szabad memóriahely tényleges megkeresésére; az operator *new()* más változatai más módokat választhatnak. Ha sikerült memóriát találni, a *new()* operátor visszaadhatja az arra hivatkozó mutatót; ha nem, a *new()* meghívja a *_new_handler*-t. Ha az talál a *malloc()* számára lefoglalható memóriát, akkor minden rendben. Ha nem, akkor a kezelő nem térhet vissza a *new()* operátorba végtelen ciklus okozása nélkül. A *_new_handler* ekkor kivételt válthat ki, ilyen módon valamelyik hívóra hagyva a helyzet tisztázását:

```
void my_new_handler()
{
    int no_of_bytes_found = find_some_memory();
    if (no_of_bytes_found < min_allocation) throw bad_alloc();    // feladjuk
}
```

Valahol lennie kell egy *try* blokknak is, a megfelelő kivételkezelővel:

```
try {
    // ...
}
catch (bad_alloc) {
    // valahogy reagálunk a memória kifogyására
}
```

A *new()* operátor ezen változatában használt *_new_handler* egy, a szabványos *set_new_handler()* függvény által fenntartott függvénymutató. Ha saját *my_new_handler()*-ünket szeretnénk *_new_handler*-ként használni, ezt írhatjuk:

```
set_new_handler(&my_new_handler);
```

Ha el akarjuk kapni a *bad_alloc* kivételt is, ezt írhatjuk:

```
void f()
{
    void(*oldnh)() = set_new_handler(&my_new_handler);

    try {
            // ...
    }
    catch (bad_alloc) {
            // ...
    }
    catch (...) {
            set_new_handler(oldnh);                // kezelő újbóli beállítása
            throw;                                 // továbbdobás
    }

    set_new_handler(oldnh);                        // kezelő újbóli beállítása
}
```

Még jobb, ha a *catch(...)* kezelőt a „kezdeti értékadás az erőforrás megszerzésével" módszernek (§14.4) a *_new_handler*-re való alkalmazásával elkerüljük (§14.12[1]).

A *_new_handler* alkalmazásával a hiba észlelésének helyétől semmiféle további információ nem jut el a segédfüggvényig. Könnyű több információt közvetíteni, ám minél több jut el belőle egy futási idejű hiba észlelésének helyétől a kijavítást segítő helyig, a két kód annál inkább függ egymástól. Ebből adódóan az egyik kódrészleten csak a másikat értve és esetleg azt módosítva lehet változtatni. A különböző helyen levő programrészek elszigetelésére jó módszer, ha az ilyen függéseket a legkisebb mértékre szorítjuk vissza. A kivételkezelő eljárás jobban támogatja az elszigetelést, mint a hívó által biztosított segédfüggvények meghívása.

Általában célszerű az erőforrások megszerzését rétegekbe, elvonatkoztatási szintekbe szervezni, és elkerülni, hogy egy réteg a hívó réteg segítségére szoruljon. Nagyobb rendszerek esetében a tapasztalat azt mutatja, hogy a sikeres rendszerek ezt az utat követik.

A kivételek kiváltásához szükség van egy „eldobandó" objektumra. Az egyes C++-változatoktól elvárjuk, hogy a memória elfogyásakor is maradjon annyi tartalék, amennyi egy *bad_alloc* kiváltásához kell, de lehetséges, hogy valamilyen más kivétel dobása a memória elfogyásához vezet.

14.4.6. Kivételek konstruktorokban

A kivételek adnak megoldást arra a problémára, hogyan jelentsünk hibát egy konstruktorból. Minthogy a konstruktorok nem adnak vissza külön visszatérési értéket, amelyet a hívó megvizsgálhatna, a hagyományos (azaz kivételeket nem alkalmazó) lehetőségek a következők:

1. Adjunk vissza egy hibás állapotú objektumot, megbízva a hívóban, hogy az majd ellenőrzi az állapotot.
2. Állítsunk be egy nem lokális változót (például az *errno*-t), hogy jelezze a létrehozás sikertelenségét, és bízzunk a felhasználóban, hogy ellenőrzi a változót.
3. Ne végezzünk kezdeti értékadást a konstruktorban és bízzunk meg a felhasználóban, hogy az első használat előtt elvégzi azt.
4. Jelöljük meg az objektumot „kezdőérték nélküli"-ként, és végezze az első meghívott tagfüggvény a kezdeti értékadást. Ez a függvény jelenti majd a hibát, ha a kezdeti értékadás nem sikerült.

A kivételkezelés lehetőséget ad arra, hogy a létrehozás sikertelenségére vonatkozó információt a konstruktorból átadjuk. Egy egyszerű *Vector* osztály például így védekezhet a túlzott igények ellen:

```
class Vector {
public:
    class Size { };

    enum { max = 32000 };

    Vector(int sz)
    {
        if (sz<0 || max<sz) throw Size();
        // ...
    }

    // ...
};
```

A *Vector*-okat létrehozó kód most elkaphatja a *Vector::Size* hibákat, és megpróbálhatunk valami értelmeset kezdeni velük:

```
Vector* f(int i)
{
    try {
        Vector* p = new Vector(i);
        // ...
        return p;
    }
```

```
    catch(Vector::Size) {
        // mérethiba kezelése
    }
}
```

Mint mindig, maga a hibakezelő az alapvető hibakezelő és -helyreállító módszerek szokásos készletét alkalmazhatja. Valahányszor a hívó egy kivételt kap, megváltozik annak értelmezése, hogy mi volt a hiba. Ha a kivétellel együtt a megfelelő információ is továbbítódik, a probléma kezeléséhez rendelkezésre álló ismeretek halmaza akár bővülhet is. Más szóval, a hibakezelő módszerek alapvető célja az, hogy a hiba felfedezésének eredeti helyéről olyan helyre jutassunk el információt, ahol elegendő ismeret áll rendelkezésre a hiba következményeinek elhárításához, és ezt ráadásul megbízhatóan és kényelmesen tegyük.

A „kezdeti értékadás az erőforrás megszerzésével" eljárás az egynél több erőforrást igénybe vevő konstruktorok kezelésének legbiztosabb és legelegánsabb módja (§14.4). Ez lényegében a sok erőforrás kezelésének problémáját az egy erőforrást kezelő egyszerűbb eljárás ismételt alkalmazására vezeti vissza.

14.4.6.1. Kivételek és a tagok kezdeti értékadása

Mi történik, ha egy tag kezdeti értékadása közvetlenül vagy közvetve kivételt vált ki? Alapértelmezés szerint a kivételt az a hívó függvény kapja meg, amelyik a tag osztályának konstruktorának meghívta, de maga a konstruktor is elkaphatja azt, ha a teljes függvénytörzset a tag kezdőérték-listájával együtt egy *try* blokkba zárjuk:

```
class X {
    Vector v;
    // ...
public:
    X(int);
    // ...
};

X::X(int s)
try
    :v(s)                           // v kezdőértéke s
{
    // ...
}
catch (Vector::Size) {              // a v által kiváltott kivételt itt kapjuk el
    // ...
}
```

14.4.6.2. Kivételek és másolás

Más konstruktorokhoz hasonlóan a másoló konstruktor is jelezheti a hibás futást kivétel kiváltásával. Ebben az esetben az objektum nem jön létre. A *vector* másoló konstruktora például memóriát foglal le és átmásolja az elemi objektumokat (16,3,4,, E.3.2.), ami kivételt válthat ki. A kivétel kiváltása előtt a másoló konstruktor fel kell szabadítsa a lefoglalt erőforrásokat. Az E.2. és E.3. függelékekben részletesen tárgyaljuk a kivételkezelés és a tárolók erőforrás-gazdálkodásának kapcsolatait.

A másoló értékadó operátor ugyanígy lefoglalhat erőforrásokat és kiválthat kivételt. Mielőtt ez utóbbit tenné, az értékadásnak biztosítania kell, hogy mindkét operandusát azonos állapotban hagyja. Más esetben megszegjük a standard könyvtárbeli előírásokat, melynek következménye nem meghatározott viselkedés lehet.

14.4.7. Kivételek destruktorokban

A kivételkezelő eljárás szemszögéből nézve egy destruktort kétféleképpen lehet meghívni:

1. Normál (szabályos) meghívás: a hatókörből való szabályos kilépéskor ((§10.4.3) vagy egy *delete* hívás folytán (§10.4.5) stb.
2. Kivételkezelés közbeni meghívás: a verem visszatekerése közben (§14.4) a kivételkezelő eljárás elhagy egy blokkot, amely destruktorral bíró objektumot tartalmaz.

Az utóbbi esetben a kivétel nem léphet ki a destruktorból. Ha megteszi, az a kivételkezelő eljárás hibájának számít, és meghívódik az *std::terminate()* (§14.7). Végülis a kivételkezelő eljárásnak és a destruktornak általában nincs módja eldönteni, hogy elfogadható-e az egyik kivételnek a másik kedvéért való elhanyagolása.

Ha a destruktor olyan függvényt hív meg, amely kivételt válthat ki, akkor ez ellen védekezhet:

```
X::~X()
try {
    f();      // kivételt válthat ki
}
catch (...) {
              // valamit csinálunk
}
```

A standard könyvtárbeli *uncaught_exception()* függvény *true*-val tér vissza, ha van olyan „eldobott” kivétel, amelyet még nem kaptak el. Ez lehetővé teszi a destruktor attól függő programozását, hogy az objektum szabályos vagy a verem visszatekerése közbeni megsemmisítéséről van-e szó.

14.5. Kivételek, amelyek nem hibák

Ha egy kivételre számítunk és elkapjuk és így annak nincs a program működésére nézve rossz következménye, akkor miért lenne hiba? Csak mert a programozó a kivételre mint hibára és a kivételkezelő eljárásokra pedig mint hibakezelő eszközökre gondol? Nos, a kivételeket úgy is tekinthetjük, mintha vezérlőszerkezetek lennének:

```
void f(Queue<X>& q)
{
    try {
        for (;;) {
            X m = q.get();              // 'Empty' kivételt vált ki, ha a sor üres
            // ...
        }
    }
    catch (Queue<X>::Empty) {
        return;
    }
}
```

Ez egészen jónak tűnik, így ebben az esetben tényleg nem teljesen világos, mi számít hibának és mi nem.

A kivételkezelés kevésbé rendezett eljárás, mint az *if*-hez vagy a *for*-hoz hasonló helyi vezérlési szerkezetek és azoknál gyakran kevésbé hatékony is, ha a kivételre ténylegesen sor kerül. Ezért kivételeket csak ott használjunk, ahol a hagyományosabb vezérlési szerkezetek nem elegánsak vagy nem használhatóak. A standard könyvtár például tartalmaz egy tetszőleges elemekből álló *queue*-t (sor) is, kivételek alkalmazása nélkül (§17.3.2).

A keresőfüggvényeknél – különösen a rekurzív hívásokat nagymértékben alkalmazó függvényeknél (például egy fában kereső függvénynél) – jó ötlet befejezésként kivételt alkalmazni:

```
void fnd(Tree* p, const string& s)
{
    if (s == p->str) throw p;           // megtalálta s-t
    if (p->left) fnd(p->left,s);
    if (p->right) fnd(p->right,s);
}

Tree* find(Tree* p, const string& s)
{
```

```
    try {
            fnd(p,s);
    }
    catch (Tree* q) {        // q->str==s
            return q;
    }
    return 0;
}
```

Ugyanakkor a kivételek ilyen használata könnyen túlzásba vihető és áttekinthetetlen kódhoz vezethet. Ha ésszerű, ragaszkodjunk „a kivételkezelés hibakezelés" elvhez. Ekkor a kód világosan két részre különül: közönséges és hibakezelő kódra. Ez érthetőbbé teszi a kódot. Sajnos a „való" világ nem ilyen tiszta, a program szerkezete pedig ezt (bizonyos fokig kívánatos módon) tükrözni fogja.

A hibakezelés lényegénél fogva nehéz. Becsüljünk meg mindent, ami segít egy világos modellt kialakítani arról, mi számít hibának és hogyan kezeljük.

14.6. Kivételek specifikációja

A kivétel kiváltása vagy elkapása befolyásolja a függvénynek más függvényekhez való viszonyát. Ezért érdemes lehet a függvény deklarációjával együtt megadni azon kivételeket is, amelyeket a függvény kiválthat.

```
void f(int a) throw (x2, x3);
```

Ha egy függvény megadja, milyen kivételek léphetnek fel végrehajtása közben, akkor ezzel valójában garanciát nyújt a használóinak. Ha a függvény futása közben valamit olyasmit próbál tenni, ami ezt a garanciát érvénytelenné tenné, akkor ez a kísérlet az *std::unexpected()* hívássá fog átalakulni. Az *unexpected()* alapértelmezett jelentése *std::terminate()*, ami szokványos esetben meghívja az *abort()*-ot. (Részletesen lásd a §9.4.1.1 pontban.) Valójában a

```
void f() throw (x2, x3)
{
    // törzs
}
```

egyenértékű a következővel:

```
void f()
try
{
    // törzs
}
catch (x2) { throw; }       // továbbdobás
catch (x3) { throw; }       // továbbdobás
catch (...) {
    std::unexpected();      // az unexpected() nem fog visszatérni
}
```

Ennek legfontosabb előnye, hogy a függvény deklarációja a hívók által elérhető felület része. A függvénydefiníciók viszont nem általánosan elérhetőek. Ha hozzá is férünk az összes könyvtár forrásához, határozottan nem szeretünk sűrűn beléjük nézegetni, ráadásul a megkivétel-specifikációkkal (exception specification) ellátott függvénydeklarációk sokkal rövidebbek és világosabbak, mint az egyenértékű kézzel írott változat.

A kivétel-specifikációk nélküli függvényekről azt kell feltételeznünk, hogy bármilyen kivételt kiválthatnak:

```
int f();            // bármilyen kivételt kiválthat
```

A kivételt ki nem váltó függvényeket üres listával adhatjuk meg:

```
int g() throw ();   // nem vált ki kivételt
```

Azt gondolhatnánk, az lenne a jó alapértelmezés, hogy a függvény nem váltana ki kivételt. Ekkor azonban alapvetően minden függvény részére szükséges lenne kivételeket meghatározni, emiatt pedig sokszor kellene újrafordítani és meg is akadályozná a más nyelveken írt programokkal való együttműködést. Ez aztán arra ösztönözné a programozókat, hogy „aknázzák alá" a kivételkezelő rendszert és hogy hibás kódot írjanak a kivételek elfojtására, ez pedig az aknamunkát észre nem vevőknek hamis biztonságérzetet adna.

14.6.1. A kivételek ellenőrzése

Egy felületnek fordítási időben nem lehet minden megsértési kísérletét ellenőrizni, de azért fordításkor sok ellenőrzés történik. A specifikált (tehát „engedélyezett") kivételeket a fordító úgy értelmezi, hogy a függvény azok mindegyikét ki fogja váltani. A kivétel-specifikációk fordítási idejű ellenőrzésének szabályai a könnyen felderíthető hibákat tiltják.

Ha egy függvény valamely deklarációja megad kivételeket is, akkor mindegyik deklarációjának (és definiciójának is) tartalmaznia kell pontosan ugyanazokat a kivételeket:

```
int f() throw (std::bad_alloc);

int f()      // hiba: a kivétel-meghatározás hiányzik
{
    // ...
}
```

Fontos szempont, hogy a kivételek fordítási egységek határain átnyúló ellenőrzése nem kötelező. Természetesen az adott nyelvi változat ellenőrizhet így, de a nagyobb rendszerek legtöbbje számára fontos, hogy ez ne történjen meg, vagy ha mégis, csak akkor jelezzen végzetes hibát, ha a kivétel-specifikációk megsértését nem lehet majd futási időben elkapni.

A lényeg az, hogy új kivétellel való bővítés ne kényszerítse ki a kapcsolódó kivétel-specifikációk kijavítását és az esetleg érintett összes kód újrafordítását. A rendszer így egy félig felújított állapotban tovább működhet és a váratlan kivételek dinamikus (futási időbeli) észlelésére hagyatkozhat, ami alapvető az olyan nagy rendszereknél, ahol a nagyobb frissítések költségesek és nincs is meg az összes forrás.

Egy virtuális függvényt csak olyan függvénnyel lehet felülírni, amely legalább annyira szigorúan határozza meg a kivételeket, mint maga az eredeti függvény:

```
class B {
public:
    virtual void f();                  // bármit kiválthat
    virtual void g() throw(X,Y);
    virtual void h() throw(X);
};

class D : public B {
public:
    void f() throw(X);                 // rendben
    void g() throw(X);                 // rendben: D::g() szigorúbb, mint B::g()
    void h() throw(X,Y);               // hiba: D::h() megengedőbb, mint B::h()
};
```

Ezt a szabályt a józan ész diktálja. Ha egy származtatott osztály olyan kivételt váltana ki, amit az eredeti függvény nem adott meg lehetségesként, annak elkapását nem várhatnánk el a hívótól. Másrészt egy felülíró függvény, amely kevesebb kivételt vált ki, betartja a felülírt függvény kivétel-specifikációjában felállított szabályokat.

Ugyanígy egy, a kivételeket szigorúbban meghatározó függvényt hozzárendelhetünk egy megengedőbb függvénymutatóhoz, de fordítva nem:

```
void f() throw(X);
void (*pf1)() throw(X,Y) = &f;          // rendben
void (*pf2)() throw() = &f;             // hiba: f() megengedőbb, mint pf2
```

Kivétel-specifikáció nélküli függvényre hivatkozó mutatót kivételeket meghatározó függvénymutatóhoz különösképpen nem rendelhetünk:

```
void g();    // bármit kiválthat

void (*pf3)() throw(X) = &g;            // hiba: g() megengedőbb, mint pf3
```

A kivétel-specifikáció nem része a függvény típusának, a *typedef*-ek pedig nem is tartalmazhatnak ilyet:

```
typedef void (*PF)() throw(X);          // hiba
```

14.6.2. Váratlan kivételek

Ha a kivételek köre specifikált, az ott nem szereplő kivételek az *unexpected()* meghívását válthatják ki. Az ilyen hívások a tesztelésen kívül általában nem kívánatosak. A kivételek gondos csoportosításával és a felületek megfelelő meghatározásával viszont elkerülhetők, de az *unexpected()* hívásokat is el lehet fogni és ártalmatlanná tenni.

Egy megfelelően kidolgozott *Y* alrendszer gyakran az *Yerr* osztályból származtatja a kivételeit:

```
class Some_Yerr : public Yerr { /* ... */ };
```

A fenti esetében a

```
void f() throw (Xerr, Yerr, exception);
```

minden *Yerr*-t továbbítani fog a hívójának. Így tehát *f()* a *Some_Yerr* osztályú hibát is továbbítani fogja és *f()*-ben semmilyen *Yerr* nem fog *unexpected()* hívást kiváltani.

A standard könyvtár által kiváltott összes kivétel az *exception* osztály (§14.10) leszármazottja.

14.6.3. Kivételek leképezése

A programot leállítani kezeletlen kivétel fellépése esetén néha túl könyörtelen eljárás. Ilyenkor az *unexpected()* viselkedését kell elfogadhatóbbá tenni.

Ennek az a legegyszerűbb módja, hogy a standard könyvtárbeli *std::bad_exception*-t felvesszük a specifikált kivételek közé. Ekkor az *unexpected()* egyszerűen egy *bad_exception*-t fog kiváltani egy kezelőfüggvény meghívása helyett:

```
class X { };
class Y { };

void f() throw(X,std::bad_exception)
{
    // ...
    throw Y();      // bad_exception-t vált ki
}
```

A kivételkezelő ekkor elkapja az elfogadhatatlan *Y* kivételt és *bad_exception* típusút vált ki helyette. A *bad_exception*-nel semmi baj, a *terminate()*-nél kevésbé drasztikus – de azért így is meglehetősen durva beavatkozás, ráadásul az információ, hogy milyen kivétel okozta a problémát, elvész.

14.6.3.1. Kivételek felhasználói leképezése

Vegyünk egy *g()* függvényt, amely nem hálózati környezethez készült. Tételezzük fel, hogy *g()* csak a saját „Y alrendszerével" kapcsolatos kivételek kiváltását engedélyezi. Tegyük fel, hogy *g()*-t hálózati környezetben kell meghívnunk.

```
void g() throw(Yerr);
```

Természetesen *g()* nem fog tudni semmit a hálózati kivételekről és az *unexpected()*-et fogja meghívni, ha ilyennel találkozik. A *g()* hálózati környezetben való használatához olyan kódra van szükségünk, amely hálózati kivételeket kezel, vagy át kell írnunk *g()*-t. Tegyük fel, hogy az újraírás nem lehetséges vagy nem kívánatos. Ekkor a problémát az *unexpected()* jelentésének felülbírálásával oldhatjuk meg.

A memória elfogyását a *_new_handler* kezeli, amelyet a *set_new_handler()* állít be. Ugyanígy a váratlan kivételre adott választ egy *_unexpected_handler* határozza meg, amelyet az *<exception>* fejállományban megadott *std::set_unexpected()* állít be:

```
typedef void(*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler);
```

A váratlan kivételek megfelelő kezeléséhez létre kell hoznunk egy osztályt, hogy a „kezdeti értékadás az erőforrás megszerzésével” módszert használhassuk az *unexpected()* függvényekben:

```
class STC {            // tárol és visszaállít
    unexpected_handler old;
public:
    STC(unexpected_handler f) { old = set_unexpected(f); }
    ~STC() { set_unexpected(old); }
};
```

Ezután definiálunk egy függvényt az *unexpected()* erre az esetre kívánt jelentésével:

```
class Yunexpected : public Yerr { };

void throwY() throw(Yunexpected) { throw Yunexpected(); }
```

A *throwY()*-t *unexpected()*-ként használva bármilyen kivételből *Yunexpected* lesz.

Végül elkészíthetjük *g()*-nek egy hálózati környezetben alkalmazható változatát:

```
void networked_g() throw(Yerr)
{
    STC xx(&throwY);      // az unexpected() most Yunexpected-et vált ki
    g();
}
```

Mivel az *Yunexpected* az *Yerr*-ből származik, a kivétel-specifikáció nem sérül. Ha a *throwY()* olyan kivételt váltott volna ki, amit a specifikáció nem engedélyez, akkor a *terminate()* hajtódott volna végre.

Azáltal, hogy mentettük és visszaállítottuk az *_unexpected_handler*-t, több alrendszer számára tettük lehetővé – egymásra való hatás nélkül – a váratlan kivételek kezelését. A váratlan kivételek engedélyezetté alakításának e módszere alapvetően a rendszer által a *bad_exception*-nal nyújtott szolgáltatás rugalmasabb változata.

14.6.3.2. Kivételek típusának visszaállítása

A váratlan kivételek *Yunexpected*-dé alakítása lehetővé tenné a *networked_g()* felhasználójának, hogy megtudja: egy váratlan kivételt képeztünk le az *Yunexpected*-re. Azt azonban nem fogja tudni, hogy melyik kivétel leképezése történt meg. Egy egyszerű eljárással lehetővé tehetjük ennek megjegyzését és továbbítását. Például így tudnánk információt gyűjteni a *Network_exception*-ökről:

```
class Yunexpected : public Yerr {
public:
    Network_exception* pe;
    Yunexpected(Network_exception* p) :pe(p?p->clone():0) { }
    ~Yunexpected() { delete p; }
};

void throwY() throw(Yunexpected)
{
    try {
            throw;                          // A továbbdobott kivételt azonnal el kell kapni!
    }
    catch(Network_exception& p) {
            throw Yunexpected(&p);          // leképezett kivétel kiváltása
    }
    catch(...) {
            throw Yunexpected(0);
    }
}
```

A kivételek továbbdobása és elkapása lehetővé teszi, hogy azon típusok összes kivételét kezeljük, amelyeket meg tudunk nevezni. A *throwY()* függvényt az *unexpected()* hívja meg, azt pedig elvileg egy *catch(...)* kezelő. Így tehát biztosan van továbbdobható kivétel. Egy *unexpected()* függvény nem tekinthet el a hibától és nem térhet vissza. Ha megpróbál így tenni, az *unexpected()* maga fog *bad_expection*-t kiváltani (§14.6.3). A *clone()* függvényt arra használjuk, hogy a kivétel egy másolata számára helyet foglaljon a szabad memóriában. Ez a másolat túléli a verem „visszatekerését”.

14.7. El nem kapott kivételek

Ha egy kivételt nem kapnak el, akkor az *std::terminate()* meghívására kerül sor. A *terminate()* fog meghívódni akkor is, ha a kivételkezelő eljárás a vermet sérültnek találja, vagy ha egy, a verem visszatekerése során meghívott destruktor kivétel kiváltásával próbál véget érni.

A váratlan kivételeket az *_unexpected_handler* kezeli, amelyet az *std::set_unexpected()* állít be. Ehhez hasonlóan, az el nem kapott kivételek kezelését az *_uncaught_handler* végzi, amelyet az *<exception>* fejállományban megadott *std::set_uncaught()* állít be:

```
typedef void(*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
```

A visszatérési érték a *set_terminate()*-nek előzőleg adott függvény.

A *terminate()* meghívásának oka, hogy időnként a kivételkezeléssel fel kell hagyni kevésbé kifinomult hibakezelési módszerek javára. A *terminate()*-et például használhatjuk arra, hogy megszakítsunk egy folyamatot vagy esetleg újraindítsunk egy rendszert (új kezdeti értékadással). A *terminate()* meghívása drasztikus intézkedés: akkor kell használni, amikor a kivételkezelő eljárás által megvalósított hibakezelő stratégia csődöt mondott és ideje a hibatűrés más szintjére áttérni.

A *terminate()* alapértelmezés szerint az *abort()*-ot hívja meg (§9.4.1.1). Ez az alapértelmezés a legtöbb felhasználó számára megfelelő választás, különösen a program hibakeresése (debugging) alatt.

Az *_uncaught_handler* függvényekről a rendszer feltételezi, hogy nem fognak visszatérni a hívójukhoz. Ha az adott függvény megpróbálja, a *terminate()* meg fogja hívni az *abort()*-ot.

Jegyezzük meg, hogy az *abort()* a programból való nem szabályos kilépést jelent. Az *exit()* függvény visszatérési értékével jelezhetjük a rendszernek, hogy a programból szabályosan vagy nem szabályosan léptünk-e ki (§9.4.1.1).

Az adott nyelvi változat határozza meg, hogy a program futásának el nem kapott kivétel miatti befejeződésénél a destruktorok meghívódnak-e. Bizonyos rendszereknél alapvető, hogy a destruktorok ne hívódjanak meg, hogy a program futtatását folytatni lehessen a hibakeresőből. Más rendszerek számára felépítésükből adódóan szinte lehetetlen nem meghívni a destruktorokat a kivétel kezelőjének keresésekor.

Ha biztosítani akarjuk a rendrakást egy el nem kapott kivétel esetében, a *main()* függvényben a ténylegesen elkapni kívánt kivételek mellé írhatunk egy minden kivételt elkapó kezelőt is (§14.3.2):

```
int main()
try {
    // ...
}
```

```
catch (std::range_error)
{
    cerr << "Tartományhiba: már megint!\n";
}
catch (std::bad_alloc)
{
    cerr << "A new kifogyott a memóriából.\n";
}
catch (...) {
    // ...
}
```

Ez a globális változók konstruktorai és destruktorai által kiváltottak kivételével minden kivételt elkap. A globális változók kezdeti értékadása alatt fellépő kivételeket nem lehet elkapni. Egy nem lokális statikus objektum kezdeti értékadása közbeni *throw* esetében csak a *set_unexpected()* (§14.6.2) segítségével kaphatjuk meg a vezérlést, ami újabb ok arra, hogy lehetőség szerint kerüljük a globális változókat.

A kivételek kiváltásánál a kivétel fellépésének pontos helye általában nem ismert, vagyis kevesebb információt kapunk, mint amit egy hibakereső (debugger) tudhat a program állapotáról. Ezért bizonyos C++ fejlesztőkörnyezetek, programok vagy fejlesztők számára előnyösebb lehet nem elkapni azokat a kivételeket, amelyek következményeit a programban nem küszöböljük ki.

14.8. A kivételek és a hatékonyság

Elvileg lehetséges a kivételkezelést úgy megtervezni, hogy az ne járjon a futási idő növekedésével, ha nem kerül sor kivétel kiváltására. Ráadásul ezt úgy is meg lehet tenni, hogy egy kivétel kiváltása ne legyen különösebben költséges egy függvényhíváshoz képest. Ha azonban a memóriaigényt is csökkenteni szeretnénk, illetve a kivételkezelést a C hívási sorrendjével és a hibakeresők szabványos eljárásaival is össze szeretnénk egyeztetni, ha nem is lehetetlen, de nehéz feladatra vállalkozunk – de emlékezzünk arra, hogy a kivételek használatának alternatívái sincsenek ingyen. Nem szokatlan olyan hagyományos rendszerekkel találkozni, amelyeknél a kód felét a hibakeresésre szánták.

Vegyünk egy egyszerű *f()* függvényt, amelynek látszólag semmi köze nincs a kivételkezeléshez:

```
void g(int);

void f()
{
    string s;
    // ...
    g(1);
    g(2);
}
```

Ám *g()* kivételt válthat ki, így *f()*-nek tartalmaznia kell a kivétel fellépte esetén *s*-et megsemmisítő kódot. Ha *g()* nem váltott volna ki kivételt, valamilyen más módon kellett volna a hibát jeleznie. Így a fentivel összehasonlítható hagyományos kód, amely kivételek helyett hibákat kezel, nem a fenti egyszerű kód lenne, hanem valami ilyesmi:

```
bool g(int);

bool f()
{
    string s;
    // ...
    if (g(1))
            if (g(2))
                    return true;
            else
                    return false;
    else
            return false;
}
```

A programozók szokás szerint persze nem kezelik ennyire módszeresen a hibákat, és az nem is mindig létfontosságú. De ha gondos és módszeres hibakezelés szükséges, akkor jobb azt a számítógépre, vagyis a kivételkezelő rendszerre hagyni.

A kivételek specifikációja (§14.6) nagyon hasznos lehet a fordító által létrehozott kód javítására. Ha az alábbi módon kijelentettük volna, hogy *g()* nem vált ki kivételt, akkor az *f()* számára létrehozott kódon javíthattunk volna:

```
void g(int) throw();
```

Érdemes megjegyezni, hogy a hagyományos C függvények nem váltanak ki kivételt, így a legtöbb programban az összes C függvény az üres *throw()* kivétel-meghatározással adható meg. Az adott nyelvi változat tudhatja, hogy a C-nek csak néhány standard könyvtárbeli függvénye vált ki kivételt, például az *atexit()* és a *qsort()*, és ennek ismeretében jobb kódot készíthet.

Mielőtt egy „C függvényt" ellátnánk az üres kivétel-meghatározással (a *throw()*-val), gondoljuk meg, nem válthat-e ki kivételt. Például átírhatták, hogy a *new* operátort használja, amely viszont *bad_alloc*-ot válthat ki vagy olyan C++-könyvtárat hívhat meg, amely ugyanezt teheti.

14.9. A hibakezelés egyéb módjai

A kivételkezelő eljárás célja, hogy lehetővé tegye egy programrész számára, hogy a program más részeit értesítse egy kivételes körülmény észleléséről. A feltevés az, hogy a két rész függetlenül készült és a kivételt kezelő rész tud valami értelmeset kezdeni a hibával.

A kivételkezelők hatékony használatához átfogó stratégiára van szükségünk. Vagyis a program különböző részeinek egyet kell érteniük abban, hogyan használják a kivételeket és hol kezelik a hibákat. A kivételkezelő eljárás lényegénél fogva nem lokális, így alapvető jelentőségű, hogy átfogó stratégia érvényesüljön. Ebből következik, hogy a hibakezelés módjára legjobb a rendszer tervezésének legkorábbi szakaszában gondolni, és hogy az alkalmazott módszernek (a teljes program összetettségéhez képest) egyszerűnek és pontosan meghatározottnak kell lennie. Egy, a lényegénél fogva annyira kényes területen, mint a hibakezelés, egy bonyolult módszerhez nem tudnánk következetesen alkalmazkodni.

Először is fel kell adni azt a tévhitet, hogy az összes hibát egyetlen eljárással kezelhetjük; ez csak bonyolítaná a helyzetet. A sikeres hibatűrő rendszerek többszintűek. Mindegyik szint annyi hibával birkózik meg, amennyivel csak tud, anélkül, hogy túlságosan megszenvedne vele, a maradék kezelését viszont a magasabb szintekre hagyja. A *terminate()* ezt a kezelési módot azzal támogatja, hogy menekülési utat hagy arra az esetre, ha maga a kivételkezelő rendszer sérülne vagy nem tökéletes használata miatt maradnának nem elkapott kivételek. Az *unexpected()* célja ugyanígy az, hogy menekülési utat hagyjon arra az esetre, ha a kivételek specifikációjára épülő hibakezelő rendszer mégsem jelentene a kivételek számára áthatolhatatlan falat.

Nem minden függvény kell, hogy „tűzfalként" viselkedjen. A legtöbb rendszerben nem lehet minden függvényt úgy megírni, hogy vagy teljes sikerrel járjon vagy egy pontosan meghatározott módon legyen sikertelen. Hogy ez miért nem lehetséges, az programról programra és programozóról programozóra változik. Nagyobb programok esetében a következő okokat sorolhatjuk fel:

1. Túl sok munka kellene ennek a „megbízhatóságnak" olyan biztosításához, hogy következetesen mindenhol érvényesüljön.
2. A szükséges tárterület és futási idő növekedése nagyobb lenne az elfogadhatónál (mert rendszeresen ugyanazon hibák – például érvénytelen paraméterek – ismételt ellenőrzésére kerül sor).
3. Más nyelven írt függvények nem alkalmazkodnának a szabályokhoz.
4. Ez a pusztán helyi értelemben vett „megbízhatóság" olyan bonyolultsághoz vezetne, amely végül aláásná a teljes rendszer megbízhatóságát.

Ugyanakkor egy program olyan különálló részrendszerekre bontása, amelyek vagy teljes sikerrel járnak, vagy meghatározott módon lesznek sikertelenek, alapvető, megtehető és gazdaságos. Egy főbb könyvtárat, részrendszert vagy kulcsfontosságú függvényt így kell megtervezni. A kivételeket ilyen könyvtárak vagy részrendszerek felületei számára célszerű meghatározni.

Rendszerint nem adatik meg az a luxus, hogy egy rendszer összes kódját „a nulláról" készíthessük el. Ezért egy általános, minden programrészre kiterjedő hibakezelési stratégia megalkotásakor figyelembe kell vennünk az egyes, a miénktől eltérő módszert alkalmazó programrészleteket is. Ehhez pedig tekintetbe kell vennünk olyan kérdéseket, mint hogy a programrészlet hogyan kezeli az erőforrásokat, vagy milyen állapotba kerül egy hibája után a rendszer. Az a cél, hogy a programrészlet látszólag akkor is az általános hibakezelési módszer szerint kezelje a hibákat, ha valójában eltérő belső eljárást alkalmaz.

Alkalmasint szükséges lehet az egyik stílusú hibajelzésről a másikra áttérni. Például egy C könyvtári függvény meghívása után ellenőrizhetjük az *errno*-t és kivételt válthatunk ki, vagy fordítva, elkaphatunk egy kivételt és az *errno*-t beállíthatjuk, mielőtt a C++-könyvtárból visszatérünk egy C programba:

```
void callC() throw(C_blewit)
{
    errno = 0;
    c_function();
    if (errno) {
            // takarítás (ha lehetséges és szükséges)
            throw C_blewit(errno);
    }
}

extern "C" void call_from_C() throw()
{
```

```
        try {
                c_plus_plus_function();
        }
        catch (...) {
                // takarítás (ha lehetséges és szükséges)
                errno = E_CPLPLFCTBLEWIT;
        }
}
```

Ilyenkor fontos, hogy következetesen járjunk el, hogy a hibajelentő stílusok átalakítása teljes legyen.

A hibakezelés – a lehetőségekhez mérten – hierarchikus legyen. Ha egy függvény futási idejű hibát észlel, ne kérjen segítséget vagy erőforrást a hívójától. Az ilyen kérések az egymástól függő elemek között körbe-körbe járó végtelen ciklusokat okoznak, ami a programot áttekinthetetlenné teszi, a végtelen ciklusok lehetőségével pedig a hibákat kezelő kódba egy nem kívánatos lehetőséget épít.

Alkalmazzunk olyan egyszerűsítő módszereket, mint a „kezdeti értékadás az erőforrás megszerzésével", illetve olyan egyszerűsítő feltevéseket, mint „a kivételek hibákat jelentenek". Ezzel a hibakezelő kódot szabályosabbá tehetjük. Lásd még a §24.3.7.1 pontot, ahol elmagyarázzuk, hogyan lehet állapotbiztosítókat (invariánsokat) és feltevéseket használni, hogy a kivételek kiváltása szabályosabb legyen.

14.10. Szabványos kivételek

A következő táblázat a szabványos kivételeket és az azokat kiváltó függvényeket, operátorokat, általános eszközöket mutatja be:

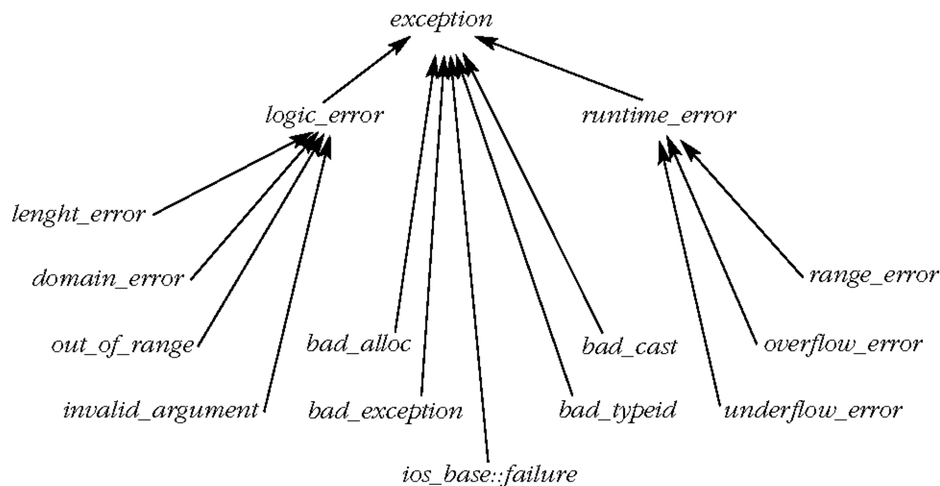
Szabványos (a nyelvbe beépített) kivételek			
Név	Kiváltó	Hivatkozás	Fejállomány
bad_alloc	*new*	§6.2.6.2, §19.4.5	<*new*>
bad_cast	*dynamic_cast*	§15.4.1.1	<*typeinfo*>
bad_typeid	*typeid*	§15.4.4	<*typeinfo*>
bad_exception	*kivétel_specifikáció*	§14.6.3	<*exception*>

Szabványos (a standard könyvtár által kiváltott) kivételek			
Név	Ki dobja	Hivatkozás	Fejállomány
out_of_range	*at()*	§16.3.3,§20.3.3	*<stdexcept>*
	bitset<>::operator[]()	§17.5.3	*<stdexcept>*
invalid_argument	*bitset konstruktor*	§17.5.3.1	*<stdexcept>*
overflow_error	*bitset<>::to_ulong()*	§17.5.3.3	*<stdexcept>*
ios_base::failure	*ios_base::clear()*	§21.3.6	*<ios>*

A könyvtári kivételek a standard könyvtár *exception* nevű – az *<exception>* fejállományban megadott – kivételosztályból kiinduló osztályhierarchia tagjai:

```
class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();

    virtual const char* what() const throw();
private:
    // ...
};
```

A hierarchia így néz ki:



Ez meglehetősen körülményesnek tűnik ahhoz képest, hogy nyolc szabványos kivételt ölel fel. Oka, hogy a hierarchia megpróbál a standard könyvtárban meghatározott kivételeken kívüli kivételek számára is használható keretrendszert adni. A logikai hibák (logic error) azok, amelyeket elvileg a program indulása előtt, függvény- vagy konstruktorparaméterek ellenőrzésével el lehetne kapni. A többi futási idejű hiba (run-time error). Némelyek ezt az összes hiba és kivétel számára hasznos keretrendszernek látják – én nem.

A standard könyvtárbeli kivételek az *exception* által meghatározott függvényeket nem bővítik újakkal, csak megfelelően definiálják a megkívánt virtuális függvényeket. Így az alábbit írhatjuk:

```
void f()
try {
    // a standard könyvtár használata
}
catch (exception& e) {
    cout << "standard könyvtári kivétel " << e.what() << '\n';        // talán
    // ...
}
catch (...) {
    cout << "másik kivétel\n";
    // ...
}
```

A standard könyvtárbeli kivételek az *exception*-ből származnak, a többi viszont nem feltétlenül, így hiba lenne az összes kivételt az *exception* elkapásával megpróbálni lekezelni. Hasonlóan hiba lenne feltételezni, hogy az összes, az *exception*-ből származó kivétel standard könyvtárbeli kivétel, a felhasználók ugyanis hozzáadhatják saját kivételeiket az *exception* hierarchiához.

Jegyezzük meg, hogy az *expection*-műveletek maguk nem váltanak ki kivételt. Ebből következően egy standard könyvtárbeli kivétel kiváltása nem vált ki *bad_alloc* kivételt. A kivételkezelő rendszer fenntart a saját céljaira egy kis memóriát (például a veremben), hogy ott tárolhassa a kivételeket. Természetesen írhatunk olyan kódot, amely a rendszer összes memóriáját elfogyasztja és így hibát kényszerít ki.

Íme egy függvény, amit ha meghívunk, kipróbálja, hogy a függvényhívásnál vagy a kivételkezelésnél fogy-e ki először a memória:

```
void perverted()
{
    try {
        throw exception();                          // ismétlődő kivétel
    }
```

```
        catch (exception& e) {
                perverted();                              // ismétlődő függvényhívás
                cout << e.what();
        }
}
```

A kimeneti utasítás egyedül azt a célt szolgálja, hogy megakadályozza a fordítóprogramot az *e* nevű kivétel által felhasznált memória újrahasznosításában.

14.11. Tanácsok

[1] Hibakezelésre használjunk kivételeket. §14.1, §14.5, §14.9.
[2] Ne használjunk kivételeket, ha a helyi vezérlési szerkezetek elégségesek. §14.1.
[3] Az erőforrások kezelésére a „kezdeti értékadás az erőforrás megszerzésével" módszert használjuk. §14.4.
[4] Nem minden programnak kell „kivételbiztosnak" lennie. §14.4.3.
[5] Az invariánsok érvényességének megőrzésére használjuk a „kezdeti értékadás az erőforrás megszerzésével" módszert és a kivételkezelőket. §14.3.2.
[6] Minél kevesebb *try* blokkot használjunk. Meghatározott kezelőkód helyett a „kezdeti értékadás az erőforrás megszerzésével" módszert használjuk. §14.4.
[7] Nem minden függvénynek kell minden lehetséges hibát kezelnie. §14.9.
[8] A konstruktorhibák jelzésére váltsunk ki kivételt. §14.9.
[9] Ha egy értékadás vált ki kivételt, előtte gondoskodjon paramétereinek következetes állapotba hozásáról. §14.4.6.2.
[10] A destruktorokban kerüljük a kivételek kiváltását. §14.4.7.
[11] A *main()* függvény kapja el és jelentse az összes hibát. §14.7.
[12] Különítsük el a közönséges kódot a hibakezeléstől. §14.4.5 és §14.5.
[13] Gondoskodjunk arról, hogy egy konstruktorban minden lefoglalt erőforrást felszabadítsunk, ha a konstruktor kivételt vált ki. §14.4.
[14] Az erőforrások kezelése hierarchikus legyen. §14.4.
[15] A főbb felületekben határozzuk meg az engedélyezett kivételeket. §14.9.
[16] Legyünk résen, nehogy a *new* által lefoglalt és a kivételek fellépte esetén fel nem szabadított memória elszivárogjon. §14.4.1, §14.4.2, §14.4.4.
[17] A függvényekről tételezzük fel, hogy minden kivételt kiválthatnak, amit megengedtek nekik. §14.6.
[18] Ne tételezzük fel, hogy minden kivétel az *exception* osztály leszármazottja. §14.10.
[19] Egy könyvtár ne fejezze be egyoldalúan a program futását. Ehelyett váltson ki kivételt és hadd döntsön a hívó. §14.1.

[20] Egy könyvtár ne bocsásson ki a végfelhasználónak szánt diagnosztikai üzeneteket. Ehelyett váltson ki kivételt és hadd döntsön a hívó. §14.1.
[21] A tervezés során minél hamarabb alakítsuk ki a hibakezelési stratégiát. §14.9

14.12. Gyakorlatok

1. (*2) Általánosítsuk a §14.6.3.1 pontbeli *STC* osztályt sablonná, amely a „kezdeti értékadás az erőforrás megszerzésével" módszert használja különféle típusú függvények tárolására és visszaállítására.
2. (*3) Egészítsük ki a §11.11 pontbeli *Ptr_to_T* osztályt mint egy olyan sablont, amely kivételekkel jelzi a futási idejű hibákat.
3. (*3) Írjunk függvényt, amely egy bináris fa csúcsaiban keres egy *char** típusú mező alapján. A *hello*-t tartalmazó csúcs megtalálásakor a *find("hello")* a csúcsra hivatkozó mutatóval térjen vissza; a keresés sikertelenségét kivétellel jelezzük.
4. (*3.) Hozzunk létre egy *Int* osztályt, amely pontosan úgy viselkedik, mint az elemi *int* típus, csak túl- és alulcsordulás esetén kivételt vált ki.
5. (*2.5) Vegyük a felhasznált operációs rendszer C felületének állományok megnyitására, lezárására, olvasására, írására való alapvető műveleteit, és írjunk hozzájuk egyenértékű C++ függvényeket, amelyek a megfelelő C függvényeket hívják meg, de hiba esetén kivételt váltanak ki.
6. (*2.5) Írjunk egy teljes *Vector* sablont *Range* és *Size* kivételekkel.
7. (*1) Írjunk egy ciklust, ami kiszámítja a §14.12[6]-beli *Vector* összegét a *Vector* méretének lekérdezése nélkül. Ez miért nem jó ötlet?
8. (*2.5) Gondoljuk meg, mi lenne, ha egy *Exception* osztályt használnánk az összes kivételként használt osztály őseként (bázisosztályaként). Hogyan nézne ki? Hogyan lehetne használni? Mire lenne jó? Milyen hátrányok származnának abból a megkötésből, hogy ezt az osztályt kell használni?
9. (*1) Adott a következő függvény:

   ```
   int main() { /* ... */ }
   ```

   Változtassuk meg úgy, hogy minden kivételt elkapjon, hibaüzenetté alakítsa azokat, majd meghívja az *abort()*-ot. Vigyázat: a §14.9 pontbeli *call_from_C()* függvény nem teljesen kezel minden esetet.
10. (*2) Írjunk visszahívások (callback) megvalósítására alkalmas osztályt vagy sablont.
11. (*2.5) Írjunk egy *Lock* osztályt, amely valamely rendszer számára a konkurens hozzáférést támogatja.

15

Osztályhierarchiák

„Az absztrakció szelektív tudatlanság.”
(Andrew Koenig)

Többszörös öröklődés • A többértelműség feloldása • Öröklődés és *using* deklarációk • Ismétlődő bázisosztályok • Virtuális bázisosztályok • A többszörös öröklődés használata • Hozzáférés-szabályozás • Védett tagok • Bázisosztályok elérése • Futási idejű típusinformáció • *dynamic_cast* • Statikus és dinamikus konverzió • *typeid* • Kiterjesztett típusinformáció • A futási idejű típusinformáció helyes és helytelen használata • Tagra hivatkozó mutatók • Szabad tár • „Virtuális konstruktorok” • Tanácsok • Gyakorlatok

15.1. Bevezetés és áttekintés

Ez a fejezet a származtatott osztályoknak és a virtuális függvényeknek a más nyelvi elemekkel, például a hozzáférés-szabályozással, a névfeloldással, a szabad tár kezelésével, a konstruktorokkal, a mutatókkal és a típuskonverziókkal való kölcsönhatását tárgyalja. Öt fő részből áll:

§15.2 Többszörös öröklődés
§15.3 Hozzáférés-szabályozás

§15.4 Futási idejű típusazonosítás
§15.5 Tagokra hivatkozó mutatók
§15.6 A szabad tár használata

Az osztályokat általában bázisosztályok „hálójából” hozzuk létre. Mivel a legtöbb ilyen háló hagyományosan fa szerkezetű, az osztályokból álló hálókat vagy *osztályhálókat* (class lattice) gyakran nevezik *osztályhierarchiának* (class hierarchy) is. Az osztályokat célszerű úgy megtervezni, hogy a felhasználóknak ne kelljen indokolatlan mértékben azzal törődniük, hogy egy osztály milyen módon épül fel más osztályokból. Így például a virtuális függvények meghívási módja biztosítja, hogy ha egy *f()* függvényt meghívunk egy objektumra, akkor mindig ugyanaz a függvény hajtódik végre, függetlenül attól, hogy a hierarchia melyik osztálya deklarálta a függvényt a meghíváshoz. Ez a fejezet az osztályhálók összeállításának és az osztálytagok elérésének módjairól, illetve az osztályhálók fordítási és futási idejű bejárásának eszközeiről szól.

15.2. Többszörös öröklődés

Ahogy a §2.5.4 és §12.3 pontokban láttuk, egy osztálynak több közvetlen bázisosztálya is lehet, azaz több osztályt is megadhatunk a *:* jel után az osztály deklarációjában. Vegyünk egy szimulációs programot, ahol a párhuzamos tevékenységeket a *Task* (Feladat) osztállyal, az adatgyűjtést és -megjelenítést pedig a *Displayed* (Megjelenítés) osztállyal ábrázoljuk. Ekkor olyan szimulált egyedeket határozhatunk meg, mint a *Satellite* (Műhold):

```
class Satellite : public Task, public Displayed {
    // ...
};
```

Több közvetlen bázisosztály használatát szokás szerint *többszörös öröklődésnek* (multiple inheritance) nevezik. Az *egyszeres öröklődésnél* (single inheritance) csak egy közvetlen bázisosztály van.

A *Satellite*-okra saját műveleteiken kívül a *Task*-ok és *Displayed*-ek műveleteinek uniója is alkalmazható:

```
void f(Satellite& s)
{
    s.draw();       // Displayed::draw()
    s.delay(10);    // Task::delay()
    s.transmit();   // Satellite::transmit()
}
```

Hasonlóan, ha egy függvény *Task* vagy *Displayed* paramétert vár, akkor adhatunk neki egy *Satellite*-ot is:

```
void highlight(Displayed*);
void suspend(Task*);

void g(Satellite* p)
{
    highlight(p);  // mutató átadása a Satellite Displayed részére
    suspend(p);    // mutató átadása a Satellite Task részére
}
```

A program létrehozása nyilván valamilyen egyszerű eljárás alkalmazását követeli meg a fordítóprogramtól, hogy a *Task*-ot váró függvények más részét lássák egy *Satellite*-nak, mint a *Displayed*-et várók. A virtuális függvények a megszokott módon működnek:

```
class Task {
    // ...
    virtual void pending() = 0;
};

class Displayed {
    // ...
    virtual void draw() = 0;
};

class Satellite : public Task, public Displayed {
    // ...
    void pending();        // felülírja a Task::pending() függvényt
    void draw();           // felülírja a  Displayed::draw() függvényt
};
```

Ez biztosítja, hogy *Satellite::draw()*, illetve *Satellite::pending()* fog meghívódni, ha egy *Satellite*-ot *Displayed*-ként, illetve *Task*-ként kezelünk.

Jegyezzük meg, hogy ha csak egyszeres öröklődést használhatnánk, akkor ez a körülmény korlátozná a programozót a *Satellite*, *Displayed* és *Task* osztályok megvalósításának megválasztásában. Egy *Satellite* vagy *Task*, vagy *Displayed* lehetne, de nem mindkettő (hacsak a *Task* nem a *Displayed*-ből származik vagy fordítva). Ezen lehetőségek mindegyike csökkenti a rugalmasságot.

Mi szüksége lehet bárkinek egy *Satellite* osztályra? Nos, bármilyen meglepő, a *Satellite* példát a valóságból merítettük. Volt – és talán még mindig van – egy olyan program, amely a többszörös öröklődés leírására itt használt minta szerint épült fel. A program műholdakat,

földi állomásokat stb. magába foglaló hírközlési rendszerek szerkezetének tanulmányozására szolgált. Egy ilyen szimuláció birtokában meg tudjuk válaszolni a forgalmi adatokra vonatkozó kérdéseket, meg tudjuk határozni, mi történik, ha egy földi állomást vihar akadályoz, műholdas és földi kapcsolatok előnyeit-hátrányait tudjuk elemezni stb. A különböző körülmények utánzása számos megjelenítési és hibakeresési műveletet igényel, és szükség van a *Satellite*-okhoz hasonló objektumok, illetve részegységeik állapotának tárolására is, az elemzés, illetve a hibakeresés és -elhárítás céljából.

15.2.1. A többértelműség feloldása

Két bázisosztálynak lehetnek azonos nevű tagfüggvényei is:

```
class Task {
    // ...
    virtual debug_info* get_debug();
};

class Displayed {
    // ...
    virtual debug_info* get_debug();
};
```

Ha egy *Satellite-ot* használunk, akkor ezeket a függvényeket egyértelműen kell megneveznünk:

```
void f(Satellite* sp)
{
    debug_info* dip = sp->get_debug();          // hiba: többértelmű
    dip = sp->Task::get_debug();                // rendben
    dip = sp->Displayed::get_debug();           // rendben
}
```

Az explicit megnevezés azonban zavaró, ezért általában legjobb elkerülni az ilyen problémákat. Ennek legegyszerűbb módja, ha a származtatott osztályban készítünk egy új függvényt:

```
class Satellite : public Task, public Displayed {
    // ...

    debug_info* get_debug()          // felülírja a Task::get_debug() és
                                     // Displayed::get_debug() függvényeket
    {
```

```
        debug_info* dip1 = Task::get_debug();
        debug_info* dip2 = Displayed::get_debug();
        return dip1->merge(dip2);
    }
};
```

Ezáltal a *Satellite* bázisosztályaira vonatkozó információk felhasználását helyhez kötöttük. Mivel a *Satellite::get_debug()* elfedi mindkét bázisosztályának *get_debug()* függvényét, a *Satellite::get_debug()* hívódik meg, valahányszor *get_debug()*-ot hívunk meg egy *Satellite* objektumra.

A *Telstar::draw* minősített név a *Telstar*-ban vagy valamelyik bázisosztályában megadott *draw*-ra vonatkozhat:

```
class Telstar : public Satellite {
    // ...
    void draw()
    {
        draw();                          // hoppá!: rekurzív hívás
        Satellite::draw();               // megtalálja a Displayed::draw-t
        Displayed::draw();
        Satellite::Displayed::draw();    // felesleges kétszeri minősítés
    }
};
```

Vagyis ha a *Satellite::draw* nem a *Satellite* osztályban bevezetett *draw*-t jelenti, akkor a fordítóprogram végignézi a bázisosztályokat, vagyis *Task::draw*-t és *Displayed::draw*-t keres. Ha csak egyet talál, akkor azt fogja használni, ha többet vagy egyet sem, a *Satellite::draw* ismeretlen vagy többértelmű lesz.

15.2.2. Öröklődés és using deklarációk

A túlterhelés feloldására nem kerül sor különböző osztályok hatókörén keresztül (§7.4), a különböző bázisosztályok függvényei közötti többértelműségek feloldása pedig nem történik meg a paramétertípus alapján.

Egymással alapvetően nem rokon osztályok egyesítésekor, például a *Task* és *Displayed* osztályok *Satellite*-té való „összegyúrásánál” az elnevezésekben megmutatkozó hasonlóság általában nem jelent közös célt. Amikor az ilyen nevek ütköznek, ez gyakran meglepetésként éri a felhasználót:

```
class Task {
    // ...
    void debug(double p);          // információ kiírása csak akkor, ha a prioritás
                                   // alacsonyabb p-nél
};

class Displayed {
    // ...
    void debug(int v);     // minél nagyobb 'v,' annál több hibakeresési információ íródik ki
};

class Satellite : public Task, public Displayed {
    // ...
};

void g(Satellite* p)
{
    p->debug(1);                   // hiba, többértelmű: Displayed::debug(int) vagy
                                   // Task::debug(double) ?
    p->Task::debug(1);             // rendben
    p->Displayed::debug(1);        // rendben
}
```

De mi van akkor, ha a különböző bázisosztályokban tudatos tervezési döntés következtében szerepelnek azonos nevek, és a felhasználó számára kívánatos lenne a paramétertípus alapján választani közülük? Ebben az esetben a függvényeket a *using* deklarációval (§8.2.2) hozhatjuk közös hatókörbe:

```
class A {
public:
    int f(int);
    char f(char);
    // ...
};

class B {
public:
    double f(double);
    // ...
};

class AB: public A, public B {
public:
    using A::f;
    using B::f;
    char f(char);   // elfedi A::f(char)-t
    AB f(AB);
};
```

```
void g(AB& ab)
{
    ab.f(1);            // A::f(int)
    ab.f('a');          // AB::f(char)
    ab.f(2.0);          // B::f(double)
    ab.f(ab);           // AB::f(AB)
}
```

A *using* deklarációk lehetővé teszik, hogy a bázis- és származtatott osztályok függvényeiből túlterhelt függvények halmazát állítsuk elő. A származtatott osztályban megadott függvények elfedik (hide) a bázisosztály függvényeit, amelyek egyébként elérhetőek lennének. A bázisosztály virtuális függvényei ugyanúgy felülírhatók (override), mint egyébként (§15.2.3.1).

Egy osztálydeklaráció *using* deklarációjának (§8.2.2) egy bázisosztály tagjára kell vonatkoznia. Egy osztály tagjára vonatkozó *using* deklaráció nem szerepelhet az osztályon, annak származtatott osztályán, illetve annak tagfüggvényein kívül, a *using* direktívák (§8.2.3) pedig nem szerepelhetnek egy osztály definiciójában és nem vonatkozhatnak osztályra.

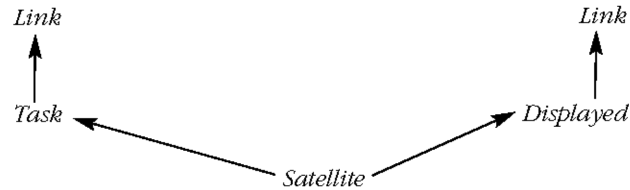
A *using* deklarációk nem szolgálhatnak kiegészítő információ elérésére sem, csak az egyébként is hozzáférhető információk kényelmesebb használatát teszik lehetővé (§15.3.2.2).

15.2.3. Ismétlődő bázisosztályok

Azáltal, hogy több bázisosztály lehet, előfordulhat, hogy egy osztály kétszer fordul elő a bázisosztályok között. Például ha mind a *Task*, mind a *Displayed* a *Link* (Kapcsolat) osztályból származott volna, a *Satellite*-oknak két *Link*-je lenne:

```
struct Link {
    Link* next;
};

class Task : public Link {
    // a Link-et a Task-ok listájához (az ütemező listához) használjuk
    // ...
};

class Displayed : public Link {
    // a Link-et a Displayed objektumok listájához (a megjelenítési listához) használjuk
    // ...
};
```

Ez nem gond. Két külön *Link* objektum szolgál a listák ábrázolására és a két lista nem zavarja egymást. Természetesen a *Link* osztály tagjaira nem hivatkozhatunk a kétértelműség veszélye nélkül (§15.2.3.1). Egy *Satellite* objektumot így rajzolhatunk le:



Ha a közös bázisosztályt nem szabad két külön objektummal ábrázolni, virtuális bázisosztályokat (§15.2.4) alkalmazhatunk.

A *Link*-hez hasonlóan többször szereplő bázisosztályok olyan elemek, amelyeket nem szabad a közvetlenül öröklő osztályon kívül használni. Ha egy ilyen osztályra olyan pontról kell hivatkozni, ahonnét annak több példánya is látható, a többértelműség elkerülése érdekében a hivatkozást minősíteni kell:

```
void mess_with_links(Satellite* p)
{
    p->next = 0;                    // hiba: többértelmű (melyik Link?)
    p->Link::next = 0;              // hiba: többértelmű (melyik Link?)
    p->Task::Link::next = 0;        // rendben
    p->Displayed::Link::next = 0;   // rendben
    // ...
}
```

Ez pontosan ugyanaz az eljárás, mint amit a tagokra való többértelmű hivatkozások feloldására használunk (§15.2.1).

15.2.3.1. Felülírás

A többször szereplő bázisosztályok valamely virtuális függvényét a származtatott osztály (egyetlen) függvénye *felülírhatja* (felülbírálhatja, override). Egy objektumnak saját magát egy fájlból kiolvasni vagy oda visszaírni való képességét például így ábrázolhatjuk:

```
class Storable {
public:
    virtual const char* get_file() = 0;
    virtual void read() = 0;
```

```
    virtual void write() = 0;
    virtual ~Storable() { }
};
```

Természetesen több felhasználó építhet erre, hogy olyan osztályokat írjon, amelyek függetlenül vagy együtt használva jobban kidolgozott osztályokat adnak. Például leállíthatunk és újraindíthatunk egy szimulációt, ha mentjük az alkotóelemeket és később visszaállítjuk azokat. Ezt az ötletet így valósíthatjuk meg:

```
class Transmitter : public Storable {
public:
    void write();
    // ...
};

class Receiver : public Storable {
public:
    void write();
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    const char* get_file();
    void read();
    void write();
    // ...
};
```

A felülíró függvény általában meghívja a bázisosztálybeli változatokat és a származtatott osztályra jellemző tennivalókat végzi el:

```
void Radio::write()
{
    Transmitter::write();
    Receiver::write();
    // kiírja a Radio-ra jellemző adatokat
}
```

Az ismétlődő bázisosztályokról származtatott osztályokra való típuskonverziót a §15.4.2 pont írja le. Arról, hogyan lehet az egyes *write()* függvényeket a származtatott osztályok külön függvényeivel felülírni, a §25.6 pont szól.

15.2.4. Virtuális bázisosztályok

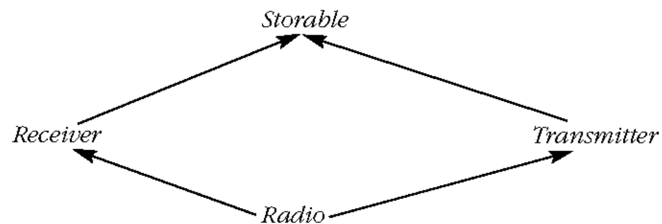
Az előző pont *Radio* példája azért működik, mert a *Storable* osztályt biztonságosan, kényelmesen és hatékonyan lehet többszörözni. Ez azonban az olyan osztályok esetében rendszerint nem igaz, amelyek jó építőkövei más osztályoknak. A *Storable* osztályt például úgy is meghatározhatnánk, mint ami tartalmazza az objektum mentésére használt fájl nevét:

```
class Storable {
public:
    Storable(const char* s);
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
private:
    const char* store;

    Storable(const Storable&);
    Storable& operator=(const Storable&);
};
```

A *Storable* ezen látszólag csekély módosítása után meg kell változtatnunk a *Radio* szerkezetét is. Az objektum összes része a *Storable* azonos példányán kell, hogy osztozzék; különben szükségtelenül nehéz feladat lenne az objektum többszöri tárolásának megakadályozása. A *virtuális bázisosztályok* (virtual base class) ezt a megosztást segítik. A származtatott osztály minden virtuális bázisosztályát ugyanaz a (megosztott) objektum ábrázolja:

```
class Transmitter : public virtual Storable {
public:
    void write();
    // ...
};

class Receiver : public virtual Storable {
public:
    void write();
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    void write();
    // ...
};
```

Ábrával:



Hasonlítsuk össze ezt az ábrát a *Satellite* objektum §15.2.3-beli rajzával, hogy lássuk a különbséget a közönséges és a virtuális öröklődés között. Az öröklődési gráfban egy adott nevű osztály minden virtuálisként megadott bázisosztályát az osztály egyetlen objektuma ábrázolja, a nem virtuális bázisosztályokat viszont saját részobjektumuk.

15.2.4.1. Virtuális bázisosztályok programozása

Amikor a programozó függvényeket készít egy olyan osztály számára, amelynek virtuális bázisosztálya van, nem tudhatja, hogy a bázisosztályt meg kell-e osztani egyéb származtatott osztályokkal, ami gondot jelenthet, ha egy szolgáltatást úgy kell megvalósítani, hogy a bázisosztály egy adott függvényének meghívására pontosan egyszer kerüljön sor, például mert a nyelv előírja, hogy egy virtuális bázisosztály konstruktora csak egyszer futhat le. A virtuális bázisosztály konstruktorát a teljes objektum, azaz a legtávolabbi származtatott osztály konstruktora hívja meg (automatikusan vagy közvetlenül):

```
class A {          // nincs konstruktor
    // ...
};

class B {
public:
    B();           // alapértelmezett konstruktor
    // ...
};

class C {
public:
    C(int);        // nincs alapértelmezett konstruktor
};
```

```
class D : virtual public A, virtual public B, virtual public C
{
    D() { /* ... */ }                    // hiba: C-nek nincs alapértelmezett konstruktora
    D(int i) : C(i) { /* ... */ };       // rendben
    // ...
};
```

A virtuális bázisosztály konstruktora a származtatott osztályok konstruktora előtt hívódik meg. Szükség esetén a programozó ezt a működést utánozhatja is, ha a virtuális bázisosztály függvényét csak a legtávolabbi származtatott osztályból hívja meg. Tegyük fel, hogy van egy alapvető *Window* osztályunk, amely ki tudja rajzolni tartalmát:

```
class Window {
    // alapkód
    virtual void draw();
};
```

Az ablakokat emellett többféle módon díszíthetjük és szolgáltatásokkal egészíthetjük ki:

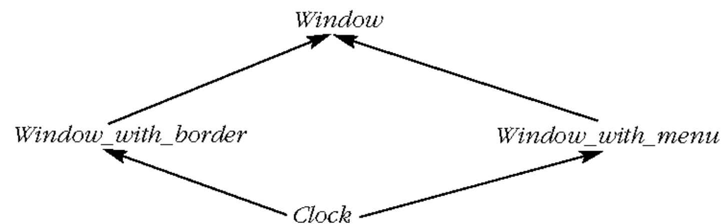
```
class Window_with_border : public virtual Window {
    // a szegély kódja
    void own_draw();     // a szegély megjelenítése
    void draw();
};

class Window_with_menu : public virtual Window {
    // a menü kódja
    void own_draw();     // a menü megjelenítése
    void draw();
};
```

Az *own_draw()* függvényeknek nem kell virtuálisaknak lenniük, mert egy virtuális *draw()* függvényből akarjuk meghívni azokat, ami pontosan ismeri az objektum típusát, amelyre meghívták.

Ebből egy működőképes *Clock* osztályt állíthatunk össze:

```
class Clock : public Window_with_border, public Window_with_menu {
    // az óra kódja
    void own_draw();     // az óralap és a mutatók megjelenítése
    void draw();
};
```

Ábrával:



A *draw()* függvényeket most már úgy írhatjuk meg az *own_draw()* függvények felhasználásával, hogy bármelyik *draw()* meghívása pontosan egyszer hívja meg a *Window::draw()*-t, függetlenül attól, milyen fajta *Window*-ra hívták meg:

```
void Window_with_border::draw()
{
    Window::draw();
    own_draw();          // a szegély megjelenítése
}

void Window_with_menu::draw()
{
    Window::draw();
    own_draw();          // a menü megjelenítése
}

void Clock::draw()
{
    Window::draw();
    Window_with_border::own_draw();
    Window_with_menu::own_draw();
    own_draw();          // az óralap és a mutatók megjelenítése
}
```

A virtuális bázisosztályokról származtatott osztályokra való konverziót a §15.4.2 pont írja le.

15.2.5. A többszörös öröklődés használata

A többszörös öröklődés legegyszerűbb és legnyilvánvalóbb felhasználása két egyébként egymással rokonságban nem álló osztály „összeragasztása" egy harmadik osztály részeként. A *Task* és *Displayed* osztályokból a §15.2 pontban összerakott *Satellite* osztály is ilyen.

A többszörös öröklődés ilyen módon való használata egyszerű, hatékony és fontos – de nem túl érdekes, hiszen alapjában véve csak a továbbító függvények megírásától kíméli meg a programozót. Az eljárás nem befolyásolja számottevően a program általános szerkezetét és alkalmasint ütközhet azzal a kívánalommal, hogy a megvalósítás részletei maradjanak rejtve. Egy módszernek azonban nem kell „okosnak" lennie ahhoz, hogy hasznos legyen.

A többszörös öröklődés használata absztrakt osztályok készítésére már nagyobb jelentőségű, annyiban, hogy befolyásolja a program tervezésének módját. A §12.4.3-beli *BB_ival_slider* osztály egy példa erre:

```
class BB_ival_slider
    : public Ival_slider      // felület
    , protected BBslider      // megvalósítás
{
    // az 'Ival_slider' és a 'BBslider' által igényelt függvények megvalósítása
    // a 'BBslider' szolgáltatásainak használata
};
```

Ebben a példában a két bázisosztály logikailag különböző szerepet játszik. Az egyik egy nyilvános absztrakt osztály, amely a felületet nyújtja, a másik pedig egy védett (protected) konkrét osztály, amely a megvalósításról gondoskodik. A kétféle szerep az osztályok stílusában és az alkalmazott hozzáférési szabályokban tükröződik. A többszörös öröklődés szerepe itt lényegbevágó, mert a származtatott osztálynak mind a felület, mind a megvalósítás virtuális függvényeit felül kell írnia.

A többszörös öröklődés lehetővé teszi a testvérosztályok számára, hogy egyetlen közös ős jelentette függés bevezetése nélkül osztozhassanak adatokon. Ez az eset, amikor az úgynevezett ***káró alakú öröklődés*** (diamond-shaped inheritance) lép fel. (Lásd a *Radio* (§15.2.4) és a *Clock* (§15.2.4.1) példákat.) Ha a bázisosztály nem ismételhető, akkor virtuális (és nem közönséges) bázisosztályra van szükség.

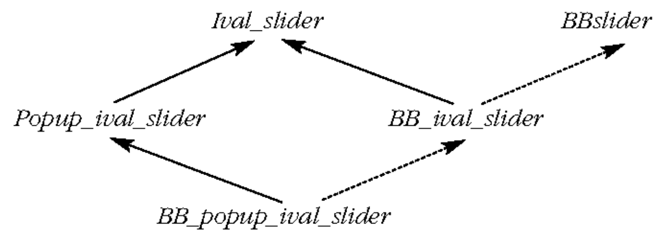
Az én véleményem az, hogy a káró alakú öröklési háló akkor kezelhető a legjobban, ha vagy a virtuális bázisosztály vagy a belőle közvetlenül származó osztályok absztraktak. Vegyük például újra a §12.4 pont *Ival_box* osztályait. Ott végül az összes *Ival_box* osztályt absztrakttá tettük, hogy kifejezzük szerepüket, vagyis hogy kizárólag felületek. Ez lehetővé tette, hogy a lényegi programkód minden részét a megfelelő megvalósító osztályokba rejtsük, és az egyes részeken való osztozás is csak a megvalósítás céljára használt ablakozó rendszer klasszikus hierarchiájában történt.

Persze lenne értelme annak, hogy a *Popup_ival_slider*-t megvalósító osztály nagy része közös legyen a sima *Ival_slider*-t megvalósító osztályéval, így az adatbekérő mezők kezelésén

kívül azonosak lennének. Ekkor az is természetes lenne, hogy az előálló csúszka-osztályban elkerüljük az *Ival_slider* objektumok ismétlődését. Ehhez az *Ival_slider*-t virtuálissá tesszük:

```
class BB_ival_slider : public virtual Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public virtual Ival_slider { /* ... */ };
class BB_popup_ival_slider
    : public virtual Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

Ábrával:

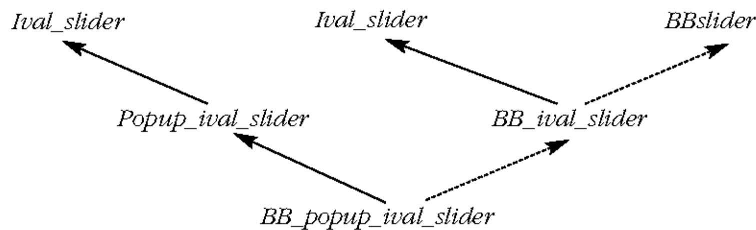


Könnyen elképzelhetjük a *Popup_ival_slider*-ből származó további felületeket és az azokból és a *BB_popup_ival_slider*-ből származó további megvalósító osztályokat.

Ha az ötletet végigvisszük, akkor a program felületét képező absztrakt osztályokból való minden származtatást virtuálissá teszünk. Ez valóban a leglogikusabb, legáltalánosabb, és legrugalmasabb megoldásnak tűnik. Hogy miért nem tettem ezt, annak egyrészt a hagyományok figyelembe vétele az oka, másrészt a virtuális bázisosztályok megvalósításának legnyilvánvalóbb és leggyakoribb módszerei annyira hely- és időigényesek, hogy kiterjedt használatuk egy osztályon belül nem vonzó. Mielőtt ez a tárigényben és futási időben mért költség visszariasztana bennünket egy más szemszögből vonzó szerkezet választásától, gondoljuk meg, hogy az *Ival_box*-ot ábrázoló objektum általában csak egy mutatót tartalmaz a virtuális táblára. Ahogy a §15.2.4 pontban láttuk, egy változókat nem tartalmazó absztrakt osztály veszély nélkül ismételhető. Így a virtuális bázisosztály helyett közönségeset alkalmazhatunk:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
class BB_popup_ival_slider
    : public Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

Ábrával:



Ez valószínűleg megvalósítható, optimalizált változata az előzőekben bemutatott, bevallottan világosabb szerkezetnek.

15.2.5.1. A virtuális bázisosztályok függvényeinek felülírása

A származtatott osztályok felülírhatják (override) közvetlen vagy közvetett virtuális bázisosztályaik virtuális függvényeit. Két különböző osztály akár a virtuális bázisosztály különböző függvényeit is felülírhatja, így több származtatott osztály járulhat hozzá a virtuális bázisosztály által adott felület megvalósításához. A *Window* osztálynak lehetnek például *set_color()* és *prompt()* függvényei. Ekkor a *Window_with_border* felülbírálhatja a *set_color()*-t, mint a színellenőrző séma részét, a *Window_with_menu* pedig a *prompt()*-ot, mint a felhasználói felület kezelésének részét:

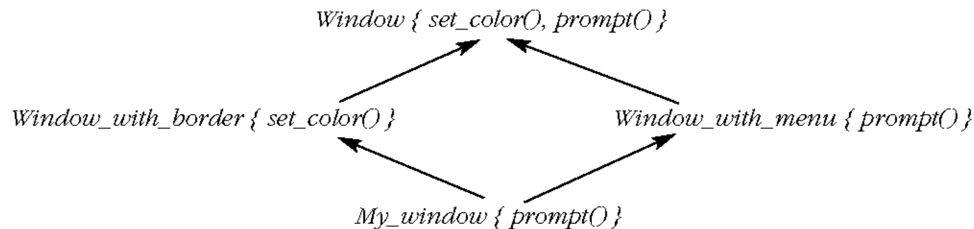
```
class Window {
    // ...
    virtual void set_color(Color) = 0;          // háttérszín beállítása
    virtual void prompt() = 0;
};

class Window_with_border : public virtual Window {
    // ...
    void set_color(Color);                      // háttérszín kezelése
};

class Window_with_menu : public virtual Window {
    // ...
    void prompt();                              // felhasználói tevékenységek kezelése
};
```

```
class My_window : public Window_with_menu, public Window_with_border {
    // ...
};
```

Mi történik, ha különböző származtatott osztályok ugyanazt a függvényt írják felül? Ez csak akkor megengedett, ha az egyik származtatott osztály minden olyan osztály örököse, amely felülírja a függvényt, vagyis egy függvénynek az összeset felül kell írnia. A *My_window* például felülírhatja a *prompt()*-t, hogy a *Window_with_menu*-belinél jobb változatot adjon:

```
class My_window : public Window_with_menu, public Window_with_border {
    // ...
    void prompt();// a felhasználói tevékenységek kezelését nem hagyjuk a bázisosztályra
};
```

Ábrával:



Ha két osztály felülír egy bázisosztálybeli függvényt, de a két függvény egyike nem írja felül a másikat, akkor az osztályhierarchia hibás. Ilyenkor nem lehet virtuális függvénytáblát építeni, mert a teljes objektumra vonatkozó függvényhívás kétértelmű lenne. Például ha a §15.2.4. pontbeli *Radio* nem adta volna meg a *write()* függvényt, akkor a *Receiver* és *Transmitter* osztályokbeli *write()* deklarációk a *Radio*-ban hibát okoztak volna. Mint a *Radio* esetében is, az ilyen konfliktust a felülíró függvénynek a legtávolabbi származtatott osztályból való meghívásával oldhatjuk meg.

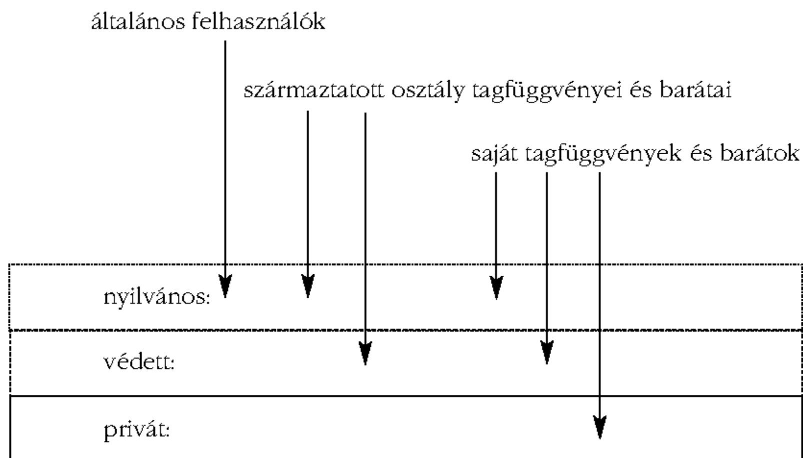
Az olyan osztályokat, amelyek egy virtuális bázisosztály némelyik (de nem mindegyik) függvényének megvalósítását tartalmazzák, gyakran „*mixin*”-nek nevezik.

15.3. Hozzáférés-szabályozás

Egy osztálytag lehet *privát* (private), *védett* (protected) vagy *nyilvános* (public):

- Ha privát, a nevét csak a tagfüggvényekben és a deklaráló osztály barátaiban (friend-jeiben) lehet felhasználni.
- Ha védett, a nevét csak a deklaráló osztály és barátai tagfüggvényeiben, valamint az osztályból származtatott osztályok és barátaik tagfüggvényeiben lehet felhasználni.
- Ha nyilvános, a nevét mindenhol fel lehet használni.

Ez azt a nézetet tükrözi, hogy egy osztályt háromféle függvény érhet el: az osztályt megvalósító függvények (azaz a barátok és a tagok), egy származtatott osztályt megvalósító függvények (azaz a származtatott osztályok barátai és a tagok), és az egyéb függvények. Ezt így ábrázolhatjuk:



A hozzáférési szabályok egyöntetűen vonatkoznak a nevekre. Hogy a név mit jelöl, az érdektelen a hozzáférés szempontjából. Ez azt jelenti, hogy ugyanúgy lehetnek privát tagfüggvények, típusok, állandók stb., mint privát adattagok. Például egy hatékony nem tolakodó (non-intrusive, §16.2.1) listaosztálynak valószínűleg szüksége van az elemeket nyilvántartó adatszerkezetekre. Az ilyen információ legjobb, ha privát:

```
template<class T> class List {
private:
    struct Link { T val; Link* next; };
    struct Chunk {
            enum { chunk_size = 15 };
            Link v[chunk_size];
            Chunk* next;
    };
    Chunk* allocated;
    Link* free;
    Link* get_free();
    Link* head;
public:
    class Underflow { };                // kivételosztály

    void insert(T);
    T get();
    // ...
};

template<class T> void List<T>::insert(T val)
{
    Link* lnk = get_free();
    lnk->val = val;
    lnk->next = head;
    head = lnk;
}

template<class T> List<T>::Link* List<T>::get_free()
{
    if (free == 0) {
            // új Chunk lefoglalása és Link-jeinek a szabad listára helyezése
    }
    Link* p = free;
    free = free->next;
    return p;
}

template<class T> T List<T>::get()
{
    if (head == 0) throw Underflow();

    Link* p= head;
    head = p->next;
    p->next = free;
    free = p;
    return p->val;
}
```

A *List<T>* hatókörbe azáltal lépünk be, hogy a tagfüggvényben *List<T>::*-t írunk. Mivel a *get_free()* visszatérési értékét előbb említjük, mint a *List<T>::get_free()* nevet, a *Link* rövidítés helyett a teljes *List<T>::Link* nevet kell használnunk. A nem tag függvényeknek – a barát (friend) függvények kivételével – nincs ilyen hozzáférésük:

```
void would_be_meddler(List<T>* p)
{
    List<T>::Link* q = 0;              // hiba: List<T>::Link privát
    // ...
    q = p->free;                       // hiba: List<T>::free privát
    // ...
    if (List<T>::Chunk::chunk_size > 31) {        // hiba: List<T>::Chunk::chunk_size privát
            // ...
    }
}
```

Az osztályok (class) tagjai alapértelmezés szerint privátok (private), a struktúrák (struct) tagjai nyilvánosak (public, §10.2.8).

15.3.1. Védett tagok

A védett (protected) tagok használatának bemutatására vegyük a §15.2.4.1 pontbeli *Window* példát. Az *own_draw()* függvények (akarattal) nem teljes körű szolgáltatást adnak. Arra a célra terveztük őket, hogy csak a származtatott osztályok számára szolgáljanak építőkövekül, az általános felhasználás számára nem biztonságosak, nem kényelmesek. Másfelől a *draw()* műveletek az általános felhasználást szolgálják. Ezt a különbséget a *Window* osztály felületének két, egy védett és egy nyilvános felületre való szétválasztásával lehet kifejezni:

```
class Window_with_border {
public:
    virtual void draw();
    // ...
protected:
    void own_draw();
    // egyéb kirajzoló kód
private:
    // ábrázolás stb.
};
```

A származtatott osztály a bázisosztály védett tagjai közül csak a saját osztályába tartozó objektumokat tudja elérni:

```
class Buffer {
protected:
    char a[128];
    // ...
};

class Linked_buffer : public Buffer { /* ... */ };

class Cyclic_buffer : public Buffer {
    // ...
    void f(Linked_buffer* p) {
            a[0] = 0;       // rendben: Cyclic_buffer saját védett tagját éri el
            p->a[0] = 0;    // hiba: más típus védett tagját próbáltuk elérni
    }
};
```

Ez megakadályozza az olyan hibákat, amelyek azáltal léphetnének fel, hogy az egyik származtatott osztály összezavarja a másik származtatott osztály adatait.

15.3.1.1. A védett tagok használata

Az adatok elrejtésének egyszerű privát/nyilvános modellje jól szolgálja a konkrét típusokat (§10.3). De ha származtatott osztályokat használunk, akkor kétféle felhasználója lesz egy osztálynak: a származtatott osztályok és „a nagyközönség". A műveleteket megvalósító tagok és barátok ezen felhasználók érdekében kezelik az objektumokat. A privát/nyilvános modell lehetővé teszi a megvalósítás és az általános felhasználás pontos megkülönböztetését, de a származtatott osztályok megfelelő kezelését nem.

A *protected*-ként deklarált tagokkal sokkal könnyebb visszaélni, mint a privátként bevezetettekkel. Ezért a tagok védettként való megadása általában tervezési hiba. Ha jelentős mennyiségű adatot úgy helyezünk el egy közös osztályban, hogy az összes származtatott osztály használhatja azokat, az adatok sérülhetnek. Még rosszabb, hogy a védett tagokat – csakúgy, mint a nyilvánosakat – nem könnyű átszervezni, mert nincs jó módszer az összes használat felderítésére; így a védett tagok megnehezítik a program módosítását.

Szerencsére nem kell feltétlenül védett tagokat használni; az osztályokra a privát az alapértelmezett hozzáférési kategória és általában ez a jobb választás. Az én tapasztalatom az, hogy az összes származtatott osztály által közvetlenül használható, jelentős mennyiségű adat közös osztályban való elhelyezése helyett mindig adódik más megoldás.

Jegyezzük meg, hogy ezek a kifogások nem érvényesek a védett tagfüggvényekre; a *protected* minősítővel remekül adhatunk meg a származtatott osztályokban használható műveleteket. A §12.4.2 pontbeli *Ival_slider* is példa erre. Ha ebben a példában a megvalósító osztály privát lett volna, a további öröklés lehetetlenné vált volna. A tagok elérhetőségére példákat a §C.11.1 tartalmaz.

15.3.2. Bázisosztályok elérése

A tagokhoz hasonlóan egy bázisosztály is lehet nyilvános, védett vagy privát:

```
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */ };
```

A nyilvános öröklés a származtatott osztályt a bázisosztály egy *altípusává* (subtype) teszi; ez az öröklés legáltalánosabb formája. A védett és a privát öröklés a megvalósítás módjának jelölésére használatos. A védett öröklés olyan osztályhierarchiákban a leghasznosabb, ahol jellemzően további öröklés történik (a §12.4.2 pontbeli *Ival_slider* jó példa erre). A privát bázisosztályok akkor a leghasznosabbak, amikor egy osztályt a felület korlátozásával határozunk meg, ami által erősebb garanciák adhatók. A mutatókra vonatkozó *Vector* például érték-ellenőrzéssel bővíti ki *Vector<void*>* bázisosztályát (§13.5). Ha azt is biztosítani szeretnénk, hogy a *Vec*-hez (§3.7.2) való minden hozzáférés ellenőrzött legyen, bázisosztályát privátként kell megadnunk, így a *Vec*-ek nem konvertálódnak nem ellenőrzött *vector*-rá:

```
template <class T > class Vec : private Vector <T> { /* ... */ };
```

Az osztály hozzáférési szintjét nem muszáj explicit megadni. Ilyenkor az alapértelmezett hozzáférése *class* esetén privát, *struct* esetén nyilvános lesz:

```
class XX : B { /* ... */ };   // B privát bázisosztály
struct YY : B { /* ... */ };  // B nyilvános bázisosztály
```

Az olvashatóság szempontjából azonban legjobb kiírni a hozzáférési szintet megadó kulcsszót.

A bázisosztály hozzáférési szintje az osztály tagjainak elérhetősége mellett a származtatott osztályra hivatkozó mutatóknak vagy referenciáknak a bázisosztály típusára való konvertálhatóságát is jelzi. Vegyünk egy *B* bázisosztályból származó *D* osztályt:

- Ha *B* privát bázisosztály, akkor nyilvános és védett tagjai csak *D* tagfüggvényeiből és barátaiból érhetők el. Csak *D* barátai és tagjai konvertálhatnak egy *D** mutatót *B**-gá.

- Ha *B* védett bázisosztály, akkor nyilvános és védett tagjai csak *D* tagfüggvényeiből és barátaiból, valamint a *D*-ből származó osztályok tagfüggvényeiből és barátaiból érhetők el. Csak *D* tagfüggvényei és barátai, valamint a *D*-ből származó osztályok tagfüggvényei és barátai végezhetnek konverziót *D**-ról *B**-ra.
- Ha *B* nyilvános bázisosztály, nyilvános tagjai bárhol használhatók. Ezenkívül védett tagjai *D* tagfüggvényeiből és barátaiból, valamint a *D*-ből származó osztályok tagfüggvényeiből és barátaiból érhetők el. Bármely függvény végezhet *D**-ról *B**-ra való konverziót.

Ez alapvetően azonos a tagok elérhetőségi szabályaival (§15.3). A bázisosztályok elérhetőségét ugyanazon szempontok szerint adjuk meg, mint a tagokét. A *BBwindow*-t például azért adtam meg az *Ival_slider* védett bázisosztályaként (§12.4.2), mert a *BBwindow* inkább az *Ival_slider* megvalósításának, mint felületének része. A *BBwindow*-t azonban nem rejthettem el teljesen úgy, hogy privát bázisosztállyá teszem, mert az *Ival_slider*-ből további osztályokat akartam származtatni és azoknak el kellett érniük a megvalósítást.

A bázisosztályok elérhetőségére a §C.11.2 mutat példákat.

15.3.2.1. A többszörös öröklődés és az elérhetőség

Ha egy nevet vagy bázisosztályt egy többszörös öröklődési háló több útvonalán is elérhetünk, akkor abban az esetben lesz elérhető, ha valamelyik út mentén elérhető:

```
struct B {
    int m;
    static int sm;
    // ...
};

class D1 : public virtual B { /* ... */ } ;
class D2 : public virtual B { /* ... */ } ;
class DD : public D1, private D2 { /* ... */ };

DD* pd = new DD;
B* pb = pd;              // rendben: elérhető D1-en keresztül
int i1 = pd->m;          // rendben: elérhető D1-en keresztül
```

Ha egy bizonyos elemet több út mentén is elérhetünk, attól még egyértelműen, azaz többértelműségi hiba nélkül hivatkozhatunk rá:

```
class X1 : public B { /* ... */ } ;
class X2 : public B { /* ... */ } ;
class XX : public X1, public X2 { /* ... */ };
```

```
XX* pxx = new XX;
int i1 = pxx->m;          // hiba, többértelmű: XX::X1::B::m vagy XX::X2::B::m
int i2 = pxx->sm;         // rendben: csak egy B::sm van egy XX-ben
```

15.3.2.2. A using deklarációk és az elérhetőség

A *using* deklarációk nem szolgálhatnak több információ elérésére, csak az egyébként is hozzáférhető adatok kényelmesebb használatát teszik lehetővé. Másrészt viszont, ha egy információ elérhető, akkor a hozzáférési jog más felhasználók felé továbbadható:

```
class B {
private:
    int a;
protected:
    int b;
public:
    int c;
};

class D : public B {
public:
    using B::a;    // hiba: B::a privát
    using B::b;    // B::b nyilvánosan elérhető D-n keresztül
};
```

Ha egy *using* deklaráció privát vagy védett öröklődéssel jár együtt, akkor a bázisosztály által rendesen felkínált szolgáltatások egy részéhez felületet adhat:

```
class BB : private B {    // hozzáférést ad a B::b és B::c nevekhez, de a B::a-hoz nem
public:
    using B::b;
    using B::c;
};
```

Lásd még a §15.2.2 pontot.

15.4. Futási idejű típusinformáció

A §12.4 pontban leírt *Ival_box*-ok valószerű használata lenne, ha átadnánk azokat egy képernyőkezelő rendszernek, majd az visszaadná őket a programnak, valahányszor valamilyen tevékenység történt. Sok felhasználói felület működik így. Egy felhasználói felületet kezelő rendszer azonban nem feltétlenül tud a mi *Ival_box*-ainkról. A rendszer felületét a rendszer saját osztályai és objektumai nyelvén adják meg, nem a mi alkalmazásunk osztályainak nyelvén. Ez szükségszerű és rendjén is való, de azzal a kellemetlen következménnyel jár, hogy információt vesztünk a rendszernek átadott és később nekünk visszaadott objektumok típusáról.

Az „elveszett" adatok visszanyeréséhez valahogy meg kell tudnunk kérdezni az objektumtól a típusát. Bármilyen műveletet akarunk is végezni az objektummal, alkalmas típusú, az objektumra hivatkozó mutatóra vagy referenciára van szükségünk. Következésképpen egy objektum típusának futási időben való lekérdezéséhez a legnyilvánvalóbb és leghasznosabb művelet az, amely érvényes mutatót ad vissza, ha az objektum a várt típusú, illetve „nullpointert", ha nem. Pontosan ezt teszi a *dynamic_cast* operátor. Például tegyük fel, hogy a rendszer a *my_event_handler()*-t arra a *BBwindow*-ra hivatkozó mutatóval hívja meg, amellyel egy tevékenység történt. Ekkor az *Ival_box* osztály *do_something()* függvényét használva meghívhatnánk programunkat:

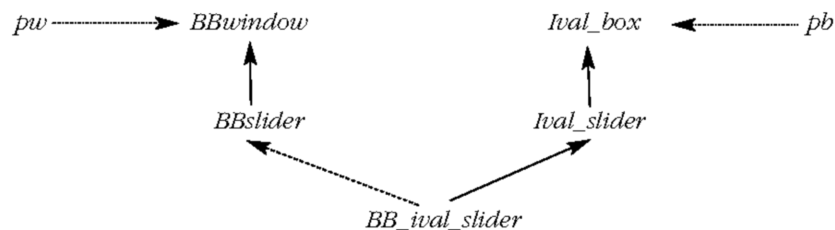
```
void my_event_handler(BBwindow* pw)
{
    if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))      // Vajon pw egy Ival_box-ra mutat?
            pb->do_something();
    else {
            // hoppá! nem várt esemény
    }
}
```

A folyamatot úgy is magyarázhatjuk, hogy a *dynamic_cast* fordít a felhasználói felületet kezelő rendszer sajátos nyelvéről az alkalmazás nyelvére. Fontos észrevenni, hogy mi nem nyert említést ebben a példában: az objektum tényleges típusa. Az objektum az *Ival_box* egy bizonyos fajtája lesz, mondjuk *Ival_slider*, amelyet a *BBwindow* egy bizonyos típusa valósít meg, mondjuk a *BBslider*. Az objektum tényleges típusának kiderítése és megemlítése se nem szükséges, se nem kívánatos a rendszer és a program közötti ezen párbeszédben. Létezik felület a párbeszéd lényegének leírására, egy jól megtervezett felület pedig elrejti a lényegtelen részleteket.

Rajzban a

pb = dynamic_cast<Ival_box>(pw)*

utasítás hatását így ábrázolhatjuk:



A *pw*-ből és *pb*-ből kiinduló nyilak az átadott objektumra hivatkozó mutatókat jelölik, míg a többi nyíl az átadott objektum különböző részei közötti öröklődési viszonyokat ábrázolja.

A típusinformációk futási időben való használatát hagyományosan *futási idejű típusinformációnak* (run-time type information) hívják és gyakran *RTTI*-nek rövidítik.

A bázisosztályról származtatott osztályra történő konverziót gyakran „lefelé történő vagy származtatott irányú konverziónak" (*downcast*) hívják, mert az öröklési fák a hagyományos ábrázolás szerint a gyökértől „lefelé nőnek". Ehhez hasonlóan a származtatott osztályról bázisosztályra történő konverzió neve „felfelé történő konverzió", vagy „bázisirányú konverzió" (*upcast*). A bázisosztályról testvérre – például *BBwindow*-ról *Ival_box*-ra – való konverziót „keresztbe történő konverziónak" (*crosscast*) hívják.

15.4.1. Dynamic_cast

A *dynamic_cast* (dinamikus típuskényszerítés) operátor két paramétert vár, egy <> közé írt típust és egy *()* közé írt mutatót vagy referenciát.

Vegyük először a mutató esetét:

dynamic_cast<T>(p)*

Ha a *p* a *T** típusba, vagy olyan *D** típusba tartozik, ahol *T* bázisosztálya *D*-nek, akkor az eredmény ugyanaz, mintha a *p*-t egy *T** változónak adtuk volna értékül:

```
class BB_ival_slider : public Ival_slider, protected BBslider {
    // ...
};

void f(BB_ival_slider* p)
{
    Ival_slider* pi1 = p;                       // rendben
    Ival_slider* pi2 = dynamic_cast<Ival_slider*>(p);      // rendben

    BBslider* pbb1 = p;                         // hiba: BBslider védett bázisosztály
    BBslider* pbb2 = dynamic_cast<BBslider*>(p);           // rendben: pbb2 értéke 0 lesz
}
```

Ez nem túl érdekes. Azt azonban jó tudni, hogy a *dynamic_cast* nem engedi meg a védett vagy privát bázisosztályok védelmének véletlen megsértését.

A *dynamic_cast* célja azon esetek kezelése, amelyeknél a fordítóprogram nem tudja a konverzió helyességét megítélni:

```
dynamic_cast<T*>(p)
```

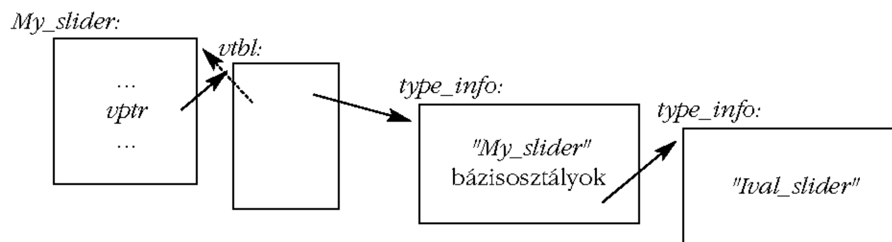
A fenti kód megvizsgálja a *p* által mutatott objektumot (ha van ilyen). Ha az objektum *T* osztályú vagy van egy egyértelmű *T* típusú őse, akkor a *dynamic_cast* egy, az objektumra hivatkozó *T** típusú mutatót ad vissza, más esetben *0*-át. Ha *p* értéke *0*, a *dynamic_cast<T*>(p)* eredménye *0* lesz. Jegyezzük meg, hogy a konverzió csak egyértelműen azonosított objektumoknál működik. Lehet olyan példákat hozni, ahol a konverzió nem sikerül és *0*-át ad, mert a *p* által mutatott objektumnak több *T* típusú bázisosztályt képviselő részobjektuma van (§15.4.2).

A *dynamic_cast*-nak a lefelé vagy keresztbe történő konvertáláshoz többalakú (polimorf) mutatóra vagy hivatkozásra van szüksége:

```
class My_slider: public Ival_slider {   // többalakú bázisosztály (Ival_slider rendelkezik
                                        // virtuális függvényekkel)
    // ...
};

class My_date : public Date {           // nem többalakú bázisosztály (Date nem rendelkezik
                                        // virtuális függvényekkel)
    // ...
};
```

```
void g(Ival_box* pb, Date* pd)
{
    My_slider* pd1 = dynamic_cast<My_slider*>(pb);     // rendben
    My_date* pd2 = dynamic_cast<My_date*>(pd);         // hiba: Date nem többalakú
}
```

Az a megkötés, hogy a mutatónak többalakúnak kell lennie, egyszerűsíti a *dynamic_cast* megvalósítását, mert megkönnyíti az objektum típusának tárolásához szükséges információ helyének megtalálását. Általános megoldás, hogy a típust jelző mutatót az objektum virtuális függvénytáblájába (§2.5.5) helyezik, ezáltal egy „típus-információ objektumot" fűznek az objektumhoz:



A szaggatott nyíl az eltolást (offset) jelöli, amelynek segítségével a teljes objektum kezdete megtalálható, ha csak egy többalakú részobjektumra hivatkozó mutató adott. Világos, hogy a *dynamic_cast* hatékonyan felhasználható; csak néhány, a bázisosztályt leíró *type_info* objektumot kell összehasonlítani, nincs szükség hosszadalmas keresésre vagy karakterláncok összehasonlítására.

A *dynamic_cast*-nak többalakú típusokra való korlátozása logikai szempontból nézve is értelmes. Ha egy objektumnak nincs virtuális függvénye, akkor pontos típusának ismerete nélkül nem kezelhető biztonságosan, ezért ügyelni kell, hogy az ilyen objektum ne kerüljön olyan környezetbe, ahol nem ismeretes a pontos típusa. Ha azonban a típusa ismert, nincs szükség *dynamic_cast*-ra.

A *dynamic_cast* céltípusa nem kell, hogy többalakú legyen, ezért egy konkrét típust többalakúba csomagolhatunk, mondjuk, hogy egy objektumokat kezelő ki- és bemeneti rendszeren keresztül továbbítsuk (§25.4.1), majd később kicsomagoljuk belőle a konkrét típust:

```
class Io_obj {                          // bázisosztály-objektum az I/O rendszer számára
    virtual Io_obj* clone() = 0;
};

class Io_date : public Date, public Io_obj { };

void f(Io_obj* pio)
{
    Date* pd = dynamic_cast<Date*>(pio);
    // ...
}
```

Egy többalakú objektum kezdőcímét egy *void**-ra való dinamikus típuskényszerítéssel határozhatjuk meg:

```
void g(Ival_box* pb, Date* pd)
{
    void* pd1 = dynamic_cast<void*>(pb);       // rendben
    void* pd2 = dynamic_cast<void*>(pd);       // hiba: Date nem többalakú
}
```

Ez azonban csak nagyon alacsony szintű függvényekkel való együttműködés céljára hasznos.

15.4.1.1. Referenciák dinamikus típuskényszerítése

Egy objektum többalakú (polymorph) viselkedéséhez akkor férünk hozzá, ha mutatón vagy referencián át kezeljük. A *dynamic_cast* művelet sikertelenségét *0*-val jelzi. Ez referenciákra se nem kivitelezhető, se nem kívánatos.

Ha egy mutatóról, mint eredményről van szó, akkor fel kell készülnünk arra a lehetőségre, hogy az eredmény *0* lesz, azaz a mutató nem mutat semmilyen objektumra. Ezért egy *dynamic_cast* művelet eredményét mindig ellenőriznünk kell. A *p* mutatón végzett *dynamic_cast<T*>(p)* egy kérdésként fogható fel: „a *p* által mutatott objektum *T* típusú?”

Másrészt viszont jogosan tételezhetjük fel, hogy egy referencia mindig egy objektumra vonatkozik. Következésképpen egy *r* referencia esetén a *dynamic_cast<T&>(r)* nem kérdés, hanem állítás: „a *r* által mutatott objektum *T* típusú.” A *dynamic_cast* művelet eredményét maga a *dynamic_cast*-ot megvalósító kód ellenőrzi automatikusan, és ha a *dynamic_cast* operandusa nem a várt típusú hivatkozás, akkor *bad_cast* kivételt vált ki:

```
void f(Ival_box* p, Ival_box& r)
{
    if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {  // Vajon p egy Ival_slider-re mutat?
            // 'is' használata
    }
    else {
            // *p nem slider
    }

    Ival_slider& is = dynamic_cast<Ival_slider&>(r);      // az r egy Ival_slider-re hivatkozik!
    // 'is' használata
}
```

A sikertelen dinamikus mutató- illetve referencia-átalakítások eredményének eltérésében a mutatók, illetve a referenciák közötti alapvető különbség tükröződik. Ha egy felhasználó védekezni akar a referencia-átalakítás sikertelensége ellen, akkor egy megfelelő kivételkezelőre van szüksége:

```
void g()
{
    try {
            f(new BB_ival_slider,*new BB_ival_slider);    // a paraméterek Ival_box-ként
                                                          // adódnak át
            f(new BBdial,*new BBdial);            // a paraméterek Ival_box-ként adódnak át
    }
    catch (bad_cast) {                          // §14.10
            // ...
    }
}
```

Az *f()* függvény első meghívása sikeresen fog visszatérni, míg a második *bad_cast* kivételt vált ki, amit *g()* elkap.

A *0* érték ellenőrzése elhagyható, így alkalmasint véletlenül el is fog maradni. Ha ez aggasztja az olvasót, akkor írhat egy olyan konverziós függvényt, amely sikertelenség esetén a *0* érték visszaadása helyett kivételt vált ki (§15.8[1]).

15.4.2. Osztályhierarchiák bejárása

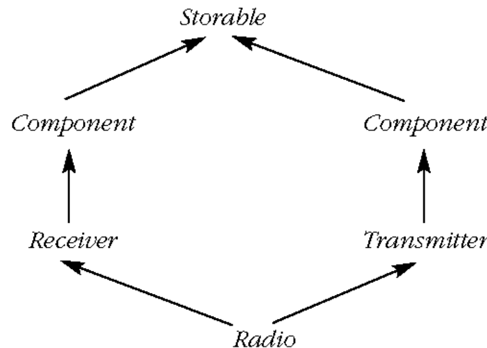
Ha csak egyszeres öröklődést használunk, az osztály és bázisosztályai egyetlen bázisosztályban gyökerező fát alkotnak. Ez egyszerű, de gyakran túl erős megszorítást jelent. Többszörös öröklődés használata esetén nincs egyetlen gyökér. Ez önmagában nem bonyolítja nagyon a helyzetet, de ha egy osztály többször fordul elő a hierarchiában, akkor némi óvatossággal kell az adott osztályú objektumra vagy objektumokra hivatkoznunk.

Természetesen a hierarchiákat igyekszünk annyira egyszerűnek venni – de nem egyszerűbbnek –, amennyire programunk megengedi. De ha már kialakult egy bonyolultabb hierarchia, hamarosan szükségünk lesz annak bejárására (vagyis végignézésére), hogy alkalmas osztályt találjunk, amelyet felületként használhatunk. Ez az igény két esetben merülhet fel. Néha kifejezetten meg akarunk nevezni egy bázisosztályt vagy annak egy tagját (például §15.2.3 és §15.2.4.1). Máskor egy, a bázisosztályt vagy annak egy származtatott osztályát megjelenítő objektumra hivatkozó mutatóra van szükségünk, ha adott egy mutató a teljes objektumra vagy annak valamely részobjektumára (§15.4 és §15.4.1).

Itt azt tekintjük át, hogyan lehet típuskényszerítést (*cast*) használva a kívánt típusú mutatóhoz jutni. Hogy szemléltessük az elérhető módszereket és a rájuk vonatkozó szabályokat, vegyünk egy többször szereplő és virtuális bázisosztályt egyaránt tartalmazó hálót:

```
class Component : public virtual Storable { /* ... */ };
class Receiver : public Component { /* ... */ };
class Transmitter : public Component { /* ... */ };
class Radio : public Receiver, public Transmitter { /* ... */ };
```

Ábrával:



Itt a *Radio* objektumnak két *Component* osztályú részobjektuma van. Ezért a *Radio*-n belüli, *Storable*-ről *Component*-re való dinamikus típuskényszerítés többértelmű lesz és nullát ad. Ilyenkor egyszerűen nem lehet tudni, melyik *Component*-re gondolt a programozó:

```
void h1(Radio& r)
{
    Storable* ps = &r;
    // ...
    Component* pc = dynamic_cast<Component*>(ps); // pc = 0
}
```

Ez a többértelműség fordítási időben általában nem deríthető fel:

```
void h2(Storable* ps)       // ps-ről nem tudjuk, hogy Component-re mutat-e
{
    Component* pc = dynamic_cast<Component*>(ps);
    // ...
}
```

A többértelműségnek ez a fajta felderítése csak virtuális bázisosztályoknál szükséges. Közönséges bázisosztályok és lefelé (azaz a származtatott osztály felé történő; §15.4) konverzió esetén a kívánt típusú részobjektum mindig egyértelmű (ha létezik). Ezzel egyenértékű többértelműség lép fel felfelé (azaz a bázisosztály felé történő) konverzió esetén és ezek a többértelműségek fordítási időben kiderülnek.

15.4.2.1. Statikus és dinamikus konverzió

A *dynamic_cast* művelet többalakú bázisosztályról származtatott osztályra vagy testvérosztályra való átalakítást tud végezni (§15.4.1). A *static_cast* (§6.2.7) nem vizsgálja a kiinduló objektum típusát, így nem képes ezekre:

```
void g(Radio& r)
{
    Receiver* prec = &r;                    // a Receiver a Radio közönséges bázisosztálya
    Radio* pr = static_cast<Radio*>(prec);              // rendben, nincs ellenőrzés
    pr = dynamic_cast<Radio*>(prec);                    // rendben, futási idejű ellenőrzés

    Storable* ps = &r;                      // a Storable a Radio virtuális bázisosztálya
    pr = static_cast<Radio*>(ps);           // hiba: virtuális bázisosztályról nem lehet átalakítani
    pr = dynamic_cast<Radio*>(ps);                      // rendben, futási idejű ellenőrzés
}
```

A *dynamic_cast*-nak többalakú operandusra van szüksége, mert egy nem többalakú objektum nem tárol olyan információt, amelynek alapján meg lehetne keresni azon objektumokat, amelyeknek őse (bázisa). Olyan objektum is lehet például virtuális bázisosztály, amelynek a memóriában való elhelyezkedését egy másik nyelv, például a Fortran vagy a C határozza meg. Ezekre vonatkozóan csak statikus adatok állnak rendelkezésre, de a *dynamic_cast* megvalósításához szükséges információt a futási idejű típusinformáció tartalmazza.

Miért akarna valaki *static_cast*-ot használni egy osztályhierarchia bejárására? A *dynamic_cast* némileg növeli a futási időt (§15.4.1), de ennél jelentősebb ok, hogy millió sornyi kód van a *dynamic_cast* bevezetése előtti időkből. Az ilyen kódok más módokon

biztosítják az alkalmazott típusátalakítások helyességét, így a *dynamic_cast*-tal végeztetett ellenőrzés feleslegesnek tűnik. Az ilyen – általában C stílusú típuskonverzióval (§6.2.7) íródott – kódban azonban gyakran maradhatnak rejtett hibák, így, hacsak lehet, használjuk a biztonságosabb *dynamic_cast*-ot.

A fordítóprogram nem tételezhet fel semmit egy *void** mutató által mutatott memóriaterületről. Ebből következik, hogy az objektum típusa felől érdeklődő *dynamic_cast* nem képes *void**-ról konvertálni. Ehhez *static_cast* kell:

```
Radio* f(void* p)
{
    Storable* ps = static_cast<Storable*>(p);      // Bízzunk a programozóban!
    return dynamic_cast<Radio*>(ps);
}
```

Mind a *dynamic_cast*, mind a *static_cast* tiszteletben tartja a *const* minősítést és a hozzáférési korlátozásokat:

```
class Users : private set<Person> { /* ... */ };

void f(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person>*>(pu);                   // hiba: nem férhet hozzá
    dynamic_cast<set<Person>*>(pu);                  // hiba: nem férhet hozzá

    static_cast<Receiver*>(pcr);                     // hiba: const minősítés nem vész el
    dynamic_cast<Receiver*>(pcr);                    // hiba: const minősítés nem vész el

    Receiver* pr = const_cast<Receiver*>(pcr);       // rendben
    // ...
}
```

Privát bázisosztályra nem lehet konvertálni, *const*-ot nem konstanssá konvertálni pedig csak *const_cast*-tal lehet (§6.2.7), és még akkor is csak úgy kapunk helyes eredményt, ha az objektumot eredetileg nem *const*-ként deklaráltuk (§10.2.7.1).

15.4.3. Osztályobjektumok felépítése és megsemmisítése

Egy valamilyen osztályba tartozó objektum több, mint egyszerűen a memória egy része (§4.9.6). Az osztályobjektumokat konstruktoraik építik fel a „nyers memóriából” és destruktoraik lefutásával válnak újra „nyers memóriává”. Az objektum felépítése alulról felfelé, megsemmisítése felülről lefelé történik, és az osztályobjektum olyan mértékben létező

objektum, amennyire felépítése, illetve megsemmisítése megtörtént. Ez tükröződik a futási idejű típusazonosításra (RTTI), a kivételkezelésre (§14.4.7) és a virtuális függvényekre vonatkozó szabályokban.

Nem bölcs dolog az objektumfelépítés vagy -megsemmisítés sorrendjére támaszkodni, de a sorrendet megfigyelhetjük, ha virtuális függvényeket, *dynamic_cast*-ot vagy *typeid*-t (§15.4.4) hívunk akkor, amikor az objektum még nincs készen. Például ha a §15.4.2 pontbeli hierarchia *Component* konstruktora egy virtuális függvényt hív, akkor a *Storable* vagy *Component*-beli változatot fogja meghívni, nem a *Receiver*, *Transmitter* vagy *Radio*-belit. Az objektum létrehozásának ezen pontján az objektum még nem *Radio*, csak egy részben felépített objektum. Ennek fényében legjobb elkerülni a virtuális függvényeknek konstruktorból vagy destruktorból való meghívását.

15.4.4. Typeid és kiterjesztett típusinformáció

A *dynamic_cast* operátor az objektumok típusára vonatkozó, futási időben jelentkező információigény legnagyobb részét kielégíti. Fontos tulajdonsága, hogy biztosítja a felhasználó kód helyes működését a programozó által használt osztályokból származó osztályokkal is. Így a *dynamic_cast* a virtuális függvényekhez hasonlóan megőrzi a rugalmasságot és bővíthetőséget.

Néha azonban alapvető fontosságú tudni az objektum pontos típusát. Például tudni szeretnénk az objektum osztályának nevét vagy memóriakiosztását. A *typeid* operátor ezt az operandusa típusát jelző objektum visszaadásával támogatja. Ha a *typeid()* függvény lenne, valahogy így adhatnánk meg:

```
class type_info;
const type_info& typeid(type_name) throw();              // ál-deklaráció
const type_info& typeid(expression) throw(bad_typeid);   // ál-deklaráció
```

Azaz a *typeid()* egy standard könyvtárbeli, a <*typeinfo*> fejállományban definiált *type_info* nevű típusra való referenciát ad vissza. Ha operandusként egy típusnevet kap, a *typeid()* az azt ábrázoló *type_info*-ra való referenciával tér vissza, ha kifejezést, a kifejezés által jelölt objektumot ábrázoló *type_info*-ra fog hivatkozni. A *typeid()* leginkább egy referenciával vagy mutatóval jelölt objektum típusának lekérdezésére használatos:

```
void f(Shape& r, Shape* p)
{
    typeid(r);        // az r által hivatkozott objektum típusa
    typeid(*p);       // a p által mutatott objektum típusa
    typeid(p);        // a mutató típusa, vagyis Shape* (nem gyakori, leginkább tévedés)
}
```

Ha a mutató vagy referencia operandus értéke *0*, a *typeid() bad_typeid* kivételt vált ki.

A *type_info* megvalósítás-független része így néz ki:

```
class type_info {
public:
    virtual ~type_info();                          // többalakú

    bool operator==(const type_info&) const;       // összehasonlítható
    bool operator!=(const type_info&) const;
    bool before(const type_info&) const;           // rendezés

    const char* name() const;                      // a típus neve
private:
    type_info(const type_info&);                   // másolás megakadályozása
    type_info& operator=(const type_info&);        // másolás megakadályozása
    // ...
};
```

A *before()* függvény lehetővé teszi a *type_info* objektumok rendezését. A meghatározott rendezési sorrendnek nincs köze az öröklési viszonyokhoz.

Nem biztos, hogy a rendszer minden egyes típusát egyetlen *type_info* objektum képviseli. Dinamikus csatolású könyvtárak használata esetén például valóban nehéz a több *type_info* objektumot elkerülő megvalósítás elkészítése. Ezért a == művelettel az egyenlőséget a *type_info* objektumok és nem az azokra hivatkozó mutatók esetében vizsgáljuk.

Néha tudni akarjuk egy objektum pontos típusát, hogy valamilyen szabványos műveletet végezzünk az egész objektumon (és nem csak annak egy ősén). Ideális esetben az ilyen műveletek virtuális függvények formájában állnak rendelkezésre, így nem szükséges a pontos típus ismerete. Egyes esetekben azonban nem tételezhető minden egyes kezelt objektumra vonatkozó közös felület, így a megoldás útja a pontos típuson keresztül vezet (§15.4.4.1). Egy másik, sokkal egyszerűbb használat az osztály nevének diagnosztikai kimenet céljára való lekérdezése:

```
#include<typeinfo>

void g(Component* p)
{
    cout << typeid(*p).name();
}
```

Az osztályok nevének szöveges ábrázolása az adott nyelvi változattól függ. C stílusú karakterláncokkal történik, amelyek a rendszerhez tartozó memóriarészben vannak, ezért ne próbáljuk meg a *delete[ ]* műveletet alkalmazni rájuk.

15.4.4.1. Kiterjesztett típusinformáció

Egy objektum pontos típusának meghatározása általában csak az első lépés a rá vonatkozó részletesebb információk megszerzése és használata felé.

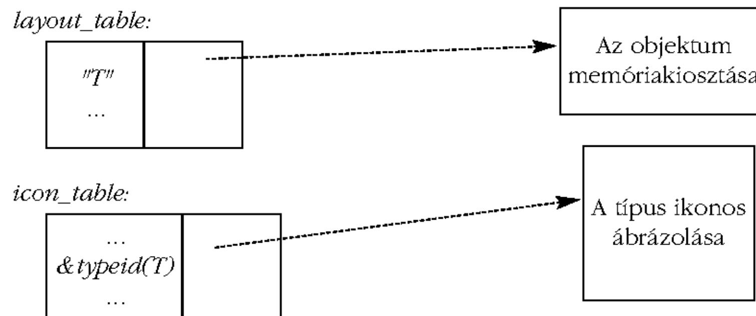
Gondoljuk meg, hogy egy program vagy programozást segítő eszköz hogyan tudna futási időben típusokról szóló információt adni a felhasználóknak. Tegyük fel, hogy van egy eszközünk, amely minden felhasznált osztályról megmondja az objektum memóriakiosztását. Ezeket a leírókat egy *map*-be tehetem, hogy annak alapján a felhasználói kód megtalálhassa a memóriakiosztási információt:

```
map<string, Layout> layout_table;

void f(B* p)
{
    Layout& x = layout_table[typeid(*p).name()];
    // x használata
}
```

Valaki más egészen eltérő információt adhat:

```
struct TI_eq {
    bool operator()(const type_info* p, const type_info* q) { return *p==*q; }
};

struct TI_hash {
    int operator()(const type_info* p);              // hasítóérték kiszámítása (§17.6.2.2)
};

hash_map<const type_info*,Icon,hash_fct,TI_hash,TI_eq> icon_table;         // §17.6

void g(B* p)
{
    Icon& i = icon_table[&typeid(*p)];
    // i használata
}
```

A *typeid*-ekhez információ rendelésének ez a módja több programozó vagy eszköz számára teszi lehetővé, hogy a típusokhoz egymástól teljesen független információkat rendeljenek:



Ez nagyon fontos, mert annak valószínűsége, hogy valaki információknak olyan halmazával tud előállni, amely egymagában minden felhasználó igényeit kielégíti, a nullával egyenlő.

15.4.5. Az RTTI helyes és helytelen használata

Csak szükség esetén használjunk explicit futási idejű típusinformációt. A statikus (fordítási időben történő) ellenőrzés biztonságosabb, „olcsóbb" és – alkalmasint – jobban szerkesztett programokhoz vezet. A futási idejű típusinformációt például arra használhatjuk, hogy egy rosszul álcázott *switch* utasítást írjunk:

```
// a futási idejű típusinformáció helytelen használata

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
            // nem csinálunk semmit
    }
    else if (typeid(r) == typeid(Triangle)) {
            // háromszög forgatása
    }
    else if (typeid(r) == typeid(Square)) {
            // négyzet forgatása
    }
    // ...
}
```

A *dynamic_cast*-nak a *typeid* helyett való használata alig javítana ezen a kódon.

Sajnos ez nem egy légből kapott példa; ilyen kódot tényleg írnak. A C-hez, a Pascalhoz, a Modula-2-höz, vagy az Adához hasonló nyelveken nevelkedett programozó majdnem ellenállhatatlan kísértést érez, hogy a programot *switch* utasítások halmazaként építse fel. Ennek a kísértésnek általában ellen kell állni. Futási idejű típusazonosítás helyett inkább virtuális függvényeket (§2.5.5, §12.2.6) használjunk a legtöbb olyan eset kezelésére, amikor a típuson alapuló futási idejű megkülönböztetés szükséges.

Az RTTI helyes használata sokszor merül fel akkor, amikor a kódban valamilyen szolgáltatást egy bizonyos osztályhoz kapcsolunk és a felhasználó öröklődéssel akar további szolgáltatásokat hozzáadni. A §15.4 pontbeli *Ival_box* használata ennek egy példája. Ha a felhasználó hajlandó és képes a könyvtári osztályok – például a *BBwindow* – módosítására, akkor az RTTI használata elkerülhető; különben viszont szükséges. De ha a felhasználó hajlandó is a könyvtári osztályok módosítására, az ilyen módosítás problémákhoz vezethet. Például szükségessé válhat a virtuális függvények ál-megvalósítása olyan osztályok esetében, amelyeknél azok nem szükségesek vagy nem értelmesek. Ezt a problémát a §24.4.3 némileg részletesebben tárgyalja. Az RTTI-nek egy egyszerű, objektumokat kezelő ki- és bemeneti rendszer elkészítésére szolgáló használatát a §25.4.1 írja le.

Azok számára, akik a Smalltalkon vagy a Lispen, esetleg más, nagymértékben a dinamikus típusellenőrzésre építő nyelveken nevelkedtek, csábító dolog az RTTI-t túlságosan általános típusokkal együtt használni. Vegyük ezt a példát:

```
// a futási idejű típusinformáció helytelen használata

class Object { /* ... */ };   // többalakú

class Container : public Object {
public:
     void put(Object*);
     Object* get();
     // ...
};

class Ship : public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
     c->put(ps);
     // ...
     Object* p = c->get();
     if (Ship* q = dynamic_cast<Ship*>(p)) {          // futási idejű ellenőrzés
               return q;
     }
```

```
        else {
                // valami mást csinálunk (általában hibakezelést végzünk)
        }
}
```

Itt az *Object* osztály szükségtelen és mesterkélt. Túlságosan általános, mert az adott alkalmazásban nem felel meg semmilyen elvonatkoztatási szintnek és a programozót egy megvalósítás-szintű fogalom használatára kényszeríti. Az ilyen jellegű problémákat gyakran jobban oldja meg, ha kizárólag egy adott típusú mutatót tartalmazó tároló sablonokat használunk:

```
Ship* f(Ship* ps, list<Ship*>& c)
{
    c.push_front(ps);
    // ...
    return c.pop_front();
}
```

Virtuális függvényekkel együtt használva így majdnem minden esetet megoldhatunk.

15.5. Tagra hivatkozó mutatók

Sok osztálynak van egyszerű, nagyon általános felülete, amelyet sokféle használatra szántak. Például sok „objektumorientált" felhasználói felület határoz meg egy sor kérést, amelyre minden, a képernyőn megjelenített objektumnak tudnia kell válaszolni. Ráadásul az ilyen kérések közvetett vagy közvetlen módon programoktól érkezhetnek. Vegyük ezen elv egy egyszerű változatát:

```
class Std_interface {
public:
    virtual void start() = 0;
    virtual void suspend() = 0;
    virtual void resume() = 0;
    virtual void quit() = 0;
    virtual void full_size() = 0;
    virtual void small() = 0;

    virtual ~Std_interface() {}
};
```

Minden művelet pontos jelentését az az objektum definiálja, amelyre alkalmazzák. Gyakran a kérést kiadó személy vagy program és a fogadó objektum között egy szoftverréteg van. Ideális esetben az ilyen köztes rétegeknek nem kell semmit tudniuk az egyes műveletekről (*resume()*, *full_size()* stb.). Ha tudnának, a köztes rétegeket fel kellene újítani, valahányszor a műveletek halmaza megváltozik. Következésképpen az ilyen köztes rétegek csupán továbbítanak valamely, az alkalmazandó műveletre vonatkozó adatot a kérés forrásától a fogadóhoz.

Ennek egyszerű módja az alkalmazandó műveletet jelölő karakterlánc küldése. Például a *suspend()* meghívása céljából a „suspend” (felfüggesztés) szöveget küldhetnénk. Valakinek azonban létre kell hoznia a karakterláncot, valakinek pedig meg kell fejtenie, melyik művelethez tartozik, ha egyáltalán tartozik valamelyikhez. Ez gyakran túlságosan közvetettnek és fáradságosnak tűnik. Ehelyett küldhetnénk csak egy, a műveletet jelölő egész értéket. Mondjuk a *2* jelenthetné a *suspend*-et. De amíg egy egész számot a számítógépek kényelmesen kezelnek, az emberek számára ez meglehetősen zavaró lehet, ráadásul még mindig meg kell írnunk a kódot, amely meghatározza, hogy a *2* a *suspend()*-et jelenti és meg kell hívnunk a *suspend()*-et.

A C++ nyelv lehetővé teszi az osztályok tagjainak közvetett elérését. A tagra hivatkozó mutató olyan érték, amely az osztály egy tagját azonosítja. Gondolhatunk rá úgy, mint egy, az adott osztályhoz tartozó objektumban levő tag helyére, de a fordító természetesen figyelembe veszi az adattagok, a virtuális és nem virtuális függvények stb. közötti különbséget.

Vegyük az *Std_interface*-t. Ha meg akarjuk hívni a *suspend()*-et valamely objektumra anélkül, hogy közvetlenül megneveznénk, akkor egy olyan mutatóra lesz szükségünk, ami az *Std_interface::suspend()* tagra mutat. Ugyancsak szükségünk lesz a *suspend()* révén felfüggesztendő objektumra hivatkozó mutatóra vagy referenciára. Vegyünk egy igen egyszerű példát:

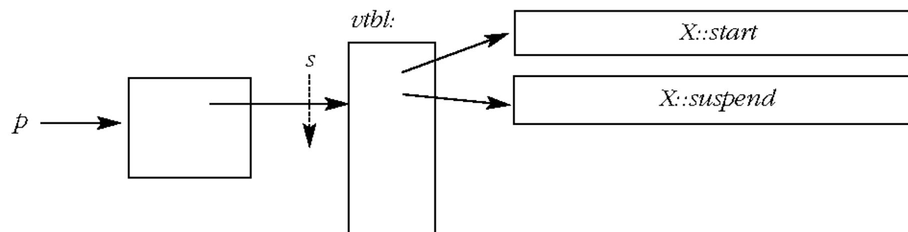
```
typedef void (Std_interface::* Pstd_mem)();          // tagra hivatkozó mutató

void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend;

    p->suspend();            // közvetlen hívás

    (p->*s)();               // hívás tagra hivatkozó mutatón keresztül
}
```

Egy *tagra hivatkozó mutatót* (pointer to member) úgy kapunk, hogy a címképző & operátort egy teljes nevű („teljesen minősített”, fully qualified) osztálytagra alkalmazzuk (például *&Std_interface::suspend()*). Az „*X* osztály egy tagjára hivatkozó mutató” típusú változókat az *X::** formában deklarálhatjuk.

A C szintaxis áttekinthetőségének hiányát általában a *typedef* alkalmazásával ellensúlyozzák, az *X::** forma viszont láthatóan nagyon szépen megfelel a hagyományos * deklarátornak.

Egy *m* tagra hivatkozó mutatót egy objektummal kapcsolatban használhatunk. A kapcsolatot a ->* és .* operátorokkal fejezhetjük ki. Például a *p->*m* az *m*-et a *p* által mutatott objektumhoz köti, az *obj.*m* pedig az *obj* objektumhoz. Az eredmény az *m* típusának megfelelően használható, de a ->* vagy .* művelet eredményét nem lehet későbbi használatra félretenni.

Természetesen ha tudnánk, melyik tagot akarjuk meghívni, közvetlenül meghívnánk azt, a tagra hivatkozó mutatókkal való bajlódás helyett. A közönséges függvénymutatókhoz hasonlóan a tagfüggvényekre hivatkozó mutatókat akkor használjuk, amikor a tag nevének ismerete nélkül akarunk egy függvényre hivatkozni. A tagra hivatkozó mutató azonban nem egyszerűen egy mutató egy memóriaterületre, mint egy változó címe vagy egy függvénymutató, inkább egy adatszerkezeten belüli eltolásra (offszetre) vagy egy tömbbeli indexre hasonlít. Amikor egy tagra hivatkozó mutatót a megfelelő típusú objektumra hivatkozó mutatóval párosítjuk, olyasvalami keletkezik, ami egy bizonyos objektum egy bizonyos tagját azonosítja. Ezt rajzzal így ábrázolhatjuk:



Mivel egy virtuális tagra hivatkozó mutató (a fenti példában *s*) egyfajta eltolás, nem függ az objektum helyétől a memóriában, ezért biztonságosan átadható egy másik címtérnek (address space), feltéve, hogy az objektum elhelyezkedése a kettőben azonos. A közönséges függvényekre hivatkozó mutatókhoz hasonlóan a nem virtuális tagfüggvényekre hivatkozó mutatókat sem lehet másik címtérnek átadni.

Jegyezzük meg, hogy a függvénymutatón keresztül meghívott függvény lehet virtuális is. Például amikor egy függvénymutatón keresztül meghívjuk a *suspend()*-et, akkor azon objektum típusának megfelelő *suspend()*-et kapjuk, amelyre a függvénymutatót alkalmaztuk. Ez a függvénymutatóknak igen lényeges tulajdonsága.

Egy tolmácsoló (továbbító, interpreter) függvény egy tagra hivatkozó mutató segítségével meghívhat egy karakterlánccal (vagyis szöveggel) megadott függvényt:

```
map<string,Std_interface*> variable;
map<string,Pstd_mem> operation;

void call_member(string var, string oper)
{
    (variable[var]->*operation[oper])();          // var.oper()
}
```

A tagfüggvényekre hivatkozó mutatók legfontosabb használatát a §3.8.5 és §18.4 pontbeli *mem_fun()* függvényben vizsgálhatjuk meg.

A statikus tagok nem tartoznak egy bizonyos objektumhoz, így egy statikus tagra hivatkozó mutató csupán egy közönséges mutató:

```
class Task {
    // ...
    static void schedule();
};

void (*p)() = &Task::schedule;              // rendben
void (Task::* pm)() = &Task::schedule;      // hiba: egyszerű mutatót adtunk értékül
                                            // tagra hivatkozó mutatónak
```

Az adattagokra hivatkozó mutatókat a §C.12 pont írja le.

15.5.1. Bázis- és származtatott osztályok

Egy származtatott osztály legalább azokkal a tagokkal rendelkezik, amelyeket a bázisosztályból örököl, de gyakran többel. Ez azt jelenti, hogy egy bázisosztály egy tagjára hivatkozó mutatót biztonságosan értékül adhatunk egy származtatott osztálytagra hivatkozó mutatónak, de fordítva nem. Ezt a tulajdonságot gyakran ***kontravarianciának*** (contravariance) hívják:

```
class text : public Std_interface {
public:
    void start();
    void suspend();
    // ...
    virtual void print();
private:
    vector s;
};

void (Std_interface::* pmi)() = &text::print;         // hiba
void (text::*pmt)() = &Std_interface::start;           // rendben
```

A fel nem cserélhetőség ezen szabálya látszólag ellenkezik azzal a szabállyal, hogy egy származtatott osztályra hivatkozó mutatót értékül adhatunk a bázisosztályára hivatkozó mutatónak. Valójában azonban mindkét szabály azért van, hogy biztosítsa az alapvető garanciát arra, hogy egy mutató soha nem mutat olyan objektumra, amelynek nincsenek meg a mutató típusa által ígért tulajdonságai. Ebben az esetben az *Std_interface::** bármilyen *Std_interface*-re alkalmazható, és a legtöbb ilyen objektum vélhetőleg nem *text* típusú lesz. Ezért aztán nincs meg a *text::print* tagjuk, amellyel *pmi*-nek kezdőértéket próbáltunk adni. A kezdeti értékadás megtagadásával a fordítóprogram egy futási idejű hibától ment meg bennünket.

15.6. A szabad tár

Az *operator new()* és az *operator delete()* definiálásával át lehet venni a memóriakezelést egy osztálytól (§6.2.6.2). Ezen globális függvények kicserélése azonban nem a félénkeknek való, hiszen előfordulhat, hogy valaki az alapértelmezett viselkedésre számít vagy ezen függvények valamilyen más változatát már meg is írta.

Gyakran jobb megközelítés ezeket a műveleteket egy bizonyos osztályra megírni. Ez az osztály több származtatott osztály alapja is lehet. Tegyük fel, hogy a §12.2.6-beli *Employee* osztályt és annak minden leszármazottját szeretnénk egyedi memóriafoglalóval (allokátorral) és -felszabadítóval (deallokátorral) ellátni:

```
class Employee {
    // ...
public:
    // ...
```

```
    void* operator new(size_t);
    void operator delete(void*, size_t);
};
```

Az *operator new()* és az *operator delete()* tagfüggvények automatikusan statikusak lesznek, ezért nincs *this* mutatójuk és nem változtatnak meg egy objektumot, csak olyan tárterületet adnak, amelyet a konstruktorok feltölthetnek és a destruktorok kitakaríthatnak.

```
void* Employee::operator new(size_t s)
{
    // lefoglal 's' bájtnyi memóriát és rá hivatkozó mutatót ad vissza
}

void Employee::operator delete(void* p, size_t s)
{
   if (p) {             // törlés csak ha p!=0; lásd §6.2.6, §6.2.6.2
     // feltesszük, hogy 'p' 's' bájt memóriára mutat, amit az Employee::operator new() foglalt le
     // felszabadítjuk a memóriát
   }
}
```

Az eddig rejtélyes *size_t* paraméter haszna most világossá válik. Ez ugyanis a ténylegesen törlendő objektum mérete. Ha egy „sima" *Employee*-t törlünk, akkor paraméterértékként *sizeof(Employee)*-t; ha egy *Manager*-t, akkor *sizeof(Manager)*-t kapunk. Természetesen az adott osztály egyedi memóriafoglalója tárolhatja az ilyen információt (mint ahogy az általános célúnak is meg kell ezt tennie) és nem törődhet az *operator delete()* függvény *size_t* paraméterével. Ez azonban nehezebbé teszi egy általános célú memóriafoglaló sebességének és memóriafogyasztásának javítását.

Honnan tudja a fordítóprogram a *delete()* operátort a megfelelő mérettel ellátni? Könnyen, amíg a *delete* műveletnek átadott típus azonos az objektum tényleges típusával. Ez azonban nincs mindig így:

```
class Manager : public Employee {
    int level;
    // ...
};

void f()
{
    Employee* p = new Manager;     // problémás (a pontos típus ismerete elvész)
    delete p;
}
```

Ekkor a fordítóprogram nem fogja tudni a pontos méretet. A tömb törléséhez hasonlóan itt is a programozó segítségére van szükség; az Employee bázisosztályban meg kell adni egy virtuális destruktort:

```
class Employee {
public:
    void* operator new(size_t);
    void operator delete(void*, size_t);
    virtual ~Employee();
    // ...
};
```

Akár egy üres destruktor is megteszi:

```
Employee::~Employee() { }
```

A memória felszabadítását a (méretet ismerő) destruktor végzi. Sőt ha az *Employee*-ben van destruktor, akkor ez biztosítja, hogy minden belőle származtatott osztálynak lesz destruktora (és így tudja a méretet), akkor is, ha a származtatott osztályban nem szerepel felhasználói destruktor:

```
void f()
{
    Employee* p = new Manager;
    delete p;              // most már jó (az Employee többalakú)
}
```

A memória lefoglalása egy (a fordítóprogram által létrehozott) hívással történik:

```
Employee::operator new(sizeof(Manager))
```

A felszabadításról szintén egy (a fordítóprogram által létrehozott) hívás gondoskodik:

```
Employee::operator delete(p,sizeof(Manager))
```

Vagyis ha olyan memóriafoglaló/felszabadító párt akarunk írni, amely származtatott osztályokkal is jól működik, akkor vagy virtuális destruktort kell írni a bázisosztályban, vagy nem szabad felhasználni a felszabadítóban a *size_t* paramétert. Természetesen meg lehetett volna úgy tervezni a nyelvet, hogy ne legyen szükség ilyen megfontolásokra, de ez csak a kevésbé biztonságos rendszerekben lehetséges optimalizálások előnyeiről való lemondás árán történhetett volna.

15.6.1. Memóriafoglalás tömbök számára

Az *operator new()* és az *operator delete()* függvények lehetővé teszik, hogy a programozó végezze az egyes objektumok számára a memóriafoglalást és -felszabadítást. Az *operator new[ ]()* és az *operator delete[ ]()* pontosan ugyanezt a szerepet játssza a tömbök esetében:

```
class Employee {
public:
    void* operator new[ ](size_t);
    void operator delete[ ](void*);
    // ...
};

void f(int s)
{
    Employee* p = new Employee[s];
    // ...
    delete[ ] p;
}
```

Itt a szükséges memóriát a

```
    Employee::operator new[ ](sizeof(Employee)*s+delta)
```

hívás fogja biztosítani, ahol a *delta* az adott fordítótól függő minimális többlet. A memóriát az alábbi hívás szabadítja fel:

```
    Employee::operator delete[ ](p); // felszabadít s*sizeof(Employee)+delta bájtot
```

Az elemek *s* számát (illetve a *delta*-t) a rendszer „megjegyzi". Ha a *delete[ ]()*-et egy- helyett kétparaméteres formában adtuk volna meg, a hívásban a második paraméter *s*sizeof(Employee) +delta* lett volna.

15.6.2. „Virtuális konstruktorok"

Miután virtuális destruktorokról hallottunk, nyilvánvaló a kérdés: „lehetnek-e a konstruktorok is virtuálisak?". A rövid válasz az, hogy nem. A kicsit hosszabb: nem, de a kívánt hatás könnyen elérhető. Egy objektum létrehozásához a konstruktornak tudnia kell a létrehozandó objektum pontos típusát. Ezért a konstruktor nem lehet virtuális. Ráadásul a konstruktor nem egészen olyan, mint a közönséges függvények. Például olyan módokon működik együtt a memóriakezelő eljárásokkal, ahogy a közönséges függvények nem. Ezért nincs konstruktorra hivatkozó mutató.

Mindkét megszorítás megkerülhető egy konstruktort meghívó és a létrehozott objektumot visszaadó függvény készítésével. Ez kedvező, mert gyakran hasznos lehet, ha a pontos típus ismerete nélkül tudunk új objektumot létrehozni. Az *Ival_box_maker* osztály (§12.4.4) pontosan erre a célra szolgált. Itt az ötlet egy másik változatát mutatom be, ahol egy osztály objektumai a felhasználóiknak képesek saját maguk másolatát átadni:

```
class Expr {
public:
    Expr();                     // alapértelmezett konstruktor
    Expr(const Expr&);          // másoló konstruktor

    virtual Expr* new_expr() { return new Expr(); }
    virtual Expr* clone() { return new Expr(*this); }
    // ...
};
```

Mivel a *new_expr()*-hez és a *clone()*-hoz hasonló függvények virtuálisak és közvetett úton objektumokat hoznak létre, gyakran hívják őket „virtuális konstruktornak”, bár az elnevezés némileg félrevezető. Mindegyik egy konstruktort használ, hogy megfelelő objektumot hozzon létre.

Egy származtatott osztály felülírhatja a *new_expr()* és/vagy *clone()* függvényt, hogy egy saját típusú objektumot adjon vissza:

```
class Cond : public Expr {
public:
    Cond();
    Cond(const Cond&);

    Cond* new_expr() { return new Cond(); }
    Cond* clone() { return new Cond(*this); }
    // ...
};
```

Ez azt jelenti, hogy egy *Expr* osztályú objektumhoz a felhasználó „pontosan ugyanolyan típusú” objektumot tud létrehozni:

```
void user(Expr* p)
{
    Expr* p2 = p->new_expr();
    // ...
}
```

A *p2*-höz rendelt mutató megfelelő, bár ismeretlen típusú.

A *Cond::new_expr()* és a *Cond::clone()* visszatérési típusa *Cond** és nem *Expr** , ezért egy *Cond* információvesztés nélkül lemásolható („klónozható"):

```
void user2(Cond* pc, Expr* pe)
{
    Cond* p2 = pc->clone();
    Cond* p3 = pe->clone();          // hiba
    // ...
}
```

A felülíró függvények típusa ugyanaz kell, hogy legyen, mint a felülírt virtuális függvényé, de a visszatérési érték típusa kevésbé szigorúan kötött. Vagyis ha az eredeti visszatérési érték *B** volt, akkor a felülíró függvény visszatérési értéke lehet *D** is, feltéve, hogy *B* nyilvános bázisosztálya *D*-nek. Ugyanígy egy *B&* visszatérési érték *D&*-ra módosítható.

Jegyezzük meg, hogy a paramétertípusok hasonló módosítása típushibához vezetne (lásd §15.8[12]).

15.7. Tanácsok

[1] Ha tulajdonságok unióját akarjuk kifejezni, használjunk közönséges többszörös öröklődést. §15.2, §15.2.4.
[2] A tulajdonságoknak a megvalósító kódtól való elválasztására használjunk többszörös öröklődést. §15.2.5.
[3] Használjunk virtuális bázisosztályt, ha egy hierarchia némely (de nem mindegyik) osztályára nézve közös dolgot akarunk kifejezni. §15.2.5.
[4] Kerüljük a típuskényszerítést (cast). §15.4.5.
[5] Ha az adott osztályhierarchia bejárása elkerülhetetlen, használjuk a *dynamic_cast*-ot. §15.4.1.
[6] A *typeid* helyett részesítsük előnyben a *dynamic_cast*-ot. §15.4.4.
[7] A *protected*-del szemben részesítsük előnyben a *private*-et. §15.3.1.1.
[8] Adattagokat ne adjunk meg védettként. §15.3.1.1.
[9] Ha egy osztály definiálja az *operator delete()*-et, akkor legyen virtuális destruktora. §15.6.
[10] A konstruktor vagy destruktor futása alatt ne hívjunk virtuális függvényt. §15.4.3.
[11] Ritkán – és lehetőleg csak felülíró virtuális függvényekben – használjuk a tagnevek explicit minősítését feloldás céljára. §15.2.1.

15.8. Gyakorlatok

1. (*1) Írjunk olyan *ptr_cast* sablont, amely ugyanúgy működik, mint a *dynamic_cast*, de a *0*-val való visszatérés helyett *bad_cast* kivételt vált ki.
2. (*2) Írjunk olyan programot, amely egy objektum létrehozása közben az RTTI-hez viszonyítva mutatja be a konstruktorhívások sorrendjét. Hasonlóan mutassuk be az objektum lebontását.
3. (*3.5) Írjuk meg a Reversi/Othello társasjáték egy változatát. Minden játékos lehessen élő személy vagy számítógép. Először a program helyes működésére összpontosítsunk, és csak azután arra, hogy a program annyira „okos" legyen, hogy érdemes legyen ellene játszani.
4. (*3) Javítsunk a §15.8[3]-beli játék felhasználói felületén.
5. (*3) Készítsünk egy grafikus objektumosztályt egy művelethalmazzal, amely alapján grafikus objektumok egy könyvtára számára közös bázisosztály lehet. Nézzünk bele egy grafikus könyvtárba, hogy ott milyen műveletek vannak. Hozzunk létre egy adatbázis-objektum osztályt egy művelethalmazzal, amely alapján elhihető, hogy közös bázisosztálya az adatbázisban mezők soraként tárolt objektumoknak. Vizsgáljunk meg egy adatbázis-könyvtárat, hogy ott milyen műveletek vannak. Határozzunk meg egy grafikus adatbázis-objektumot többszörös örökléssel és anélkül, és hasonlítsuk össze a két megoldás előnyeit és hátrányait.
6. (*2) Írjuk meg a §15.6.2-beli *clone()* művelet egy olyan változatát, amely a paraméterben megkapott *Arena*-ba (§10.4.11) teszi a lemásolt objektumot. Készítsünk egy „egyszerű *Arena*"-t mint az *Arena*-ból származó osztályt.
7. (*2) Anélkül, hogy belenéznénk a könyvbe, írjunk le annyi C++-kulcsszót, amennyit csak tudunk.
8. (*2) Írjunk szabványos C++-programot, amelyben legalább tíz különböző kulcsszó szerepel egymás után, úgy, hogy nincs azonosítókkal, operátorokkal vagy írásjelekkel elválasztva.
9. (*2.5) Rajzoljuk le a §15.2.4-beli *Radio* objektum memóriakiosztásának egy lehetséges változatát. Magyarázzuk el, hogyan lehetne egy virtuális függvényt meghívni.
11. (*3) Gondoljuk meg, hogyan lehet a *dynamic_cast*-ot megvalósítani. Készítsünk egy *dcast* sablont, amely úgy viselkedik, mint a *dynamic_cast*, de csak az általunk meghatározott függvényekre és adatokra támaszkodik. Gondoskodjunk arról, hogy a rendszert a *dcast* vagy az előzőleg megadott osztályok megváltoztatása nélkül lehessen új osztályokkal bővíteni.
12. (*2) Tegyük fel, hogy a függvényparaméterek típus-ellenőrzési szabályai a visszatérési értékre vonatkozóakhoz hasonlóan enyhítettek, tehát egy *Derived** paraméterű függvény felülírhat egy *Base**-ot. Írjunk egy programot, ami konverzió nélkül elrontana egy *Derived* típusú objektumot. Írjuk le a paramétertípusokra vonatkozó felülírási szabályok egy biztonságos enyhítését.

Harmadik rész
A standard könyvtár

Ebben a részben a C++ standard könyvtárát mutatjuk be. Megvizsgáljuk a könyvtár szerkezetét és azokat az alapvető módszereket, amelyeket az egyes elemek megvalósításához használtak. Célunk az, hogy megértsük, hogyan kell használni ezt a könyvtárat, illetve szemléltessük azokat az általános módszereket, amelyeket tervezéskor és programozáskor használhatunk. Ezenkívül bemutatjuk azt is, hogyan bővíthetjük a könyvtárat, pontosabban a rendszer fejlesztői hogyan képzelték el a továbbfejlesztést.

Fejezetek

16. A könyvtár szerkezete és a tárolók
17. Szabványos tárolók
18. Algoritmusok és függvényobjektumok
19. Bejárók és memóriafoglalók
20. Karakterláncok
21. Adatfolyamok
22. Számok

„...s te, Marcus, oly sok mindent adtál nekem, ím én adok neked most egy jótanácsot. Sok ember légy. Hagyd el a régi játékot, hogy mindig csak ugyanaz a Marcus Cocoza vagy. Túlontúl sokat vessződtél már Marcus Cocozával, míg végül valósággal a rabszolgája lettél. Szinte semmit sem teszel anélkül, hogy ne mérlegelnéd, vajon milyen hatással lesz Marcus Cocoza boldogságára és tekintélyére. Szüntelen félelemben élsz, hogy Marcus Cocoza esetleg valami ostobaságot követ el, vagy elunja magát. Dehát mit számít mindez valójában? Az emberek az egész világon ostobaságokat művelnek... Szeretném ha felszabadulnál, ha szíved újra megtalálná békéjét. Mostantól fogva ne csak egy, hanem több ember légy, annyi, amennyit csak el tudsz gondolni...”

(Karen Blixen: Álmodozók; Kertész Judit fordítása)

16

A könyvtár szerkezete és a tárolók

„Újdonság volt.
Egyedi volt. Egyszerű volt.
Biztos, hogy sikeres lesz!"
(H. Nelson)

Tervezési szempontok a standard könyvtárhoz • A könyvtár szerkezete • Szabványos fejállományok • Nyelvi támogatás • A tárolók szerkezete • Bejárók • Bázisosztállyal rendelkező tárolók • Az STL tárolói • *vector* • Bejárók • Elemek elérése • Konstruktorok • Módosítók • Listaműveletek • Méret és kapacitás • *vector<bool>* • Tanácsok • Gyakorlatok

16.1. A standard könyvtár

Minek kell szerepelnie a C++ standard könyvtárában? A programozó szempontjából az tűnik ideálisnak, ha egy könyvtárban megtalál minden olyan osztályt, függvényt, és sablont, ami érdekes, jelentős és eléggé általános. A kérdés azonban most nem az, hogy *egy* könyvtárban mi legyen benne, hanem az, hogy a *standard* könyvtár milyen elemeket tartalmazzon. Az előbbi kérdés esetében ésszerű megközelítés, hogy minden elérhető legyen, az utóbbi esetben azonban ez nem alkalmazható. A standard könyvtár olyan eszköz, amelyet minden C++-változat tartalmaz, így minden programozó számíthat rá.

A C++ standard könyvtára a következő szolgáltatásokat nyújtja:

1. Támogatja a nyelv lehetőségeinek használatát, például a memóriakezelést (§6.2.6) és a futási idejű típusinformáció (RTTI, §15.4) kezelését.
2. Információkat ad az adott nyelvi változat egyedi tulajdonságairól, például megadja a legnagyobb *float* értéket (§22.2).
3. Elérhetővé teszi azokat a függvényeket, amelyeket nem lehet a nyelven belül optimálisan elkészíteni minden rendszer számára (például *sqrt()*, §22.3 vagy *memmove()*, §19.4.6).
4. Olyan magas szintű szolgáltatásokat nyújt, amelyekre a programozó más rendszerre átvihető (hordozható) programok készítésekor támaszkodhat, például listákat (§17.2.2), asszociatív tömböket (map) (§17.4.1), rendező függvényeket (§18.7.1) és bemeneti/kimeneti adatfolyamokat (21. fejezet).
5. Keretet biztosít a könyvtár elemeinek továbbfejlesztésére, például szabályokkal és eszközökkel segíti, hogy a felhasználó ugyanolyan bemeneti/kimeneti felületet biztosíthasson saját típusaihoz, mint a könyvtár a beépített típusokhoz.
6. További könyvtárak közös alapjául szolgál.

Néhány további eszközt – például a véletlenszám-előállítókat (§22.7) – csak azért helyeztek a standard könyvtárba, mert így használatuk kényelmes és hagyományosan itt a helyük.

A könyvtár tervezésekor leginkább az utolsó három szerepkört vették figyelembe. Ezek a szerepek erősen összefüggnek. A hordozhatóság például olyan általános fogalom, amely fontos tervezési szempont minden egyedi célú könyvtár esetében. A közös tárolótípusok (például a listák vagy asszociatív tömbök) pedig igen jelentősek a külön fejlesztett könyvtárak kényelmes együttműködésének biztosításában.

Az utolsó pont különösen fontos a tervezés szempontjából, mert ezzel korlátozható a standard könyvtár hatóköre és gátat szabhatunk a szolgáltatások özönének. A karakterlánc- és listakezelő lehetőségek például a standard könyvtárban kaptak helyet. Ha ez nem így történt volna, akkor a külön fejlesztett könyvtárak csak a beépített típusok segítségével működhetnének együtt egymással. Ugyanakkor a mintaillesztő és a grafikai lehetőségek nem szerepelnek itt, mert – bár tagadhatatlanul széles körben használatosak – nem kimondottan a külön fejlesztett könyvtárak közötti együttműködést szolgálják.

Ha egy lehetőség nem feltétlenül szükséges a fenti szerepkörök teljesítéséhez, akkor azt megvalósíthatjuk külön, a standard könyvtáron kívül is. Azzal, hogy kihagyunk egy szolgáltatást a standard könyvtárból, azt a lehetőséget is nyitva hagyjuk a további könyvtárak számára, hogy ugyanazt az ötletet más-más formában valósítsák meg.

16.1.1. Tervezési korlátozások

A standard könyvtárak szerepköreiből számos korlátozás következik a könyvtár szerkezetére nézve. A C++ standard könyvtára által kínált szolgáltatások tervezésekor az alábbi szempontokat tartották szem előtt:

1. Komoly segítséget jelentsen lényegében minden tanuló és profi programozó számára, közéjük értve a további könyvtárak fejlesztőit is.
2. Közvetve vagy közvetlenül minden programozó felhasználhassa az összes olyan célra, ami a könyvtár hatáskörébe esik.
3. Elég hatékony legyen ahhoz, hogy a későbbi könyvtárak előállításában komoly vetélytársa legyen a saját kezűleg előállított függvényeknek, osztályoknak és sablonoknak.
4. Mentes legyen az eljárásmód szabályozásától, vagy lehetőséget adjon rá, hogy a felhasználó paraméterként határozza meg az eljárásmódot.
5. Legyen primitív, a matematikai értelemben. Egy olyan összetevő, amely két, gyengén összefüggő feladatkört tölt be, kevésbé hatékony, mint két önálló komponens, amelyeket kimondottan az adott szerep betöltésére fejlesztettek ki.
6. Legyen kényelmes, hatékony és elég biztonságos a szokásos felhasználási területeken.
7. Nyújtson teljeskörű szolgáltatásokat ahhoz, amit vállal. A standard könyvtár fontos feladatok megvalósítását is ráhagyhatja más könyvtárakra, de ha egy feladat teljesítését vállalja, akkor elegendő eszközt kell biztosítania ahhoz, hogy az egyes felhasználóknak és fejlesztőknek ne kelljen azokat más módszerekkel helyettesíteniük.
8. Legyen összhangban a beépített típusokkal és műveletekkel és biztasson azok használatára.
9. Alapértelmezés szerint legyen típusbiztos (mindig helyesen kezelje a különböző típusokat).
10. Támogassa az általánosan elfogadott programozási stílusokat.
11. Legyen bővíthető úgy, hogy a felhasználó által létrehozott típusokat hasonló formában kezelhessük, mint a beépített és a standard könyvtárban meghatározottakat.

Rossz megoldás például, ha egy rendező függvénybe beépítjük az összehasonlítási módszert, hiszen ugyanazok az adatok más-más szempont szerint is rendezhetők. Ezért a C standard könyvtárának *qsort()* függvénye paraméterként veszi át az összehasonlítást végző függvényt, és nem valamilyen rögzített műveletre (például a < operátorra) hivatkozik (§7.7). Másrészről ebben a megoldásban minden egyes összehasonlítás alkalmával meg kell hív-

nunk egy függvényt, ami többletterhet jelent a *qsort()* eljárást felhasználó további eljárásokban. Szinte minden adattípus esetében könnyen elvégezhetjük az összehasonlítást, anélkül, hogy függvényhívással rontanánk a rendszer teljesítményét.

Jelentős ez a teljesítményromlás? A legtöbb esetben valószínűleg nem. Egyes algoritmusok esetében azonban ezek a függvényhívások adják a végrehajtási idő jelentős részét, ezért ilyenkor a felhasználók más megoldásokat fognak keresni. Ez a problémát a §13.4 pontban oldottuk meg, ahol bemutattuk, hogyan adhatunk meg összehasonlítási feltételt egy sablonparaméterrel. A példa szemléltette, hogy a hatékonyság és az általánosság két egymással szemben álló követelmény. Egy szabvány könyvtárnak azonban nem csak meg kell valósítania feladatait, hanem elég hatékonyan kell azokat elvégeznie ahhoz, hogy a felhasználóknak eszébe se jusson saját eljárásokat írni az adott célra. Ellenkező esetben a bonyolultabb eszközök fejlesztői kénytelenek kikerülni a standard könyvtár szolgáltatásait, hiszen csak így maradhatnak versenyképesek. Ez pedig nem csak a könyvtárak fejlesztőinek okoz sok többletmunkát, hanem azon felhasználók életét is megnehezíti, akik platformfüggetlen programokat szeretnének készíteni vagy több, külön fejlesztett könyvtárat szeretnének használni.

A „primitívség" és a „kényelmesség a szokásos felhasználási területeken" követelmények is szemben állnak egymással. Egy szabvány könyvtárban az első követelmény azonnal kizár minden optimalizálási lehetőséget arra nézve, hogy felkészüljünk a gyakori esetekre. Az olyan összetevők azonban, amelyek általános, de nem primitív szolgáltatásokat nyújtanak, szerepelhetnek a standard könyvtárban a „primitívek" mellett, de nem helyettük. A kölcsönös kizárás nem akadályozhat bennünket abban, hogy mind a profi, mind az alkalmi programozó életét egyszerűbbé tegyük. Azt sem engedhetjük meg, hogy egy összetevő alapértelmezett viselkedése homályos vagy veszélyes legyen.

16.1.2. A standard könyvtár szerkezete

A standard könyvtár szolgáltatásait az *std* névtérben definiálták és fejállományok segítségével érhetjük el azokat. Ezek a fejállományok jelzik a könyvtár legfontosabb részeit, így felsorolásukból áttekintést kaphatunk a kínált eszközökről. Ezek szerint nézzük végig a könyvtárat a most következő fejezetekben is.

Ezen alfejezet további részében a fejállományokat soroljuk fel, szerepeik szerint csoportosítva. Mindegyikről adunk egy rövid leírást is és megemlítjük, hol található részletes elemzésük. A csoportosítást úgy választottuk meg, hogy illeszkedjen a standard könyvtár szerkezetéhez. Ha a szabványra vonatkozó hivatkozást adunk meg (például §s.18.1), akkor az adott lehetőséget itt nem vizsgáljuk részletesen.

Ha egy szabványos fejállomány neve *c* betűvel kezdődik, akkor az egy szabványos C könyvtárbeli fejállomány megfelelője. Minden <*X.h*> fejállományhoz, amely a C standard könyvtárának részeként neveket ad meg a globális névtérben, létezik egy <cX> megfelelő, amely ugyanazon neveket az *std* névtérbe helyezi (§9.2.2).

Tárolók		
<*vector*>	T típusú elemek egydimenziós tömbje	§16.3
<*list*>	T típusú elemek kétirányú listája	§17.2.2
<*deque*>	T típusú elemek kétvégű sora	§17.2.3
<*queue*>	T típusú elemekből képzett sor	§17.3.2
<*stack*>	T típusú elemekből képzett verem	§17.3.1
<*map*>	T típusú elemek asszociatív tömbje	§17.4.1
<*set*>	T típusú elemek halmaza	§17.4.3
<*bitset*>	logikai értékek tömbje	§17.5.3

A *multimap* és *multiset* asszociatív tárolókat sorrendben a <*map*>, illetve a <*set*> állományban találhatjuk meg. A *priority_queue* osztály a <*queue*> fájlban szerepel.

Általános eszközök		
<*utility*>	operátorok és párok	§17.1.4, §17.4.1.2
<*functional*>	függvényobjektumok	§18.4
<*memory*>	memóriafoglalók a tárolókhoz	§19.4.4
<*ctime*>	C stílusú dátum- és időkezelés	§s.20.5

A <*memory*> fejállomány tartalmazza az *auto_ptr* sablont is, amely elsősorban arra használható, hogy simábbá tegyük a mutatók és kivételek együttműködését (§14.4.2).

Bejárók		
<*iterator*>	bejárók és kezelésük	19. fejezet

A bejárók (iterator) lehetővé teszik, hogy a szabványos algoritmusokat általánosan használhassuk a szabványos tárolókban és más hasonló típusokban (§2.7.2, §19.2.1).

Algoritmusok		
<algorithm>	általános algoritmusok	18.fejezet
<cstdlib>	*bsearch() qsort()*	§18.11

Egy szokványos általános algoritmus egyformán alkalmazható bármilyen típusú elemek bármilyen sorozatára (§3.8, §18.3). A C standard könyvtárában szereplő *bsearch()* és *qsort()* függvények csak a beépített tömbökre használhatók, melyek elemeihez a felhasználó nem határozhat meg sem másoló konstruktort, sem destruktort (§7.7).

Ellenőrzések, diagnosztika		
<exception>	kivételosztály	§14.10
<stdexcept>	szabványos kivételek	§14.10
<cassert>	hibaellenőrző makró (feltételezett érték biztosítása)	§24.3.7.2
<cerrno>	C stílusú hibakezelés	§20.4.1

A kivételkezelésre támaszkodó hibaellenőrzést a §24.3.7.1 pontban vizsgáljuk meg.

Karakterláncok		
<string>	T típusú elemekből álló karakterlánc	20. fejezet
<cctype>	karakterek osztályozása	§20.4.2
<cwtype>	„széles” karakterek osztályozása	§20.4.2
<cstring>	C stílusú karakterláncokat kezelő függvények	§20.4.1
<cwchar>	C stílusú széles karakterláncok kezelése	§20.4
<cstdlib>	C stílusú karakterláncokat kezelő függvények	§20.4.1

A <*cstring*> fejállományban szerepelnek az *strlen()*, *strcpy()* stb. függvények. A <*cstdlib*> adja meg az *atof()* és *atoi()* függvényeket, amelyek a C stílusú karakterláncokat alakítják számértékekre.

Ki- és bemenet		
<*iosfwd*>	előzetes deklarációk az I/O szolgáltatásokhoz	§21.1
<*iostream*>	szabványos bemeneti adatfolyamok objektumai és műveletei	§21.2.1
<*ios*>	bemeneti adatfolyamok bázisosztályai	§21.2.1
<*streambuf*>	átmeneti tár adatfolyamokhoz	§21.6
<*istream*>	bemeneti adatfolyam sablon	§21.3.1
<*ostream*>	kimeneti adatfolyam sablon	§21.2.1
<*iomanip*>	adatfolyam-módosítók (manipulátorok)	§21.4.6.2
<*sstream*>	adatfolyamok karakterláncból/karakterláncba	§21.5.3
<*cstdlib*>	karakterosztályozó függvények	§20.4.2
<*fstream*>	adatfolyamok fájlokból/fájlokba	§21.5.1
<*cstdio*>	a *printf()* függvénycsalád	§21.8
<*cwchar*>	*printf()* szolgáltatások „széles” karakterekre	§21.8

Az adatfolyam-módosítók (manipulator) olyan objektumok, melyek segítségével az adatfolyamok állapotát megváltoztathatjuk (§21.4.6). (Például módosíthatjuk a lebegőpontos számok kimeneti formátumát.)

Nemzetközi szolgáltatások		
<*locale*>	kulturális eltérések ábrázolása	§21.7
<*clocale*>	kulturális eltérések ábrázolása C stílusban	§21.7

A *locale* az olyan kulturális eltérések meghatározására szolgál, mint a dátumok írásmódja, a pénzegységek jelölésére használt szimbólumok vagy a karakterláncok rendezésére vonatkozó szabályok, melyek a különböző természetes nyelvekben és kultúrákban jelentősen eltérhetnek.

A programnyelvi elemek támogatása		
<limits>	numerikus értékhatárok	§22.2
<climits>	C stílusú, numerikus, skalár értékhatárokat megadó makrók	§22.2.1
<cfloat>	C stílusú, numerikus, lebegőpontos értékhatárokat megadó makrók	§22.2.1
<new>	dinamikus memóriakezelés	§16.1.3
<typeinfo>	futási idejű típusazonosítás támogatása	§15.4.1
<exception>	kivételkezelés támogatása	§14.10
<cstddef>	C könyvtárak nyelvi támogatása	§6.2.1
<cstdarg>	változó hosszúságú paraméterlistával rendelkező függvények kezelése	§7.6
<csetjmp>	C stílusú verem-visszatekerés	§s.18.7
<cstdlib>	programbefejezés	§9.4.1.1
<ctime>	rendszeróra	§D.4.4.1
<csignal>	C stílusú szignálkezelés	§D.4.4.1

A *<cstddef>* fejállomány határozza meg azt a típust, amit a *sizeof()* függvény visszaad (*size_t*), a mutatóknak egymásból való kivonásakor keletkező érték típusát (*ptrdiff_t*) és a hírhedt *NULL* makrót (§5.1.1).

Numerikus értékek		
<complex>	komplex számok és műveletek	§22.5
<valarray>	numerikus vektorok és műveletek	§22.4
<numeric>	általánosított numerikus műveletek	§22.6
<cmath>	általános matematikai műveletek	§22.3
<cstdlib>	C stílusú véletlenszámok	§22.7

A hagyományt követve az *abs()*, és *div()* függvény a *<cstdlib>* fejállományban található, bár matematikai függvények lévén jobban illenének a *<cmath>* fájlba. (§22.3)

A felhasználók és a könyvtárak fejlesztői nem bővíthetik vagy szűkíthetik a szabványos fejállományokban szereplő deklarációk körét. A fejállományok jelentését úgy sem módosíthatjuk, hogy megpróbálunk makrókat megadni az állományok beszerkesztése előtt vagy megváltoztatjuk a deklarációk jelentését azzal, hogy saját deklarációkat készítünk a fejállo-

mány környezetében (§9.2.3). Bármely program, amely nem tartja be ezeket a játékszabályokat, nem nevezheti magát a szabványhoz igazodónak, és az ilyen trükköket alkalmazó programok általában nem vihetők át más rendszerre. Lehet, hogy ma működnek, de az adott környezet legkisebb változása használhatatlanná teheti őket. Tartózkodjunk az ilyen szemfényvesztéstől.

Ahhoz, hogy a standard könyvtár valamely lehetőséget használhassuk, a megfelelő fejállományt be kell építenünk (*#include*). Nem jelent a szabványnak megfelelő megoldást, ha a megfelelő deklarációkat saját kezűleg adjuk meg programunkban. Ennek oka az, hogy bizonyos nyelvi változatok a beépített szabványos fejállományok alapján optimalizálják a fordítást, mások pedig a standard könyvtár szolgáltatásait optimalizálják, attól függően, milyen fejállományokat használtunk. Általában igaz, hogy a fejlesztők olyan formában használhatják a szabványos fejállományokat, amelyre a programozók nem készülhetnek fel és programjaik készítésekor nem is kell tudniuk ezek szerkezetéről.

Ezzel szemben a programozó specializálhatja a szolgáltatás-sablonokat, például a *swap()* sablon függvényt (§16.3.9), így ezek jobban igazodhatnak a nem szabványos könyvtárakhoz és a felhasználói típusokhoz.

16.1.3. Nyelvi elemek támogatása

A standard könyvtárnak egy kis része a nyelvi elemek támogatásával foglalkozik, azaz olyan szolgáltatásokat nyújt, amelyekre a program futtatásához azért van szükség, mert a nyelv eszközei ezektől függően működnek.

Azon könyvtári függvényeket, amelyek a *new* és *delete* operátorok kezelését segítik, a §6.2.6, a §10.4.11, a §14.4.4 és a §15.6 pontban mutattuk be. Ezek a *<new>* fejállományban szerepelnek.

A futási idejű típusazonosítás elsősorban a *type_info* osztályt jelenti, amely a *<typeinfo>* fejállományban található és a §15.4.4 pont írt le.

A szabványos kivételek osztályait a §14.10 pontban vizsgáltuk, helyük a *<new>*, a *<typeinfo>*, az *<ios>*, az *<exception>*, illetve az *<stdexcept>* fejállományban van.

A program elindításáról és befejezéséről a §3.2, a §9.4 és a §10.4.9 pontban volt szó.

16.2. A tárolók szerkezete

A *tároló* (container) olyan objektum, amely más objektumokat tartalmaz. Példaképpen megemlíthetjük a listákat (*list*), a vektorokat (*vector*) és az asszociatív tömböket (*map*). Általában lehetőségünk van rá, hogy a tárolókban objektumokat helyezzünk el vagy eltávolítsuk azokat onnan. Természetesen ez az ötlet számtalan formában megvalósítható. A C++ standard könyvtárának tárolóit két feltétel szem előtt tartásával fejlesztették ki: biztosítsák a lehető legnagyobb szabadságot a felhasználónak új, egyéni tárolók fejlesztésében, ugyanakkor nyújtsanak hasonló kezelői felületet. Ez a megközelítés lehetővé teszi, hogy a tárolók hatékonysága a lehető legnagyobb legyen és a felhasználók olyan programot írhassanak, amelyek függetlenek az éppen használt tároló típusától.

A tárolók tervezői általában vagy csak az egyik, vagy csak a másik feltétellel foglalkoznak. A standard könyvtárban szereplő tárolók és algoritmusok bemutatják, hogy lehetséges egyszerre általános és hatékony eszközöket létrehozni. A következőkben két hagyományos tárolótípus erősségeit és gyengeségeit mutatjuk be, így megismerkedhetünk a szabványos tárolók szerkezetével.

16.2.1. Egyedi célú tárolók és bejárók

A legkézenfekvőbb megközelítés egy vektor és egy lista megvalósítására az, hogy mindkettőt olyan formában készítjük el, ami legjobban megfelel a tervezett céloknak:

```
template<class T> class Vector {          // optimális
public:
   explicit Vector(size_t n);             // kezdetben n darab, T() értékű elemet tárol

   T& operator[ ](size_t);                // indexelés
   // ...
};

template<class T> class List {            // optimális
public:
   class Link { /* ... */ };

   List();                                // kezdetben üres
   void put(T*);                          // az aktuális elem elé helyezés
   T* get();                              // az aktuális elem megszerzése

   // ...
};
```

Mindkét osztály olyan műveleteket kínál, amely ideális felhasználásukhoz, és mindkettőben szabadon kiválaszthatjuk a megfelelő ábrázolást, anélkül, hogy más tárolótípusokkal foglalkoznánk. Ez lehetővé teszi, hogy a műveletek megvalósítása közel optimális legyen. Különösen fontos, hogy a leggyakrabban használt műveletek (tehát a *put()* a *List* esetében, illetve az *operator[ ]()* a *Vector* esetében) egészen rövidek és könnyen optimalizálhatóak (pl. helyben (inline) kifejthetők) legyenek.

A tárolók egyik leggyakoribb felhasználási módja, hogy egymás után végiglépkedünk a tárolóban tárolt elemeken. Ezt nevezzük a tároló *bejárásának*, a feladat megvalósításához pedig általában létrehozunk egy *bejáró* (iterator) osztályt, amely megfelel az adott tároló típusának. (§11.5 és §11.14[7])

Bizonyos esetekben, amikor a felhasználó bejár egy tárolót, nem is akarja tudni, hogy az adatokat ténylegesen egy listában vagy egy vektorban tároljuk. Ezekben a helyzetekben a bejárásnak nem szabad attól függnie, hogy a *List* vagy a *Vector* osztályt használtuk. Az ideális az, ha mindkét esetben pontosan ugyanazt a programrészletet használhatjuk.

A megoldást a bejáró (iterator) osztály jelenti, amely biztosít egy „kérem a következőt” műveletet, amely minden tároló számára megvalósítható. Például:

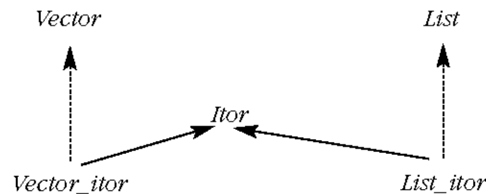
```
template<class T> class Itor {            // közös felület (absztrakt osztály §2.5.4, §12.3)
public:
   // 0 visszaadásával jelezzük, hogy nincs több elem

   virtual T* first() = 0;                // mutató az első elemre
   virtual T* next() = 0;                 // mutató a következő elemre
};
```

Ezt az osztályt később elkészíthetjük külön a *Vector* és külön a *List* osztályhoz:

```
template<class T> class Vector_itor : public Itor<T> {       // vektor-megvalósítás
   Vector<T>& v;
   size_t index;                                              // az aktuális elem indexértéke
public:
   Vector_itor(Vector<T>& vv) : v(vv), index(0) { }
   T* first() { return (v.size()) ? &v[index=0] : 0; }
   T* next() { return (++index<v.size()) ? &v[index] : 0; }
};

template<class T> class List_itor : public Itor<T> {          // lista-megvalósítás
   List<T>& lst;
   List<T>::Link p;                                           // az aktuális elemre mutat
```

```
public:
  List_itor(List<T>&);
  T* first();
  T* next();
};
```

Grafikus formában (szaggatott vonallal jelezve a „megvalósítás a … felhasználásával" kapcsolatot):



A két bejáró (iterator) belső szerkezete jelentősen eltér, de ez a felhasználó számára láthatatlan marad. Ezek után bármit bejárhatunk, amire az *Itor* osztályt meg tudjuk valósítani. Például:

```
int count(Itor<char>& ii, char term)
{
  int c = 0;
  for (char* p = ii.first(); p; p=ii.next()) if (*p==term) c++;
  return c;
}
```

Van azonban egy kis probléma. A bejárón elvégzendő művelet lehet rendkívül egyszerű, mégis mindig végre kell hajtanunk egy (virtuális) függvényhívást. A legtöbb esetben ez a teljesítményromlás nem jelent nagy veszteséget az egyéb tevékenységek mellett. A nagyteljesítményű rendszerekben azonban gyakran éppen egy egyszerű tároló gyors bejárása jelenti a létfontosságú feladatot, és a függvények meghívása sokkal „költségesebb" lehet, mint egy egész számokkal végzett összeadás vagy egy mutatóérték-számítás (ami a *Vector* és a *List* esetében megvalósítja a *next()* függvényt). Ezért a fenti modell nem használható, vagy legalábbis nem tökéletes egy szabványos könyvtár szolgáltatásaként.

Ennek ellenére a tároló-bejáró szemlélet nagyon jól használható sok rendszer esetében. Évekig ez volt programjaim kedvenc megoldása. Az előnyöket és a hátrányokat az alábbiakkal foglalhatjuk össze:

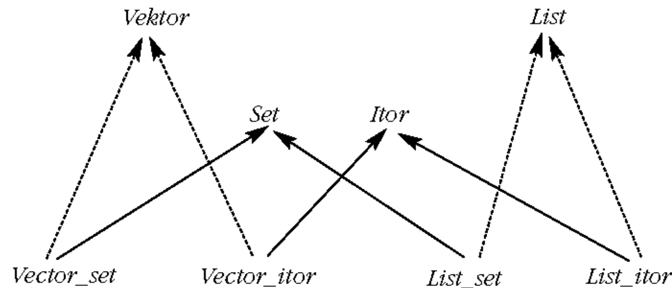
+ Az önálló tárolók (container) egyszerűek és hatékonyak.
+ A tárolóknak nem kell hasonlítaniuk egymásra. A bejáró (iterator) és a beburkoló (wrapper) osztályok (§25.7.1) segítségével a külön fejlesztett tárolókat is egységes formában érhetjük el.
+ A közös felhasználói felületet a bejárók biztosítják (nem egy általános tárolótípus, §16.2.2).
+ Ugyanahhoz a tárolóhoz több bejárót is definiálhatunk a különböző igényeknek megfelelően.
+ A tárolók alapértelmezés szerint típusbiztosak és homogének (azaz a tároló minden eleme ugyanolyan típusú). Heterogén tárolókat úgy hozhatunk létre, hogy olyan mutatókból hozunk létre egy homogén tárolót, amelyek azonos őssel rendelkező objektumokra mutatnak.
+ A tárolók nem tolakodók (non-intrusive), (azaz nem igénylik, hogy a tároló tagjai egy közös őstől származzanak, vagy valamilyen hivatkozó mezővel rendelkezzenek). A nem tolakodó tárolók jól használhatóak a beépített típusok, vagy a mi hatáskörünkön kívül létrehozott adatszerkezetek esetében.
- Minden bejárón keresztüli hozzáférés egy virtuális függvényhívással rontja a rendszer hatékonyságát. Az egyszerű hozzáférési eljárásokhoz képest ez a módszer komoly időveszteséget is jelenthet.
- A bejáró-osztályok hierarchiája könnyen áttekinthetetlenné válhat.
- Semmilyen közös alapot nem találhatunk a különböző tárolók között, még kevésbé a tárolókban tárolt objektumok között. Ez megnehezíti az olyan általános feladatokra való felkészülést, mint a perzisztencia (persistence) biztosítása vagy az objektum be- és kivitel.

A + jellel az előnyöket, - jellel a hátrányokat soroltuk fel.

A bejárók által biztosított rugalmasságra külön felhívnánk a figyelmet. Egy olyan közös felhasználói felület, mint az *Itor*, akkor is megvalósítható, amikor a tároló (esetünkben a *Vector* és a *List*) megtervezése és elkészítése már rég megtörtént. Amikor a tárolót tervezzük, általában konkrét formában gondolkodunk, például egy tömböt vagy egy listát készítünk. Csak később látjuk meg az elvont ábrázolás azon lehetőségeit, amelyek egy adott környezetben egyaránt alkalmazhatóak a tömbökre és a listákra is.

Ezt a „kései elvonatkoztatást” tulajdonképpen többször is elvégezhetjük. Például képzeljük el, hogy egy halmazt szeretnénk létrehozni. A halmaz teljesen más jellegű elvont ábrázolás,

mint az *Itor*, de ugyanazzal a módszerrel, amelyet az *Itor* esetében alkalmaztunk, készíthetünk egy halmaz jellegű felületet is a *Vector* és a *List* osztályhoz:



Ezért a kései elvonatkoztatás az absztrakt osztályok segítségével lehetővé teszi, hogy ugyanazokat a fogalmakat több, különböző formában is ábrázolhassuk, és ez akkor is igaz, ha az egyes megvalósításokban semmilyen hasonlóság sincs. A listák és a vektorok esetében még találhatunk néhány nyilvánvaló hasonlóságot, de az *Itor* felületet akár egy *istream* számára is könnyen elkészíthetnénk.

Elméletileg a listában szereplő utolsó két pont jelenti a szemlélet igazi hátrányát. Ez azt jelenti, hogy ez a megközelítés még akkor sem lehet ideális megoldás egy szabványos könyvtárban, ha a bejárók függvényhívásaiból illetve a hasonló tároló-felületekből eredő teljesítményromlást sikerült is kiküszöbölnünk (amire bizonyos helyzetekben lehetőség van).

A nem tolakodó (non-intrusive) tárolók néhány esetben egy kicsit lassabbak és kicsit több helyet igényelnek, mint a tolakodó tárolók. Én magam ezt nem tartom gondnak; ha mégis az lenne, az *Itor*-hoz hasonló bejárókat tolakodó tárolókhoz is elkészíthetjük (§16.5[11]).

16.2.2. Tárolók közös őssel

Egy tolakodó tároló elkészíthető anélkül is, hogy sablonokat (template) használnánk vagy bármely más módon paramétereket adnánk meg a típushoz:

```
struct Link {
  Link* pre;
  Link* suc;
  // ...
};
```

```
class List {
   Link* head;
   Link* curr;                // az aktuális elem
public:
   Link* get();               // az aktuális elem eltávolítása és visszaadása
   void put(Link*);           // beszúrás az aktuális elem elé
   // ...
};
```

A *List* itt nem más, mint *Link* típusú objektumok listája, azaz olyan objektumok tárolására képes, amelyek a *Link* adatszerkezetből származnak:

```
class Ship : public Link { /* ... */ };

void f(List* lst)
{
   while (Link* po = lst->get()) {
            if (Ship* ps = dynamic_cast<Ship*>(po)) {  // a Ship-nek többalakúnak kell lennie
                                                         // (§15.4.1)
                     // a ship használata
            }
            else {
                     // hoppá, valami mást csinálunk
            }
   }
}
```

A Simula ebben a stílusban határozta meg szabványos tárolóit, tehát ezt a megoldást tekinthetjük az objektumorientált programozást támogató nyelvek eredeti szemléletének. Napjainkban minden objektum közös bázisosztályának neve *Object* vagy valami hasonló. Az *Object* osztály általában számos más szolgáltatást is nyújt azon kívül, hogy összekapcsolja a tárolókat.

Ez a szemlélet gyakran (bár nem szükségszerűen) kiegészül egy általános tárolótípussal is:

```
class Container : public Object {
public:
   virtual Object* get();                  // az aktuális elem eltávolítása és visszaadása
   virtual void put(Object*);              // beszúrás az aktuális elem elé
   virtual Object*& operator[](size_t);    // indexelés
   // ...
};
```

Figyeljük meg, hogy a *Container* osztályban szereplő műveletek virtuálisak, így a különböző tárolók saját igényeiknek megfelelően felülírhatják azokat:

```
class List : public Container {
public:
   Object* get();
   void put(Object*);
   // ...
};

class Vector : public Container {
public:
   Object*& operator[ ](size_t);
   // ...
};
```

Sajnos azonnal jelentkezik egy probléma. Milyen műveleteket kell biztosítania a *Container* osztálynak? Csak azok a függvények szerepelhetnek, amelyeket minden tároló meg tud valósítani, de az összes tároló művelethalmazának metszete nevetségesen szűk felület. Sőt, az az igazság, hogy az igazán érdekes esetekben ez a metszet teljesen üres. Tehát gyakorlatilag a támogatni kívánt tárolótípusokban szereplő, lényeges műveletek unióját kell szerepeltetnünk. A szolgáltatásokhoz biztosított felületek ilyen unióját ***kövér*** vagy ***bőséges felületnek*** nevezzük. (fat interface, §24.4.3)

Vagy e felület leírásában készítünk valamilyen alapértelmezett változatot a függvényekhez, vagy azokat tisztán virtuálisakká (pure virtual) téve arra kötelezzük a származtatott osztályokat, hogy ők fejtsék ki a függvényeket. Mindkét esetben sok-sok olyan függvényt kapunk, amelyek egyszerűen futási idejű hibát váltanak ki:

```
class Container : public Object {
public:
   struct Bad_op {     // kivételosztály
            const char* p;
            Bad_op(const char* pp) :p(pp) { }
   };

   virtual void put(Object*) { throw Bad_op("put-hiba"); }
   virtual Object* get() { throw Bad_op("get-hiba"); }
   virtual Object*& operator[ ](int) { throw Bad_op("[ ]"); }
   // ...
};
```

Ha védekezni szeretnénk azon lehetőséggel szemben, hogy egy tároló nem támogatja a *get()* függvény használatát, el kell kapnunk valahol a *Container::Bad_op* kivételt. Ezek után a korábbi *Ship* példát a következő formában valósíthatjuk meg:

```
class Ship : public Object { /* ... */ };

void f1(Container* pc)
{
  try {
        while (Object* po = pc->get()) {
              if (Ship* ps = dynamic_cast<Ship*>(po)) {
                    // a ship használata
              }
              else {
                    // hoppá, valami mást csinálunk
              }
        }
  }
  catch (Container::Bad_op& bad) {
                    // hoppá, valami mást csinálunk
  }
}
```

Ez így túlságosan bonyolult, ezért a *Bad_op* kivétel ellenőrzését érdemes máshova helyeznünk. Ha számíthatunk rá, hogy a kivételeket máshol kezelik, akkor a példát az alábbi formára rövidíthetjük:

```
void f2(Container* pc)
{
  while (Object* po = pc->get()) {
        Ship& s = dynamic_cast<Ship&>(*po);
        // a Ship használata
  }
}
```

Ennek ellenére az az érzésünk, hogy a futási idejű típusellenőrzés stílustalan és nem is túl hatékony. Ezért inkább a statikus típusellenőrzés mellett maradunk:

```
void f3(Itor<Ship>* i)
{
  while (Ship* ps = i->next()) {
        // a Ship használata
  }
}
```

A tárolótervezés „objektumok közös bázisosztállyal" megközelítésének előnyeit és hátrányait az alábbiakkal foglalhatjuk össze (nézzük meg a §16.5[10] feladatot is):

- \- A különböző tárolókon végzett műveletek virtuális függvényhívást eredményeznek.
- \- Minden tárolót a *Container* osztályból kell származtatnunk. Ennek következtében kövér felületet kell készítenünk, és nagy mértékű előrelátásra, illetve futási idejű típusellenőrzésre van szükség. Egy külön fejlesztett tárolót egy általános keretbe szorítani a legjobb esetben is körülményes (§16.5[12]).
- \+ A közös *Container* bázisosztály egyszerű megoldást kínál olyan tárolók fejlesztéséhez, amelyek hasonló műveleteket biztosítanak.
- \- A tárolók heterogének és alapértelmezés szerint nem típusbiztosak. (Mindössze arra számíthatunk, hogy az elemek típusa *Object* *.) Ha szükség van rá, sablonok (template) segítségével hozhatunk létre típusbiztos és homogén tárolókat.
- \- A tárolók tolakodóak (ami esetünkben azt jelenti, hogy minden elemnek az *Object* osztályból kell származnia). Beépített típusokba tartozó objektumokat, illetve a mi hatáskörünkön kívül meghatározott adatszerkezeteket közvetlenül nem helyezhetünk el bennük.
- \- A tárolóból kiemelt elemekre megfelelő típuskonverziót kell alkalmaznunk, mielőtt használhatnánk azokat.
- \+ A *Container* és az *Object* osztály olyan szolgáltatások kialakításához használható, amelyek minden tárolóra vagy minden objektumra alkalmazhatók. Ez nagymértékben leegyszerűsíti az olyan általános szolgáltatások megvalósítását, mint a perzisztencia biztosítása vagy az objektumok ki- és bevitele.

Ugyanúgy, mint korábban (§16.2.1), a + az előnyöket, a - a hátrányokat jelöli.

Az egymástól független tárolókhoz és bejárókhoz viszonyítva a közös bázisosztállyal rendelkező objektumok ötlete feleslegesen sok munkát hárít a felhasználóra, jelentős futási idejű terhelést jelent és korlátozza a tárolóban elhelyezhető objektumok körét. Ráadásul sok osztály esetében az *Object* osztályból való származtatás a megvalósítás részleteinek felfedését jelenti, ezért ez a megközelítés nagyon távol áll az ideális megoldástól egy szabványos könyvtár esetében.

Ennek ellenére az ezen megoldás által kínált általánosságot és rugalmasságot nem szabad alábecsülnünk. Számtalan változatát, számtalan programban sikeresen felhasználták már. Ennek a megközelítésnek azokon a területeken van jelentősége, ahol a hatékonyság kevésbé fontos, mint az egyszerűség, amelyet az egyszerű *Container* felület és az objektum ki- és bevitelhez hasonló szolgáltatások biztosítanak.

16.2.3. A standard könyvtár tárolói

A standard könyvtár tárolói (container) és bejárói (iterator) – amelyet gyakran nevezünk STL (Standard Template Library) keretrendszernek, (§3.10) – olyan megközelítésként foghatók fel, amely a korábban bemutatott két hagyományos modell előnyeit a lehető legjobban kiaknázza. Az STL azonban egyik módszert sem alkalmazza közvetlenül; célja az, hogy egyszerre hatékony és általános algoritmusokat kínáljon, mindenféle megalkuvás nélkül. A hatékonyság érdekében az alkotók a gyakran használt adatelérő függvények esetében elvetették a nehezen optimalizálható virtuális függvényeket, így nem adhattak szabványos felületet a tárolók és a bejárók számára absztrakt osztály formájában. Ehelyett mindegyik tárolótípus támogatja az alapvető műveleteknek egy szabványos halmazát. A kövér felületek problémájának (§16.2.2, §24.4.3) elkerülése érdekében az olyan műveletek, amelyek nem minden tárolóban valósíthatók meg hatékonyan, nem szerepelnek ebben a közös halmazban. Az indexelés például alkalmazható a *vector* esetében, de nem használható a *list* tárolókra. Ezenkívül minden tárolótípus saját bejárókat biztosít, amelyek a szokásos bejáróműveleteket teszik elérhetővé.

A szabványos tárolók nem közös bázisosztályból származnak, hanem minden tároló önállóan tartalmazza a szabványos tárolófelületet. Ugyanígy nincs közös bejáró-ős sem. A szabványos tárolók és bejárók használatakor nem történik futási idejű típusellenőrzés, sem közvetlen (explicit), sem automatikus (implicit) formában.

A minden tárolóra kiterjedő közös szolgáltatások biztosítása igen fontos és bonyolult feladat. Az STL ezt a *memóriafoglalók* (allokátor, allocator) segítségével valósítja meg, amelyeket sablonparaméterként adunk meg (§19.4.3), ezért nincs szükség közös bázisosztályra.

Mielőtt a részleteket megvizsgálnánk és konkrét példákat mutatnánk be, foglaljuk össze az STL szemlélete által biztosított előnyöket és hátrányokat:

+ Az önálló tárolók egyszerűek és hatékonyak (nem egészen olyan egyszerűek, mint a tényleg teljesen független tárolók, de ugyanolyan hatékonyak).
+ Mindegyik tároló biztosítja a szabványos műveleteket szabványos néven és jelentéssel. Ha szükség van rá, az adott tárolótípusnak megfelelő további műveletek is megtalálhatók. Ezenkívül, a becsomagoló vagy beburkoló (wrapper) osztályok (§25.7.1) segítségével a külön fejlesztett tárolók is beilleszthetők a közös keretrendszerbe. (§16.5[14])
+ A további közös használati formákat a szabványos bejárók biztosítják. Minden tároló biztosít bejárókat, amelyek lehetővé teszik bizonyos műveletek végrehajtását szabványos néven és jelentéssel. Minden bejáró-típus külön létezik minden tárolótípushoz, így ezek a bejárók a lehető legegyszerűbbek és a lehető leghatékonyabbak.

+ A különböző szükségletek kielégítésére minden tárolóhoz létrehozhatunk saját bejárókat vagy általános felületeket, a szabványos bejárók mellett.
+ A tárolók alapértelmezés szerint típusbiztosak és homogének (azaz az adott tároló minden eleme ugyanolyan típusú). Heterogén tárolókat úgy készíthetünk, hogy egy olyan homogén tárolót hozunk létre, amely közös bázisosztályra hivatkozó mutatókat tartalmaz.
+ A tárolók nem tolakodóak (azaz a tároló tagjainak nem kell egy adott bázisosztályból származniuk, vagy hivatkozó mezőket tartalmazniuk). A nem tolakodó tárolók a beépített típusok esetében használhatók jól, illetve az olyan adatszerkezetekhez, melyeket a mi hatáskörünkön kívül határoztak meg.
+ Az általános keretrendszer lehetővé teszi tolakodó tárolók használatát is. Természetesen ez az elemek típusára nézve komoly megkötéseket jelenthet.
+ Minden tároló használ egy paramétert, az úgynevezett *memóriafoglalót* (allocator), amely olyan szolgáltatások kezeléséhez nyújt segítséget, amelyek minden tárolóban megjelennek. Ez nagymértékben megkönnyíti az olyan általános feladatok megvalósítását, mint a perzisztencia biztosítása vagy az objektum ki- és bevitel.(§19.4.3)
- A tárolók és bejárók nem rendelkeznek olyan szabványos, futási idejű ábrázolással, amelyet például függvényparaméterként átadhatnánk (bár a szabványos tárolók és bejárók esetében könnyen létrehozhatnánk ilyet, ha erre az adott programban szükségünk van, §19.3).

Ugyanúgy, mint eddig (§16.2.1) a + az előnyöket, a - a hátrányokat jelöli.

A tárolóknak (container) és a bejáróknak (iterator) tehát nincs rögzített, általános ábrázolása. Ehelyett minden tároló ugyanazt a szabványos felületet biztosítja szabványos műveletek formájában. Ennek következtében a tárolókat egyformán kezelhetjük és fel is cserélhetjük. A bejárókat is hasonlóan használhatjuk. Ez az idő- és tárhasználati hatékonyságot csak kevéssé rontja, a felhasználó viszont kihasználhatja az egységességet, mind a tárolók szintjén (a közös bázisosztályból származó tárolók esetében), mind a bejárók szintjén (a specializált tárolók esetében).

Az STL megközelítése erősen épít a sablonok (template) használatára. Ahhoz, hogy elkerüljük a felesleges kódismétléseket, gyakran van szükség arra, hogy mutatókat tartalmazó tárolókban részlegesen specializált változatok készítésével közösen használható összetevőket hozunk létre (§13.5).

16.3. A vektor

Itt a *vector*-t úgy írjuk le, mint a teljes szabványos tárolók egyik példáját. Ha mást nem mondunk, a *vector*-ra vonatkozó állítások változtatás nélkül alkalmazhatók az összes többi szabványos tárolóra is. A 17. fejezet foglalkozik azokkal a lehetőségekkel, amelyek kizárólag a *list*, a *set*, a *map* vagy valamelyik másik tárolóra vonatkoznak. Azokat a lehetőségeket, amelyeket kifejezetten a *vector* – vagy egy hasonló tároló – valósít meg, csak bizonyos mértékig részletezzük. Célunk az, hogy megismerjük a *vector* lehetséges felhasználási területeit és megértsük a standard könyvtár globális szerkezetében elfoglalt szerepét.

A §17.1 pontban áttekintjük a standard tárolók tulajdonságait és az általuk kínált lehetőségeket. Az alábbiakban a *vector* tárolót különböző szempontok szerint mutatjuk be: a tagtípusok, a bejárók, az elemek elérése, a konstruktorok, a veremműveletek, a listaműveletek, a méret és kapacitás, a segédfüggvények, illetve a *vector<bool>* szempontjából.

16.3.1. Típusok

A szabványos *vector* egy sablon (template), amely az *std* névtérhez tartozik és a *<vector>* fejállományban található. Először is néhány szabványos típusnevet definiál:

```
template <class T, class A = allocator<T> > class std::vector {
public:
  // típusok

  typedef T value_type;                                    // elemtípus
  typedef A allocator_type;                                // memóriakezelő-típus
  typedef typename A::size_type size_type;
  typedef typename A::difference_type difference_type;

  typedef megvalósítás_függő1 iterator;                     // T*
  typedef megvalósítás_függő2 const_iterator;               // const T*
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

  typedef typename A::pointer pointer;                     // elemmutató
  typedef typename A::const_pointer const_pointer;
  typedef typename A::reference reference;                 // hivatkozás elemre
  typedef typename A::const_reference const_reference;

  // ...
};
```

Minden szabványos tároló definiálja ezeket a típusokat, mint saját tagjait, de a saját megvalósításának legmegfelelőbb formában.

A tároló elemeinek típusát az első sablonparaméter határozza meg, amit gyakran nevezünk értéktípusnak (*value_type*) is. A memóriafoglaló típusa (*allocator_type*) – amelyet (nem kötelezően) a sablon második paraméterében adhatunk meg – azt határozza meg, hogy a *value_type* hogyan tart kapcsolatot a különböző memóriakezelő eljárásokkal. Ezen belül, a memóriafoglaló adja meg azokat a függvényeket, amelyeket a tároló az elemek tárolására szolgáló memória lefoglalására és felszabadítására használ. A memóriafoglalókról a §19.4 pontban lesz szó részletesen. Általában a *size_type* határozza meg azt a típust, amelyet a tároló indexeléséhez használunk, míg a *difference_type* annak az értéknek a típusát jelöli, amelyet két bejáró különbségének képzésekor kapunk. A legtöbb tároló esetében ezek a *size_t*, illetve a *ptrdiff_t* típust jelentik (§6.2.1)

A bejárókat a §2.7.2 pontban mutattuk be és a 19. fejezetben foglalkozunk velük részletesen. Úgy képzelhetjük el őket, mint egy tároló valamelyik elemére hivatkozó mutatót. Minden tároló leír egy *iterator* nevű típust, amellyel az elemekre mutathatunk. Rendelkezésünkre áll egy *const_iterator* típus is, amelyet akkor használunk, ha nincs szükség az elemek módosítására. Ugyanúgy, mint a mutatók esetében, itt is mindig használjuk a biztonságosabb *const* változatot, hacsak nem feltétlenül a másik lehetőségre van szükségünk. A *vector* bejáróinak konkrét típusa a megvalósítástól függ, a legnyilvánvalóbb megoldás egy hagyományosan definiált *vector* esetében a *T**, illetve a *const T**.

A visszafelé haladó bejárók típusát a *vector* számára a szabványos *reverse_iterator* sablon segítségével határozták meg. (§19.2.5) Ez az elemek sorozatát fordított sorrendben szolgáltatja.

A §3.8.1 pontban bemutattuk, hogy ezen típusok segítségével a felhasználó úgy írhat tárolókat használó programrészleteket, hogy a ténylegesen használt típusokról semmit sem tud, sőt, olyan kódot is írhat, amely minden szabványos tároló esetében használható:

```
template<class C> typename C::value_type sum(const C& c)
{
  typename C::value_type s = 0;
  typename C::const_iterator p = c.begin();     // kezdés az elején
  while (p!=c.end()) {                          // folytatás a végéig
        s += *p;                                // elem értékének megszerzése
        ++p;                                    // p a következő elemre fog mutatni
  }
  return s;
}
```

A *typename* használata egy sablonparaméter tagjának neve előtt elég furcsán néz ki, de a fordítóprogram nem akad fenn rajta, ugyanis nincs általános módszer arra, hogy egy sablonparaméter valamelyik tagjáról eldöntsük, hogy az típusnév-e. (§C.13.5)

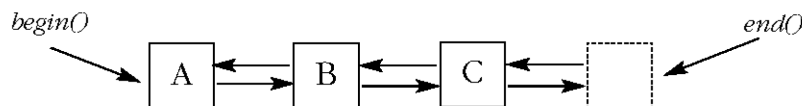
Ugyanúgy, mint a mutatók esetében, az előtagként használt * a bejáró indirekcióját jelenti (§2.7.2, §19.2.1), míg a ++ a bejáró növelését végzi.

16.3.2. Bejárók

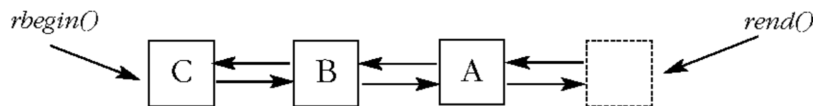
Ahogy az előző alfejezetben már bemutattuk, a programozók a bejárókat arra használhatják, hogy bejárják a tárolót az elemek típusának pontos ismerete nélkül. Néhány tagfüggvény teszi lehetővé, hogy elérjük az elemek sorozatának valamelyik végét:

```
template <class T, class A = allocator<T> > class vector {
public:
  // ...
  // iterators:

  iterator begin();                              // az első elemre mutat
  const_iterator begin() const;
  iterator end();                                // az "utolsó utáni" elemre mutat
  const_iterator end() const;

  reverse_iterator rbegin();                     // hátulról az első elemre mutat
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();                       // hátulról az "utolsó utáni" elemre mutat
  const_reverse_iterator rend() const;

  // ...
};
```

A *begin()/end()* pár a tároló elemeit a szokásos sorrendben adja meg, azaz elsőként a „nulladik” elemet (vagyis az elsőt) kapjuk, majd sorban a következőket; az *rbegin()/rend()* párnál viszont fordított sorrendben, azaz az *n-1.* elem után következik az *n-2.*, majd az *n-3.*, és így tovább. Például ha egy *iterator* használatával ilyen sorozatot kapunk:



akkor a *reverse_iterator* a következő elemsorozatot adja (§19.2.5):



Így lehetőségünk van olyan algoritmusok készítésére, amelyeknek fordított sorrendben van szüksége az elemek sorozatára. Például:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
   typename C::reverse_iterator ri = find(c.rbegin(),c.rend(),v);
   if (ri == c.rend()) return c.end();              // a c.end() jelzi, hogy "nem található"
   typename C::iterator i = ri.base();
   return --i;
}
```

Egy *reverse_iterator* esetében az *ri.base()* függvény egy olyan bejárót ad vissza, amely az *ri* által kijelölt hely után következő elemre mutat. (§19.2.5) A visszafelé haladó bejárók nélkül egy ciklust kellett volna készítenünk:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
   typename C::iterator p = c.end();       // keresés a végétől visszafelé
   while (p!=c.begin())
            if (*--p==v) return p;
   return c.end();                         // a c.end() jelzi, hogy "nem található"
}
```

A visszafelé haladó bejáró is közönséges bejáró, tehát írhattuk volna ezt is:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
   typename C::reverse_iterator p = c.rbegin();    // a sorozat átnézése visszafelé
   while (p!=c.rend()) {
            if (*p==v) {
                     typename C::iterator i = p.base();
                     return --i;
            }
            ++p;                                   // vigyázzunk: növelés, nem csökkentés (--)
   }
   return c.end();                                 // a c.end() jelzi, hogy "nem található"
}
```

Figyeljünk rá, hogy a *C::reverse_iterator* típus nem ugyanaz, mint a *C::iterator*.

16.3.3. Az elemek elérése

A *vector* igen fontos tulajdonsága a többi tárolóval összehasonlítva, hogy az egyes elemeket bármilyen sorrendben könnyen és hatékonyan elérhetjük:

```
template <class T, class A = allocator<T> > class vector {
public:
  // ...
  // hozzáférés az elemekhez

  reference operator[ ](size_type n);              // nem ellenőrzött hozzáférés
  const_reference operator[ ](size_type n) const;

  reference at(size_type n);                       // ellenőrzött hozzáférés
  const_reference at(size_type n) const;

  reference front();                               // első elem
  const_reference front() const;
  reference back();                                // utolsó elem
  const_reference back() const;

  // ...
};
```

Az indexeléshez az *operator[ ]()* vagy az *at()* függvényt használjuk. A *[ ]* operátor ellenőrizetlen hozzáférést biztosít, míg az *at()* indexhatár-ellenőrzést is végez és *out_of_range* kivételt vált ki, ha a megadott index nem a megfelelő tartományba esik:

```
void f(vector<int>& v, int i1, int i2)
try {
  for(int i = 0; i < v.size(); i++) {
           // a tartomány már ellenőrzött, itt a nem ellenőrzött v[i]-t kell használnunk
  }

  v.at(i1) = v.at(i2);          // hozzáféréskor a tartomány ellenőrzése

  // ...
}
catch(out_of_range) {
  // hoppá, kicsúsztunk a tartományon kívülre
}
```

Ez a példa bemutat egy hasznos ötletet is: ha az indexhatárokat már valamilyen módon ellenőriztük, akkor az ellenőrizetlen indexelő operátort teljes biztonsággal használhatjuk. Ellenkező esetben viszont érdemes az *at()* függvényt alkalmaznunk. Ez a különbség fontos lehet olyan programoknál, ahol a hatékonyságra is figyelnünk kell. Ha a hatékonyság nem jelentős szempont vagy nem tudjuk egyértelműen eldönteni, hogy a tartomány ellenőrzött-e, akkor biztonságosabb ellenőrzött *[ ]* operátorral rendelkező vektort használjunk (például a *Vec*-et, §3.7.2), vagy legalábbis ellenőrzött bejárót (§19.3).

A tömbök alapértelmezett hozzáférési módja ellenőrizetlen. Biztonságos (ellenőrzött) szolgáltatásokat megvalósíthatunk egy gyors alaprendszer felett, de gyors szolgáltatást lassú alaprendszer felett nem.

A hozzáférési műveletek *reference* vagy *const_reference* típusú értéket adnak vissza, attól függően, hogy konstans objektumra alkalmaztuk-e azokat. Az elemek elérésére a referencia megfelelő típus. A *vector<X>* legegyszerűbb és legkézenfekvőbb megvalósításában a *reference* egyszerűen *X&*, míg a *const_reference* megfelelője a *const X&*. Ha egy indexhatáron kívüli elemre próbálunk hivatkozni, az eredmény meghatározhatatlan lesz:

```
void f(vector<double>& v)
{
  double d = v[v.size()];      // nem meghatározható: rossz indexérték

  list<char> lst;
  char c = lst.front();        // nem meghatározható: a lista üres
}
```

A szabványos sorozatok közül csak a *vector* és a *deque* (§17.2.3) támogatja az indexelést. Ennek oka az, hogy a rendszer tervezői nem akarták a felhasználókat alapvetően rossz hatásfokú műveletek bevezetésével megzavarni. Az indexelés például nem alkalmazható a *list* típusra (§17.2.2), mert veszélyesen rossz hatásfokú eljárás lenne (konkrétabban: $O(n)$).

A *front()* és *back()* tagfüggvények sorrendben az első, illetve az utolsó elemre hivatkozó referenciákat adnak vissza. Akkor igazán hasznosak, ha biztosan tudjuk, hogy léteznek ezek az elemek és programunkban valamiért különösen fontosak. Ennek gyakori esete, amikor egy vektort veremként (*stack*, §16.3.5) akarunk használni. Jegyezzük meg, hogy a *front()* arra az elemre ad hivatkozást, amelyre a *begin()* egy bejárót. A *front()* úgy is elképzelhető, mint maga az első elem, míg a *begin()* inkább az első elemre hivatkozó mutató. A *back()* és az *end()* közötti kapcsolat egy kicsit bonyolultabb: a *back()* az utolsó elem, míg az *end()* egy mutató az „utolsó utáni" pozícióra.

16.3.4. Konstruktorok

Természetesen a *vector* a konstruktoroknak, destruktoroknak és másoló műveleteknek is teljes tárházát kínálja:

```
template <class T, class A = allocator<T> > class vector {
public:
  // ...
  // konstruktorok, stb.

  explicit vector(const A& = A());
  explicit vector(size_type n, const T& val = T(), const A& = A());  // n darab val
  template <class In>           // az In-nek bemeneti bejárónak kell lennie (§19.2.1)
              vector(In first, In last, const A& = A());          // másolás [first:last[-ből
  vector(const vector& x);

  ~vector();

  vector& operator=(const vector& x);

  template <class In>           // az In-nek bemeneti bejárónak kell lennie (§19.2.1)
              void assign(In first, In last);            // másolás [first:last[-ből
  void assign(size_type n, const T& val);          // n darab val

  // ...
};
```

A *vector* gyors hozzáférést tesz lehetővé tetszőleges eleméhez, de méretének módosítása viszonylag bonyolult. Ezért általában a *vector* létrehozásakor megadjuk annak méretét is:

```
vector<Record> vr(10000);

void f(int s1, int s2)
{
  vector<int> vi(s1);

  vector<double>* p = new vector<double>(s2);
}
```

Az ilyen módon létrehozott vektorelemeknek az elemek típusának megfelelő alapértelmezett konstruktorok adnak kezdőértéket, tehát a *vr* vektornak mind a 10 000 elemére lefut a *Record()* függvény, a *vi si* darab elemének pedig egy-egy *int()* hívás ad kezdőértéket. Gondoljunk rá, hogy a beépített típusok alapértelmezett konstruktora az adott típusnak megfelelő *0* értéket adja kezdőértékül. (§4.9.5, §10.4.2)

Ha egy típus nem rendelkezik alapértelmezett konstruktorral, akkor belőle nem hozhatunk létre vektort, hacsak nem adjuk meg az összes elem kezdőértékét. Például:

```
class Num {              // végtelen pontosság
public:
  Num(long);
  // nincs alapértelmezett konstruktor
  // ...
};

vector<Num> v1(1000);              // hiba: nincs alapértelmezett Num
vector<Num> v2(1000,Num(0));       // rendben
```

Mivel egy *vector* nem tárolhat negatív számú elemet, így méretének nemnegatív számnak kell lennie. Ez azon követelményben jut kifejezésre, hogy a vektor *size_type* típusának *unsigned*-nak kell lennie. Ez bizonyos rendszerekben nagyobb vektorméret használatát teszi lehetővé, egyes esetekben viszont meglepetésekhez is vezethet:

```
void f(int i)
{
  vector<char> vc0(-1);        // a fordító erre könnyen figyelmeztethet
  vector<char> vc1(i);
}

void g()
{
  f(-1);                 // becsapjuk f()-et, hogy elfogadjon egy igen nagy pozitív számot
}
```

Az *f(-1)* hívásban a *-1* értéket egy igen nagy pozitív számra alakítja a rendszer (§C.6.3). Ha szerencsénk van, fordítónk figyelmeztet erre.

A *vector* méretét közvetve is megadhatjuk, úgy, hogy felsoroljuk a kezdeti elemhalmazt. Ezt úgy valósíthatjuk meg, hogy a konstruktornak azon értékek sorozatát adjuk át, amelyekből a vektort fel kell építeni:

```
void f(const list<X>& lst)
{
  vector<X> v1(lst.begin(),lst.end());    // elemek másolása listából

  char p[] = "despair";
  vector<char> v2(p,&p[sizeof(p)-1]);    // karakterek másolása C stílusú karakterláncból
}
```

Minden esetben a *vector* konstruktora számítja ki a vektor méretét, miközben a bemeneti sorozatból átmásolja az elemeket.

A *vector* azon konstruktorait, melyek egyetlen paramétert igényelnek, az *explicit* kulcsszóval deklaráljuk, így véletlen konverziók nem fordulhatnak elő (§11.7.1):

```
vector<int> v1(10);                 // rendben: 10 egészből álló vektor
vector<int> v2 = vector<int>(10);   // rendben: 10 egészből álló vektor
vector<int> v3 = v2;                // rendben: v3 v2 másolata
vector<int> v4 = 10;        // hiba: automatikus konvertálás kísérlete 10-ről vector<int>-re
```

A másoló konstruktor és a másoló értékadás a *vector* elemeit másolják le. Egy sok elemből álló vektor esetében ez hosszadalmas művelet lehet, ezért a vektorokat általában referenciaként adjuk át:

```
void f1(vector<int>&);              // szokásos stílus
void f2(const vector<int>&);        // szokásos stílus
void f3(vector<int>);               // szokatlan stílus

void h()
{
  vector<int> v(10000);

  // ...

  f1(v);    // hivatkozás átadása
  f2(v);    // hivatkozás átadása
  f3(v);    // a 10 000 elem új vektorba másolása az f3() számára
}
```

Az *assign* függvények a többparaméteres konstruktorok párjainak tekinthetők. Azért van rájuk szükség, mert az = egyetlen, jobb oldali operandust vár, így ha alapértelmezett paraméterértéket akarunk használni vagy értékek teljes sorozatát akarjuk átadni, az *assign* függvényre van szükségünk:

```
class Book {
  // ...
};

void f(vector<Num>& vn, vector<char>& vc, vector<Book>& vb, list<Book>& lb)
{
  vn.assign(10,Num(0));     // Num(0) tíz példányát tároló vektor értékül adása vn-nek
```

```
    char s[] = "literál";
    vc.assign(s,&s[sizeof(s)-1]);          // a "literál" értékül adása vc-nek

    vb.assign(lb.begin(),lb.end());        // listaelemek értékül adása

    // ...
}
```

Ezek után egy *vector* objektumot feltölthetünk tetszőleges elemsorozattal, ami típus szerint megfelelő, és később is hozzárendelhetünk az objektumhoz ilyen sorozatokat. Fontos, hogy ezt anélkül értük el, hogy nagyszámú konstruktort illetve konverziós operátort kellett volna definiálnunk. Figyeljük meg, hogy az értékadás teljes egészében lecseréli a vektor elemeit. Elméletileg az összes régi elemet töröljük, majd az új elemeket beillesztjük. Az értékadás után a *vector* mérete az értékül adott elemek számával egyezik meg:

```
void f()
{
    vector<char> v(10,'x');           // v.size()==10, minden elem értéke 'x'
    v.assign(5,'a');                  // v.size()==5, minden elem értéke 'a'
    // ...
}
```

Természetesen, amit az *assign()* csinál, az megvalósítható közvetetten úgy is, hogy először létrehozzuk a kívánt vektort, majd ezt adjuk értékül:

```
void f2(vector<Book>& vh, list<Book>& lb)
{
    vector<Book> vt(lb.begin(),lb.end());
    vh = vt;
    // ...
}
```

Ez a megoldás azonban nem csak csúnya, de rossz hatásfokú is lehet.

Ha egy vektor konstruktorában két ugyanolyan típusú paramétert használunk, akkor azt kétféleképpen is értelmezhetnénk:

```
vector<int> v(10,50);  // vector(méret,érték) vagy vector(bejáró1,bejáró2)?
                       // vector(méret,érték)!
```

Az *int* nem bejáró, így a megvalósításoknak biztosítaniuk kell, hogy a megfelelő konstruktor fusson le:

```
vector(vector<int>::size_type, const int&, const vector<int>::allocator_type&);
```

A másik lehetséges konstruktor a következő:

```
vector(vector<int>::iterator, vector<int>::iterator, const vector<int>::allocator_type&);
```

A könyvtár ezt a problémát úgy oldja meg, hogy megfelelő módon túlterheli a konstruktorokat, de hasonlóan kezeli az *assign()* és az *insert()* (§16.3.6) esetében előforduló többértelműséget is.

16.3.5. Veremműveletek

A vektort általában úgy képzeljük el, mint egy egységes adatszerkezetet, melyben az elemeket indexeléssel érhetjük el. Ezt a szemléletet azonban el is felejthetjük és a vektort a legelvontabb (legáltalánosabb) sorozat megtestesítőjének tekinthetjük. Ha erre gondolunk és megfigyeljük, hogy a tömböknek, vektoroknak milyen általános felhasználási területei vannak, nyilvánvalóvá válik, hogy a *vector* osztályban a veremműveletekre is szükség lehet:

```
template <class T, class A = allocator<T> > class vector {
public:
  // ...
  // veremműveletek

  void push_back(const T& x);          // hozzáadás a végéhez
  void pop_back();                     // az utolsó elem eltávolítása
  // ...
};
```

Ezek a függvények a vektort veremnek tekintik és annak végén végeznek műveleteket:

```
void f(vector<char>& s)
{
  s.push_back('a');
  s.push_back('b');
  s.push_back('c');
  s.pop_back();
  if (s[s.size()-1] != 'b') error("Lehetetlen!");
  s.pop_back();
  if (s.back() != 'a') error("Ennek soha nem szabad megtörténnie!");
}
```

Amikor meghívjuk a *push_back()* függvényt, az *s* vektor mérete mindig nő egy elemmel és a paraméterként megadott elem a vektor végére kerül. Így az *s[s.size()-1]* elem (ami ugyanaz, mint az *s.back()* által visszaadott érték, §16.3.3) a verembe legutoljára helyezett elem lesz.

Attól eltekintve, hogy a *vector* szót használjuk a *stack* helyett, semmi szokatlant nem művelünk. A *_back* utótag azt hangsúlyozza ki, hogy az elemet a vektor végére helyezzük, nem pedig az elejére. Egy új elem elhelyezése a vektor végén nagyon költséges művelet is lehet, hiszen annak tárolásához további memóriát kell lefoglalnunk. Az adott nyelvi változatnak azonban biztosítania kell, hogy az ismétlődő veremműveletek csak ritkán okozzanak méretnövekedésből eredő teljesítménycsökkenést.

Figyeljük meg, hogy a *pop_back()* függvény sem ad vissza értéket. Egyszerűen törli a legutolsó elemet és ha azt szeretnénk megtudni, hogy a kiemelés (pop) előtt milyen érték szerepelt a verem tetején, akkor azt külön meg kell néznünk. Ezt a típusú vermet sokan nem kedvelik (§2.5.3, §2.5.4), de ez a megoldás hatékonyabb lehet és ez tekinthető a szabványnak is.

Miért lehet szükség veremszerű műveletekre egy vektor esetében? Egy nyilvánvaló indok, hogy a verem megvalósítására ez az egyik lehetőség (§17.3.1), de ennél gyakoribb, hogy a tömböt folyamatosan növekedve akarjuk létrehozni. Elképzelhető például, hogy pontokat akarunk beolvasni egy tömbbe, de nem tudjuk, összesen hány pontot kapunk. Ez esetben arra nincs lehetőségünk, hogy már kezdetben a megfelelő méretű tömböt hozzuk létre és utána csak egyszerűen beleírjuk a pontokat. Ehelyett a következő eljárást kell végrehajtanunk:

```
vector<Point> cities;

void add_points(Point sentinel)
{
  Point buf;

  while (cin >> buf) {
            if (buf == sentinel) return;
            // új pont ellenőrzése
            cities.push_back(buf);
  }
}
```

Ez a megoldás biztosítja, hogy a *vector* igény szerint növekedhessen. Ha mindössze annyi a teendőnk, hogy a pontokat elhelyezzük a vektorban, akkor a *cities* adatszerkezetet feltölthetjük közvetlenül egy konstruktor segítségével is (§16.3.4), általában azonban valamilyen módon még fel kell dolgoznunk a beérkező adatokat és csak fokozatosan tölthetjük fel a vektort a program előrehaladtával. Ilyenkor használhatjuk a *push_back()* függvényt.

A C programokban ez a leggyakoribb felhasználási területe a *realloc()* függvénynek, amely a C standard könyvtárában szerepel. Ezért a vektor – és általában minden szabványos tároló – biztosít egy általánosabb, elegánsabb, de nem kevésbé hatékony változatot a *realloc()* helyett.

A vektor méretét (*size()*) a *push_back()* automatikusan növeli, így nem fordulhat elő, hogy a vektor túlcsordul (mindaddig, amíg memória igényelhető a rendszertől, §19.4.1). Alulcsordulás ellenben előfordulhat:

```
void f()
{
  vector<int> v;
  v.pop_back();              // nem meghatározható hatás: v állapota nem
                             // meghatározható lesz
  v.push_back(7);            // nem meghatározható hatás (v állapota nem
                             // meghatározható), valószínűleg rossz
}
```

Az alulcsordulás következménye nem meghatározható és a legtöbb C++-változat esetében olyan memóriaterület felülírását eredményezi, amely nem tartozik a vektorhoz. Az alulcsordulást ugyanúgy el kell kerülnünk, mint a túlcsordulást.

16.3.6. Listaműveletek

A *push_back()*, a *pop_back()* és a *back()* művelet (§16.3.5) lehetővé teszi, hogy a *vector* osztályt veremként használjuk. Bizonyos helyzetekben azonban arra is szükségünk lehet, hogy a verem közepére illesszünk be, vagy onnan távolítsunk el egy elemet:

```
template <class T, class A = allocator<T> > class vector {
public:
  // ...
  // listaműveletek

  iterator insert(iterator pos, const T& x);              // x felvétele pos elé
  void insert(iterator pos, size_type n, const T& x);     // n darab x felvétele pos elé
  template <class In>        // az In-nek bemeneti bejárónak kell lennie (§19.2.1)
          void insert(iterator pos, In first, In last);   // elemek beillesztése sorozatból

  iterator erase(iterator pos); // a pos pozícióban levő elem eltávolítása
  iterator erase(iterator first, iterator last);          // a sorozat törlése
  void clear();                                           // minden elem törlése
  // ...
};
```

Ezen műveletek működésének bemutatásához egy gyümölcsök (fruit) neveit tartalmazó vektort hozunk létre. Először is meghatározzuk a vektort, majd feltöltjük néhány névvel:

```
vector<string> fruit;

fruit.push_back("őszibarack");
fruit.push_back("alma");
fruit.push_back("kivi");
fruit.push_back("körte");
fruit.push_back("csillaggyümölcs");
fruit.push_back("szőlő");
```

Ha hirtelen elegünk lesz azokból a gyümölcsökből, melyek neve *k* betűvel kezdődik, akkor ezeket a következő eljárással törölhetjük ki:

```
sort(fruit.begin(),fruit.end());
vector<string>::iterator p1 = find_if(fruit.begin(),fruit.end(),initial('k'));
vector<string>::iterator p2 = find_if(p1,fruit.end(),initial_not('k'));
fruit.erase(p1,p2);
```

Tehát rendezzük a vektort, megkeressük az első és az utolsó gyümölcsöt, melynek neve *k* betűvel kezdődik, végül ezeket töröljük a *fruit* objektumból. Hogyan készíthetünk olyan függvényeket, mint az *initial(x)* – amely eldönti, hogy a kezdő karakter *x* betű-e – vagy az *initial_not(x)*, amely akkor ad igaz értéket, ha a kezdőbetű nem *x*? Ezzel a kérdéssel később, a §18.4.2 pontban foglalkozunk részletesen.

Az *erase(p1,p2)* művelet *p1*-től *p2*-ig törli az elemeket (a *p2* elem marad). Ezt a következőképpen szemléltethetjük:

```
fruit[ ]:
                                p1          p2
                                |           |
                                v           v
  alma   csillaggyümölcs  kivi    körte   őszibarack    szőlő
```

Az *erase(p1,p2)* törli a *kivi* és a *körte* elemet, tehát a végeredmény a következő lesz:

```
fruit[ ]:

  alma   csillaggyümölcs  őszibarack   szőlő
```

Szokás szerint, a programozó által megadott sorozat az első feldolgozandó elemtől az utolsó feldolgozott elem utáni pozícióig tart.

Esetleg eszünkbe jut az alábbi megoldással próbálkozni:

```
vector<string>::iterator p1 = find_if(fruit.begin(),fruit.end(),initial('k'));
vector<string>::reverse_iterator p2 = find_if(fruit.rbegin(),fruit.rend(),initial('k'));
fruit.erase(p1,p2+1);              // hoppá! típushiba
```

Azonban a *vector<fruit>::iterator* és a *vector<fruit>::reverse_iterator* nem feltétlenül azonos típusú, így nem használhatjuk őket együtt az *erase()* függvény hívásakor. Ha egy *reverse_iterator* értéket egy *iterator* értékkel együtt akarunk használni, akkor az előbbit át kell alakítanunk:

```
fruit.erase(p1,p2.base());        // iterator kinyerése reverse_iterator-ból (§19.2.5)
```

Ha egy vektorból elemeket törlünk, akkor megváltozik annak mérete és a törölt elem után szereplő értékeket a felszabadult területre kell másolnunk. Példánkban a *fruit.size()* értéke 4-re változik, és az *őszibarack*, amire eddig *fruit[5]* néven hivatkozhattunk, most már *fruit[3]*-ként lesz elérhető.

Természetesen arra is lehetőségünk van, hogy egyetlen elemet töröljünk. Ebben az esetben csak az erre az elemre mutató bejáróra van szükség (és nem egy bejáró-párra):

```
fruit.erase(find(fruit.begin(),fruit.end(),"őszibarack"));
fruit.erase(fruit.begin()+1);
```

Ez a két sor törli az *őszibarack*-ot és a *csillaggyümölcs*-öt, így a *fruit* vektorban már csak két elem marad:

fruit[ ]:

alma szőlő

Arra is lehetőségünk van, hogy egy vektorba elemeket illesszünk be. Például:

```
fruit.insert(fruit.begin()+1,"cseresznye");
fruit.insert(fruit.end(),"ribizli");
```

Az új elem a megadott elem elé kerül, az utána következő elemek pedig elmozdulnak, hogy az új elemet beszúrhassuk. Az eredmény:

fruit[ ]:

alma cseresznye szőlő ribizli

Figyeljük meg, hogy az *f.insert(f.end(),x)* egyenértékű az *f.push_back(x)* művelettel.

Teljes sorozatokat is beilleszthetünk egy vektorba:

```
fruit.insert(fruit.begin()+2,citrus.begin(),citrus.end());
```

Ha a *citrus* egy másik tároló, amely így néz ki:

citrus[ ]:
citrom grapefruit narancs lime

akkor a következő eredményt kapjuk:

fruit[ ]:
alma cseresznye citrom grapefruit narancs lime szőlő ribizli

Az *insert()* függvény a *citrus* elemeit átmásolja a *fruit* vektorba. A *citrus* tároló változatlan marad.

Kétségtelen, hogy az *insert()* és az *erase()* általánosabbak, mint azok a műveletek, melyek csak a vektor végének módosítását teszik lehetővé (§16.3.5). Éppen emiatt azonban sokkal több gonddal is járhatnak. Például ahhoz, hogy az *insert()* egy új elemet beillesszen, esetleg az összes korábbi elemet át kell helyeznie a memóriában. Ha sokszor használunk teljesen általános beszúró és törlő műveleteket egy tárolón, akkor ennek a tárolónak esetleg nem is *vector*-nak, hanem inkább *list*-nek kéne lennie. A *list* tároló hatékonyan képes együttműködni az *insert()* és az *erase()* műveletekkel, de az indexelés ez esetben nehézkes (§16.3.3).

A beszúrás és a törlés a *vector* esetében esetleg sok-sok elem áthelyezésével jár, (míg a *list* vagy az asszociatív tárolók – például a *map* – esetében ez elkerülhető). Ennek következtében előfordulhat, hogy a *vector* egyik elemére mutató *iterator* egy *insert()* vagy egy *erase()* művelet végrehajtása után egy másik, vagy akár egy egyáltalán nem létező elemre mutat. Soha ne próbáljunk meg elérni elemeket érvénytelen bejárón keresztül, mert az eredmény

meghatározhatatlan lesz és általában végzetes következményekkel jár. Különösen veszélyes egy olyan bejáró használata, amely a beszúrás helyét jelölte ki, mert az *insert()* az első paraméterét érvénytelenné teszi:

```
void duplicate_elements(vector<string>& f)
{
  for(vector<string>::iterator p = f.begin(); p!=f.end(); ++p) f.insert(p,*p);     // Nem!
}
```

Erre kell gondolnunk majd a §16.5[15] feladatnál is. A *vector* adott változata az új *p* elem beszúrásához esetleg az összes elemet áthelyezné, de a *p* utániakat biztosan.

A *clear()* művelet a tároló összes elemét törli. Ezért a *c.clear()* a *c.erase(c.begin(),c.end())* rövidítésének tekinthető. A *c.clear()* végrehajtása után a *c.size()* értéke *0* lesz.

16.3.7. Az elemek kiválasztása

A legtöbb esetben az *erase()* és az *insert()* célpontja egy jól meghatározott hely, például a *begin()* vagy az *end()* eredménye, valamilyen keresési eljárás (például a *find()*) által visszaadott érték vagy egy bejárással megtalált elempozíció. Ezekben az esetekben rendelkezésünkre áll egy bejáró, amely a kívánt elemet jelöli ki. A *vector* (és a vektorszerű tárolók) elemeit azonban meghatározhatjuk indexeléssel is. Hogyan állíthatunk elő egy olyan bejárót, amely az *insert()* vagy az *erase()* utasításban megfelelő paraméter a 7-es sorszámú elem kijelöléséhez? Mivel ez a hetedik elem a vektor elejétől számítva, így a *c.begin()+7* kifejezés jelenti a megoldást. A további lehetőségek, amelyek a tömbökhöz való hasonlóság miatt felmerülhetnek bennünk, gyakran nem működnek. Például gondoljuk végig a következő eseteket:

```
template<class C> void f(C& c)
{
  c.erase(c.begin()+7);          // rendben (ha c bejárói támogatják a + műveletet
                                 // (§19.2.1))
  c.erase(&c[7]);                // nem általános
  c.erase(c+7);                  // hiba: 7 hozzáadása egy tárolóhoz nem értelmezhető
  c.erase(c.back());             // hiba: c.back() referencia, nem bejáró
  c.erase(c.end()-2);            // rendben (utolsó előtti előtti elem)
  c.erase(c.rbegin()+2);         // hiba: vector::reverse_iterator és vector::iterator
                                 // különböző típusok
  c.erase((c.rbegin()+2).base());         // zavaros, de jó (§19.2.5)
}
```

A legcsalogatóbb lehetőség a *&c[7]*, amely a *vector* legnyilvánvalóbb megvalósításában használható is, hiszen a *c[7]* egy létező elem, és ennek címe használható bejáróként. A *c* viszont lehet egy olyan tároló is, ahol a bejáró nem egy egyszerű mutató valamelyik elemre. A *map* index-operátora (§17.4.1.3) például egy *mapped_type&* típusú referenciát, és nem egy elemre hivatkozó referenciát (*value_type&*) ad vissza.

A bejáróra nem minden tároló esetében használható a + művelet. A *list* például még a *c.begin()+7* forma használatát sem támogatja. Ha feltétlenül hetet kell adnunk egy *list::iterator* objektumhoz, akkor a ++ operátort kell ismételgetnünk.

A *c+7* és a *c.back()* forma egyszerűen típushibát eredményez. A tároló nem numerikus változó, amelyhez hetet hozzáadhatnánk, a *c.back()* pedig egy konkrét elem – például a *„körte"* értékkel, amely nem határozza meg a *körte* helyét a *c* tárolóban.

16.3.8. Méret és kapacitás

A *vector* osztályt eddig úgy próbáltuk meg bemutatni, hogy a lehető legkevesebbet szóljunk a memóriakezelésről. A *vector* szükség szerint növekszik. Általában ennyit éppen elég tudnunk. Ennek ellenére lehetőségünk van rá, hogy közvetlenül a *vector* memóriahasználatáról kérdezzünk, és bizonyos helyzetekben érdemes is élnünk ezzel a lehetőséggel. Az ilyen célú műveletek a következők:

```
template <class T, class A = allocator<T> > class vector {
public:
  // ...
  // kapacitás

  size_type size() const;                    // elemek száma
  bool empty() const { return size()==0; }
  size_type max_size() const;                // a lehetséges legnagyobb vektor mérete
  void resize(size_type sz, T val = T());    // a hozzáadott elemeknek val ad kezdőértéket

  size_type capacity() const;      // az elemek számának megfelelő lefoglalt memória mérete
  void reserve(size_type n);       // hely biztosítása összesen n elem számára; nem adunk
                                   // kezdőértéket
                                   // length_error kiváltása, ha n>max_size()

  // ...
};
```

A *vector* minden pillanatban valahány elemet tárol. Az éppen tárolt elemek számát a *size()* függvénnyel kérdezhetjük le és a *resize()* segítségével módosíthatjuk. Tehát a programozó meg tudja állapítani a vektor méretét és meg tudja azt változtatni, ha a vektor szűknek vagy túl nagynak bizonyul:

```
class Histogram {
  vector<int> count;
public:
  Histogram(int h) : count(max(h,8)) {}
  void record(int i);
  // ...
};

void Histogram::record(int i)
{
  if (i<0) i = 0;
  if (count.size()<=i) count.resize(i+i);          // sok hely kell
  count[i]++;
}
```

A *resize()* használata egy *vector* esetében nagyon hasonlít arra, amikor a C standard könyvtárának *realloc()* függvényét használjuk egy dinamikusan lefoglalt C tömb esetében.

Amikor egy vektort átméretezünk, hogy több (vagy kevesebb) elemet tároljunk benne, elképzelhető, hogy az összes elem új helyre kerül a memóriában. Ezért átméretezhető vektorok elemeire hivatkozó mutatókat nem tárolhatunk akármeddig, egy *resize()* utasítás után ugyanis ezek már felszabadított memóriaterületre mutatnak. Ehelyett nyilvántarthatunk indexértékeket. Soha ne felejtsük el, hogy a *push_back()*, az *insert()* és az *erase()* is átméretezi a vektort.

Ha a programozó tudja, hogy a vektor mérete a későbbiekben mennyire nőhet, esetleg érdemes a *reserve()* függvény segítségével előre lefoglalnia a megfelelő méretű területet a későbbi felhasználáshoz:

```
struct Link {
  Link* next;
  Link(Link* n =0) : next(n) {}
  // ...
};

vector<Link> v;

void chain(size_t n)        // v feltöltése n számú Link-kel, melyek az előző Link-re mutatnak
{
```

```
    v.reserve(n);
    v.push_back(Link(0));
    for (int i = 1; i<n; i++) v.push_back(Link(&v[i-1]));
    // ...
}
```

A *v.reserve(n)* függvényhívás biztosítja, hogy a *v* méretének növelésekor mindaddig nem lesz szükség memóriafoglalásra, amíg *v.size()* meg nem haladja *n* értékét.

A szükséges memóriaterület előzetes lefoglalásának két előnye van. Az egyik, hogy még a legegyszerűbb nyelvi változat is egyetlen művelettel lefoglalhat elegendő memóriaterületet, és nem kell minden lépésben lassú memóriaigénylést végrehajtania. Ezenkívül a legtöbb esetben számolhatunk egy logikai előnnyel is, amely talán még fontosabb, mint a hatékonysági szempont. Amikor egy *vector* mérete megnő, akkor elméletileg minden elem helye megváltozhat a memóriában. Ennek következtében minden olyan jellegű láncolás, amilyet az előbbi példában létrehoztunk az elemek között, csak akkor használható, ha a *reserve()* garantálja, hogy az elemek memóriabeli helye nem változik meg a vektor felépítése közben. Tehát bizonyos esetekben a *reserve()* biztosítja programunk helyességét, amellett, hogy hatékonysági előnyöket is jelent.

Ugyanezt a biztonságot nyújtja az is, ha csak kiszámítható időközönként fordulhat elő, hogy a *vector* számára lefoglalt terület elfogy és egy bonyolult művelettel át kell helyeznünk az addig tárolt elemeket. Ez nagyon fontos lehet olyan programok esetében, melyeknek szigorú futási idejű korlátozásoknak kell megfelelniük.

Fontos megjegyeznünk, hogy a *reserve()* nem változtatja meg a vektor (logikai) méretét, ezért az új elemeknek sem kell kezdőértéket adnia. Tehát mindkét szempontból ellentétesen viselkedik, mint a *resize()*.

Ugyanúgy, ahogy a *size()* az éppen tárolt elemek számát adja meg, a *capacity()* függvény a lefoglalt memória-egységek számáról tájékoztat. Így a *c.capacity()-c.size()* kifejezés azt adja meg, hogy még hány elemet hozhatunk létre újbóli memóriafoglalás nélkül.

A vektor méretének csökkentésével nem csökkentjük annak kapacitását. Az így felszabadult területek tehát megmaradnak a *vector*-ban a későbbi növekedést segítendő.

Ha a felszabadult területet vissza szeretnénk adni a rendszernek, egy kis trükkhöz kell folyamodnunk:

```
vector <Link> tmp = v;    // v másolata alapértelmezett kapacitással
v.swap(tmp);              // most v rendelkezik az alapértelmezett kapacitással (§16.3.9)
```

A *vector* az elemek tárolásához szükséges memóriaterületet úgy foglalja le, hogy meghívja (a sablonparaméterként megadott) memóriafoglaló tagfüggvényeit. Az alapértelmezett memóriafoglaló, melynek neve *allocator* (§19.4.1), a *new* operátort használja a memóriafoglaláshoz, így egy *bad_alloc* kivételt vált ki, ha már nincs elég szabad memória. Más memóriafoglalók más módszert is követhetnek (§19.4.2).

A *reserve()* és *capacity()* függvények csak az olyan egységes, „tömör" tárolók esetében használhatók, mint a *vector*. A *list*-hez például ilyen szolgáltatás nem áll rendelkezésünkre.

16.3.9. További tagfüggvények

Nagyon sok algoritmus – köztük a fontos rendező eljárások – elemek felcserélésével dolgozik. A csere (§13.5.2) legegyszerűbb megvalósítása, hogy egyszerűen átmásolgatjuk az elemeket. A *vector* osztályt azonban általában olyan szerkezet ábrázolja, amely az elemek leíróját (handle) tárolja (§13.5, §17.1.3). Így két *vector* sokkal hatékonyabban is felcserélhető, ha a leírókat cseréljük fel. A *vector::swap()* függvény ezt valósítja meg. Az alapértelmezett *swap()* nagyságrendekkel lassabb, mint a fenti eljárás:

```
template <class T, class A = allocator<T> > class vector {
public:
  // ...

  void swap(vector&);

  allocator_type get_allocator() const;
};
```

A *get_allocator()* függvény lehetőséget ad a programozónak arra, hogy elérje a *vector* memóriafoglalóját (§16.3.1, §16.3.4). Erre a szolgáltatásra általában akkor van szükségünk, amikor egy, a vektorhoz kapcsolódó adatnak a programban ugyanúgy akarunk memóriát foglalni, mint a *vector*-nak magának (§19.4.1)

16.3.10. Segédfüggvények

Két *vector* objektumot az == és a < operátor segítségével hasonlíthatunk össze:

```
template <class T, class A>
bool std::operator==(const vector<T,A>& x, const vector<T,A>& y);
```

```
template <class T, class A>
bool std::operator<(const vector<T,A>& x, const vector<T,A>& y);
```

A *v1* és a *v2* *vector* akkor egyenlő, ha *v1.size()==v2.size()* és *v1[n]==v2[n]* minden értelmes *n* index esetén. Hasonlóan, a < a nyelvi elemek szerinti rendezést jelenti, tehát a < műveletet a *vector* esetében a következőképpen határozhatjuk meg:

```
template <class T, class A>
inline bool std::operator<(const vector<T,A>& x, const vector<T,A>& y)
{
  return lexicographical_compare (x.begin(),x.end(),y.begin(),y.end());     // lásd §18.9
}
```

Ez azt jelenti, hogy *x* akkor kisebb, mint *y*, ha az első olyan *x[i]* elem, amely nem egyezik meg a megfelelő *y[i]* elemmel, kisebb, mint *y[i]*, vagy *x.size()<y.size()* és minden *x[i]* megegyezik a megfelelő *y[i]* értékkel.

A standard könyvtár az == és a < definíciójának megfelelően meghatározza a *!=*, a <=, a >, és a >= operátort is.

Mivel a *swap()* egy tagfüggvény, *v1.swap(v2)* formában hívhatjuk meg. Nem minden típusban szerepel azonban a *swap()* tagfüggvény, így az általános algoritmusok a *swap(a,b)* formát használják. Ahhoz, hogy ez a forma a *vector*-ok esetében is működjön, a standard könyvtár a következő specializációt adja meg:

```
template <class T, class A> void std::swap(vector<T,A>& x, vector<T,A>& y)
{
  x.swap(y);
}
```

16.3.11. Vector<bool>

A *vector<bool>* specializált osztály (§13.5) logikai értékeknek egy tömörített vektorát valósítja meg. Egy *bool* típusú változót is meg kell tudnunk címezni, így az legalább egy bájtot foglal. A *vector<bool>* osztályt azonban könnyen elkészíthetjük úgy is, hogy minden elem csak egy bitet foglaljon.

A szokásos *vector* műveletek ugyanolyan jelentéssel használhatók a *vector<bool>* esetében is. Különösen fontos, hogy az indexelés és a bejárás is elvárásainknak megfelelően működnek:

```
void f(vector<bool>& v)
{
  for (int i = 0; i<v.size(); ++i) cin >> v[i];          // ciklus index használatával

  typedef vector<bool>::const_iterator VI;
  for (VI p = v.begin(); p!=v.end(); ++p) cout<<*p;       // ciklus bejárók
                                                           // használatával
}
```

Ennek eléréséhez utánozni kell a bitenkénti címzést. Mivel egy mutató az egy bájtnál kisebb memóriaegységeket nem képes azonosítani, a *vector<bool>::iterator* nem lehet mutató. A *bool** például biztosan nem használható bejáróként a *vector<bool>* osztályban:

```
void f(vector<bool>& v)
{
  bool* p = v.begin();         // hiba: nem megfelelő típus
  // ...
}
```

A különálló bitek megcímzésének módját a §17.5.3 pontban mutatjuk be.

A könyvtárban szerepel a *bitset* típus is, amely logikai értékek halmazát tárolja, a logikai halmazok szokásos műveleteivel (§17.5.3).

16.4. Tanácsok

[1] Hordozható programok készítéséhez használjuk a standard könyvtár szolgáltatásait. §16.1.
[2] Ne próbáljuk újraalkotni a standard könyvtár szolgáltatásait. §16.1.2.
[3] Ne higgyük, hogy minden esetben a standard könyvtár jelenti a legjobb megoldást.
[4] Amikor új szolgáltatásokat hozunk létre, gondoljuk végig, hogy nem valósíthatók-e meg a standard könyvtár nyújtotta kereteken belül. §16.3.
[5] Ne felejtsük, hogy a standard könyvtár szolgáltatásai az *std* névtérhez tartoznak. §16.1.2.
[6] A standard könyvtár szolgáltatásait a megfelelő fejállomány segítségével építsük be, ne használjunk közvetlen deklarációt. §16.1.2.

[7] Használjuk ki a kései elvonatkoztatás előnyeit. §16.2.1.
[8] Kerüljük a kövér felületek használatát. §16.2.2.
[9] Ha az elemeken visszafelé akarunk haladni, használjunk *reverse_iterator*-okat *iterator* helyett. §16.3.2.
[10] Ha *iterator*-t szeretnénk csinálni egy *reverse_iterator*-ból, használjuk a *base()* függvényt. §16.3.2.
[11] Tárolókat referencia szerint adjunk át. §16.3.4.
[12] Ha egy tároló elemeire akarunk hivatkozni, mutatók helyett használjunk bejárótípusokat (például *list<char>::iterator*). §16.3.1.
[13] Használjunk *const* bejárókat, ha a tároló elemeit nem akarjuk megváltoztatni. §16.3.1.
[14] Ha tartományellenőrzést akarunk végezni – akár közvetlenül, akár közvetve –, használjuk az *at()* függvényt. §16.3.3.
[15] Használjuk inkább a *push_back()* vagy a *resize()* függvényt egy tárolóban, mint a *realloc()* függvényt egy tömbben. §16.3.5.
[16] Ne használjuk egy átméretezett *vector* bejáróit. §16.3.8.
[17] A bejárók érvénytelenné válását elkerülhetjük a *reserve()* függvény használatával. §16.3.8.
[18] Ha szükséges, a *reserve()* függvényt felhasználhatjuk a teljesítmény kiszámíthatóvá tételére is. §16.3.8.

16.5. Feladatok

A fejezet legtöbb feladatának megoldását könnyen megtalálhatjuk, ha megnézzük a standard könyvtár adott változatát. Mielőtt azonban megnéznénk, hogy a könyvtár készítői hogyan közelítették meg az adott problémát, érdemes saját megoldást készítenünk.

1. (*1.5.) Készítsünk egy *vector<char>* típusú objektumot, amely az ábécé betűit tartalmazza, sorrendben. Írassuk ki ennek a tömbnek a tartalmát előre, majd visszafelé.
2. (*1.5.) Készítsünk egy *vector<string>* objektumot, majd olvassuk be gyümölcsök neveit a *cin* eszközről. Rendezzük a listát, majd írassuk ki elemeit.
3. (*1.5.) A 16.5[2] feladat vektorának felhasználásával írjunk olyan ciklust, amely az összes *'a'* betűvel kezdődő nevű gyümölcsöt jeleníti meg.
4. (*1.) A 16.5[2] feladat vektorának felhasználásával írjunk olyan ciklust, amellyel törölhetjük az összes *'a'* betűvel kezdődő nevű gyümölcsöt.
5. (*1.) A 16.5[2] feladat vektorának felhasználásával írjunk olyan ciklust, amellyel az összes citrusfélét törölhetjük.

6. (*1.5.) A 16.5[2] feladat vektorának felhasználásával írjunk olyan ciklust, amely azokat a gyümölcsöket törli, amelyeket nem szeretünk.
7. (*2.) Fejezzük be a §16.2.1 pontban elkezdett *Vector*, *List* és *Itor* osztályt.
8. (*2.5.) Az *Itor* osztályból kiindulva gondoljuk végig, hogyan készíthetünk bejárókat előre haladó bejárásokhoz, visszafelé haladó bejárásokhoz, olyan tárolóhoz, amely a bejárás alatt megváltozhat, illetve megváltoztathatatlan tárolóhoz. Szervezzük úgy ezeket a tárolókat , hogy a programozó mindig azt a bejárót használhassa, amelyik a leghatékonyabb megoldást kínálja az adott algoritmus megvalósítására. A tárolókban kerüljünk minden kódismétlést. Milyen más bejáró-fajtára lehet szüksége egy programozónak? Gyűjtsük össze az általunk használt megközelítés előnyeit és hátrányait.
9. (*2.) Fejezzük be a §16.2.2 pontban elkezdett *Container*, *Vector* és *List* osztály megvalósítását.
10. (*2.5.) Állítsunk elő 10 000 darab egyenletes eloszlású véletlenszámot a *0-1023* tartományban és tároljuk azokat a.) a standard könyvtár *vector* osztályával, b.) a §16.5[7] feladat *Vector* osztályával, c.) a §16.5[9] feladat *Vector* osztályával. Mindegyik esetben számoljuk ki a vektor elemeinek számtani közepét (mintha még nem tudnánk). Mérjük a ciklusok lefutásának idejét. Becsüljük meg vagy számoljuk ki a háromféle vektor memória-felhasználását, és hasonlítsuk össze az értékeket.
11. (*1.5.) Írjunk egy bejárót, amely lehetővé teszi, hogy a §16.2.2 pontban bemutatott *Vector* osztályt a §16.2.1 pontban használt tároló stílusában kezeljük.
12. (*1.5.) Származtassunk egy osztályt a *Container*-ből, amely lehetővé teszi, hogy a §16.2.1 vektorát a 16.2.2 pontban bemutatott tárolóstílusban használjuk.
13. (*2.) Készítsünk olyan osztályokat, amelyek lehetővé teszik, hogy a §16.2.1 és a §16.2.2 *Vector* osztályait szabványos tárolókként használjuk.
14. (*2) Írjunk egy létező (nem szabványos, nem tankönyvi példa) tárolótípushoz egy olyan sablont, amely ugyanazokkal a tagfüggvényekkel és típus tagokkal rendelkezik, mint a szabványos *vector*. Az eredeti tárolótípust ne változtassuk meg. Hogyan tudjuk felhasználni azokat a szolgáltatásokat, amelyeket a nem szabványos vektor megvalósít, de a *vector* nem?
15. (*1.5) Gondoljuk végig, hogyan viselkedik a §16.3.6 pontban bemutatott *duplicate_elements()* függvény egy olyan *vector<string>* esetében, melynek három eleme: *ne tedd ezt*.

17

Szabványos tárolók

„Itt az ideje, hogy munkánkat
végre szilárd elméleti
alapokra helyezzük."
(Sam Morgan)

Szabványos tárolók • Tárolók és műveletek - áttekintés • Hatékonyság • Ábrázolás • Megkötések az elemekre • Sorozatok • *vector* • *list* • *deque* • Átalakítók • *stack* • *queue* • *priority_queue* • Asszociatív tárolók • *map* • Összehasonlítások • *multimap* • *set* • *multiset* • „Majdnem-tárolók" • *bitset* • Tömbök • Hasító táblák • A *hash_map* egy megvalósítása • Tanácsok • Gyakorlatok

17.1. A szabványos tárolók

A standard könyvtár kétféle tárolót tartalmaz: sorozatokat (sequences) és asszociatív tárolókat (associative container). A sorozatok mindegyike nagyon hasonlít a *vector* tárolóra (§16.3). Ha külön nem említjük, ugyanazok a tagtípusok és függvények használhatók ezekre is, mint a vektorokra, és eredményük is ugyanaz lesz. Az asszociatív tárolók ezen kívül lehetőséget adnak az elemek kulcsokkal történő elérésére is (§3.7.4).

A beépített tömbök (§5.2), a karakterláncok (20. fejezet), a *valarray* osztály (§22.4) és a *bitset* (bithalmaz) típusok szintén elemek tárolására szolgálnak, így ezeket is tárolóknak tekinthetjük. E típusok azonban mégsem tökéletesen kidolgozott, szabványos tárolók. Ha a standard könyvtár elvárásainak megfelelően próbálnánk meg elkészíteni őket, akkor elsődleges céljukat nem tudnák maradéktalanul elérni. A beépített tömböktől például nem várhatjuk el, hogy egyszerre tartsák nyilván saját méretüket és ugyanakkor teljesen összeegyeztethetőek maradjanak a C tömbökkel.

A szabványos tárolók alapötlete, hogy logikailag felcserélhetőek legyenek minden értelmes helyzetben. Így a felhasználó mindig szabadon választhat közülük, az éppen használni kívánt műveletek, illetve a hatékonysági szempontok figyelembe vételével. Ha például gyakran kell elérnünk elemeket valamilyen kulcsérték segítségével, akkor a *map* (§17.4.1) tárolót használjuk. Ha legtöbbször a szokásos listaműveletekre van szükségünk, akkor a *list* (§17.2.2) a megfelelő osztály. Ha sokszor kell hozzáfűznünk elemeket a tároló végéhez, vagy éppen el kell onnan távolítanunk elemeket, akkor a *deque* (kétvégű sor, §17.2.3), a *stack* (§17.3.1) vagy a *queue* (§17.3.2) nyújtja a legtöbb segítséget. Ezeken kívül maga a programozó is kifejleszthet olyan tárolókat, melyek pontosan illeszkednek a szabványos tárolók által kínált keretrendszerbe (§17.6). Leggyakrabban a *vector* osztályt fogjuk használni, mert ennek előnyeit rengeteg felhasználási területen kiaknázhatjuk.

Az ötlet, hogy a különböző jellegű tárolókat – vagy még általánosabban, az összes információforrást – egységes formában kezeljük, elvezet minket az általánosított (generikus) programozás fogalmához (§2.7.2, §3.8). Ezt a szemléletmódot követve a standard könyvtár nagyon sok általános algoritmust biztosít (18. fejezet), melyek megkímélik a programozót attól, hogy közvetlenül az egyes tárolók részleteivel foglalkozzon.

17.1.1. Műveletek – áttekintés

Ebben a pontban felsoroljuk azokat az osztálytagokat, amelyek minden, vagy majdnem minden tárolóban megtalálhatók. Ha részleteket szeretnénk megtudni, nézzük meg a megfelelő fejállományt (<*vector*>, <*list*>, <*map*> stb, §16.1.2).

Típusok (§16.3.1)	
value_type	Az elemek típusa.
allocator_type	A memóriakezelő típusa.
size_type	Típus az indexeléshez, elemszámláláshoz stb.
difference_type	A bejárók közötti különbség típusa.
iterator	Úgy viselkedik, mint a *value_type** típus.
const_iterator	A *const value_type** megfelelője.
reverse_iterator	A tároló elemeit fordított sorrendben látjuk, egyébként a *value_type** megfelelője.
const_reverse_iterator	A tároló elemeit fordított sorrendben látjuk; a *const value_type** típushoz hasonlít.
reference	Olyan, mint a *value_type&*.
const_reference	Olyan, mint a *const value_type&*.
key_type	A kulcs típusa (csak asszociatív tárolókhoz).
mapped_type	A *mapped_value* típusa (csak asszociatív tárolókhoz).
key_compare	Az összehasonlítási szempont típusa (csak asszociatív tárolókhoz).

A tárolót tekinthetjük elemek sorozatának, akár abban a sorrendben, amelyet a tároló bejárója meghatároz, akár ellenkező irányban. Egy asszociatív tároló esetében a sorrendet a tároló összehasonlítási szempontja határozza meg (ami alapértelmezés szerint a < művelet).

Bejárók (§16.3.2)	
begin()	Az első elemre mutat.
end()	Az utolsó utáni elemre mutat.
rbegin()	Visszafelé haladó felsorolás esetén az első elemre mutat.
rend()	Visszafelé haladó felsorolás esetén az utolsó utáni elemre mutat.

Néhány elemet közvetlenül is elérhetünk:

Hozzáférés elemekhez (§16.3.3)	
front()	Az első elem.
back()	Az utolsó elem.
[ ]	Indexelés, ellenőrzés nélküli hozzáférés. (Listáknál nem használható.)
at()	Indexelés, ellenőrzött változat. (Listáknál nem használható.)

A vektorok (*vector*) és a kétvégű sorok (*deque*) olyan hatékony műveleteket is biztosítanak, amelyek az elemek sorozatának végén (*back*) végeznek módosításokat. A listák (*list*) és a kétvégű sorok (*deque*) az ezekkel egyenértékű műveleteket az elemsorozatok elején (*front*) is képesek elvégezni:

Verem- és sorműveletek (§16.3.5, §17.2.2.2)	
push_back()	Elem beszúrása a tároló végére.
pop_back()	Elem eltávolítása a tároló végéről.
push_front()	Új első elem beillesztése (csak listákhoz és kétvégű sorokhoz).
pop_front()	Az első elem eltávolítása (csak listákhoz és kétvégű sorokhoz).

A tárolók lehetővé teszik a listaműveletek használatát is:

Listaműveletek (§16.3.6)	
insert(p,x)	*x* beillesztése *p* elé.
insert(p,n,x)	*x* elem *n* darab másolatának beillesztése *p* elé.
insert(p,first,last)	Elemek beszúrása a [*first*:*last*[tartományból a *p* elé.
erase(p)	A *p* helyen lévő elem eltávolítása.
erase(first,last)	A [*first*:*last*[tartomány törlése.
clear()	Az összes elem törlése.

Minden tárolóban találhatunk az elemek számával kapcsolatos műveleteket, illetve néhány egyéb függvényt is:

További műveletek (§16.3.8, §16.3.9, §16.3.10)	
size()	Az elemek száma.
empty()	Üres a tároló?
max_size()	A legnagyobb lehetséges tároló mérete.
capacity()	A *vector* számára lefoglalt terület mérete (csak vektorokhoz).
reserve()	Memóriafoglalás a későbbi növelések gyorsításához (csak vektorokhoz).
resize()	A tároló méretének módosítása (csak vektorokhoz, listákhoz és kétvégű sorokhoz).
swap()	Két tároló elemeinek felcserélése.
get_allocator()	A tároló memóriafoglalójának lemásolása.
==	Megegyezik a két tároló tartalma?
!=	Különbözik a két tároló tartalma?
<	Az egyik tároló ábécésorrendben megelőzi a másikat?

A tárolók számos konstruktort, illetve értékadó operátor biztosítanak:

Konstruktorok stb. (§16.3.4)	
container()	Üres tároló.
container(n)	*n* darab elem, alapértelmezett értékkel (asszociatív tárolókhoz nem használható).
container(n,x)	*x* elem *n* példányából készített tároló (asszociatív tárolókhoz nem használható).
container(first,last)	A kezdőelemek a *[first:last[* tartományból származnak.
container(x)	Másoló konstruktor. A kezdeti elemeket az *x* tároló elemei adják.
~container()	A tároló és összes elemének megsemmisítése.

Értékadások (§16.3.4)	
operator=(x)	Másoló értékadás. Az elemek az *x* tárolóból származnak.
assign(n,x)	*x* elem *n* példányának beillesztése a tárolóba (asszociatív tárolóknál nem használható).
assign(first,last)	Értékadás a [*first*:*last*[tartományból.

Az asszociatív tárolók lehetővé teszik az elemek kulcs alapján történő elérését:

Asszociatív műveletek (§17.4.1)	
operator[ ](k)	A *k* kulcsú elem elérése (egyedi kulccsal rendelkező tárolókhoz).
find(k)	*k* kulccsal rendelkező elem keresése.
lower_bound(k)	Az első *k* kulcsú elem megkeresése.
upper_bound(k)	Az első olyan elem megkeresése, melynek kulcsa nagyobb, mint *k*.
equal_range(k)	A *k* kulcsú elemek alsó és felső határának megkeresése.
key_comp()	A kulcs-összehasonlító objektum másolata.
value_comp()	A *mapped_value* értékeket összehasonlító objektum másolata.

Ezen általános műveleteken kívül a legtöbb tároló néhány egyedi műveletet is biztosít.

17.1.2. Tárolók – áttekintés

A szabványos tárolókat (container) az alábbi táblázat foglalja össze:

A szabványos tárolók műveletei

	[ ]	Lista-műveletek	Műveletek a tároló elején	Műveletek a tároló végén (verem műveletek)	Bejárók
	§16.3.3	§16.3.6	§17.2.2.2	§16.3.5	§19.2.1
	§17.4.1.3	§20.3.9	§20.3.9	§20.3.12	
vector	konstans	O(n)+		konstans+	Közvetlen
list		konstans	konstans	konstans	Kétirányú
deque	konstans	O(n)	konstans	konstans	Közvetlen
stack				konstans	
queue			konstans	konstans	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Kétirányú
multimap		O(log(n))+			Kétirányú
set		O(log(n))+			Kétirányú
multiset		O(log(n))+			Kétirányú
string	konstans	O(n)+	O(n)+	konstans+	Közvetlen
array	konstans				Közvetlen
valarray	konstans				Közvetlen
bitset	konstans				

Az *Elérés* oszlopban a *Közvetlen* azt jelenti, hogy az elemek tetszőleges sorrendben elérhetők (közvetlen elérés, random access), a *Kétirányú* esetében pedig kétirányú (bidirectional) soros hozzáférésű bejárókat használhatunk. A kétirányú bejárók műveletei részhalmazát képezik a közvetlen elérésűek műveleteinek (§19.2.1).

A táblázat további elemeiből az adott művelet hatékonyságára következtethetünk. A *konstans* érték azt fejezi ki, hogy az adott művelet végrehajtási ideje nem függ a tárolóban tárolt elemek számától. A *konstans időigény* egy másik szokásos jelölése *O(1)*. Az *O(n)* érték azt fejezi ki, hogy a számítási idő arányos a feldolgozott elemek számával. A + jelölést azokban

az esetekben használtuk, ahol időnként jelentős mennyiségű többletterhelés is előfordulhat. Egy elem beszúrása egy listába például mindig ugyanannyi ideig tart, így a megfelelő helyen a *konstans* értéket olvashatjuk. Ugyanez a művelet a *vector* esetében a beszúrási pont után álló összes elem áthelyezésével jár, így ott az *O(n)* érték szerepel. Sőt, időnként az összes elemet (tehát a beszúrási pont előttieket is) át kell helyezni, ezért a + jelet is megadtuk. A „nagy O" (ordo) egy hagyományos jelölésmód, a + jelet csak azért használtam, hogy kiegészítő információt adjak azoknak a programozóknak, akik az átlagos teljesítmény mellett a művelet idejének megjósolhatóságára is hangsúlyt helyeznek. Az *O(n)*+ jelölés hagyományos jelentése *„amortizált lineáris idő" (amortized linear time*, vagyis kb. „többletterhet jelentő, egyenes arányosságban növekvő idő"*)*.

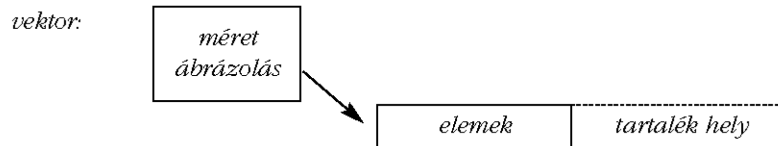
Természetesen, ha a „konstans" egy nagyon hosszú időtartamot jelent, akkor ez nagyobb lehet, mint az elemek számával egyenesen arányos időigény. Nagy adatszerkezetek esetében azonban a *konstans* jelentése mégis „olcsó", az *O(n)* jelentése pedig „drága", míg az *O(log(n))* időt tekinthetjük „elég olcsónak". Még viszonylag nagy *n* értékek esetén is, az *O(log(n))* közelebb áll a konstanshoz, mint az *O(n)*-hez. Azoknak a programozóknak, akiknek programjaik „költségeivel" komolyan foglalkozniuk kell, ennél alaposabb ismeretekre is szükségük van. Tisztában kell lenniük azzal, mely elemek figyelembe vételével alakul ki *n* értéke. Olyan alapművelet szerencsére nincs, melyet „nagyon drágának" nevezhetnénk, ami *O(n*n)* vagy még jelentősebb időigényt jelentene.

A *string* kivételével az itt közölt költségértékek megfelelnek a C++ szabvány elvárásainak. A *string* típusnál szereplő becslések saját feltételezéseim.

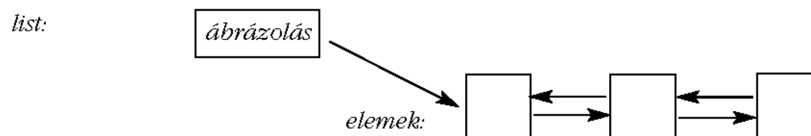
A bonyolultságnak, illetve az időigénynek ezen mérőszámai a felső határt jelentik. Szerepük abban áll, hogy a programozó megtudhatja belőlük, mit várhat el az adott szolgáltatástól. Természetesen a konkrét megvalósítások készítői igyekeznek a lényeges helyeken ezeknél jobb megoldást biztosítani.

17.1.3. Ábrázolás

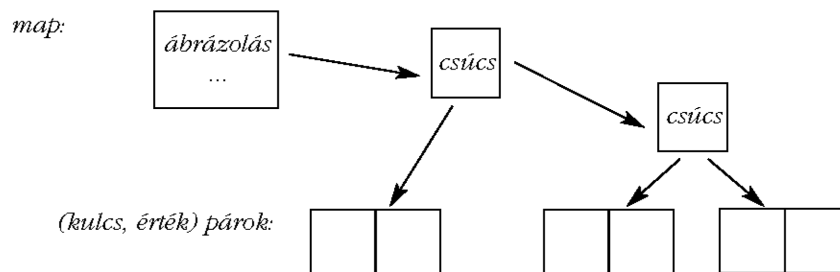
A szabvány nem ír elő semmilyen egyedi ábrázolási módot egyik tároló esetében sem. A szabvány csak a tároló felületét írja le, illetve néhány „bonyolultsági" követelményt határoz meg. Az egyes nyelvi változatok készítői maguk választhatják meg a legmegfelelőbb és gyakran nagyon jól optimalizált megvalósítási módot, amely kielégíti az általános követelményeket. A tárolókat általában olyan adatszerkezet segítségével hozzák létre, amely az elemek elérését biztosító azonosító mellett a méretre és kapacitásra vonatkozó információkat is nyilvántartja. A *vector* esetében az elemek adatszerkezete leggyakrabban egy tömb:



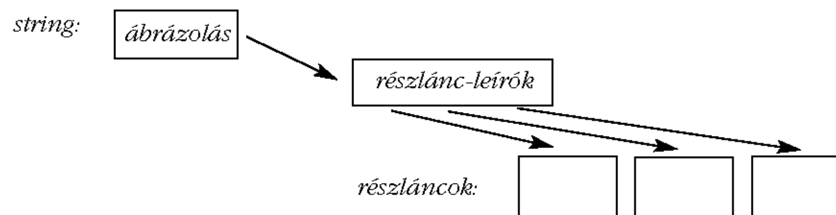
A *list* megvalósításának legegyszerűbb eszköze a láncolás, ahol az elemek egymásra mutatnak:



A *map* leggyakoribb megvalósítása a (kiegyensúlyozott) bináris fa, ahol a csúcsok vagy más néven csomópontok (kulcs, érték) párokat jelölnek ki:



A *string* megvalósítására a §11.12 pontban adtunk egy ötletet, de elképzelhetjük tömbök sorozataként is, ahol minden tömb néhány karaktert tárol:



17.1.4. Megkötések az elemekre

A tárolóban tárolt elemek a beillesztett objektumok másolatai, így ahhoz, hogy egy objektum bekerülhessen a tárolóba, olyan típusúnak kell lennie, amely lehetővé teszi, hogy a tároló másolatot készítsen róla. A tároló az elemeket egy másoló konstruktor vagy egy értékadás segítségével másolhatja le. A másolás eredményének mindkét esetben egyenértékűnek kell lennie az eredeti objektummal. Ez nagyjából azt jelenti, hogy minden elképzelhető egyenlőségvizsgálat azonosnak kell, hogy minősítse az eredeti és a másolt objektumot. Az elemek másolásának tehát úgy kell működnie, mint a beépített típusok (köztük a mutatók) szokásos másolásának. Az alábbi értékadás például nagy lépést jelent afelé, hogy az *X* típus egy szabványos tároló elemeinek típusa legyen:

```
X& X::operator=(const X& a)            // helyes értékadó operátor
{
  // 'a' minden tagjának másolása *this-be
  return *this;
}
```

Az alábbi kódrészlet *Y* típusa viszont helytelen, mert a visszatérési érték nem a szokásos és a jelentés sem megfelelő.

```
void Y::operator=(const Y& a)                  // helytelen értékadó operátor
{
  // 'a' minden tagjának törlése
}
```

A szabványos tárolók szabályaitól való eltérések egy részét már a fordító képes jelezni, de sok esetben ezt a segítséget sem kapjuk meg, csak teljesen váratlan eseményekkel kerülünk szembe. Egy olyan másolási művelet például, amely kivételt válthat ki, részlegesen átmásolt elemet is eredményezhet. Ennek következményeképpen a tároló olyan állapotba kerül, amely csak jóval később okoz problémákat. Az ilyen másolási műveletek tehát már önmagukban is rossz tervezés következményei (§14.4.6.1).

Ha az elemek másolása nem lehetséges, megoldást az jelenthet, ha a tárolóban konkrét objektumok helyett azokra hivatkozó mutatókat helyezünk el. Ennek legnyilvánvalóbb példája a többalakú (polimorf) típusok esete (§2.5.4, §12.2.6). Ha a többalakúság lehetőségét meg akarjuk őrizni, akkor a *vector<Shape>* helyett például a *vector<Shape*>* osztályra lesz szükségünk.

17.1.4.1. Összehasonlítások

Az asszociatív tárolók megkövetelik, hogy elemeiket össze lehessen hasonlítani. Ugyanez elmondható más tárolók néhány műveletéről is (például a rendező eljárásokról, mint a *sort()*). Alapértelmezés szerint a sorrend meghatározására a < operátort használjuk. Ha ez az adott esetben nem használható, akkor a programozónak valamilyen más megoldást kell találnia (§17.4.1.5, §18.4.2). A rendezési szempontnak *szigorúan megengedőnek* kell lennie (strict weak ordering), ami lényegében azt jelenti, hogy a „kisebb mint" és „egyenlő" műveletnek is tranzitívnak kell lennie. Vegyük például a következő *cmp* rendezést:

1. *cmp(x,x)* mindig hamis.
2. Ha *cmp(x,y)* és *cmp(y,z)*, akkor *cmp(x,z)*.
3. Határozzuk meg az egyenlőség *equiv(x,y)* műveletét a következő kifejezéssel: *!(cmp(x,y) || cmp(y,x))*. Így, ha *equiv(x,y)* és *equiv(y,z)*, akkor *equiv(x,z)*.

Ennek megfelelően a következőket írhatjuk:

```
template<class Ran> void sort(Ran first, Ran last);          // a < használata az
                                                              // összehasonlításra
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);
                                                              // a cmp használata
```

Az első változat a < operátort használja, míg a második a programozó által megadott *cmp* rendezést. Az előző fejezet gyümölcsöket tartalmazó tárolójának esetében például szükségünk lehet egy olyan rendező eljárásra, amely nem különbözteti meg a kis- és nagybetűket. Ezt úgy érhetjük el, hogy létrehozunk egy függvényobjektumot (function object) (§11.9, §18.4), amely egy *string*-párra elvégzi az összehasonlítást:

```
class Nocase {          // kis- és nagybetűket nem megkülönböztető karakterlánc-
                        // összehasonlítás
public:
   bool operator()(const string&, const string&) const;
};

bool Nocase::operator()(const string& x, const string& y) const
   // igazat ad vissza, ha x ábécésorrendben megelőzi y-t; a kis- és nagybetűk közötti
   // különbség nem számít
{
   string::const_iterator p = x.begin();
   string::const_iterator q = y.begin();
```

```
    while (p!=x.end() && q!=y.end() && toupper(*p)==toupper(*q)) {
            ++p;
            ++q;
    }
    if (p == x.end()) return q != y.end();
    if (q == y.end()) return false;
    return toupper(*p) < toupper(*q);
}
```

A *sort* függvényt ezután meghívhatjuk ezzel a rendezési szemponttal. Például az alábbi sorozatból kiindulva:

```
fruit:
  alma   körte   Alma   Körte   citrom
```

A *sort(fruit.begin(), fruit.end, Nocase())* rendezés eredménye a következő lesz:

```
fruit:
  Alma   alma   citrom   Körte   körte
```

Ugyanakkor az egyszerű *sort(fruit.begin(), fruit.end())* a következő eredményt adná (feltételezve, hogy olyan karakterkészletet használunk, ahol a nagybetűk megelőzik a kisbetűket):

```
fruit:
  Alma   Körte   alma   citrom   körte
```

Vigyázzunk rá, hogy a < operátor a C stílusú karakterláncoknál (tehát a *char** típusnál) nem ábécésorrend szerinti rendezést jelent (§13.5.2). Ennek például az a következménye, hogy az olyan asszociatív tárolók, melyeknek kulcsa C stílusú karakterlánc, nem úgy működnek, mint ahogy azt első ránézésre gondolnánk. Ezen kellemetlenség kijavításához egy olyan < operátort kell használnunk, amely tényleg ábécésorrend szerinti összehasonlítást végez:

```
struct Cstring_less {
  bool operator()(const char* p, const char* q) const { return strcmp(p,q)<0; }
};

map<char*,int,Cstring_less> m;          // const char* kulcsok összehasonlítására a strcmp()
                                        // függvényt használó asszociatív tömb
```

17.1.4.2. További relációs műveletek

A tárolók és az algoritmusok alapértelmezés szerint a < műveletet használják, amikor az elemek között sorrendet kell megállapítaniuk. Ha ez az alapértelmezés nem felel meg a programozónak, akkor új összehasonlítási szempontot is megadhat. Az egyenlőségvizsgálat műveletét viszont nem adhatjuk meg közvetlenül. Ha van egy sorrend-meghatározó függvényünk (például a *cmp*), akkor az egyenlőséget két sorrendvizsgálattal állapíthatjuk meg:

```
if (x == y)                 // megadott felhasználói összehasonlítás esetén nem használatos

if (!cmp(x,y) && !cmp(y,x))  // a felhasználói cmp összehasonlítás esetén használatos
```

Ez a lehetőség megkímél minket attól, hogy minden asszociatív tároló és számos algoritmus esetében külön paraméterként adjuk át az egyenlőségvizsgáló eljárást. Első ránézésre azt mondhatnánk, hogy ez a megoldás nem túl hatékony, de a tárolók rendkívül ritkán vizsgálják elemeik egyenlőségét, és ilyenkor is általában elegendő egyetlen *cmp()* hívás.

A „kisebb mint" (alapértelmezés szerint <) művelettel végzett egyenértékűség-vizsgálat gyakorlati okokból is hasznosabb lehet, mint az „egyenlő" (==) használata. Az asszociatív tárolók (§17.4) például a *! (cmp(x,y)||cmp(y,x))* egyenértékűség-vizsgálattal hasonlítják össze a kulcsaikat. Az egyenértékű kulcsok nem feltétlenül egyenlők. Egy olyan *multimap* (§17.4.2) például, amelynek sorrend-meghatározó függvénye nem különbözteti meg a kis- és nagybetűket, a *Last*, *last*, *lAst*, és *lasT* karakterláncokat egyenértékűnek fogja minősíteni, annak ellenére, hogy az == operátor különbözőnek tartja azokat. Így tehát lehetőségünk nyílik arra, hogy a számunkra lényegtelen különbségektől a rendezésben eltekintsünk.

A < és az == operátor segítségével a többi szokásos összehasonlító műveletet könnyen definiálhatjuk. A standard könyvtár az *std::rel_ops* névtérben adja meg ezeket és a *<utility>* fejállomány segítségével használhatók:

```
template<class T> bool rel_ops::operator!=(const T& x, const T& y) { return !(x==y); }
template<class T> bool rel_ops::operator>(const T& x, const T& y) { return y<x; }
template<class T> bool rel_ops::operator<=(const T& x, const T& y) { return !(y<x); }
template<class T> bool rel_ops::operator>=(const T& x, const T& y) { return !(x<y); }
```

A *rel_ops* névtér alkalmazása biztosítja, hogy az operátorokat könnyen elérhetjük, amikor szükség van rájuk, viszont meghatározásuk nem történik meg „titokban", ha explicit nem kérjük a névtér használatát.

```
void f()
{
   using namespace std;
   // a !=, > stb. alapértelmezés szerint nem jön létre
}
```

```
void g()
{
  using namespace std;
  using namespace std::rel_ops;
  // a !=, > stb. alapértelmezés szerint létrejön
}
```

A *!=*, stb. operátorokat nem az *std* névtér írja le, mert nem mindig van rájuk szükség, és bizonyos esetekben meghatározásuk meg is változtatná a felhasználó programjának működését. Például, ha egy általános matematikai könyvtárat akarunk készíteni, akkor valószínűleg a saját relációs műveleteinkre lesz szükségünk és nem a beépített függvényekre.

17.2. Sorozatok

A sorozatok körülbelül úgy néznek ki, mint a korábban (§16.3) bemutatott *vector*. A standard könyvtár által biztosított alapvető sorozatok a következők:

vector list deque

Ezekből származnak – a megfelelő felület kialakításával – az alábbi tárolók:

stack queue priority_queue

Ezeket a sorozatokat *tároló-átalakítóknak* (container adapter), *sorozat-átalakítóknak* (sequence adapter) vagy egyszerűen *átalakítóknak* (adapter, §17.3) nevezzük.

17.2.1. A vector

A szabványos *vector* osztállyal a §16.3 pontban már részletesen foglalkoztunk. Az előzetes memóriafoglalás (*reserve*) lehetősége kizárólag a vektorokra jellemző. Az indexelés a *[ ]* operátorral ellenőrizetlen adatelérést jelent. Ha ellenőrzött hozzáférést szeretnénk, használjuk az *at()* függvényt (§16.3.3), egy ellenőrzött vektort (§3.7.2) vagy egy ellenőrzött bejárót (§19.3). A *vector* osztályokban közvetlen elérésű (random-access) bejárókat (§19.2.1) használhatunk.

17.2.2. A list

A lista egy olyan sorozat, amely elemek beszúrására és törlésére a legalkalmasabb. A *vector* (és a *deque*, §17.2.3) osztállyal összehasonlítva az indexelés fájdalmasan lassú lenne, így meg sem valósították a *list* tárolóban. Ennek következménye, hogy a *list* csak kétirányú (bidirectional) bejárókat (§19.2.1) biztosít, közvetlen elérésűeket nem használhatunk. A *list* osztályt ezek alapján valamilyen kétirányú láncolt listával szokás megvalósítani (§17.8[16]).

A lista mindazon tagtípusok és tagfüggvények használatát lehetővé teszi, melyet a vektor (§16.3) esetében megtalálunk, kivéve az indexelést, a *capacity()* és a *reserve()* függvényeket:

```
template <class T, class A = allocator<T> > class std::list {
public:
  // a vector-éhoz hasonló típusok és műveletek, kivéve: [ ], at(), capacity(), és reserve()
  // ...
};
```

17.2.2.1. Áthelyezés, rendezés, összefésülés

Az általános sorozat-műveletek mellett a *list* számos, kifejezetten a listákhoz készített műveleteket kínál:

```
template <class T, class A = allocator<T> > class list {
public:
  // ...
  // kifejezetten listákhoz készített műveletek

  void splice(iterator pos, list& x);                  // x minden elemének áthelyezése
                                                        // a listában levő pos elé, másolás nélkül
  void splice(iterator pos, list& x, iterator p);      // *p áthelyezése x-ből
                                                        // a listában levő pos elé, másolás nélkül
  void splice(iterator pos, list& x, iterator first, iterator last);

  void merge(list&);                                   // rendezett listák összefésülése
  template <class Cmp> void merge(list&, Cmp);

  void sort();
  template <class Cmp> void sort(Cmp);

  // ...
};
```

Ezek a listaműveletek *stabilak* (stable), ami azt jelenti, hogy az egyenértékű értékkel rendelkező elemek egymáshoz viszonyított (relatív) sorrendje nem változik meg.

A §16.3.6 pontban példaképpen a *fruit* tárolón végeztünk műveleteket. Azok a feladatok változtatás nélkül elvégezhetők akkor is, ha a *fruit* történetesen egy lista. Ezenkívül az egyik lista elemeit egyetlen *splice* művelettel át is helyezhetjük egy másik listába. Induljunk ki az alábbi listákból:

fruit:
alma körte

citrus:
narancs grapefruit citrom

A következő művelettel tudjuk a *narancs* elemet áthelyezni a *citrus* listából a *fruit* listába:

```
list<string>::iterator p = find_if(fruit.begin(),fruit.end(),initial('k'));
fruit.splice(p,citrus,citrus.begin());
```

A művelet kiveszi a *citrus* első elemét, majd beilleszti ezt az elemet a *fruit* tároló első *k* betűvel kezdődő eleme elé. Az eredmény tehát a következő:

fruit:
alma narancs körte

citrus:
grapefruit citrom

Jegyezzük meg, hogy a *splice()* nem készít másolatokat az elemekről úgy, mint az *insert()* (§16.3.6), csak a listák adatszerkezetét rendezi át a feladatnak megfelelően.

A *splice()* segítségével nem csak egyetlen elemet, hanem tartományokat, vagy akár teljes listákat is áthelyezhetünk:

```
fruit.splice(fruit.begin(),citrus);
```

Eredménye:

fruit:
grapefruit citrom alma narancs körte

citrus:
<üres>

A *splice* függvény minden változatának a második paraméterben meg kell adnunk azt a *list* objektumot, amelyből az elemeket át kell helyezni. Ez teszi lehetővé, hogy az eredeti listából töröljük az elemet. Egy bejáró erre önmagában nem lenne alkalmas, hiszen egy konkrét elemre mutató bejáróból semmilyen általános módszerrel nem lehet megtudni, hogy az adott elem éppen melyik tárolóban található (§18.6).

Természetesen, a bejáró-paraméternek olyan érvényes bejárót kell tartalmaznia, amely a megfelelő listához kapcsolódik. Tehát vagy a *list* egyik elemére kell mutatnia, vagy a lista *end()* függvényének értékét kell tartalmaznia. Ha ez nem így van, akkor az eredmény nem meghatározható és általában végzetes problémákat okoz:

```
list<string>::iterator p = find_if(fruit.begin(),fruit.end(),initial('k'));
fruit.splice(p,citrus,citrus.begin());    // rendben
fruit.splice(p,citrus,fruit.begin());     // hiba: fruit.begin() nem mutat citrus-ban levő elemre
citrus.splice(p,fruit,fruit.begin());     // hiba: p nem mutat citrus-ban levő elemre
```

Az első *splice()* helyes még akkor is, ha a *citrus* tároló üres.

A *merge()* két rendezett listát egyesít (fésül össze). Az egyik listából törli az elemeket, miközben a másikba beszúrja azokat, a rendezés megőrzésével.

```
f1:
  alma   birsalma   körte
f2:
  citrom   grapefruit   narancs   lime
```

Ez a két lista az alábbi programrészlettel rendezhető és fűzhető össze:

```
f1.sort();
f2.sort();
f1.merge(f2);
```

Az eredmény a következő lesz:

```
f1:
  alma   birsalma   citrom   grapefruit   körte      lime   narancs
f2:
  <üres>
```

Ha az összefésült listák valamelyike nem volt rendezett, a *merge()* akkor is olyan listát eredményez, amelyben a két lista elemeinek uniója szerepel, de az elemek helyes sorrendje nem biztosított.

A *splice()* függvényhez hasonlóan a *merge()* sem másolja az elemeket, hanem a forráslistából kiveszi, majd a céllistába beilleszti azokat. Az *x.merge(y)* függvény meghívása után az *y* lista üres lesz.

17.2.2.2. Műveletek a lista elején

A lista első elemére vonatkozó műveletek azoknak a minden sorozatban megtalálható eljárásoknak a párjai, melyek az utolsó elemre hivatkoznak (§16.3.6):

```
template <class T, class A = allocator<T> > class list {
public:
  // ...
  // hozzáférés elemekhez

  reference front();                    // hivatkozás az első elemre
  const_reference front() const;

  void push_front(const T&);            // új első elem hozzáadása
  void pop_front();                     // első elem eltávolítása

  // ...
};
```

A tároló első elemének neve *front*. A *list* esetében a *front* fejelem műveletei ugyanolyan hatékonyak és kényelmesek, mint a sorozat végét kezelő függvények (§16.3.5). Ha mi dönthetünk, akkor érdemesebb az utolsó elemet használni, mert az így megírt programrészletek a *vector* és a *list* számára is megfelelőek. Így ha csak egy kicsi esély is van arra, hogy az általunk listára használt algoritmust valamikor – általános eljárásként – más tárolótípusokra is alkalmazzuk, akkor mindenképpen az utolsó elem műveleteit használjuk, hiszen ezek jóval szélesebb körben működőképesek. Itt is ahhoz a szabályhoz kell igazodnunk, miszerint a lehető legnagyobb rugalmasság elérése érdekében érdemes a lehető legkisebb művelethalmazt használnunk egy feladat elvégzéséhez (§17.1.4.1).

17.2.2.3. További műveletek

Az elemek beszúrása és törlése rendkívül hatékony művelet a *list* tárolóban. Ez természetesen arra ösztönzi a programozókat, hogy listát használjanak, ha programjaikban az ilyen műveletek gyakoriak. Ezért fontos, hogy az elemek közvetlen törlésére általános, szabványos módszereket adjunk:

```
template <class T, class A = allocator<T> > class list {
public:
  // ...

  void remove(const T& val);
  template <class Pred> void remove_if(Pred p);

  void unique();                // kétszer szereplő elemek eltávolítása == használatával
  template <class BinPred> void unique(BinPred b);     // kétszer szereplő elemek
                                                       // eltávolítása b használatával

  void reverse();               // az elemek sorrendjének megfordítása
};
```

Például tegyük fel, hogy adott az alábbi lista:

fruit:
alma narancs grapefruit citrom narancs görögdinnye körte birsalma

Ekkor a *„narancs"* értékkel rendelkező elemeket a következő paranccsal távolíthatjuk el:

```
fruit.remove("narancs");
```

Az eredmény tehát:

fruit:
alma grapefruit citrom görögdinnye körte birsalma

Gyakran bizonyos feltételt kielégítő elemeket szeretnénk törölni, nem csak egy adott értékkel rendelkezőket. A *remove_if* függvény pontosan ezt a feladatot hajtja végre. A következő utasítás például az összes *'g'* betűvel kezdődő gyümölcsnevet törli a *fruit* listából:

```
fruit.remove_if(initial('g'));
```

A függvény lefutása után a *fruit* a következőképpen néz ki:

fruit:
alma citrom körte birsalma

A törléseknek gyakran az az oka, hogy meg szeretnénk szüntetni az ismétlődéseket. A *unique()* függvény ezzel a céllal szerepel a *list* osztályban:

```
fruit.sort();
fruit.unique();
```

A rendezésre azért van szükség, mert a *unique()* csak azokat az ismétlődéseket szűri ki, amelyeknél közvetlenül egymás után szerepel két (vagy több) azonos érték. Például ha a *fruit* lista tartalma a következő:

alma körte alma alma körte

akkor a *fruit.unique()* hívás önmagában az alábbi eredményt adná:

alma körte alma körte

Ha először rendezünk, akkor a végeredmény:

alma körte

Ha csak bizonyos ismétlődéseket akarunk megszüntetni, akkor megadhatunk egy predikátumot (feltételt), amely meghatározza, mely ismétlődések nem kellenek. Például meghatározhatjuk a kétoperandusú (bináris) *initial2(x)* predikátumot (§18.4.2), amely karakterláncokat vizsgál, és csak akkor ad igaz értéket, ha a karakterlánc *x* betűvel kezdődik. Tehát ha a következő listából indulunk ki:

körte körte alma alma

akkor a következő utasítással el tudjuk tüntetni az összes *'k'* betűvel kezdődő, egymás után következő nevet:

```
fruit.unique(initial2('k'));
```

Az eredmény a következő lesz:

körte alma alma

A §16.3.2 pontban már volt róla szó, hogy a tároló elemeit néha fordított sorrendben akarjuk elérni. A *list* esetében lehetőség van arra, hogy az elemek sorrendjét teljesen megfordítsuk, azaz az utolsó elem váljon elsővé, az első pedig utolsóvá. A *reverse()* függvény ezt a műveletet az elemek másolása nélkül valósítja meg. Vegyük a következő listát:

fruit:
banán cseresznye eper körte

Ekkor a *fruit.reverse()* hívás eredménye a következő lesz:

```
fruit:
  körte eper  cseresznye banán
```

A listából eltávolított elemek teljesen törlődnek. Egy mutató törlése azonban nem jelenti azt, hogy maga a mutatott objektum is törlődik. Ha olyan mutatókból álló tárolót akarunk használni, amely automatikusan törli a belőle kivett mutatók által mutatott objektumokat, akkor azt nekünk kell megírnunk (§17.8[13]).

17.2.3. A deque

A *deque* egy kétvégű sort (double-ended queue) valósít meg. Ez azt jelenti, hogy a *deque* egy olyan sorozat, amit arra optimalizáltak, hogy egyrészt mindkét végén ugyanolyan hatékony műveleteket használhassunk, mint egy *list*-nél, másrészt az indexelés ugyanolyan hatékony legyen, mint a *vector* esetében:

```
template <class T, class A = allocator<T> > class std::deque {
  // a vector-éhoz hasonló típusok és műveletek (§16.3.3, §16.3.5, §16.3.6), kivéve:
  // capacity() és reserve()
  // a list-éhez hasonló fejelem-műveletek (§17.2.2.2)
};
```

Az adatszerkezet belsejében az elemek törlése és beszúrása ugyanolyan (rossz) hatékonyságú, mint a *vector*-nál. Ennek következtében a *deque* tárolót akkor használjuk, ha beszúrások és törlések általában csak a végeken fordulnak elő, például egy vasútszakasz modellezéséhez vagy egy kártyapakli ábrázolásához:

```
deque<car> siding_no_3;
deque<Card> bonus;
```

17.3. Sorozat-átalakítók

A *vector*, a *list* és a *deque* sorozatok nem építhetők fel egymásból komoly hatékonyságromlás nélkül. Ugyanakkor a verem (*stack*) és a sor (*queue*) elegánsan és hatékonyan megvalósítható e három alap-sorozattípus segítségével. Így ezt a két osztályt nem teljesen önálló tárolóként határozták meg, hanem az alaptárolók átalakítóiként.

Egy tároló átalakítója (adapter) egy leszűkített (korlátozott) felületet ad az adott tárolóhoz. A legszembetűnőbb, hogy az átalakítók nem biztosítanak bejárókat, mert a céljuknak megfelelő felhasználáskor a leszűkített felület elegendő szolgáltatást nyújt. Azokat a módszereket, melyekkel egy tárolóból létrehozhatunk egy átalakítót, általánosan használhatjuk olyan esetekben, amikor egy osztály szolgáltatásait a felhasználók igényeihez szeretnénk igazítani, az eredeti osztály megváltoztatása nélkül.

17.3.1. A stack

A *stack* (verem) tárolót a *<stack>* fejállomány írja le. Annyira egyszerű szerkezet, hogy leírásának legegyszerűbb módja, ha bemutatunk egy lehetséges megvalósítást:

```
template <class T, class C = deque<T> > class std::stack {
protected:
  C c;
public:
  typedef typename C::value_type value_type;
  typedef typename C::size_type size_type;
  typedef C container_type;

  explicit stack(const C& a = C()) : c(a) { }

  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }

  value_type& top() { return c.back(); }
  const value_type& top() const { return c.back(); }

  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_back(); }
};
```

Vagyis a *stack* egyszerűen egy felület egy olyan tárolóhoz, melynek típusát sablonparaméterként adjuk meg. A *stack* mindössze annyit tesz, hogy elrejti tárolójának nem verem stílusú műveleteit, valamint a *back()*, *push_back()*, *pop_back()* függvényeket hagyományos nevükön (*top()*, *push()*, *pop()*) teszi elérhetővé.

Alapértelmezés szerint a *stack* egy *deque* tárolót használ elemeinek tárolására, de bármilyen sorozatot használhatunk, melyben elérhető a *back()*, a *push_back()* és a *pop_back()* függvény:

```
stack<char> s1;                // char típusú elemek tárolása deque<char> segítségével
stack< int,vector<int> > s2;   // int típusú elemek tárolása vector<int> segítségével
```

Egy verem kezdeti feltöltéséhez felhasználhatunk egy már létező tárolót is:

```
void print_backwards(vector<int>& v)
{
  stack< int,vector<int> > state(v);       // kezdőállapot beállítása v segítségével
  while (state.size()) {
            cout << state.top();
            state.pop();
  }
}
```

De gondoljunk rá, hogy ez a művelet a paraméterként megadott tároló elemeinek másolásával jár, így rendkívül hosszadalmas lehet.

A veremhez az elemek tárolásához használt tároló *push_back()* műveletével adunk elemeket. Így a *stack* nem telhet meg mindaddig, amíg a tároló képes memóriát lefoglalni (memóriafoglalójának segítségével, §19.4).

Ugyanakkor a verem alulcsordulhat:

```
void f()
{
  stack<int> s;
  s.push(2);
  if (s.empty()) {                 // az alulcsordulás megakadályozható
            // nincs kiemelés
  }
  else {                           // de nem elképzelhetetlen
            s.pop();               // jó: s.size() értéke 0 lesz
            s.pop();               // nem meghatározott hatás, valószínűleg rossz
  }
}
```

Jegyezzük meg, hogy a felső elem használatához nincs szükségünk a *pop()* függvényre. Erre a *top()* utasítás szolgál, a *pop()* műveletre pedig akkor van szükség, ha el akarjuk távolítani a legfelső elemet. Ez a megoldás nem túlságosan kényelmetlen, és sokkal hatékonyabb, amikor a *pop()* műveletre nincs szükség:

```
void f(stack<char>& s)
{
  if (s.top()=='c') s.pop();      // nem kötelező kezdő 'c' eltávolítása
  // ...
}
```

Az önállóan, teljesen kifejlesztett tárolókkal ellentétben a veremnek (és a többi tároló-átalakítónak) nem lehet memóriafoglalót megadni sablonparaméterként, azt a *stack* megvalósításához használt tároló saját memóriafoglalója helyettesíti.

17.3.2. A queue

A *<queue>* fejállományban leírt *queue* (sor) olyan felület egy tárolóhoz, amely lehetővé teszi az elemek beszúrását az adatszerkezet végére, illetve kivételét a tároló elejéről:

```
template <class T, class C = deque<T> > class std::queue {
protected:
   C c;
public:
   typedef typename C::value_type value_type;
   typedef typename C::size_type size_type;
   typedef C container_type;

   explicit queue(const C& a = C()) : c(a) { }

   bool empty() const { return c.empty(); }
   size_type size() const { return c.size(); }

   value_type& front() { return c.front(); }
   const value_type& front() const { return c.front(); }

   value_type& back() { return c.back(); }
   const value_type& back() const { return c.back(); }

   void push(const value_type& x) { c.push_back(x); }
   void pop() { c.pop_front(); }
};
```

Alapértelmezés szerint a *queue* a *deque* tárolót használja elemeinek tárolásához, de bármely sorozat betöltheti ezt a szerepet, amely rendelkezik a *front()*, *back()*, *push_back()* és *pop_front()* függvényekkel. Mivel a *vector* nem teszi lehetővé a *pop_front()* használatát, a vektor nem lehet a sor belső tárolója.

A sor szerkezet szinte minden rendszerben előfordul valamilyen formában. Egy egyszerű üzenetalapú kiszolgálót például a következőképpen határozhatunk meg:

```
struct Message {
  // ...
};

void server(queue<Message>& q)
{
  while(!q.empty()) {
        Message& m = q.front();     // üzenet elfogása
        m.service();                // a kérést kiszolgáló függvény meghívása
        q.pop();                    // üzenet törlése
  }
}
```

Az üzeneteket a *push()* utasítás segítségével helyezhetjük a sorba.

Ha az ügyfél (kliens) és a kiszolgáló (szerver) külön-külön folyamatként vagy szálként fut, akkor a sor-hozzáféréseket valamilyen módon össze kell hangolnunk (szinkronizálás):

```
void server2(queue<Message>& q, Lock& lck)
{
  while(!q.empty()) {
        Message m;
        {     LockPtr h(lck);     // zárolás az üzenet kinyerése közben (§14.4.1)
              if (q.empty()) return;     // valaki más már megszerezte az üzenetet
              m = q.front();
              q.pop();
        }
        m.service();                // a kérést kiszolgáló függvény meghívása
  }
}
```

Az egyidejű hozzáférésnek (konkurrencia, párhuzamosság), illetve a zárolásnak (lock) még nincs szabványos definíciója sem a C++ nyelvben, sem a számítástechnika világában. Ha ezeket a lehetőségeket szeretnénk használni, akkor járjunk utána, hogy saját rendszerünk milyen lehetőségeket biztosít, és azokat hogyan érhetjük el a C++ nyelvből (§17.8[8]).

17.3.3. A priority_queue

A *priority_queue* (prioritásos sor) egy olyan sor, melyben az elemek fontossági értéket kapnak, és ez az érték szabályozza, hogy az elemek milyen sorrendben vehetők ki:

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type> >
class std::priority_queue {
protected:
  C c;
  Cmp cmp;
public:
  typedef typename C::value_type value_type;
  typedef typename C::size_type size_type;
  typedef C container_type;

  explicit priority_queue(const Cmp& a1 = Cmp(), const C& a2 = C())
        : c(a2), cmp(a1) { make_heap(c.begin(),c.end(),cmp); }        // lásd §18.8
  template <class In>
  priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }

  const value_type& top() const { return c.front(); }

  void push(const value_type&);
  void pop();
};
```

A *priority_queue*-t a *<queue>* fejállomány deklarálja.

Alapértelmezés szerint a *priority_queue* az elemeket egyszerűen a < operátor segítségével hasonlítja össze, és a *top()* mindig a legnagyobb elemet adja vissza:

```
struct Message {
  int priority;
  bool operator<(const Message& x) const { return priority < x.priority; }
  // ...
};

void server(priority_queue<Message>& q, Lock& lck)
{
  while(!q.empty()) {
        Message m;
        {     LockPtr h(lck);     // zárolás az üzenet kinyerése közben (§14.4.1)
              if (q.empty()) return;     // valaki más már megszerezte az üzenetet
              m = q.top();
              q.pop();
        }
        m.service();     // a kérést kiszolgáló függvény meghívása
  }
}
```

Ez a programrészlet abban különbözik a sornál (*queue*, §17.3.2) bemutatott példától, hogy itt az üzenetek közül a legfontosabb („legnagyobb prioritású") kerül először feldolgozásra. Az nem meghatározott, hogy az azonos fontossági értékkel rendelkező elemek közül melyik jelenik meg először a sor elején. Két elem fontossága (prioritása) akkor egyezik meg, ha egyik sem „fontosabb" a másiknál (§17.4.1.5).

Ha a < operátor nem felel meg céljainknak, akkor egy sablonparaméter segítségével megadhatjuk az összehasonlító műveletet. Például karakterláncokat rendezhetünk a kis- és nagybetűk megkülönböztetése nélkül, ha az alábbi sorban helyezzük el azokat:

```
priority_queue<string, vector<string>, Nocase> pq;   // Nocase használata az
                                                     // összehasonlításokhoz (§17.1.4.1)
```

A karakterláncokat a *pq.push()* utasítással tesszük be a sorba és a *pq.top()*, *pq.pop()* műveletekkel dolgozhatjuk fel.

Az olyan sablonokból létrehozott objektumok, melyeknél különböző sablonparamétereket használtunk, különböző típusúak (§13.6.3.1):

```
priority_queue<string>& pq1 = pq;   // hiba: nem megfelelő típus
```

Összehasonlítási szempontot megadhatunk úgy is, hogy közben nem változtatjuk meg a prioritásos sor típusát. Ehhez egy megfelelő típusú összehasonlító objektumot kell létrehoznunk, melynek konstruktor-paraméterében megadjuk az érvényes összehasonlítási szempontot:

```
struct String_cmp {              // futási idejű összehasonlítási feltételt kifejező típus
   String_cmp(int n = 0);        // az n összehasonlítási feltétel használata
   // ...
};

typedef priority_queue<string, vector<string>, String_cmp> Pqueue;

void g(Pqueue& pq)               // a pq a String_cmp()-et használja az összehasonlításokhoz
{
   Pqueue pq2(String_cmp(nocase));
   pq = pq2;                     // rendben: pq és pq2 azonos típusú, így pq most
                                 // a String_cmp(nocase)-t shasználja
}
```

Az elemek rendezése némi teljesítményromlással jár, de ez egyáltalán nem jelentős. A *priority_queue* egyik hatékony megvalósítási módja a faszerkezet, mellyel az elemek egy-

máshoz viszonyított helyét könnyedén beállíthatjuk. Ezzel a megoldással a *push()* és a *pop()* művelet költsége is *O(log(n))*.

Alapértelmezés szerint a *priority_queue* egy *vector* tárolót használ elemeinek nyilvántartására, de bármely sorozattípus megfelel e célra, amely rendelkezik a *front()*, *push_back()* és *pop_back()* függvényekkel, valamint lehetővé teszi közvetlen elérésű bejárók használatát. A prioritásos sorok leggyakrabban használt megvalósítási eszköze a *heap* (kupac vagy halom, §18.8)

17.4. Asszociatív tárolók

Az *asszociatív tömb* az egyik leghasznosabb felhasználói típus. Ennek következtében az elsősorban szöveg- vagy szimbólum-feldolgozásra kifejlesztett nyelvekben gyakran beépített típusként szerepel. Az asszociatív tömb, melynek gyakori elnevezése a *map* (hozzárendelés, leképezés) és a *dictionary* (szótár) is, értékpárokat tárol. Az egyik érték a *kulcs* (key), melyet a másik érték (a *hozzárendelt érték*, mapped value) eléréséhez használunk. Az asszociatív tömböt úgy képzelhetjük el, mint egy tömböt, melynek indexe nem feltétlenül egy egész szám:

```
template<class K, class V> class Assoc {
public:
   V& operator[ ](const K&);   // hivatkozás visszaadása a K-nak megfelelő V-re
   // ...
};
```

Itt a *K* típusú kulcs segítségével a *V* típusú hozzárendelt értéket választjuk ki.

Az asszociatív tároló az asszociatív tömb fogalmának általánosítása. A *map* sablon a hagyományos asszociatív tömbnek felel meg, ahol minden kulcsértékhez egyetlen értéket rendelünk. A *multimap* egy olyan asszociatív tömb, amely megengedi, hogy ugyanolyan kulcsértékkel több elem is szerepeljen, míg a *set* és a *multiset* olyan egyedi asszociatív tömbök, melyekben nincs hozzárendelt érték.

17.4.1. A map

A *map* (kulcs,érték) párok sorozata, melyben a bejegyzéseket a kulcs (key) alapján gyorsan elérhetjük. A *map* tárolóban a kulcsok egyediek, azaz minden kulcshoz legfeljebb egy értéket rendelünk hozzá. A *map* szerkezetben kétirányú bejárókat használhatunk (§19.2.1).

A *map* megköveteli, hogy a kulcsként használt típusokban a „kisebb mint” művelet használható legyen (§17.1.4.1), és ez alapján az elemeket rendezve tárolja. Ennek következtében a *map* bejárásakor az elemeket rendezve kapjuk meg. Ha olyan elemeink vannak, melyek között nem lehet egyértelmű sorrendet megállapítani, vagy nincs szükség arra, hogy az elemeket rendezve tároljuk, akkor használjuk inkább a *hash_map* szerkezetet (§17.6).

17.4.1.1. Típusok

A *map* a tárolók szokásos típus tagjain (§16.3.1) kívül néhány, az osztály egyedi céljának megfelelő típust is meghatároz:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class std::map {
public:
  // típusok

  typedef Key key_type;
  typedef T mapped_type;

  typedef pair<const Key, T> value_type;

  typedef Cmp key_compare;
  typedef A allocator_type;

  typedef typename A::reference reference;
  typedef typename A::const_reference const_reference;

  typedef megvalósítás_függő1 iterator;
  typedef megvalósítás_függő2 const_iterator;

  typedef typename A::size_type size_type;
  typedef typename A::difference_type difference_type;

  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
  // ...
};
```

Jegyezzük meg, hogy a *value_type* a *map* esetében a (kulcs, érték) *párt (*pair*)* jelenti. A hozzárendelt értékek típusát a *mapped_type* adja meg. Tehát a *map* nem más, mint *pair<const Key, mapped_type>* típusú elemek sorozata. Szokás szerint a konkrét bejáró-típusok az adott megvalósítástól függőek. Mivel a *map* tárolót leggyakrabban valamilyen faszerkezet segítségével valósítják meg, a bejárók általában biztosítanak valamilyen fabejárást.

A visszafelé haladó bejárókat a szabványos *reverse_iterator* sablonok (§19.2.5) segítségével határozhatjuk meg.

17.4.1.2. Bejárók és párok

A *map* a szokásos függvényeket biztosítja a bejárók eléréséhez (§16.3.2):

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > > class map {
public:
  // ...
  // bejárók

  iterator begin();
  const_iterator begin() const;

  iterator end();
  const_iterator end() const;

  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;

  reverse_iterator rend();
  const_reverse_iterator rend() const;

  // ...
};
```

A *map* bejárásakor *pair<const Key, mapped_type>* típusú elemek sorozatán haladunk végig. Például egy telefonkönyv bejegyzéseit az alábbi eljárással írathatjuk ki:

```
void f(map<string,number>& phone_book)
{
  typedef map<string,number>::const_iterator CI;
  for (CI p = phone_book.begin(); p!=phone_book.end(); ++p)
          cout << p->first << '\t' << p->second << '\n';
}
```

A *map* bejárói az elemeket kulcs szerint növekvő sorrendben adják vissza (§17.1.4.5), így a *phone_book* bejegyzései ábécésorrendben jelennek meg. Egy *pair* első elemére a *first*, második elemére a *second* névvel hivatkozhatunk, függetlenül attól, hogy éppen milyen típusúak ezek az elemek:

```
template <class T1, class T2> struct std::pair {
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;

  pair() :first(T1()), second(T2()) { }
  pair(const T1& x, const T2& y) :first(x), second(y) { }
  template<class U, class V>
          pair(const pair<U, V>& p) :first(p.first), second(p.second) { }
};
```

Az utolsó konstruktorra azért van szükség, hogy a párokon típuskonverziót is végezhessünk (§13.6.2):

```
pair<int,double> f(char c, int i)
{
  pair<char,int> x(c,i);
  // ...
  return x;           // pair<char,int>-ről pair<int,double>-ra való konverzió szükséges
.. return pair<int,double>(c,i);        // konverzió szükséges
}
```

A *map* esetében a pár első tagja a kulcs, a második tagja a hozzárendelt érték.

A *pair* adatszerkezet nem csak a *map* megvalósításában használható, így ez is egy önálló osztály a standard könyvtárban. A *pair* leírását a *<utility>* fejállományban találhatjuk meg. A *pair* típusú változók létrehozását könnyíti meg a következő függvény:

```
template <class T1, class T2> pair<T1,T2> std::make_pair(T1 t1, T2 t2)
{
  return pair<T1,T2>(t1,t2);
}
```

A *pair* alapértelmezett kezdőértékeit két elemtípusának alapértelmezett értéke adja. Fontos, hogy ha az elemek típusa beépített, akkor a kezdőérték *0* (§5.1.1), karakterláncok esetében pedig egy „üres" karakterlánc (§20.3.4). Olyan típus, melynek nincs alapértelmezett konstruktora, csak akkor lehet egy *pair* eleme, ha a *pair* kezdőértékét kifejezetten megadjuk.

17.4.1.3. Indexelés

A legjellemzőbb *map*-művelet a hozzárendeléses (asszociációs) keresés, amit az indexelő operátor valósít meg:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
  // ...
  mapped_type& operator[ ](const key_type& k); // a k kulcsú elem elérése
  // ...
};
```

Az indexelő operátor a megadott kulcsot indexnek tekintve egy keresést hajt végre és visszaadja a kulcshoz rendelt értéket. Ha a kulcs nem található meg a tárolóban, akkor ezzel a kulccsal és a *mapped_type* alapértelmezett értékével egy új elem kerül az asszociatív tömbbe:

```
void f()
{
  map<string,int> m;           // a map kezdetben üres
  int x = m["Henry"];          // új bejegyzés készítése ("Henry"); a kezdőérték 0, a
                               // visszaadott érték 0
  m["Harry"] = 7;              // új bejegyzés készítése ("Harry"), a kezdőérték 0, kapott érték 7
  int y = m["Henry"];          // a "Henry" bejegyzés értékének visszaadása
  m["Harry"] = 9;              // "Harry" értékének módosítása 9-re
}
```

Egy kicsit valóságszerűbb példa: képzeljünk el egy programot, amely kiszámítja a bemenetén megadott tárgyak darabszámát. A listában (név, mennyiség) bejegyzések szerepelnek. Az összesítést el szeretnénk végezni tárgyanként és az egész listára is. A lista lehet például a következő:

szög 100 kalapács 2 fűrész 3 fűrész 4 kalapács 7 szög 1000 szög 250

A feladat nagy részét elvégezhetjük, miközben a (név, mennyiség) párokat beolvassuk egy *map* tárolóba:

```
void readitems(map<string,int>& m)
{
  string word;
  int val = 0;
  while (cin >> word >> val) m[word] += val;
}
```

Az *m[word]* indexelés azonosítja a megfelelő *(string, int)* párt, és visszaadja az *int* értéket. A fenti programrészletben kihasználjuk azt is, hogy az új elemek *int* értéke alapértelmezés szerint *0*.

A *readitems()* függvénnyel felépített *map* egy szokásos ciklus segítségével jeleníthető meg:

```
int main()
{
   map<string,int> tbl;
   readitems(tbl);

   int total = 0;
   typedef map<string,int>::const_iterator CI;

   for (CI p = tbl.begin(); p!=tbl.end(); ++p) {
         total += p->second;
         cout << p->first << '\t' << p->second << '\n';
   }

   cout << "----------------\nösszesen\t" << total << '\n';

   return !cin;
}
```

A fenti bemenettel a függvény eredménye a következő:

```
fűrész    7
kalapács  9
szög      1350
-----------------------
összesen  1366
```

Figyeljük meg, hogy a nevek ábécésorrendben jelennek meg (§17.4.1, §17.4.1.5).

Az indexelő operátornak meg kell találnia a megadott kulcsértéket a tárolóban. Ez természetesen nem olyan egyszerű művelet, mint a tömbök esetében az egész értékkel való indexelés. A pontos költség: *O(log(a_map_mérete))*. Ez sok alkalmazás esetében még elfogadható, ha azonban a mi céljainknak túl drága, érdemesebb hasító tárolót (§17.6) használnunk. Ha a *map* tárolóban nem található meg egy kulcs, akkor az erre vonatkozó indexeléssel létrehozunk egy alapértelmezett elemet. Ennek következtében a *const map* osztályok esetében nem használhatjuk az *operator[ ]()* műveletet. Ráadásul az indexelés csak akkor használható, ha a *mapped_type* típusnak van alapértelmezett értéke. Ha csak arra vagyunk kíváncsiak, hogy egy adott kulcs megtalálható-e a tárolóban, akkor használjuk a *find()* műveletet (§17.4.1.6), ami a *map* megváltoztatása nélkül keresi meg a kulcsot.

17.4.1.4. Konstruktorok

A *map* a szokásos konstruktorokat és egyéb függvényeket biztosítja (§16.3.4):

```
template <class Key, class T, class Cmp =less<Key>,
          class A =allocator<pair<const Key,T> > >
class map {
public:
  // ...
  // létrehozás/másolás/megsemmisítés:

  explicit map(const Cmp& = Cmp(), const A& = A());
  template <class In> map(In first, In last, const Cmp& = Cmp(), const A& = A());
  map(const map&);

  ~map();

  map& operator=(const map&);

  // ...
};
```

A tároló lemásolása azt jelenti, hogy helyet foglalunk az elemeknek, majd mindegyikről másolatot készítünk (§16.3.4). Ez nagyon költséges művelet is lehet, ezért csak akkor használjuk, ha elkerülhetetlen. Ebből következik, hogy az olyan tárolókat, mint a *map*, csak referencia szerint érdemes átadni.

A sablon konstruktor *pair<const Key, T>* elemek sorozatát kapja paraméterként, amelyet egy bemeneti bejáró-párral adunk meg. A függvény a sorozat elemeit az *insert()* művelet segítségével szúrja be az asszociatív tömbbe.

17.4.1.5. Összehasonlítások

Ahhoz, hogy egy adott kulcsú elemet megtaláljunk egy *map*-ben, a *map* műveleteinek össze kell tudnia hasonlítani a kulcsokat. A bejárók is a kulcsok növekvő értékei szerint haladnak végig a tárolón, így a beszúrások is kulcs-összehasonlításokat végeznek (ahhoz, hogy az elemeket elhelyezzék a *map* tárolót ábrázoló faszerkezetben).

Alapértelmezés szerint a kulcsok összehasonlításához használt művelet a < (kisebb mint) operátor, de egy sablon- vagy konstruktor-paraméterben más függvényt is megadhatunk (§17.3.3). Az itt megadott rendezési szempont a kulcsokat hasonlítja össze, míg a *map* ese-

tében a *value_type* (kulcs, érték) párokat jelent. Ezért van szükség a *value_comp()* függvényre, amely a kulcsokat összehasonlító eljárás alapján a párokat hasonlítja össze:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
  // ...

  typedef Cmp key_compare;

  class value_compare : public binary_function<value_type,value_type,bool> {
  friend class map;
  protected:
          Cmp cmp;
          value_compare(Cmp c) : cmp(c) {}
  public:
          bool operator()(const value_type& x, const value_type& y) const
                  { return cmp(x.first, y.first); }
  };

  key_compare key_comp() const;
  value_compare value_comp() const;
  // ...
};
```

Például:

```
map<string,int> m1;
map<string,int,Nocase> m2;            // összehasonlítás típusának megadása (§17.1.4.1)
map<string,int,String_cmp> m3;        // összehasonlítás típusának megadása (§17.1.4.1)
map<string,int,String_cmp> m4(String_cmp(literary));    // összehasonlítandó objektum
                                                        // átadása
```

A *key_comp()* és a *value_comp()* tagfüggvény lehetővé teszi, hogy az asszociatív tömbben az egész elemekre a csak kulcsokra, illetve a csak értékekre vonatkozó összehasonlító műveleteket használjuk. Erre legtöbbször akkor van szükség, ha ugyanazt az összehasonlítást szeretnénk használni egy másik tárolóban vagy algoritmusban is:

```
void f(map<string,int>& m)
{
  map<string,int> mm;        // összehasonlítás alapértelmezés szerint < operátorral
  map<string,int> mmm(m.key_comp());         // m szerinti összehasonlítás
  // ...
}
```

A §17.1.4.1 pontban példát láthatunk arra, hogyan adhatunk meg egyedi összehasonlításokat, a §18.4 pontban pedig a függvényobjektumok általános bemutatásával foglalkozunk.

17.4.1.6. Műveletek asszociatív tömbökkel

A *map* és természetesen az összes asszociatív tároló legfontosabb tulajdonsága, hogy egy kulcs alapján férhetünk hozzá az információkhoz. Ezen cél megvalósításához számos egyedi függvény áll rendelkezésünkre:

```
template <class Key, class T, class Cmp = less<Key>,
            class A = allocator< pair<const Key,T> > >
class map {
public:
   // ...
   // map-műveletek

   iterator find(const key_type& k);               // a k kulcsú elem megkeresése
   const_iterator find(const key_type& k) const;

   size_type count(const key_type& k) const;   // a k kulcsú elemek számának
                                                // meghatározása

   iterator lower_bound(const key_type& k);   // az első k kulcsú elem megkeresése
   const_iterator lower_bound(const key_type& k) const;
   iterator upper_bound(const key_type& k);   // az első k-nál nagyobb kulcsú elem
                                               // megkeresése
   const_iterator upper_bound(const key_type& k) const;

   pair<iterator,iterator> equal_range(const key_type& k);
   pair<const_iterator,const_iterator> equal_range(const key_type& k) const;

   // ...
};
```

Az *m.find(k)* művelet egyszerűen visszaad egy *k* kulcsú elemre hivatkozó bejárót. Ha ilyen elem nincs, akkor a visszaadott bejáró az *m.end()*. Egy egyedi kulcsokkal rendelkező tároló esetében (mint a *map* és a *set*) az eredmény az egyetlen *k* kulcsú elemre mutató bejáró lesz. Ha a tároló nem teszi kötelezővé egyedi kulcsok használatát (mint a *multimap* és a *multiset*), akkor a visszaadott bejáró az első megfelelő kulcsú elem lesz:

```
void f(map<string,int>& m)
{
   map<string,int>::iterator p = m.find("Arany");
```

```
    if (p!=m.end()) {                                   // ha "Arany"-at találtunk
            // ...
    }
    else if (m.find("Ezüst")!=m.end()) {                // "Ezüst" keresése
            // ...
    }
    // ...
}
```

Egy *multimap* (§17.4.2) esetében az első *k* kulcsú elem megkeresése helyett általában az összes ilyen elemre szükségünk van. Az *m.lower_bound(k)* és az *m.upper_bound(k)* függvényekkel az *m k* kulcsú elemeiből álló részsorozat elejét, illetve végét kérdezhetjük le. Szokás szerint, a sorozat végét jelző bejáró az utolsó utáni elemre mutat:

```
void f(multimap<string,int>& m)
{
    multimap<string,int>::iterator lb = m.lower_bound("Arany");
    multimap<string,int>::iterator ub = m.upper_bound("Arany");

    for (multimap<string,int>::iterator p = lb; p!=ub; ++p) {
            // ...
    }
}
```

Az alsó és a felső határ meghatározása két külön művelettel nem túl elegáns és nem is hatékony. Ezért áll rendelkezésünkre az *equal_range()* függvény, amely mindkét értéket visszaadja:

```
void f(multimap<string,int>& m)
{
    typedef multimap<string,int>::iterator MI;
    pair<MI,MI> g = m.equal_range("Arany");

    for (MI p = g.first; p!=g.second; ++p) {
            // ...
    }
}
```

Ha a *lower_bound(k)* nem találja meg a *k* kulcsot, akkor az első olyan elemre hivatkozó bejárót adja vissza, melynek kulcsa nagyobb, mint *k*, illetve az *end()* bejárót, ha nem létezik *k*-nál nagyobb kulcs. Ugyanezt a hibajelzési módot használja az *upper_bound()* és az *equal_range()* is.

17.4.1.7. Listaműveletek

Ha egy asszociatív tömbbe új értéket szeretnénk bevinni, akkor a hagyományos megoldás az, hogy egyszerű indexeléssel értéket adunk az adott kulcsú elemnek:

```
phone_book["Rendelési osztály"] = 8226339;
```

Ez a sor biztosítja, hogy a *phone_book* tárolóban meglesz a *Rendelési osztály* bejegyzés a megfelelő értékkel, függetlenül attól, hogy korábban létezett-e már ilyen kulcs. Elemeket beilleszthetünk közvetlenül az *insert()* függvény segítségével is, és az *erase()* szolgál az elemek törlésére:

```
template <class Key, class T, class Cmp = less<Key>,
            class A = allocator< pair<const Key,T> > >
class map {
public:
  // ...
  // listaműveletek

  pair<iterator, bool> insert(const value_type& val);         // (kulcs,érték) pár beszúrása
  iterator insert(iterator pos, const value_type& val);       // a pos csak javaslat
  template <class In> void insert(In first, In last);         // elemek beszúrása sorozatból

  void erase(iterator pos);                      // a mutatott elem törlése
  size_type erase(const key_type& k);            // a k kulcsú elem törlése (ha van ilyen)
  void erase(iterator first, iterator last);     // tartomány törlése
  void clear();                                  // minden elem törlése

  // ...
};
```

Az *m.insert(val)* függvény beszúrja a *val* értéket, amely egy *(Key, T)* párt ad meg. Mivel a *map* egyedi kulcsokkal foglalkozik, a beszúrásra csak akkor kerül sor, ha az *m* tárolóban még nincs ilyen kulccsal rendelkező elem. Az *m.insert(val)* visszatérési értéke egy *pair<iterator,bool>*. A pár második, logikai tagja akkor igaz, ha a *val* érték ténylegesen bekerült a tárolóba. A bejáró az *m* azon elemét jelöli ki, melynek kulcsa megegyezik a *val* kulcsával (*val.first*):

```
void f(map<string,int>& m)
{
  pair<string,int> p99("Pali",99);

  pair<map<string,int>::iterator,bool> p = m.insert(p99);
  if (p.second) {
```

```
                // "Pali"-t beszúrtuk
        }
        else {
                // "Pali" már szerepelt
        }
        map<string,int>::iterator i = p.first;    // points to m["Pali"]
        // ...
}
```

Általában nem érdekel minket, hogy a kulcs korábban már szerepelt-e a *map*-ben vagy új elemként került be. Ha mégis erre vagyunk kíváncsiak, annak oka általában az, hogy a kívánt kulcs egy másik *map* tárolóba is kerülhet az általunk vizsgált helyett, és ezt észre kell vennünk. Az *insert()* másik két változata nem jelzi, hogy az új érték tényleg bekerült-e a tárolóba.

Ha az *insert(pos,val)* forma szerint egy pozíciót is megadunk, akkor csak egy ajánlást adunk, hogy a rendszer a *val* kulcsának keresését a *pos* pozíciótól kezdje. Ha az ajánlás helyes, jelentős teljesítménynövekedést érhetünk el. Ha nem tudunk jó ajánlást adni, használjuk inkább az előző változatot, mind az olvashatóság, mind a hatékonyság érdekében:

```
void f(map<string,int>& m)
{
    m["Dilbert"] = 3;                          // elegáns, de valószínűleg kevésbé hatékony
    m.insert(m.begin(),make_pair(const string("Dogbert"),99));        // csúnya
}
```

Valójában a *[ ]* egy kicsit több, mint egyszerűen az *insert()* kényelmesebb alakja. Az *m[k]* eredménye a következő kifejezéssel egyenértékű:

```
(*(m.insert(make_pair(k,V())).first)).second
```

ahol a *V()* a hozzárendelt érték alapértelmezett értékét jelöli. Ha ezt az egyenértékűséget megértettük, akkor valószínűleg már értjük az asszociatív tárolókat is. Mivel a *[ ]* mindenképpen használja a *V()* értéket, nem használhatjuk az indexelést olyan *map* esetében, melynek értéktípusa nem rendelkezik alapértelmezett értékkel. Ez egy sajnálatos hiányossága az asszociatív tárolóknak, annak ellenére, hogy az alapértelmezett érték megléte nem alapvető követelményük (ld. §17.6.2). Az elemek törlését is kulcs szerint végezhetjük el:

```
void f(map<string,int>& m)
{
    int count = m.erase("Ratbert");
    // ...
}
```

A visszaadott egész érték a törölt elemek számát adja meg. Tehát a *count* tartalma *0*, ha nincs *„Ratbert"* kulcsú elem a tárolóban. A *multimap* és a *multiset* esetében a visszaadott érték egynél nagyobb is lehet. Egy elemet egy rá mutató bejáró segítségével törölhetünk, elemek sorozatát pedig a megfelelő tartomány kijelölésével:

```
void g(map<string,int>& m)
{
  m.erase(m.find("Catbert"));
  m.erase(m.find("Alice"),m.find("Wally"));
}
```

Természetesen gyorsabban lehet törölni egy olyan elemet, melynek már megtaláltuk a bejáróját, mint egy olyat, melyet először a kulcs alapján meg kell keresnünk. Az *erase()* után a bejárót tovább már nem használhatjuk, mivel az általa mutatott elem megsemmisült. Az *m.erase(b,e)* hívása, ahol *e* értéke *m.end()* veszélytelen (*b* vagy *m* valamelyik elemére hivatkozik, vagy *m.end()*). Ugyanakkor az *m.erase(p)* meghívása, amennyiben *p* értéke *m.end()*, súlyos hiba és tönkreteheti a tárolót.

17.4.1.8. További műveletek

A *map* biztosítja azokat a szokásos függvényeket, melyek az elemek számával foglalkoznak, illetve egy specializált *swap()* függvényt:

```
template <class Key, class T, class Cmp = less<Key>,
            class A = allocator< pair<const Key,T> > >
class map {
public:
  // ...
  // kapacitás:

  size_type size() const;              // elemek száma
  size_type max_size() const;          // a lehetséges legnagyobb map mérete
  bool empty() const { return size()==0; }

  void swap(map&);
};
```

Szokás szerint a *size()* és a *max_size()* által visszaadott érték valamilyen elemszám.

Ezenkívül a *map* használatakor rendelkezésünkre állnak az összehasonlító operátorok (==, !=, <, >, <=, >=), valamint a *swap()* eljárás is, ezek azonban nem tagfüggvényei a *map* osztálynak:

```
template <class Key, class T, class Cmp, class A>
bool operator==(const map<Key,T,Cmp,A>&, const map<Key,T,Cmp,A>&);

// ugyanígy !=, <, >, <=, and >=

template <class Key, class T, class Cmp, class A>
void swap(map<Key,T,Cmp,A>&, map<Key,T,Cmp,A>&);
```

Miért lehet szükség arra, hogy két asszociatív tömböt összehasonlítsunk? Ha két *map* objektumot hasonlítunk össze, általában nem csak azt akarjuk megtudni, hogy egyenlőek-e, hanem azt is, hogy miben különböznek, ha különböznek. Ilyenkor tehát az *==* és a *!=* nem használható. Az *==*, a *<* és a *swap()* megvalósításával azonban lehetővé válik, hogy olyan algoritmusokat készítsünk, melyek minden tárolóra alkalmazhatók. Ezekre a függvényekre például akkor lehet szükségünk, ha asszociatív tömbökből álló vektort szeretnénk rendezni vagy *map* tárolók halmazára van szükségünk.

17.4.2. A multimap

A *multimap* nagyon hasonlít a *map* tárolóra, azzal a kivétellel, hogy ugyanaz a kulcs többször is szerepelhet:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class std::multimap {
public:
  // mint a map, kivéve a következőt:

  iterator insert(const value_type&);     // bejárót ad vissza, nem párt

  // nincs indexelő operátor ([ ])
};
```

Például (felhasználva a §17.1.4.1 pontban C stílusú karakterláncok összehasonlításához bemutatott *CString_less* osztályt):

```
void f(map<char*,int,Cstring_less>& m, multimap<char*,int,Cstring_less>& mm)
{
  m.insert(make_pair("x",4));
  m.insert(make_pair("x",5));// nincs hatása: "x" már szerepel (§17.4.1.7)
  // most m["x"] == 4
```

```
    mm.insert(make_pair("x",4));
    mm.insert(make_pair("x",5));
    // mm most ("x",4)-et és ("x",5)-öt is tartalmazza
}
```

Tehát a *multimap* tárolóban nem lehet olyan indexelést megvalósítani, mint a *map* esetében. Itt az *equal_range()*, a *lower_bound()* és az *upper_bound()* művelet (§17.4.1.6) jelenti az azonos kulcsú elemek elérésére szolgáló legfőbb eszközt.

Természetesen ha ugyanahhoz a kulcshoz több értéket is hozzárendelhetünk, akkor a *map* helyett feltétlenül a *multimap* szerkezetet használjuk. Ez pedig sokkal gyakrabban előfordul, mint gondolnánk. Bizonyos tekintetben a *multimap* sokkal tisztább és elegánsabb megoldást ad, mint a *map*.

Mivel egy embernek több telefonszáma is lehet, a telefonkönyvet is érdemesebb *multimap* formában elkészíteni. Saját telefonszámaimat például a következő programrészlettel jeleníthetem meg:

```
void print_numbers(const multimap<string,int>& phone_book)
{
    typedef multimap<string,int>::const_iterator I;
    pair<I,I> b = phone_book.equal_range("Stroustrup");
    for (I i = b.first; i != b.second; ++i) cout << i->second << '\n';
}
```

A *multimap* esetében az *insert()* mindig beilleszti a paramétereként megadott elemet, így nincs szükség a *pair<iterator, bool>* típusú visszatérési értékre; a *multimap::insert()* egyetlen bejárót ad vissza. Az egységesség érdekében a könyvtár tartalmazhatná az *insert()* általános formáját, mind a *map*-hez, mind a *multimap*-hez, annak ellenére, hogy a *bool* rész felesleges a *multimap* osztályban. Egy másik tervezési szemlélet szerint elkészíthették volna az egyszerű *insert()* függvényt mindkét tárolóhoz, amely nem ad vissza logikai értéket. Ez esetben viszont a *map* felhasználóinak valamilyen más lehetőséget kellett volna biztosítani annak megállapításához, hogy új kulcs került-e a tárolóba. Ebben a kérdésben a különböző felülettervezési elméletek ütköztek, és nem született megegyezés.

17.4.3. A set

A *set* (halmaz) olyan egyedi *map* (§17.4.1) tárolónak tekinthető, ahol a hozzárendelt értékek érdektelenek, így csak a kulcsokat tartjuk meg. Ez a felületnek csupán apró módosítását jelenti:

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::set {
public:
  // mint a map, kivéve a következőt

  typedef Key value_type;                  // a kulcs maga az érték
  typedef Cmp value_compare;
  // nincs indexelő operátor ([ ])
};
```

A *value_type* típust tehát itt a *key_type* típussal azonosként határozzuk meg. Ez a trükk lehetővé teszi, hogy nagyon sok esetben ugyanazzal a programmal kezeljünk egy *map* és egy *set* tárolót.

Figyeljük meg, hogy a *set* egy sorrendvizsgáló operátort használ (alapértelmezés szerint <), és nem egyenlőséget (==) ellenőriz. Ennek következtében az elemek egyenértékűségét a nem-egyenlőség határozza meg (§17.1.4.1) és a halmaz bejárásakor az elemeket meghatározott sorrendben kapjuk meg.

Ugyanúgy, mint a *map* esetében, itt is rendelkezésünkre állnak a halmazokat összehasonlító operátorok (==, !=,< , >, <=, >=) és a *swap()* függvény is.

17.4.4. A multiset

A *multiset* egy olyan halmaz, amely megengedi, hogy ugyanazok a kulcsok többször is előforduljanak:

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::multiset {
public:
  // mint a set, kivéve a következőt:
  iterator insert(const value_type&);               // bejárót ad vissza, nem párt
};
```

A többször előforduló kulcsok eléréséhez elsősorban az *equal_range()*, a *lower_bound()* és az *upper_bound()* függvényeket használhatjuk.

17.5. Majdnem-tárolók

A beépített tömbök (§5.2), a karakterláncok (20. fejezet), valamint a *valarray* (§22.4) és a *bitset* (§17.5.3) osztályok is elemeket tárolnak, így sok szempontból tárolónak tekinthetjük őket. Mindegyik hordoz azonban néhány olyan vonást, amely ellentmond a szabványos tárolók elveinek, ezért ezek a „majdnem-tárolók” nem mindig cserélhetők fel a teljesen kifejlesztett tárolókkal, például a *vector* vagy a *list* osztályokkal.

17.5.1. Karakterláncok

A *basic_string* osztály lehetőséget ad indexelésre, használhatunk közvetlen elérésű bejárókat és a tárolóknál megszokott kényelmi lehetőségek többsége is rendelkezésünkre áll (20. fejezet). A *basic_string* azonban nem teszi lehetővé, hogy annyiféle típust használjunk elemként, mint a tárolókban. Leginkább karakterláncokhoz felel meg és általában a tárolóktól jelentősen eltérő formában használjuk.

17.5.2. A valarray

A *valarray* kifejezetten a számokkal végzett műveletekhez készült vektor, így nem is akar általános tároló lenni. Ehelyett számos hasznos matematikai műveletet biztosít, míg a szabványos tárolók műveletei (§17.1.1) közül csak a *size()* és az indexelés áll a rendelkezésünkre (§22.4.2). A *valarray* elemeit közvetlen elérésű bejáróval érhetjük el (§19.2.1).

17.5.3. Bithalmazok

Egy rendszer szempontjából gyakran van szükség arra, hogy például egy bemeneti adatfolyamot (§21.3.3) egy sor kétértékű (például jó/rossz, igaz/hamis, kikapcsolt/bekapcsolt) állapotjelzővel írjunk le. A C++ egész értékekre megvalósított bitszintű műveletek segítségével támogatja az ilyen állapotjelzők hatékony kezelését (amennyiben csak néhány van belőlük). Ilyen művelet az *&* (és), a | (vagy), a ^ (kizáró vagy), a << (léptetés balra) és a >> (léptetés jobbra). A *bitset<N>* osztály ezt a fogalmat általánosítja. Segítségével egyszerűen végezhetünk műveleteket *N* darab biten, melyeket *0*-tól *N–1*-ig indexelünk. Az *N* értékét fordítási időben rögzítjük. Egy olyan bitsorozat esetében, amely nem fér el egy *long int* változóban, a *bitset* használata sokkal kényelmesebb, mint az *int* értékek közvetlen kezelése. Kevesebb jelzőbit esetében hatékonysági szempontok alapján kell döntenünk. Ha a biteket sorszámozás helyett el szeretnénk nevezni, halmazt (*set*, §17.4.3), felsorolási típust (§4.8), vagy bitmezőt (*bitfield*, §C.8.1) használhatunk.

A *bitset<N>* *N* darab bit tömbjének tekinthető. A *bitset* a *vector<bool>* osztálytól abban különbözik, hogy rögzített méretű, a *set* tárolótól abban, hogy egészekkel és nem asszociatív értékkel indexelhető, illetve mindkettőtől eltér abban, hogy közvetlen bitműveleteket kínál.

Egy beépített mutató (§5.1) segítségével nem tudunk biteket kijelölni, ezért a *bitset* osztálynak meg kell határoznia egy „hivatkozás bitre" típust. Ez a módszer általánosan használható olyan objektumok esetében, melyekhez a beépített mutatók valamilyen okból nem használhatók:

```
template<size_t N> class std::bitset {
public:

  class reference {                              // hivatkozás egyetlen bitre
        friend class bitset;
        reference();
  public:                                        // b[i] az (i+1)-edik bitre hivatkozik
        ~reference();
        reference& operator=(bool x);                          // b[i] = x;
        reference& operator=(const reference&);                // b[i] = b[j];
        bool operator~() const;                                // return ~b[i]
        operator bool() const;                                 // x = b[i];
        reference& flip();                                     // b[i].flip();
  };

  // ...
};
```

A *bitset* sablon az *std* névtérhez tartozik és a *<bitset>* fejállomány segítségével érhetjük el.

A C++ történetére visszanyúló okokból a *bitset* néhány szempontból eltér a standard könyvtár osztályainak stílusától. Például, ha egy index (amit gyakran *bitpozíciónak* nevezünk) kívül esik a megengedett tartományon, akkor egy *out_of_range* kivétel keletkezik. Nem állnak rendelkezésünkre bejárók. A bitpozíciókat jobbról balra számozzuk, ugyanúgy, ahogy a biteket egy gépi szóban. Így a *b[i]* bit értéke *pow(2,i)*. A *bitset* tehát egy *N* bites bináris számnak tekinthető:

pozició:	9	8	7	6	5	4	3	2	1	0
bitset<10>(989)	1	1	1	1	0	1	1	1	0	1

17.5.3.1. Konstruktorok

Egy *bitset* objektumot létrehozhatunk alapértelmezett értékkel, valamint egy *unsigned long int* vagy egy *string* bitjeiből is:

```
template<size_t N> class bitset {
public:
  // ...
  // konstruktorok

  bitset();                                        // N nulla bit
  bitset(unsigned long val);                       // bitek a val-ból

  template<class Ch, class Tr, class A>                  // Tr karakter-jellemző (§20.2)
  explicit bitset(const basic_string<Ch,Tr,A>& str,      // bitek az str karakterláncból
          basic_string<Ch,Tr,A>::size_type pos = 0,
          basic_string<Ch,Tr,A>::size_type n = basic_string<Ch,Tr,A>::npos);

  // ...
};
```

A bitek alapértelmezett értéke *0*. Ha *unsigned long int* paramétert adunk meg, akkor a szám minden egyes bitje a neki megfelelő bit kezdőértékét határozza meg. A *basic_string* (20. fejezet) ugyanígy működik; a *'0'* karakter jelenti a *0* bitértéket, az *'1'* karakter az *1* bitértéket. Minden más karakter *invalid_argument* kivételt vált ki. Alapértelmezés szerint a rendszer a bitek beállításához teljes karakterláncot használ, de a *basic_string* konstruktorának (§20.3.4) megfelelő stílusban megadhatjuk, hogy a karaktereknek csak egy résztartományát használjuk, a *pos* pozíciótól kezdve a lánc végéig vagy a *pos+n* pozícióig:

```
void f()
{
  bitset<10> b1;     // all 0

  bitset<16> b2 = 0xaaaa;                    // 1010101010101010
  bitset<32> b3 = 0xaaaa;                    // 00000000000000001010101010101010

  bitset<10> b4("1010101010");               // 1010101010
  bitset<10> b5("10110111011110",4);         // 0111011110
  bitset<10> b6("10110111011110",2,8);       // 0011011101

  bitset<10> b7("n0g00d");              // invalid_argument váltódott ki
  bitset<10> b8 = "n0g00d";             // hiba: nincs char*-ról bitset-re való átalakítás
}
```

A *bitset* osztály alapvető célja az, hogy egyetlen gépi szó helyigényű bitsorozathoz hatékony megoldást adhassunk. A felületet ennek a szemléletnek megfelelően alakították ki.

17.5.3.2. Bitkezelő műveletek

A *bitset* számos olyan műveletet biztosít, melyekkel egyes biteket érhetünk el vagy az összes bitet egyszerre kezelhetjük:

```
template<size_t N> class std::bitset {
public:
  // ...
  // bithalmaz-műveletek

  reference operator[ ](size_t pos);                    // b[i]

  bitset& operator&=(const bitset& s);                  // és
  bitset& operator|=(const bitset& s);                  // vagy
  bitset& operator^=(const bitset& s);                  // kizáró vagy

  bitset& operator<<=(size_t n);                        // logikai eltolás balra (feltöltés nullákkal)
  bitset& operator>>=(size_t n);                        // logikai eltolás jobbra (feltöltés nullákkal)

  bitset& set();                                        // minden bit 1-re állítása
  bitset& set(size_t pos, int val = 1);                 // b[pos]=val

  bitset& reset();                                      // minden bit 0-ra állítása
  bitset& reset(size_t pos);                            // b[pos]=0

  bitset& flip();                                       // minden bit értékének módosítása
  bitset& flip(size_t pos);                             // b[pos] értékének módosítása

  bitset operator~() const { return bitset<N>(*this).flip(); }    // komplemens halmaz
                                                                  // létrehozása
  bitset operator<<(size_t n) const { return bitset<N>(*this)<<=n; } // eltolt halmaz
                                                                     // létrehozása
  bitset operator>>(size_t n) const { return bitset<N>(*this)>>=n; } // eltolt halmaz
                                                                     // létrehozása

  // ...
};
```

Az indexelő operátor *out_of_range* kivételt vált ki, ha a megadott index kívül esik a megengedett tartományon. Nem ellenőrzött indexelésre nincs lehetőség.

A műveletek által visszaadott *bitset&* érték a **this*. Azok a műveletek, melyek a *bitset&* helyett *bitset* értéket adnak vissza, egy másolatot készítenek a **this* objektumról, ezen a másolaton végzik el a kívánt műveletet, és ennek eredményét adják vissza. Fontos, hogy a << és a >> műveletek itt tényleg eltolást jelentenek és nem be- vagy kimeneti műveleteket.

A *bitset* kimeneti operátora egy olyan << függvény, melynek két paramétere egy *ostream* és egy *bitset* (§17.5.3.3).

Amikor a biteket eltoljuk, logikai eltolás történik (tehát nem ciklikus). Ez azt jelenti, hogy néhány bit „kiesik", mások pedig az alapértelmezett *0* értéket kapják meg. Mivel a *size_t* típus előjel nélküli, arra nincs lehetőségünk, hogy negatív számmal toljuk el a bitsorozatot. Ennek következtében a *b<<-1* utasítás egy igen nagy pozitív számmal tolja el a biteket, így a *b bitset* minden bitjének értéke 0 lesz. A fordítók általában figyelmeztetnek erre a hibára.

17.5.3.3. További műveletek

A *bitset* is támogatja az olyan szokásos műveleteket, mint a *size()*, az ==, a ki- és bemenet stb.:

```
template<size_t N> class bitset {
public:
  // ...

  unsigned long to_ulong() const;

  template <class Ch, class Tr, class A> basic_string<Ch,Tr,A> to_string() const;

  size_t count() const;                    // 1 értékű bitek száma
  size_t size() const { return N; }        // bitek száma

  bool operator==(const bitset& s) const;
  bool operator!=(const bitset& s) const;

  bool test(size_t pos) const;             // igaz, ha b[pos] értéke 1
  bool any() const;                        // igaz, ha bármelyik bit értéke 1
  bool none() const;                       // igaz, ha egyik bit értéke sem 1
};
```

A *to_ulong()* és a *to_string()* függvény a megfelelő konstruktor fordított (inverz) művelete. A félreérthető átalakítások elkerülése érdekében a szokásos konverziós operátorok helyett a fenti műveletek állnak rendelkezésünkre. Ha a *bitset* objektumnak olyan bitjei is értéket tartalmaznak, melyek nem ábrázolhatók *unsigned long int* formában, a *to_ulong()* függvény *overflow_error* kivételt vált ki.

A *to_string()* művelet a megfelelő típusú karakterláncot állítja elő, amely *'0'* és *'1'* karakterekből épül fel. A *basic_string* a karakterláncok ábrázolásához használt sablon (20. fejezet).

A *to_string()* függvényt használhatjuk egy *int* bináris alakjának kiíratásához is:

```
void binary(int i)
{
  bitset<8*sizeof(int)> b = i;  // 8 bites bájtot tételezünk fel (lásd még §22.2)
  cout << b.template to_string< char,char_traits<char>,allocator<char> >() << '\n';
}
```

Sajnos egy minősített sablon tag meghívása igen bonyolult és ritkán használt utasításformát követel (§C.13.6). A tagfüggvényeken kívül a *bitset* lehetővé teszi a logikai & (és), a | (vagy), a ^ (kizáró vagy), valamint a szokásos ki- és bemeneti operátorok használatát is:

```
template<size_t N> bitset<N> std::operator&(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std::operator|(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std::operator^(const bitset<N>&, const bitset<N>&);

template <class charT, class Tr, size_t N>
basic_istream<charT,Tr>& std::operator>>(basic_istream<charT,Tr>&, bitset<N>&);
template <class charT, class Tr, size_t N>
basic_ostream<charT,Tr>& std::operator<<(basic_ostream<charT,Tr>&, const bitset<N>&);
```

Tehát egy *bitset* objektumot kiírathatunk anélkül is, hogy előbb karakterlánccá alakítanánk:

```
void binary(int i)
{
  bitset<8*sizeof(int)> b = i;  // 8 bites bájtot tételezünk fel (lásd még §22.2)
  cout << b << '\n';
}
```

Ez a programrészlet nullák és egyesek formájában írja ki a biteket és a legnagyobb helyiértékű bit lesz a bal oldalon.

17.5.4. Beépített tömbök

A beépített tömbök támogatják az indexelést és – a szokásos mutatók formájában – a közvetlen elérésű bejárók használatát is (§2.7.2). A tömb azonban nem tudja a saját méretét, így a programozónak kell azt nyilvántartania. A tömbök általában nem biztosítják a szabványos tagműveleteket és -típusokat.

Néha nagyon hasznos az a lehetőség, hogy egy beépített tömböt elrejthetünk a szabványos tárolók kényelmes jelölésrendszere mögé, úgy, hogy közben megtartjuk alacsonyszintű természetének előnyeit:

```
template<class T, int max> struct c_array {
   typedef T value_type;

   typedef T* iterator;
   typedef const T* const_iterator;

   typedef T& reference;
   typedef const T& const_reference;

   T v[max];
   operator T*() { return v; }

   reference operator[ ](size_t i) { return v[i]; }
   const_reference operator[ ](ptrdiff_t i) const { return v[i]; }

   iterator begin() { return v; }
   const_iterator begin() const { return v; }

   iterator end() { return v+max; }
   const_iterator end() const { return v+max; }

   ptrdiff_t size() const { return max; }
};
```

A tömbökkel való kompatibilitás miatt az előjeles *ptrdiff_t* típust (16.2.2) használom indexként az előjel nélküli *size_t* helyett.

A *c_array* sablon nem része a standard könyvtárnak. Mindössze egyszerű példaként szerepel itt arra, hogy egy „idegen" tárolót hogyan igazíthatunk a szabványos tárolók keretrendszeréhez. Ez az osztály olyan szabványos algoritmusokhoz (18. fejezet) használható, melyeknek a *begin()*, *end()* stb. műveletekre van szükségük. A verembe anélkül helyezhető, hogy közvetve dinamikus memóriát használnánk, és ezenkívül átadható olyan C stílusú függvényeknek is, melyek mutatót várnak paraméterként:

```
void f(int* p, int sz);          // C stílus

void g()
{
   c_array<int,10> a;

   f(a,a.size());                                           // C stílusú használat
   c_array<int,10>::iterator p = find(a.begin(),a.end(),777);  // C++/STL stílusú
                                                            // használat
   // ...
}
```

17.6. Új tárolók létrehozása

A szabványos tárolók olyan keretrendszert biztosítanak, melyet a programozó szabadon bővíthet. Az alábbiakban azt mutatom be, hogyan készíthetünk olyan tárolókat, melyek felcserélhetők a szabványos tárolókkal, ha indokolt. A bemutatott példa a gyakorlatban is működik, de nem optimális. A felületet úgy választottam meg, hogy az nagyon közel álljon a *hash_map* létező, széles körben elérhető, magas szintű megvalósításaihoz. Az itt bemutatott *hash_map* arra használható, hogy megtanuljuk az általános alapelveket. Komoly munkához használjunk inkább a támogatott *hash_map* típust.

17.6.1. A hash_map

A *map* olyan asszociatív tároló, melynek szinte bármilyen típusú eleme lehet, hiszen az egyetlen követelmény egy „kisebb mint” művelet az elemek összehasonlításához (§17.4.1.5). Ha azonban többet tudunk a kulcs típusáról, általában lerövidíthetjük az elemek megtalálásához szükséges időt, azzal, hogy hasító függvényt (hash function) készítünk és a tárolót hasító táblaként (hash table) valósítjuk meg.

A hasító függvény egy olyan eljárás, amely egy értékhez gyorsan hozzá tud rendelni egy indexet, mégpedig úgy, hogy két különböző érték ritkán kapja ugyanazt. A hasító tábla ezután lényegében úgy működik, hogy az értéket a kiszámított indexre helyezi, ha az még üres, ellenkező esetben az index „közelébe”. Ha egy elem a hozzá rendelt indexen található, akkor annak megtalálása nagyon gyors lehet, de az index „közelében” elhelyezett elem keresése sem túl lassú, ha az elemek egyenlőségének vizsgálata megfelelően gyors. Ezért aztán nem ritka, hogy olyan nagyobb tárolók esetében, ahol a keresés a legjelentősebb művelet, a *hash_map* akár 5–10-szer gyorsabban végzi el ezt a feladatot, mint a *map*. Ugyanakkor az is igaz, hogy a *hash_map* egy rosszul megválasztott hasító függvénnyel még a *map*-nél is sokkal lassabb lehet.

Hasító táblát sokféleképpen készíthetünk. A *hash_table* felületének csak ott kell különböznie a szabványos tárolók felületétől, ahol a hasítás hatékonysága ezt megköveteli. A legalapvetőbb különbség a *map* és a *hash_map* között, hogy a *map* a < művelettel hasonlítja össze az elemeit, míg a *hash_map* az == műveletet használja és szüksége van egy hasító függvényre is. Tehát ha a *hash_map* objektumot nem alapértelmezett beállításokkal akarjuk létrehozni, akkor a paraméterek jelentősen eltérnek a *map* tárolónál használtaktól:

```
map<string,int> m1;           // karakterláncok összehasonlítása < használatával
map<string,int,Nocase> m2;    // karakterláncok összehasonlítása Nocase()
                              // használatával (§17.1.4.1)
```

```
hash_map<string,int> hm1;               // hasítás Hash<string>() használatával (§17.6.2.3),
                                        // összehasonlítás == használatával
hash_map<string,int,hfct> hm2;          // hasítás hfct() használatával, összehasonlítás ==
                                        // használatával
hash_map<string,int,hfct,eql> hm3;      // hasítás hfct() használatával, összehasonlítás eql
                                        // használatával
```

A hasításos keresést használó tárolót egy vagy több táblázat segítségével hozhatjuk létre. Az elemek tárolásán kívül a tárolónak nyilván kell tartania azt is, milyen értéket milyen hasított értékhez (az előzőekben „index") rendelt hozzá. Erre szolgál a „hasító tábla". A hasító táblák teljesítménye általában jelentősen romlik, ha túlságosan telítetté válnak, tehát mondjuk 75 százalékig megteltek. Ezért az alábbiakban leírt *hash_map* automatikusan növeli méretét, ha túl telítetté válik. Az átméretezés azonban rendkívül költséges művelet lehet, így mindenképpen lehetővé kell tennünk egy kezdeti méret megadását.

Ezek alapján a *hash_map* első változata a következő lehet:

```
template<class Key, class T, class H = Hash<Key>,
            class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
class hash_map {
  // mint a map, kivéve a következőket:

  typedef H Hasher;
  typedef EQ key_equal;

  hash_map(const T& dv =T(), size_type n =101, const H& hf =H(), const EQ& =EQ());
  template<class In> hash_map(In first, In last,
            const T& dv =T(), size_type n =101, const H& hf =H(), const EQ& =EQ());
};
```

Ez lényegében megegyezik a *map* felületével (§17.4.1.4), csak a < műveletet az == operátorral helyettesítettük és megadtunk egy hasító függvényt is.

A könyvben eddig használt *map* objektumok (§3.7.4, §6.1, §17.4.1) könnyedén helyettesíthetők a *hash_map* szerkezettel. Nem kell mást tennünk, csak a *map* nevet át kell írnunk *hash_map*-re. A *map* cseréjét *hash_map*-re általában leegyszerűsíthetjük egy *typedef* utasítással:

```
typedef hash_map<string,record> Map;
Map dictionary;
```

A *typedef* arra is használható, hogy a szótár (dictionary) tényleges típusát elrejtsük a felhasználók elől.

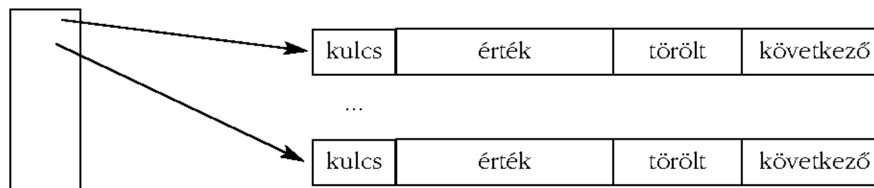
Bár a meghatározás nem egészen pontos, a *map* és a *hash_map* közötti ellentétet tekinthetjük egyszerűen tár-idő ellentétnek. Ha a hatékonyság nem fontos, nem érdemes időt vesztegetni a közöttük való választásra: mindkettő jól használható. Nagy és nehézkesen használható táblák esetében a *hash_map* jelentős előnye a sebesség, és ezt érdemes használnunk, hacsak nem fontos a kis tárigény. Még ha takarékoskodnunk is kell a memóriával, akkor is érdemes megvizsgálnunk néhány egyéb lehetőséget a tárigény csökkentésére, mielőtt az „egyszerű" *map* szerkezetet választjuk. Az aktuális méret kiszámítása nagyon fontos ahhoz, hogy ne egy alapjaiban rossz kódot próbáljunk optimalizálni.

A hatékony hasítás megvalósításának legfontosabb része a megfelelő hasító függvény megválasztása. Ha nem találunk jó hasító függvényt, akkor a *map* könnyen túlszárnyalhatja a *hash_map* hatékonyságát. A C stílusú karakterláncokra vagy egész számokra épülő hasítás általában elég hatékony tud lenni, de mindig érdemes arra gondolnunk, hogy egy hasító függvény hatékonysága nagymértékben függ az aktuális értékektől, melyeket hasítanunk kell (§17.8[35]). A *hash_map* tárolót kell használnunk akkor, ha a kulcshoz nem adhatunk meg < műveletet vagy az nem felel meg elvárásainknak. Megfordítva: a hasító függvény nem határoz meg az elemek között olyan sorrendet, mint amilyet a < operátor, így a *map* osztályra van szükségünk, ha az elemek sorrendje fontos. A *map* osztályhoz hasonlóan a *hash_map* is biztosít egy *find()* függvényt, amely megállapítja, hogy egy kulcs szerepel-e már a tárolóban.

17.6.2. Ábrázolás és létrehozás

A *hash_map* sokféleképpen megvalósítható. Itt egy olyan formát mutatok be, amely viszonylag gyors, de legfontosabb műveletei elég egyszerűek. Ezek a műveletek a konstruktorok, a keresés (a *[ ]* operátor), az átméretezés és az egy elem törlését végző függvény (*erase()*).

A hasító tábla itt bemutatott egyszerű megvalósítása egy olyan vektort használ, amely a bejegyzésekre hivatkozó mutatókat tárolja. Minden bejegyzésben (*Entry*) szerepel egy kulcs (*key*), egy érték (*value*) , egy mutató a következő ugyanilyen hasítókóddal rendelkező *Entry* bejegyzésre (ha van ilyen), illetve egy töröltséget jelző (*erased*) bit:



Ez deklaráció formájában a következőképpen néz ki.

```
template<class Key, class T, class H = Hash<Key>,
            class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
class hash_map {
  // ...
private:                          // ábrázolás
  struct Entry {
            key_type key;
            mapped_type val;
            bool erased;
            Entry* next;        // hash-túlcsordulás esetére
            Entry(key_type k, mapped_type v, Entry* n)
                     : key(k), val(v), next(n), erased(false) { }
  };

  vector<Entry> v;   // az aktuális bejegyzések
  vector<Entry*> b; // a hasító tábla: mutató v-be

  // ...
};
```

Vizsgáljuk meg az *erased* bit szerepét. Az ugyanolyan hasítókóddal rendelkező elemek kezelésének módja miatt nagyon nehéz lenne egy bizonyos elemet törölni. Ezért a tényleges törlés helyett az *erase()* meghívásakor csak az *erased* bitet jelöljük be, és az elemet mindaddig figyelmen kívül hagyjuk, amíg a táblát át nem méretezzük.

A fő adatszerkezet mellett a *hash_map* osztálynak egyéb adatokra is szüksége van. Természetesen minden konstruktornak az összes adattagot be kell állítania:

```
template<class Key, class T, class H = Hash<Key>,
            class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
class hash_map {
  // ...

  hash_map(const T& dv =T(), size_type n =101, const H& h =H(), const EQ& e =EQ())
            : default_value(dv), b(n), no_of_erased(0), hash(h), eq(e)
  {
            set_load();                              // alapértelmezés
            v.reserve(max_load*b.size());            // hely biztosítása a növekedéshez
  }

  void set_load(float m = 0.7, float g = 1.6) { max_load = m; grow = g; }

  // ...
```

```
private:
  float max_load;            // v.size()<=b.size()*max_load megtartása
  float grow;                // ha szükséges, resize(bucket_count()*grow)

  size_type no_of_erased;    // v azon bejegyzéseinek száma, amelyeket törölt elem foglal el

  Hasher hash;               // hasító függvény
  key_equal eq;              // egyenlőség

  const T default_value;     // a [ ] által használt alapértelmezett érték
};
```

A szabványos asszociatív tárolók megkövetelik, hogy a hozzárendelt érték típusa alapértelmezett értékkel rendelkezzen (§17.4.1.7). Ez a követelmény valójában nem elengedhetetlen és bizonyos esetekben nagyon kényelmetlen is lehet. Ezért az alapértelmezett értéket paraméterként vesszük át, ami lehetővé teszi a következő sorok leírását:

```
hash_map<string,Number> phone_book1;               // alapértelmezés: Number()
hash_map<string,Number> phone_book2(Number(411)); // alapértelmezés: Number(411)
```

17.6.2.1. Keresések

Végül elérkeztünk a kritikus keresési függvényekhez:

```
template<class Key, class T, class H = Hash<Key>,
          class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
class hash_map {
  // ...
  mapped_type& operator[ ](const key_type&);

  iterator find(const key_type&);
  const_iterator find(const key_type&) const;
  // ...
};
```

Az értékek megtalálásához az *operator[ ]()* a hasító függvényt használja, mellyel kiszámítja a kulcshoz tartozó indexet a hasító táblában. Ezután végignézi az adott index alatt található bejegyzéseket, amíg meg nem találja a kívánt kulcsot. Az így megtalált *Entry* objektumban található az a *value* érték, amelyet kerestünk. Ha nem találtuk meg a kulcsot, akkor alapértelmezett értékkel felveszünk egy új elemet:

```
template<class Key, class T, class H = Hash<Key>,
            class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
hash_map<Key,T,H,EQ,A>::mapped_type& hash_map<Key,T,H,EQ,A>::operator[ ](const
key_type& k)
{
  size_type i = hash(k)%b.size();          // hasítás

  for(Entry* p = b[i]; p; p = p->next)     // keresés a hasítás eredményeképpen i-be került
                                           // elemek között
            if (eq(k,p->key)) {                      // megvan
                      if (p->erased) {               // újbóli beszúrás
                                p->erased = false;
                                no_of_erased--;
                                return p->val = default_value;
                      }
                      return p->val;
            }
  // ha nincs meg

  if (size_type(b.size()*max_load) <= v.size()) { //ha "telített"
            resize(b.size()*grow);                   // növekedés
            return operator[ ](k);                   // újrahasítás
  }

  v.push_back(Entry(k,default_value,b[i]));          // Entry hozzáadása
  b[i] = &v.back();                                  // az új elemre mutat

  return b[i]->val;
}
```

A *map* megoldásától eltérően a *hash_map* nem a „kisebb mint” műveletből származtatható egyenlőségvizsgálatot (§17.1.4.1) használja, hiszen az azonos hasítókóddal rendelkező elemeket végignéző ciklusban az *eq()* függvény meghívása pontosan ezt a feladatot látja el. Ez a ciklus nagyon fontos a keresés hatékonysága szempontjából, a leggyakoribb kulcstípusok (például a *string*, vagy a C stílusú karakterláncok) szempontjából pedig egy felesleges összehasonlítás akár jelentős teljesítményromlást is eredményezhet.

Az azonos hasítókóddal rendelkező elemek tárolásához használhattuk volna a *set<Entry>* tárolót is, de ha elég jó hasítófüggvényünk (*hash()*) van és a hasítótábla (*b*) mérete is megfelelő, akkor ezen halmazok többségében egyetlen elem lesz. Ezért ezeket az elemeket egyszerűen az *Entry* osztály *next* mezőjével kapcsoltam össze (§17.8[27]).

Figyeljük meg, hogy a *b* az elemekre hivatkozó mutatókat tárolja és az elemek valójában a *v*-be kerülnek. A *push_back()* általában megtehetné, hogy áthelyezi az elemeket, és ezzel

a mutatók érvénytelenné válnának (§16.3.5), most azonban a konstruktorok (§17.6.2) és a *resize()* függvény előre helyet foglalnak az elemek számára a *reserve()* eljárással, így elkerüljük a váratlan áthelyezéseket.

17.6.2.2. Törlés és átméretezés

A hasítófüggvényes keresés nagyon rossz hatásfokúvá válhat, ha a tábla túlságosan telített. Az ilyen kellemetlenségek elkerülése érdekében az indexelő operátor automatikusan átméretezheti a táblát. A *set_load()* (§17.6.2) függvény szabályozza, hogy ez az átméretezés mikor és hogyan menjen végbe, néhány további függvénnyel pedig lehetővé tesszük a programozó számára, hogy lekérdezze a *hash_map* állapotát:

```
template<class Key, class T, class H = Hash<Key>,
            class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
class hash_map {
  // ...

  void resize(size_type n);                        // a hasító tábla méretének n-re állítása

  void erase(iterator position);                   // a mutatott elem törlése

  size_type size() const { return v.size()-no_of_erased; }  // az elemek száma

  size_type bucket_count() const { return b.size(); }       // a hasító tábla mérete

  Hasher hash_fun() const { return hash; }                  // a használt hasító függvény
  key_equal key_eq() const { return eq; }                   // a használt egyenlőségvizsgálat

  // ...
};
```

A *resize()* függvény rendkívül fontos, viszonylag egyszerű, de néha rendkívül „drága” művelet:

```
template<class Key, class T, class H = Hash<Key>,
            class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
void hash_map<Key,T,H,EQ,A>::resize(size_type s)
{
  size_type i = v.size();
  while (no_of_erased) {                  // a "törölt" elemek tényleges eltávolítása
            if (v[--i].erased) {
                      v.erase(&v[i]);
                      --no_of_erased;
            }
  }
```

```
    if (s <= b.size()) return;
    b.resize(s);                    // s-b.size() számú mutató hozzáadása
    fill(b.begin(),b.end(),0);      // bejegyzések törlése (§18.6.6)
    v.reserve(s*max_load);          // ha v-nek újbóli memóriafoglalásra van szüksége, most
                                    // történjen meg
    if (no_of_erased) {                         // a "törölt" elemek tényleges eltávolítása
            for (size_type i = v.size()-1; 0<=i; i--)
                    if (v[i].erased) {
                            v.erase(&v[i]);
                            if (--no_of_erased == 0) break;
                    }
    }

    for (size_type i = 0; i<v.size(); i++) {                // újrahasítás
            size_type ii = hash(v[i].key)%b.size();         // hasítás
            v[i].next = b[ii];                              // láncolás
            b[ii] = &v[i];
    }
}
```

Ha szükség van rá, a programozó maga is meghívhatja a *resize()* eljárást, így biztosíthatja, hogy az időveszteség kiszámítható helyen következzen be. Tapasztalataim szerint bizonyos alkalmazásokban a *resize()* művelet rendkívül fontos, de a hasítótáblák szempontjából nem nélkülözhetetlen. Egyes megvalósítási módszerek egyáltalán nem használják.

Mivel a tényleges munka nagy része máshol történik (és csak akkor, amikor átméretezzük a *hash_map* tárolót), az *erase()* függvény nagyon egyszerű:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator< pair<const Key,T> > >
void hash_map<Key,T,H,EQ,A>::erase(iterator p)          // a mutatott elem törlése
{
  if (p->erased == false) no_of_erased++;
  p->erased = true;
}
```

17.6.2.3. Hasítás

A *hash_map::operator[ ]()* teljessé tételéhez még meg kell határoznunk a *hash()* és az *eq()* függvényt. Bizonyos okokból (amelyeket majd a §18.4 pontban részletezünk) a hasító függvényt érdemesebb egy függvényobjektum *operator()()* műveleteként elkészítenünk:

```
template <class T> struct Hash : unary_function<T, size_t> {
  size_t operator()(const T& key) const;
};
```

A jó hasító függvény paraméterében egy kulcsot kap és egy egész értéket ad vissza, amely különböző kulcsok esetében nagy valószínűséggel különböző. A jó hasítófüggvény kiválasztása nagyon nehéz. Gyakran az vezet elfogadható eredményhez, ha a kulcsot jelölő biteken és egy előre meghatározott egész értéken „kizáró vagy" műveletet hajtunk végre:

```
template <class T> size_t Hash<T>::operator()(const T& key) const
{
   size_t res = 0;

   size_t len = sizeof(T);
   const char* p = reinterpret_cast<const char*>(&key);   //objektumok elérése bájtok
                                                            // sorozataként

   while (len--) res = (res<<1)^*p++;                // a kulcsábrázolás bájtjainak használata
   return res;
}
```

A *reinterpret_cast* (§6.2.7) használata jól jelzi, hogy valami „csúnya" dolgot műveltünk, és ha többet tudunk a hasított értékről, akkor ennél hatékonyabb megoldást is találhatunk. Például, ha az objektum mutatót tartalmaz, ha nagy objektumról van szó, vagy ha az adattagok igazítása miatt az objektum ábrázolásában felhasználatlan területek („lyukak", holes) vannak, mindenképpen jobb hasítófüggvényt készíthetünk (§17.8[29]).

A C stílusú karakterláncok mutatók (a karakterekre) és a *string* is tartalmaz mutatót. A specializációk ezekre az esetekre sorrendben a következők:

```
size_t Hash<char*>::operator()(const char* key) const
{
   size_t res = 0;

   while (*key) res = (res<<1)^*key++;              // a karakterek egész értékének használata
   return res;
}

template <class C>
size_t Hash< basic_string<C> >::operator()(const basic_string<C>& key) const
{
   size_t res = 0;

   typedef basic_string<C>::const_iterator CI;
   CI p = key.begin();
   CI end = key.end();

   while (p!=end) res = (res<<1)^*p++;              // a karakterek egész értékének használata
   return res;
}
```

A *hash_map* minden megvalósításában legalább az egész és a karakterlánc típusú kulcsokhoz szerepelniük kell hasítófüggvénynek. Komolyabb kulcsok esetében a programozót segíthetjük megfelelő specializációkkal is. A jó hasítófüggvény kiválasztásában sokat segíthet a megfelelő mérőszámokra támaszkodó kísérletezés. Megérzéseinkre nagyon ritkán hagyatkozhatunk ezen a területen.

A *hash_map* akkor válik teljessé, ha bejárókat is definiálunk és néhány további egyszerű függvényt is készítünk. Ez azonban maradjon meg feladatnak (§17.8[34]).

17.6.3. További hasított asszociatív tárolók

Az egységesség és a teljesség érdekében mindenképpen el kell készítenünk a *hash_map* „testvéreit" is: a *hash_set*, a *hash_multimap* és a *hash_multiset* tárolót. Ezek létrehozása a *hash_map*, a *map*, a *multimap*, a *set* és a *multiset* alapján egyszerű, így ezeket is meghagyom feladatnak §17.8[34]. Ezeknek a hasított asszociatív tárolóknak komoly irodalma van és kész kereskedelmi változatok is elérhetők belőlük. Tényleges programokban ezek a változatok jobban használhatók, mint a házilag összeeszkábáltak (köztük az általam bemutatott is).

17.7. Tanácsok

[1] Ha tárolóra van szükségünk, általában a *vector* osztályt használjuk. §17.1.
[2] Ha gyakran használunk egy műveletet, érdemes ismerni annak költségeit (bonyolultság, „nagy O" mérték) §17.1.2.
[3] A tároló felülete, megvalósítása és ábrázolása külön-külön fogalom. Ne keverjük össze őket. §17.1.3.
[4] Keresést vagy rendezést bármilyen szempont szerint megvalósíthatunk. §17.1.4.1.
[5] Ne használjunk kulcsként C stílusú karakterláncokat, vagy biztosítsunk megfelelő összehasonlítási műveletet. §17.1.4.1.
[6] Az összehasonlítási szemponttal elemek egyenértékűségét határozhatjuk meg, ami eltérhet attól, hogy a kulcsok teljesen egyenlők-e. §17.1.4.1.
[7] Lehetőleg használjuk a sorozat végét módosító függvényeket (*back*-műveletek), ha elemek beszúrására vagy törlésére van szükségünk. §17.1.4.1.

[8] Ha sok beszúrásra, illetve törlésre van szükség a sorozat elején vagy belsejében, használjuk a *list* tárolót.
[9] Ha az elemeket legtöbbször kulcs alapján érjük el, használjuk a *map* vagy a *multimap* szerkezetet. §17.4.1.
[10] A lehető legnagyobb rugalmasságot úgy érhetjük el, ha a lehető legkevesebb műveletet használjuk. §17.1.1.
[11] Ha fontos az elemek sorrendje, a *hash_map* helyett használjuk a *map* osztályt. §17.6.1.
[12] Ha a keresés sebessége a legfontosabb, a *hash_map* hasznosabb, mint a *map*. §17.6.1.
[13] Ha nem tudunk az elemekre „kisebb mint" műveletet megadni, a *hash_map* tárolót használhatjuk. §17.6.1.
[14] Ha azt akarjuk ellenőrizni, hogy egy kulcs megtalálható-e egy asszociatív tárolóban, használjuk a *find()* függvényt. §17.4.1.6.
[15] Ha az adott kulccsal rendelkező összes elemet meg akarjuk találni, használjuk az *equal_range()*-et. §17.4.1.6.
[16] Ha ugyanazzal a kulccsal több elem is szerepelhet, használjuk a *multimap* tárolót. §17.4.2.
[17] Ha csak a kulcsot kell nyilvántartanunk, használjuk a *set* vagy a *multiset* osztályt. §17.4.3.

17.8. Gyakorlatok

Az itt szereplő gyakorlatok többségének megoldása kiderül a standard könyvtár bármely változatának forráskódjából. Mielőtt azonban megnéznénk, hogy a könyvtár megalkotói hogyan közelítették meg az adott problémát, érdemes saját megoldást készítenünk. Végül nézzük át, hogy saját rendszerünkben milyen tárolók és milyen műveletek állnak rendelkezésünkre.

1. (*2.5) Értsük meg az *O()* jelölést (§17.1.2). Végezzünk néhány mérést a szabványos tárolók műveleteire és határozzuk meg a konstans szorzókat.
2. (*2) Sok telefonszám nem ábrázolható egy *long* értékkel. Készítsünk egy *phone_number* típust és egy osztályt, amely meghatározza az összes olyan műveletet, melyeknek egy telefonszámokat tároló tárolóban hasznát vehetjük.
3. (*2) Írjunk programot, amely kiírja egy fájl szavait ábécésorrendben. Készítsünk két változatot: az egyikben a szó egyszerűen üreshely karakterekkel határolt karaktersorozat legyen, a másikban olyan betűsorozat, melyet nem betű karakterek sorozata határol.

4. (*2.5) Írjunk egy egyszerű Pasziánsz kártyajátékot.
5. (*1.5) Írjunk programot, amely eldönti, hogy egy szó palindrom-e (azaz ábrázolása szimmetrikus-e, például: *ada, otto, tat*). Írjuk eljárást, amely egy egész számról dönti el ugyanezt, majd készítsünk mondatokat vizsgáló függvényt. Általánosítsunk.
6. (*1.5) Készítsünk egy sort (queue) két verem segítségével.
7. (*1.5) Készítsünk egy vermet, amely majdnem olyan, mint a *stack*, csak nem másolja az elemeit és lehetővé teszi bejárók használatát.
8. (*3) Számítógépünk minden bizonnyal támogat valamilyen konkurens (párhuzamos végrehajtási) lehetőséget: szálakat (*thread*), taszkokat (*task*) vagy folyamatokat (*process*). Derítsük ki, hogyan működik ez. A konkurens hozzáférést lehetővé tevő rendszer valamilyen módot ad rá, hogy megakadályozzuk, hogy két folyamat ugyanazt a memóriaterületet egyszerre használja. Saját rendszerünk zárolási eljárása alapján készítsünk egy osztályt, amely a programozó számára egyszerűen elérhetővé teszi ezt a lehetőséget.
9. (*2.5) Olvassuk be dátumok egy sorozatát (például: *Dec85, Dec50, Jan76*), majd jelenítsük meg azt úgy, hogy a legkésőbbi időpont legyen az első a sorban. A dátum formátuma a következő legyen: hónapnév három karakteren, majd évszám két karakteren. Tételezzük fel, hogy mindegyik dátum ugyanarra az évszázadra vonatkozik.
10. (*2.5) Általánosítsuk a dátumok bemeneti formátumát úgy, hogy felismerje az alábbi dátumformátumokat: *Dec1985*, *12/3/1990*, *(Dec,30,1950)*, *3/6/2001*, stb. Módosítsuk a §17.8[9] feladatot úgy, hogy működjön ezekre a formátumokra is.
11. (*1.5) Használjuk a *bitset* tárolót néhány szám bináris alakjának kiíratásához. Például *0, 1, -1, 18, -18* és a legnagyobb pozitív *int* érték.
12. (*1.5) A *bitset* segítségével ábrázoljuk, hogy egy osztály mely tanulói voltak jelen egy adott napon. Olvassuk be 12 nap *bitset* objektumát és állapítsuk meg, kik voltak jelen minden nap, és kik voltak legalább 8 napot az iskolában.
13. (*1.5) Készítsünk egy olyan mutatókból álló listát, amely törli a mutatott objektumokat is, ha egy mutatót törlünk belőle vagy ha az egész listát megszüntetjük.
14. (*1.5) Írassuk ki rendezve egy adott *stack* objektum elemeit anélkül, hogy az eredeti vermet megváltoztatnánk.
15. (*2.5) Fejezzük be a *hash_map* (§17.6.1) megvalósítását. Ehhez meg kell írnunk a *find()* és az *equal_range()* függvényt, valamint módot kell adnunk a kész sablon ellenőrzésére. Próbáljuk ki a *hash_map* osztályt legalább egy olyan kulcstípusra, melyre az alapértelmezett hasítófüggvény nem használható.
16. (*2.5) Készítsünk el egy listát a szabványos *list* stílusában és teszteljük.
17. (*2) Bizonyos helyzetekben a *list* túlzott memória-felhasználása problémát jelent. Készítsünk egy egyirányú láncolt listát a szabványos tárolók stílusában.

18. (*2.5) Készítsünk el egy listát, amely olyan, mint a szabványos *list*, csak támogatja az indexelést is. Hasonlítsunk össze néhány listára az indexelés költségét egy ugyanilyen méretű *vector* indexelési költségével.
19. (*2) Készítsünk egy sablon függvényt, amely két tárolót összefésül.
20. (*1.5) Állapítsuk meg, hogy egy C stílusú karakterlánc palindrom-e. Vizsgáljuk meg, hogy a karakterlánc (legalább) első három szavából álló sorozat palindrom-e.
21. (*2) Olvassuk be *(name,value)* (név, érték) párok egy sorozatát és készítsünk egy rendezett listát *(name, total, mean, median)* (név, összesen, átlag, középérték) sorokból.
22. (*2.5) Vizsgáljuk meg, mekkora az általunk készített tárolók tárigénye.
23. (*3.5) Gondolkozzunk el azon, milyen megvalósítási stratégiát használhatnánk egy olyan *hash_map* tárolóhoz, melynél a lehető legkisebb tárigény a legfőbb követelmény. Hogyan készíthetnénk olyan *hash_map* osztályt, melynek keresési ideje minimális? Nézzük át, mely műveleteket érdemes kihagyni a megvalósításból az optimálishoz (felesleges memóriafoglalás, illetve időveszteség nélküli) közeli megoldás eléréséhez. Segítség: a hasítótábláknak igen kiterjedt irodalma van.
24. (*2) Dolgozzunk ki olyan elvet a *hash_map* túlcsordulásának (különböző értékek ugyanazon hasítókódra kerülésének) kezelésére, amely az *equal_range()* elkészítését egyszerűvé teszi.
25. (*2.5) Becsüljük meg a *hash_map* tárigényét, majd mérjük is le azt. Hasonlítsuk össze a becsült és a számított értékeket. Hasonlítsuk össze az általunk megvalósított *hash_map* és *map* tárigényét.
26. (*2.5) Vizsgáljuk meg saját *hash_map* osztályunkat abból a szempontból, hogy melyik művelettel telik el a legtöbb idő. Tegyük meg ugyanezt saját *map* tárolónkra, illetve a *hash_map* kereskedelmi változataira is.
27. (*2.5) Készítsük el a *hash_map* tárolót egy *vector<map<K,V>*>* szerkezet segítségével. Minden *map* tárolja az összes olyan kulcsot, melyek hasítókódja megegyezik.
28. (*3) Készítsük el a *hash_map* osztályt Splay fák segítségével. (Lásd: D. Sleator, R. E. Tarjan: *Self-Adjusting Binary Search Trees*, JACM, 32. kötet, 1985)
29. (*2) Adott egy struktúra, amely egy karakterlánc-szerű egyedet ír le:

```
struct St {
        int size;
        char type_indicator;
        char* buf;                  // méretre mutat
        St(const char* p);          // memóriafoglalás és a buf feltöltése
};
```

Készítsünk 1000 *St* objektumot és használjuk ezeket egy *hash_map* kulcsaként. Készítsünk programot, mellyel tesztelni lehet a *hash_map* hatékonyságát. Írjunk egy hasítófüggvényt (*Hash*, §17.6.2.3) kifejezetten az *St* típusú kulcsok kezeléséhez.

30. (*2) Adjunk legalább négyféle megoldást a törlésre kijelölt (*erased*) elemek eltávolítására a *hash_map* tárolóból. Ciklus helyett használjuk a standard könyvtár algoritmusait. (§3.8, 18. fejezet)
31. (*3) Készítsünk olyan *hash_map* tárolót, amely azonnal törli az elemeket.
32. (*2) A §17.6.2.3 pontban bemutatott hasítófüggvény nem mindig használja a kulcs teljes ábrázolását. Mikor hagy figyelmen kívül részleteket ez a megoldás? Írjunk olyan hasítófüggvényt, amely mindig a kulcs teljes ábrázolását használja. Adjunk rá példát, mikor lehet jogos a kulcs egy részének „elfelejtése" és írjunk olyan hasítófüggvényt, amely a kulcsnak csak azt a részét veszi figyelembe, amelyet fontosnak nyilvánítunk.
33. (*2.5) A hasítófüggvények forráskódja meglehetősen hasonló: egy ciklus sorra veszi az adatokat, majd előállítja a hasítóértéket. Készítsünk olyan *Hash* (§17.6.2.3) függvényt, amely az adatokat egy, a felhasználó által megadott függvény ismételt meghívásával gyűjti össze. Például:

```
size_t res = 0;
while (size_t v = hash(key)) res = (res<<3)^v;
```

    Itt a felhasználó a *hash(K)* függvényt minden olyan *K* típusra megadhatja, amely alapján hasítani akar.
34. (*3) A *hash_map* néhány megvalósításából kiindulva készítsük el a *hash_multimap*, a *hash_set* és a *hash_multiset* tárolót.
35. (*2.5) Készítsünk olyan hasítófüggvényt, amely egyenletes eloszlású *int* értékeket képez le egy körülbelül 1024 méretű hasítótáblára. Ezen hasítófüggvény ismeretében adjunk meg 1024 olyan kulcsértéket, amelyet a függvény ugyanarra a hasítókódra képez le.

18

Algoritmusok és függvényobjektumok

„A forma szabaddá tesz.”
(a mérnökök közmondása)

Bevezető • A szabványos algoritmusok áttekintése • Sorozatok • Függvényobjektumok • Predikátumok • Aritmetikai objektumok • Lekötők • Tagfüggvény-objektumok • *for_each* • Elemek keresése • *count* • Sorozatok összehasonlítása • Keresés • Másolás • *transform* • Elemek lecserélése és eltávolítása • Sorozatok feltöltése • Átrendezés • *swap* • Rendezett sorozatok • *binary_search* • *merge* • Halmazműveletek • *min* és *max* • Kupac • Permutációk • C stílusú algoritmusok • Tanácsok • Gyakorlatok

18.1. Bevezető

Egy tároló önmagában nem túlságosan érdekes dolog. Ahhoz, hogy tényleg hasznossá váljon, számos alapvető műveletre is szükség van, melyekkel például lekérdezhetjük a tároló méretét, bejárhatjuk, másolhatjuk, rendezhetjük, vagy elemeket kereshetünk benne. Szerencsére a standard könyvtár biztosítja mindazokat a szolgáltatásokat, melyekre a programozóknak szükségük van a tárolók használatához.

Ebben a fejezetben a szabványos algoritmusokat foglaljuk össze és néhány példát mutatunk be használatukra. Kiemeljük azokat a legfontosabb elveket és módszereket, melyeket ismernünk kell az algoritmusok lehetőségeinek a C++-ban való kiaknázásához. Néhány alapvető algoritmust részletesen is megvizsgálunk.

Azokat az eljárásokat, melyek segítségével a programozók saját igényeikhez alakíthatják a szabványos algoritmusok viselkedését, a függvényobjektumok biztosítják. Ezek adják meg azokat az alapvető információkat is, melyekre a felhasználók adatainak kezeléséhez szükség van.

Éppen ezért nagy hangsúlyt helyezünk arra, hogy bemutassuk a függvényobjektumok létrehozásának és használatának módszereit.

18.2. A standard könyvtár algoritmusainak áttekintése

Első pillantásra úgy tűnhet, hogy a standard könyvtár algoritmusainak száma szinte végtelen, pedig „mindössze" 60 darab van belőlük. Találkoztam már olyan osztállyal, melynek önmagában több tagfüggvénye volt. Ráadásul nagyon sok algoritmus ugyanazt az általános viselkedésformát, illetve felületstílust mutatja, és ez nagymértékben leegyszerűsíti megértésüket. Ugyanúgy, mint a programnyelvi lehetőségek esetében, a programozónak itt is csak azokat az elemeket kell használnia, amelyekre éppen szüksége van és amelyek működését ismeri. Semmilyen előnyt nem jelent, ha minden apróságához szabványos algoritmust keresünk, és azért sem kapunk jutalmat, ha az algoritmusokat rendkívül „okosan", de áttekinthetetlenül alkalmazzuk. Ne felejtsük el, hogy a programkód leírásának elsődleges célja, hogy a későbbi olvasók számára a program működése érthető legyen (A „későbbi olvasók" valószínűleg mi magunk leszünk néhány év múlva.) Másrészt, ha egy tároló elemeivel valamilyen feladatot kell elvégeznünk, gondoljuk végig, hogy a művelet nem fogalmazható-e meg a standard könyvtár algoritmusainak stílusában. Az is elképzelhető, hogy az algoritmus már meg is van, csak olyan általános formában, hogy első ránézésre rá sem ismerünk. Ha megszokjuk az általánosított (generikus) algoritmusok világát, nagyon sok felesleges munkától kímélhetjük meg magunkat.

Mindegyik algoritmus egy sablon függvény (template function) (§13.3) formájában jelenik meg vagy sablon függvények egy csoportjaként. Ez a megoldás lehetővé teszi, hogy az algoritmusok sokféle elemsorozaton legyenek képesek működni és természetesen az elemek típusa is módosítható legyen. Azok az algoritmusok, melyek eredményképpen egy bejárót (iterator) (§19.1) adnak vissza, általában a bemeneti sorozat végét használják a sikertelen végrehajtás jelzésére:

```
void f(list<string>& ls)
{
   list<string>::const_iterator p = find(ls.begin(),ls.end(),"Frici");

   if (p == ls.end()) {
            // "Frici" nem található
   }
   else {
            // p "Frici"-re mutat
   }
}
```

Az algoritmusok nem végeznek tartományellenőrzést a be- vagy kimenetükön. A rossz tartományból eredő hibákat más módszerekkel kell elkerülnünk (§18.3.1, §19.3) Ha az algoritmus egy bejárót ad vissza, annak típusa ugyanolyan lesz, mint a bemeneti sorozat valamelyik bejárójának. Tehát például az algoritmus paramétere határozza meg, hogy a visszatérési érték *const_iterator* vagy nem konstans *iterator* lesz-e:

```
void f(list<int>& li, const list<string>& ls)
{
   list<int>::iterator p = find(li.begin(),li.end(),42);
   list<string>::const_iterator q = find(ls.begin(),ls.end(),"Ring");
}
```

A standard könyvtár algoritmusai között megtaláljuk a tárolók leggyakoribb általános műveleteit, például a bejárásokat, kereséseket, rendezéseket, illetve az elemek beszúrását és törlését is. A szabványos algoritmusok kivétel nélkül az *std* névtérben találhatók és az *<algorithm>* fejállomány deklarálja őket. Érdekes, hogy az igazán általános algoritmusok nagy része annyira egyszerű, hogy általában helyben kifejtett (inline) sablon függvények valósítják meg azokat. Ezért az algoritmusok ciklusait a hatékony, függvényen belüli optimalizációs eljárások jelentősen javíthatják.

A szabványos függvényobjektumok is az *std* névtérben találhatók, de ezek deklarációját a *<functional>* fejállományban érhetjük el. A függvényobjektumok felépítése is olyan, hogy könnyen használhatók helyben kifejtett függvényként.

A nem módosító sorozatműveletek arra használhatók, hogy adatokat nyerjünk ki egy sorozatból vagy bizonyos elemek helyét meghatározzuk bennük:

Nem módosító sorozatműveletek (§18.5) <*algorithm*>	
for_each()	Művelet végrehajtása egy sorozat összes elemére.
find()	Egy érték első előfordulásának megkeresése egy sorozatban.
find_if()	Az első olyan elem megkeresése egy sorozatban, amire egy állítás teljesül.
find_first_of()	Egy sorozat egy elemének megkeresése egy másik sorozatban.
adjacent_find()	Két szomszédos érték keresése.
count()	Egy érték előfordulásainak száma egy sorozatban.
count_if()	Azon elemek száma egy sorozatban, melyre teljesül egy állítás.
mismatch()	Az első olyan elemek keresése, ahol két sorozat különbözik.
equal()	Igazat ad vissza, ha két sorozat elemei páronként megegyeznek.
search()	Egy sorozat első előfordulását keresi meg részsorozatként.
find_end()	Egy sorozat részsorozatként való utolsó előfordulását keresi meg.
search_n()	Egy érték *n*-edik előfordulását keresi meg egy sorozatban.

A legtöbb algoritmus lehetővé teszi, hogy a programozó határozza meg azt a feladatot, amelyet minden elemen, illetve elempáron el akar végezni. Ezáltal az algoritmusok sokkal általánosabbak és hasznosabbak, mint azt első ránézésre gondolhatnánk. A programozó határozhatja meg, mikor tekintünk két elemet azonosnak és mikor különbözőnek (§18.4.2), és a leggyakrabban végrehajtott, leghasznosabb műveletet alapértelmezettként is kijelölheti.

A sorozatmódosító műveletek között igen kevés hasonlóságot találhatunk azon kívül, hogy megváltoztatják a sorozat egyes elemeinek értékét:

Sorozatmódosító műveletek (§18.6) <*algorithm*>	
transform()	Művelet végrehajtása a sorozat minden elemén.
copy()	Sorozat másolása az első elemtől kezdve.
copy_backward()	Sorozat másolása az utolsó elemtől kezdve.
swap()	Két elem felcserélése.
iter_swap()	Bejárók által kijelölt két elem felcserélése.
swap_ranges()	Két sorozat elemeinek felcserélése.
replace()	Adott értékű elemek helyettesítése.
replace_if()	Állítást kielégítő elemek helyettesítése.
replace_copy()	Sorozat másolása adott értékű elemek helyettesítésével.
replace_copy_if()	Sorozat másolása állítást kielégítő elemek helyettesítésével.
fill()	Az összes elem helyettesítése egy adott értékre.
fill_n()	Az első *n* elem helyettesítése egy adott értékkel.
generate()	Az összes elem helyettesítése egy művelettel előállított értékre.
generate_n()	Az első *n* elem helyettesítése egy művelettel előállított értékre.
remove()	Adott értékkel rendelkező elemek törlése.
remove_if()	Állítást kielégítő elemek törlése.
remove_copy()	Sorozat másolása adott értékű elemek törlésével.
remove_copy_if()	Sorozat másolása állítást kielégítő elemek törlésével.
unique()	Szomszédos egyenértékű elemek törlése.
unique_copy()	Sorozat másolása szomszédos egyenértékű elemek törlésével.
reverse()	Az elemek sorrendjének megfordítása.
reverse_copy()	Elemek másolása fordított sorrendben.
rotate()	Elemek körbeforgatása.
rotate_copy()	Sorozat másolása az elemek körbeforgatásával.
random_shuffle()	Elemek átrendezése egyenletes eloszlás szerint.

Minden jó rendszer magán viseli megalkotójának érdeklődési körét, illetve személyes jellemvonásait. A standard könyvtár tárolói és algoritmusai tökéletesen tükrözik a klasszikus adatszerkezetek mögötti elveket és az algoritmusok tervezési szempontjait. A standard könyvtár nem csak a tárolók és algoritmusok legalapvetőbb fajtáit biztosítja, melyekre minden programozónak szüksége van, hanem olyan eszközöket is, melyekkel ezek az algoritmusok megvalósíthatók, és lehetőséget ad a könyvtár bővítésére is.

A hangsúly most nem igazán azon van, hogy ezek az algoritmusok hogyan valósíthatók meg, sőt – a legegyszerűbb algoritmusoktól eltekintve – nem is azok használatán. Ha az algoritmusok szerkezetéről és elkészítési módjairól többet szeretnénk megtudni, más könyveket kell fellapoznunk (például [Knuth, 1968] vagy [Tarjan, 1983]). Itt azzal foglalkozunk, mely algoritmusok állnak rendelkezésünkre a standard könyvtárban és ezek hogyan jelennek meg a C++ nyelvben. Ez a nézőpont lehetővé teszi, hogy – amennyiben tisztában vagyunk az algoritmusokkal – hatékonyan használjuk a standard könyvtárat, illetve olyan szellemben fejlesszük azt tovább, ahogy megszületett.

A standard könyvtár sok-sok olyan eljárást kínál, melyekkel sorozatokat rendezhetünk, kereshetünk bennük, vagy más, sorrenden alapuló műveleteket végezhetünk velük:

Rendezett sorozatok műveletei (§18.7) *<algorithm>*	
sort()	Átlagos hatékonysággal rendez.
stable_sort()	Az egyenértékű elemek sorrendjének megtartásával rendez.
partial_sort()	Egy sorozat elejét rendezi.
partial_sort_copy()	Másol a sorozat elejének rendezésével.
nth_element()	Az *n*-edik elemet a megfelelő helyre teszi.
lower_bound()	Egy érték első előfordulását keresi meg.
upper_bound()	Egy érték utolsó előfordulását keresi meg.
equal_range()	Egy adott értéket tartalmazó részsorozatot ad meg.
binary_search()	Igazat ad vissza, ha a megadott érték szerepel a sorozatban.
merge()	Két rendezett sorozatot fésül össze.
inplace_merge()	Két egymást követő rendezett részsorozatot fésül össze.
partition()	Elemeket helyez el egy állításnak (feltételnek) megfelelően.
stable_partition()	Elemeket helyez el egy állításnak megfelelően, a relatív sorrend megtartásával.

Halmazműveletek (§18.7.5) <*algorithm*>	
includes()	Igazat ad vissza, ha egy sorozat részsorozata egy másiknak.
set_union()	Rendezett uniót állít elő.
set_intersection()	Rendezett metszetet állít elő.
set_difference()	Azon elemek rendezett sorozatát állítja elő, melyek az első sorozatban megtalálhatók, de a másodikban nem.
set_symmetric_difference()	Azon elemek rendezett sorozatát állítja elő, melyek csak az egyik sorozatban találhatók meg.

A kupacműveletek (heap-műveletek) olyan állapotban tartják a sorozatot, hogy az könnyen rendezhető legyen, amikor arra szükség lesz:

Kupacműveletek (§18.8) <*algorithm*>	
make_heap()	Egy sorozatot felkészít kupacként való használatra.
push_heap()	Elemet ad a kupachoz.
pop_heap()	Elemet töröl a kupacból.
sort_heap()	Rendezi a kupacot.

A könyvtár biztosít néhány olyan eljárást is, melyek összehasonlítással kiválasztott elemeket adnak meg:

Minimum és maximum (§18.9) <*algorithm*>	
min()	Két érték közül a kisebb.
max()	Két érték közül a nagyobb.
min_element()	A legkisebb érték egy sorozatban.
max_element()	A legnagyobb érték egy sorozatban.
lexicographical_compare()	Két sorozat közül az ábécésorrend szerint korábbi.

Végül a könyvtár lehetővé teszi azt is, hogy előállítsuk egy sorozat permutációit (az elemek összes lehetséges sorrendjét):

Permutációk (§18.10) *<algorithm>*	
next_permutation()	Az ábécésorrend szerinti rendezés alapján következő permutáció.
prev_permutation()	Az ábécésorrend szerinti rendezés alapján előző permutáció.

Ezeken kívül, a *<numeric>* (§22.6) fejállományban néhány általános matematikai algoritmust is találhatunk.

Az algoritmusok leírásában a sablonparaméterek neve nagyon fontos. Az *In, Out, For, Bi* és *Ran* elnevezések sorrendben bemeneti bejárót, kimeneti bejárót, előre haladó bejárót, kétirányú bejárót, illetve közvetlen elérésű (véletlen elérésű) bejárót jelentenek (§19.2.1). A *Pred* egyparaméterű, a *BinPred* kétparaméterű predikátumot (állítást, logikai értékű függvényt, §18.4.2) határoz meg, míg a *Cmp* összehasonlító függvényre utal. Az *Op* egyoperandosú műveletet, a *BinOp* kétoperandosút vár. A hagyományok szerint sokkal hosszabb neveket kellett volna használnom a sablonparaméterek megnevezésére, de úgy vettem észre, hogy ha már egy kicsit is ismerjük a standard könyvtárat, a hosszú nevek inkább csak rontják az olvashatóságot.

A közvetlen elérésű bejárók használhatók kétirányú bejáróként is, a kétirányú bejárók előre haladó bejáróként, az előre haladó bejárók pedig akár bemeneti, akár kimeneti bejáróként (§19.2.1). Ha a sablonnak olyan típust adunk át, amely nem biztosítja a szükséges műveleteket, a sablon példányosításakor hibaüzenetet kapunk (§C.13.7). Ha olyan típust használunk, amelyben megvannak a szükséges műveletek, de jelentésük (szerepük) nem a megfelelő, akkor kiszámíthatatlan futási idejű viselkedésre kell felkészülnünk (§17.1.4).

18.3. Sorozatok és tárolók

Általános szabály, hogy mindennek a leggyakoribb felhasználási módja legyen a legrövidebb, a legegyszerűbb és a legbiztonságosabb. A standard könyvtár az általánosság érdekében itt-ott megsérti ezt a szabályt, de egy szabványos könyvtár esetében az általánosság mindennél fontosabb. A *42* első két előfordulását egy sorozatban például az alábbi programrészlettel kereshetjük meg

```
void f(list<int>& li)
{
  list<int>::iterator p = find(li.begin(),li.end(),42);          // első előfordulás
  if (p != li.end()) {
            list<int>::iterator q = find(++p,li.end(),42);    // második előfordulás
            // ...
  }
  // ...
}
```

Mivel a *find()* egy tárolókon működő művelet, valamilyen további eszköz segítségével lehetővé kell tennünk, hogy a második előfordulást is elérhessük. A „további eszköz” fogalmát általánosítani minden tárolóra és algoritmusra nagyon nehéz lenne, ezért a standard könyvtár algoritmusai csak sorozatokon működnek. Az algoritmusok bemenetét gyakran egy bejáró-pár adja, amely egy sorozatot határoz meg. Az első bejáró az első elemet jelöli ki, míg a második az utolsó utáni elemet (§3.8, §19.2). Az ilyen sorozatot „félig nyíltnak” nevezzük, mivel az első megadott elemet tartalmazza, de a másodikat nem. A félig nyílt sorozatok lehetővé teszik, hogy a legtöbb algoritmusnak ne kelljen egyedi esetként kezelnie az üres sorozatot.

Egy sorozatot gyakran tekinthetünk *tartománynak* (range) is, főleg ha elemeit közvetlenül elérhetjük. A félig nyílt tartományok hagyományos matematikai jelölése az *[első,utolsó)* vagy az *[első, utolsó[*. Fontos, hogy egy ilyen sorozat lehet tároló vagy annak részsorozata is, de bizonyos sorozatok, például a ki- és bemeneti adatfolyamok, egyáltalán nem kapcsolódnak tárolókhoz. A sorozatokkal kifejezett algoritmusok mindegyik esetben tökéletesen működnek.

18.3.1. Bemeneti sorozatok

Az *x.begin(), x.end()* párral gyakran jelöljük azt, hogy az *x* összes elemére szükségünk van, pedig ez a jelölés hosszadalmas, ráadásul sok hibalehetőséget hordoz magában. Például ha több bejárót is használunk, nagyon könnyen hívunk meg egy algoritmust sorozatot nem alkotó paraméterpárral:

```
void f(list<string>& fruit, list<string>& citrus)
{
  typedef list<string>::const_iterator LI;

  LI p1 = find(fruit.begin(),citrus.end(),"alma");      // helytelen! (különböző sorozatok)
  LI p2 = find(fruit.begin(),fruit.end(),"alma");       // rendben
  LI p3 = find(citrus.begin(),citrus.end(),"körte");    // rendben
  LI p4 = find(p2,p3,"peach");                          // helytelen! (különböző sorozatok)
  // ...
}
```

A példában két hiba is szerepel. Az első még elég nyilvánvaló (főleg ha várjuk a hibát), de a fordító már ezt sem könnyen találja meg. A második hibát viszont egy valódi programban nagyon nehéz felderíteni, még egy gyakorlott programozó számára is. Ha a bejárók számát sikerül csökkentenünk, akkor ezen hibák előfordulásának valószínűségét is csökkenthetjük. Az alábbiakban körvonalazunk egy megközelítést, amely a bemeneti sorozatok fogalmának bevezetésével ezt a problémát próbálja megoldani. Az algoritmusok későbbi bemutatásakor azonban nem használjuk majd a bemeneti sorozatokat, mert azok a standard könyvtárban nem szerepelnek, és ebben a fejezetben kizárólag a standard könyvtárral akarunk foglalkozni.

Az alapötlet az, hogy paraméterként egy sorozatot adunk meg:

```
template<class In, class T> In find(In first, In last, const T& v)    // szabványos
{
  while (first!=last && *first!=v) ++first;
  return first;
}

template<class In, class T> In find(Iseq<In> r, const T& v)           // bővítés
{
  return find(r.first,r.second,v);
}
```

A túlterhelés (overloading) (§13.3.2) általában lehetővé teszi, hogy az algoritmus bemeneti sorozat formáját használjuk, ha *Iseq* paramétert adunk át.

A bemeneti sorozatot természetesen bejáró-párként (§17.4.1.2) valósítjuk meg:

```
template<class In> struct Iseq : public pair<In,In> {
  Iseq(In i1, In i2) : pair<In,In>(i1,i2) { }
};
```

A *find()* függvény második változatának használatához közvetlenül is előállíthatunk *Iseq* bemeneti sorozatot:

```
  LI p = find(Iseq<LI>(fruit.begin(),fruit.end()),"alma");
```

Ez a megoldás azonban még körülményesebb, mint az eredeti *find()* függvény. Ahhoz, hogy ennek a megoldásnak hasznát vehessük, még egy egyszerű segédfüggvényre van szükség. Egy tárolóhoz megadott *Iseq* valójában nem más, mint az elemek sorozata az elsőtől (*begin()*) az utolsóig (*end()*):

```
template<class C> Iseq<C::iterator> iseq(C& c)          // tárolóra
{
  return Iseq<C::iterator>(c.begin(),c.end());
}
```

Ez a függvény lehetővé teszi, hogy a teljes tárolókra vonatkozó algoritmusokat tömören, ismétlések nélkül írjuk le:

```
void f(list<string>& ls)
{
  list<string>::iterator p = find(ls.begin(),ls.end(),"szabványos");
  list<string>::iterator q = find (iseq(ls),"bővítés");
  // ...
}
```

Az *iseq()* függvénynek könnyen elkészíthetjük olyan változatait is, amelyek tömbökhöz, bemeneti adatfolyamokhoz, vagy bármely más tárolóhoz (§18.13[6]) nyújtanak ilyen hozzáférést.

Az *Iseq* legfontosabb előnye, hogy a bemeneti sorozat lényegét világossá teszi. Gyakorlati haszna abban áll, hogy az *iseq()* megszünteti a kényelmetlen és hibalehetőséget magában hordozó ismétlődést, amelyet a bemeneti sorozat két bejáróval való megadása jelent.

A kimeneti sorozat fogalmának meghatározása szintén hasznos lehet, bár kevésbé egyszerű és kevésbé közvetlenül használható, mint a bemeneti sorozatok (§18.13[7], lásd még: §19.2.4).

18.4. Függvényobjektumok

Nagyon sok olyan algoritmus van, amely a sorozatokat csak bejárók és értékek segítségével kezeli. A *7* első előfordulását egy sorozatban például az alábbi módon találhatjuk meg:

```
void f(list<int>& c)
{
  list<int>::iterator p = find(c.begin(),c.end(),7);
  // ...
}
```

Egy kicsit érdekesebb eset, ha mi adunk meg egy függvényt, amelyet az algoritmus használ (§3.8.4). Az első, hétnél kisebb elemet például a következő programrészlet keresi meg:

```
bool less_than_7(int v)
{
  return v<7;
}

void f(list<int>& c)
{
  list<int>::iterator p = find_if(c.begin(),c.end(),less_than_7);
  // ...
}
```

Nagyon sok egyszerű helyzetben nyújtanak segítséget a paraméterként átadott függvények: logikai feltételként (predikátum), aritmetikai műveletként, olyan eljárásként, mellyel információt nyerhetünk ki az elemekből stb. Nem szokás és nem is hatékony minden felhasználási területre külön függvényt írni. Másrészt egyetlen függvény néha nem is képes megvalósítani igényeinket. Gyakran előfordul például, hogy a minden egyes elemre meghívott függvénynek a lefutások között meg kell tartania valamilyen információt. Az osztályok tagfüggvényei az ilyen feladatokat jobban végre tudják hajtani, mint az önálló eljárások, hiszen az objektumok képesek adatok tárolására és lehetőséget adnak azok kezdeti értékének beállítására is.

Nézzük, hogyan is készíthetünk egy függvényt – vagy pontosabban egy függvényszerű osztályt – egy összegzéshez:

```
template<class T> class Sum {
  T res;
public:
  Sum(T i = 0) : res(i) { }                 // kezdeti értékadás
  void operator()(T x) { res += x; }         // összegzés
```

```
    T result() const { return res; }              // összegzés visszaadása
};
```

Látható, hogy a *Sum* osztály olyan aritmetikai típusokhoz használható, melyek kezdőértéke nulla lehet és alkalmazható rájuk a += művelet:

```
void f(list<double>& ld)
{
    Sum<double> s;
    s = for_each(ld.begin(),ld.end(),s);          // s() meghívása ld minden elemére
    cout << "Az összeg: " << s.result() << '\n';
}
```

Ebben a programrészletben a *for_each()* (§18.5.1) függvény az *ld* minden egyes elemére meghívja a *Sum<double>::operator()(double)* tagfüggvényt, és eredményül a harmadik paraméterben megadott objektumot adja vissza.

Annak, hogy ez az utasítás egyáltalán működik, az az oka, hogy a *for_each()* nem teszi kötelezővé, hogy harmadik paramétere tényleg egy függvény legyen. Mindössze annyit feltételez, hogy ez a valami meghívható a megfelelő paraméterrel. Ezt a szerepet egy meghatározott objektum is betöltheti, sokszor hatékonyabban is, mint egy egyszerű függvény. Egy osztály függvényhívó operátorát például könnyebb optimalizálni, mint egy függvényt, melyet függvényre hivatkozó mutatóként adtunk át. Ezért a függvényobjektumok gyakran gyorsabban futnak, mint a szokásos függvények. Azon osztályok objektumait, melyekhez elkészítettük a függvényhívó operátort (§11.9), *függvényszerű objektumoknak*, *funktoroknak* (functor) vagy egyszerűen *függvényobjektumoknak* nevezzük.

18.4.1. Függvényobjektumok bázisosztályai

A standard könyvtár számos hasznos függvényobjektumot kínál. A függvényobjektumok elkészítésének megkönnyítéséhez a könyvtár két bázisosztályt tartalmaz:

```
template <class Arg, class Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

template <class Arg, class Arg2, class Res> struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};
```

Ezen osztályok célja, hogy szabványos neveket adjanak a paramétereknek és a visszatérési érték típusának, így a *unary_function* és a *binary_function* osztály leszármazottait használó programozók elérhetik azokat. A standard könyvtár következetesen használja ezeket az osztályokat, ami abban is segíti a programozót, hogy megállapítsa, mire is jó az adott függvényobjektum (§18.4.4.1).

18.4.2. Predikátumok

A *predikátum* (*állítás, feltétel; predicate*) egy olyan függvényobjektum (vagy függvény), amely logikai (bool) értéket ad vissza. A *<functional>* fejállományban például az alábbi definíciókat találhatjuk:

```
template <class T> struct logical_not : public unary_function<T,bool> {
  bool operator()(const T& x) const { return !x; }
};

template <class T> struct less : public binary_function<T,T,bool> {
  bool operator()(const T& x, const T& y) const { return x<y; }
};
```

Az egy- és kétoperandosú (unáris/bináris) predikátumokra gyakran van szükség a szabványos algoritmusok esetében. Például összehasonlíthatunk két sorozatot úgy, hogy megkeressük az első elemet, amely az egyik sorozatban nem kisebb, mint a neki megfelelő elem a másikban:

```
void f(vector<int>& vi, list<int>& li)
{
  typedef list<int>::iterator LI;
  typedef vector<int>::iterator VI;
  pair<VI,LI> p1 = mismatch(vi.begin(),vi.end(),li.begin(),less<int>());
  // ...
}
```

A *mismatch()* függvény a sorozatok összetartozó elemeire alkalmazza a megadott kétoperandusú predikátumot, mindaddig, amíg az igaz értéket nem ad vissza (§18.5.4). A visszatérési érték az a két bejáró, amelyeket az összehasonlítás nem összetartozónak talált. Mivel nem típust, hanem objektumot kell átadnunk, a *less<int>()* kifejezést használjuk (zárójelekkel) a *less<int>* forma helyett.

Elképzelhető, hogy az első olyan elem helyett, amely nem kisebb a párjánál, pont arra van szükségünk, amely kisebb annál. Ezt a feladatot az első olyan elempár megkeresésével végezhetjük el, amelyre a nagyobb vagy egyenlő (*greater_equal*) kiegészítő feltétel nem teljesül:

```
p1 = mismatch(vi.begin(),vi.end(),li.begin(),greater_equal<int>());
```

Egy másik lehetőség, hogy a sorozatokat fordított sorrendben adjuk meg és a kisebb vagy egyenlő (*less_equal*) műveletet használjuk:

```
pair<LI,VI> p2 = mismatch(li.begin(),li.end(),vi.begin(),less_equal<int>());
```

A §18.4.4.4 pontban azt is megnézzük, hogyan írhatjuk le közvetlenül a „nem kisebb” feltételt.

18.4.2.1. A predikátumok áttekintése

A <*functional*> fejállományban néhány általános predikátum található:

Predikátumok <*functional*>		
equal_to	bináris	arg1==arg2
not_equal_to	bináris	arg1!=arg2
greater	bináris	arg1>arg2
less	bináris	arg1<arg2
greater_equal	bináris	arg1>=arg2
less_equal	bináris	arg1<=arg2
logical_and	bináris	arg1&&arg2
logical_or	bináris	arg1 || arg2
logical_not	unáris	!arg

Az unáris egyparaméterű, a bináris kétparaméterű függvényt jelent, az *arg* (argumentum) a paramétereket jelöli. A *less* és a *logical_not* műveletet a §18.4.2 pont elején írtuk le.

A könyvtár által kínált predikátumokon kívül a programozó saját maga is létrehozhat ilyen függvényeket. Ezek a programozó által megadott predikátumok nagy szerepet játszanak a standard könyvtár algoritmusainak egyszerű és elegáns használatában. A predikátumok meghatározásának lehetősége különösen fontos akkor, ha olyan osztályra akarunk használni egy algoritmust, amely teljesen független a standard könyvtártól és annak algoritmusaitól. Például képzeljük el a §10.4.6 pontban bemutatott *Club* osztály következő változatát:

```
class Person { /* ... */ };

struct Club {
  string name;
  list<Person*> members;
  list<Person*> officers;
  // ...

  Club(const string& n);
};
```

Igen logikus feladat lenne, hogy megkeressünk egy adott nevű klubot egy *list<Club>* tárolóban. A standard könyvtár *find_if()* algoritmusa azonban egyáltalán nem ismeri a *Club* osztályt. A könyvtári algoritmusok tudják, hogyan lehet egyenlőséget vizsgálni, de a klubot mi nem a teljes értéke alapján akarjuk megtalálni, mindössze a *Club::name* adattagot akarjuk kulcsként használni. Tehát írunk egy olyan predikátumot, amely ezt a feltételt tükrözi:

```
class Club_eq : public unary_function<Club,bool> {
  string s;
public:
  explicit Club_eq(const string& ss) : s(ss) { }
  bool operator()(const Club& c) const { return c.name==s; }
};
```

A megfelelő predikátumok meghatározása általában nagyon egyszerű, és ha az általunk létrehozott típusokhoz megadtuk a megfelelő predikátumokat, akkor ezekre a szabványos algoritmusok ugyanolyan egyszerűen és hatékonyan használhatók lesznek, mint az egyszerű típusokból felépített tárolók esetében:

```
void f(list<Club>& lc)
{
  typedef list<Club>::iterator LCI;
  LCI p = find_if(lc.begin(),lc.end(),Club_eq("Étkező filozófusok"));
  // ...
}
```

18.4.3. Aritmetikai függvényobjektumok

Amikor numerikus osztályokat használunk, gyakran van szükségünk a szokásos aritmetikai műveletekre függvényobjektumok formájában. Ezért a *<functional>* állományban a következő műveletek deklarációja is szerepel:

Aritmetikai műveletek <*functional*>		
plus	bináris	arg1+arg2
minus	bináris	arg1–arg2
multiplies	bináris	arg1*arg2
divides	bináris	arg1/arg2
modulus	bináris	arg1%arg2
negate	unáris	–arg

A *multiplies* műveletet használhatjuk például arra, hogy két *vector* elemeit összeszorozzuk és ezzel egy harmadik vektort hozzunk létre:

```
void discount(vector<double>& a, vector<double>& b, vector<double>& res)
{
   transform(a.begin(),a.end(),b.begin(),back_inserter(res),multiplies<double>());
}
```

A *back_inserter()* eljárásról a §19.2.4. pontban lesz szó. Az aritmetikai algoritmusok némelyikével a §22.6 fejezetben részletesen foglalkozunk.

18.4.4. Lekötők, átalakítók és tagadók

Használhatunk olyan predikátumokat és aritmetikai függvényobjektumokat is, melyeket mi magunk írtunk, de hivatkozunk bennük a standard könyvtár által kínált eljárásokra. Ha azonban egy új predikátumra van szükségünk, előállíthatjuk azt egy már létező predikátum apró módosításával is. A standard könyvtár támogatja a függvényobjektumok ilyen felépítését:

§18.4.4.1 A *lekötők* (binder) lehetővé teszik, hogy egy kétparaméterű függvényobjektumot egyparaméterű függvényként használjuk, azáltal, hogy az egyik paraméterhez egy rögzített értéket kötnek.
§18.4.4.2 A *tagfüggvény-átalakítók* (member function adapter) lehetővé teszik, hogy tagfüggvényeket használjunk az algoritmusok paramétereként.
§18.4.4.3 A *függvényre hivatkozó mutatók átalakítói* (pointer to function adapter) lehetővé teszik, hogy függvényre hivatkozó mutatókat használjunk algoritmusok paramétereként.
§18.4.4.4 A *tagadók* (negater) segítségével egy állítás (predikátum) tagadását, ellentétét (negáltját) fejezhetjük ki.

Ezeket a függvényobjektumokat együtt *átalakítóknak* (adapter) nevezzük. Mindegyik átalakító azonos felépítésű és a *unary_function* és *binary_function* függvényobjektum-bázisosztályokon (§18.4.1) alapul. Mindegyikhez rendelkezésünkre áll egy segédfüggvény, mely

paraméterenként egy függvényobjektumot kap és a megfelelő függvényobjektumot adja vissza. Ha ezeket az osztályokat az *operator()()* művelettel hívjuk meg, akkor a kívánt feladat kerül végrehajtásra. Tehát az átalakító egyszerűen egy magasabb szintű függvény: egy függvényt kap paraméterként és ebből egy másik függvényt állít elő:

Lekötők, átalakítók, tagadók <*functional*>		
bind2nd(y)	*binder2nd*	Kétparaméterű függvény meghívása úgy, hogy *y* a második paraméter.
bind1st(x)	*binder1st*	Kétparaméterű függvény meghívása úgy, hogy *x* az első paraméter.
mem_fun()	*mem_fun_t*	Paraméter nélküli tagfüggvény meghívása mutatón keresztül.
	mem_fun1_t	Egyparaméterű tagfüggvény meghívása mutatón keresztül.
	const_mem_fun_t	Paraméter nélküli konstans tagfüggvény meghívása mutatón keresztül.
	const_mem_fun1_t	Egyparaméterű konstans tagfüggvény meghívása mutatón keresztül.
mem_fun_ref()	*mem_fun-ref_t*	Paraméter nélküli tagfüggvény meghívása referencián keresztül.
	mem_fun1_ref_t	Egyparaméterű tagfüggvény meghívása referencián keresztül.
	const_mem_fun_ref_t	Paraméter nélküli konstans tagfüggvény meghívása referencián keresztül.
	const_mem_fun1_ref_t	Egyparaméterű konstans tagfüggvény meghívása referencián keresztül.
ptr_fun()	*pointer_to_unary_function*	Egyparaméterű függvényre hivatkozó mutató meghívása.
ptr_fun()	*pointer_to_binary_function*	Kétparaméterű függvényre hivatkozó mutató meghívása.
not1()	*unary_negate*	Egyparaméterű predikátum negálása.
not2()	*binary_negate*	Kétparaméterű predikátum negálása.

18.4.4.1. Lekötők

Az olyan kétparaméterű predikátumok, mint a *less* (§18.4.2), igen hasznosak és rugalmasak. Gyakran tapasztaljuk azonban, hogy a legkényelmesebb predikátum az lenne, amely a tároló minden elemét ugyanahhoz a rögzített elemhez hasonlítaná. A §18.4. pontban bemutatott *less_than_7()* egy jellemző példa erre. A *less* műveletnek mindenképpen két paramétert kell megadnunk minden egyes híváskor, így közvetlenül nem használhatjuk ezt az eljárást. A megoldás a következő lehet:

```
template <class T> class less_than : public unary_function<T,bool> {
  T arg2;
public:
  explicit less_than(const T& x) : arg2(x) { }
  bool operator()(const T& x) const { return x<arg2; }
};
```

Ezek után már leírhatjuk a következőt:

```
void f(list<int>& c)
{
  list<int>::const_iterator p = find_if(c.begin(),c.end(),less_than<int>(7));
  // ...
}
```

A *less_than(7)* forma helyett a *less_than<int>(7)* alakot kell használnunk, mert az *<int>* sablonparaméter nem vezethető le a konstruktor paraméterének (7) típusából (§13.3.1).

A *less_than* predikátum általában igen hasznos. A fenti művelet viszont úgy jött létre, hogy rögzítettük, lekötöttük (bind) a *less* függvény második paraméterét. Az ilyen szerkezet, melyben tehát egy paramétert rögzítünk, minden helyzetben ugyanúgy megvalósítható, és annyira általános és hasznos, hogy a standard könyvtár egy külön osztályt kínál erre a célra:

```
template <class BinOp>
class binder2nd : public unary_function<BinOp::first_argument_type, BinOp::result_type> {
protected:
  BinOp op;
  typename BinOp::second_argument_type arg2;
public:
  binder2nd(const BinOp& x, const typename BinOp::second_argument_type& v)
          : op(x), arg2(v) { }
  result_type operator()(const argument_type& x) const { return op(x,arg2); }
};
```

```
template <class BinOp, class T> binder2nd<BinOp> bind2nd(const BinOp& op, const T& v)
{
  return binder2nd<BinOp>(op,v);
}
```

A *bind2nd()* függvényobjektumot használhatjuk például arra, hogy meghatározzuk a „kisebb, mint 7” egyparaméterű feltételt a *less* függvény és a *7* érték felhasználásával:

```
void f(list<int>& c)
{
  list<int>::const_iterator p = find_if(c.begin(),c.end(),bind2nd(less<int>(),7));
  // ...
}
```

Elég olvasható ez a megoldás? Elég hatékony? Egy átlagos C++-változat megvalósításával összehasonlítva bizony hatékonyabb, mint az eredeti, melyben a §18.4. pont *less_than_7()* függvényét használtuk – akár időigény, akár tárhasználat szempontjából! Az összehasonlítás ráadásul könnyen fordítható helyben kifejtett függvényként.

A jelölés logikus, de megszokásához kell egy kis idő, ezért érdemes konkrét névvel megadni a kötött paraméterű műveletet is:

```
template <class T> struct less_than : public binder2nd< less<T> > {
  explicit less_than(const T& x) : binder2nd(less<T>(),x) { }
};

void f(list<int>& c)
{
  list<int>::const_iterator p = find_if(c.begin(),c.end(),less_than<int>(7));
  // ...
}
```

Fontos, hogy a *less_than* eljárást a *less* művelettel határozzuk meg, és nem közvetlenül a < operátorral, mert így a *less_than* használni tudja a *less* bármely elképzelhető specializációját (§13.5, §19.2.2).

A *bind2nd()* és a *binder2nd* mellett a *<functional>* fejállományban megtalálhatjuk a *bind1st()* és *binder1st* osztályt is, melyekkel egy kétparaméterű függvény első paraméterét rögzíthetjük.

Egy paraméter lekötése, amit a *bind1st()* és a *bind2nd()* nyújt, nagyon hasonlít ahhoz az általános szolgáltatáshoz, amelyet *Currying*-nek neveznek.

18.4.4.2. Tagfüggvény-átalakítók

A legtöbb algoritmus felhasznál valamilyen szabványos vagy programozó által megadott műveletet. Természetesen gyakran tagfüggvényt szeretnénk meghívni. Például (§3.8.5):

```
void draw_all(list<Shape*>& c)
{
  for_each(c.begin(),c.end(),&Shape::draw);     // hoppá! hiba
}
```

A problémát az jelenti, hogy egy *mf()* tagfüggvény meghívásához az objektumot is meg kell adnunk: *p->mf()*. Az olyan algoritmusok azonban, mint a *for_each()*, a paraméterként kapott függvényeket az egyszerű függvényhívó utasítással hajtják végre: *f()*. Ezért szükségünk van egy olyan következetes és hatékony módszerre, amivel létrehozhatunk valamit, ami képes rávenni az algoritmusokat, hogy tagfüggvényeket hívjanak meg. Egy lehetséges megoldás az lenne, hogy minden algoritmusnak két példányát hozzuk létre: az egyik a tagfüggvényekkel lenne használható, a másik az egyszerű függvényekkel. A helyzet még ennél is rosszabb, mert olyan változatokra is szükségünk lenne, amelyek objektumok tárolóin működnének (nem objektumokra hivatkozó mutatókon). Ugyanúgy, mint a lekötők (§18.4.4.1) esetében, a megoldást itt is egy új osztály és egy függvény megírása jelenti. Először vizsgáljuk meg azt az általános esetet, amikor egy tagfüggvényt paraméterek nélkül szeretnénk meghívni egy mutatókat tartalmazó tároló minden elemére:

```
template<class R, class T> class mem_fun_t : public unary_function<T*,R> {
  R (T::*pmf)();
public:
  explicit mem_fun_t(R (T::*p)()) :pmf(p) {}
  R operator()(T* p) const { return (p->*pmf)(); }         // meghívás mutatón keresztül
};

template<class R, class T> mem_fun_t<R,T> mem_fun(R (T::*f)())
{
  return mem_fun_t<R,T>(f);
}
```

Ez megoldja a példában szereplő *Shape::draw()* hívást:

```
void draw_all(list<Shape*>& lsp)      // paraméter nélküli tag meghívása objektumra
                                      // hivatkozó mutatón keresztül
{
  for_each(lsp.begin(),lsp.end(),mem_fun(&Shape::draw));      // minden alakzat
                                                              // kirajzolása
}
```

Ezenkívül szükségünk van egy olyan osztályra és *mem_fun()* függvényre is, amelyek a paraméteres tagfüggvények kezelésére képesek. Olyan változatok is kellenek, melyekkel közvetlenül objektumokat használhatunk, nem pedig objektumokra hivatkozó mutatókat. Ezek neve *mem_fun_ref()*. Végül szükség van a *const* tagfüggvényeket kezelő változatokra is:

```
template<class R, class T> mem_fun_t<R,T> mem_fun(R (T::*f)());
// és az egyparaméterű tagokra, a const tagokra, és az egyparaméterű const tagokra
// vonatkozó változatok (lásd a §18.4.4 táblázatot)

template<class R, class T> mem_fun_ref_t<R,T> mem_fun_ref(R (T::*f)());
// és az egyparaméterű tagokra, a const tagokra, és az egyparaméterű const tagokra
// vonatkozó változatok (lásd a §18.4.4 táblázatot)
```

A *<functional>* fejállomány tagfüggvény-átalakítóinak felhasználásával a következőket írhatjuk:

```
void f(list<string>& ls)          // paraméter nélküli tagfüggvény használata objektumra
{
  typedef list<string>::iterator LSI;
  LSI p = find_if(ls.begin(),ls.end(),mem_fun_ref(&string::empty));          // "" keresése
}

void rotate_all(list<Shape*>& ls, int angle)
  // egyparaméterű tagfüggvény használata objektumra hivatkozó mutatón keresztül
{
  for_each(ls.begin(),ls.end(),bind2nd(mem_fun(&Shape::rotate),angle));
}
```

A standard könyvtárnak nem kell foglalkoznia azokkal a tagfüggvényekkel, melyek egynél több paramétert várnak, mert a standard könyvtárban nincs olyan algoritmus, amely kettőnél több paraméterű függvényt várna operandusként.

18.4.4.3. Függvényre hivatkozó mutatók átalakítói

Egy algoritmus nem foglalkozik azzal, hogy a „függvényparaméter" milyen formában adott: függvény, függvényre hivatkozó mutató vagy függvényobjektum. Ellenben a lekötők (§18.4.4.1) számára ez fontos, mert tárolniuk kell egy másolatot a későbbi felhasználáshoz. A standard könyvtár két átalakítót kínál a függvényekre hivatkozó mutatók szabványos algoritmusokban való felhasználásához. A definíció és a megvalósítás nagyon hasonlít a tagfüggvény-átalakítóknál (§18.4.4.2) használt megoldásra. Most is két függvényt és két osztályt használunk:

```
template <class A, class R> pointer_to_unary_function<A,R> ptr_fun(R (*f)(A));

template <class A, class A2, class R>
  pointer_to_binary_function<A,A2,R> ptr_fun(R (*f)(A, A2));
```

A függvényre hivatkozó mutatók ezen átalakítói lehetővé teszik, hogy a szokásos függvényeket a lekötőkkel együtt használjuk:

```
class Record { /* ... */ };

bool name_key_eq(const Record&, const char*);// összehasonlítás nevek alapján
bool ssn_key_eq(const Record&, long);          // összehasonlítás számok alapján

void f(list<Record>& lr)       // függvényre hivatkozó mutató használata
{
  typedef typename list<Record>::iterator LI;
  LI p = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(name_key_eq),"John Brown"));
  LI q = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(ssn_key_eq),1234567890));
  // ...
}
```

A fenti utasítások azokat az elemeket keresik meg az *lr* listában, melyekben a *John Brown*, illetve az *1234567890* kulcsérték szerepel.

18.4.4.4. Tagadók

A predikátum-tagadók (negater) a lekötőkhöz kapcsolódnak abból a szempontból, hogy egy műveletet kapnak paraméterként és ebből egy másik műveletet állítanak elő. A tagadók definíciója és megvalósítása követi a tagfüggvény-átalakítóknál (§18.4.4.2) alkalmazott formát. Meghatározásuk rendkívül egyszerű, de ezt az egyszerűséget kicsit elhomályosítja a hosszú szabványos nevek használata:

```
template <class Pred>
class unary_negate : public unary_function<typename Pred::argument_type,bool> {

  Pred op;
public:
  explicit unary_negate(const Pred& p) : op(p) { }
  bool operator()(const argument_type& x) const { return !op(x); }
};

template <class Pred>
class binary_negate : public binary_function<typename Pred::first_argument_type,
                                        typename Pred::second_argument_type, bool> {
```

```
  typedef first_argument_type Arg;
  typedef second_argument_type Arg2;

  Pred op;
public:
  explicit binary_negate(const Pred& p) : op(p) { }
  bool operator()(const Arg& x, const Arg2& y) const { return !op(x,y); }
};

template<class Pred> unary_negate<Pred> not1(const Pred& p);    // unáris tagadása
template<class Pred> binary_negate<Pred> not2(const Pred& p);   // bináris tagadása
```

Ezek az osztályok és függvények is a *<functional>* fejállományban kaptak helyet. A *first_argument_type*, *second_argument_type* stb. elnevezések a *unary_function*, illetve a *binary_function* szabványos bázisosztályokból erednek.

Ugyanúgy, mint a lekötők, a tagadók is kényelmesen használhatók segédfüggvényeiken keresztül. Például a „nem kisebb, mint” kétparaméterű predikátumot is egyszerűen leírhatjuk, és megkereshetjük vele az első két olyan szomszédos elemet, melyek közül az első nagyobb vagy egyenlő, mint a második:

```
void f(vector<int>& vi, list<int>& li)    // a §18.4.2 példájának javított változata
{
  // ...
  p1 = mismatch(vi.begin(),vi.end(),li.begin(),not2(less<int>()));
  // ...
}
```

Tehát a *p1* kapja meg az első olyan elempárt, melyre a „nem kisebb, mint” művelet hamis értéket ad vissza.

A predikátumok logikai értékekkel dolgoznak, így a bitenkénti operátoroknak (|, &, ^, ~) nincs megfelelőjük.

Természetesen a lekötők, az átalakítók és a tagadók együtt is használhatók:

```
extern "C" int strcmp(const char*,const char*); // a <cstdlib> fejállományból

void f(list<char*>& ls)                          // függvényre hivatkozó mutató használata
{
  typedef typename list<char*>::const_iterator LI;
  LI p = find_if(ls.begin(),ls.end(),not1(bind2nd(ptr_fun(strcmp),"vicces")));
}
```

Ez a kódrészlet az első olyan elemet keresi meg az *ls* listában, amely a *"vicces"* C stílusú karakterláncot tartalmazza. A tagadóra azért van szükség, mert a *strcmp()* függvény akkor ad vissza *0* értéket, ha a két karakterlánc egyenlő.

18.5. Nem módosító algoritmusok sorozatokon

A sorozatok nem módosító algoritmusai elsősorban arra szolgálnak, hogy a sorozatokban anélkül kereshessünk meg bizonyos elemeket, hogy ciklust írnánk. Ezen kívül lehetőséget adnak arra, hogy az elemekről megtudjunk minden létező információt. Ezek az algoritmusok csak konstans bejárókat (§19.2.1) használnak és a *for_each()* kivételével nem használhatók olyan műveletek elvégzésére, melyek a sorozat elemeit megváltoztatnák.

18.5.1. A for_each

Könyvtárakat azért használunk, hogy ne nekünk kelljen azzal fáradozni, amit valaki más már megvalósított. Egy könyvtár függvényeinek, osztályainak, algoritmusainak stb. használata megkönnyíti egy program megtervezését, megírását, tesztelését és dokumentálását is. A standard könyvtár használata ezenkívül olvashatóbbá is teszi programunkat olyanok számára, akik ismerik a könyvtárat, hiszen nem kell időt tölteniük a „házilag összeeszkábált" algoritmusok értelmezésével.

A standard könyvtár algoritmusainak legfőbb előnye, hogy a programozónak nem kell megírnia bizonyos ciklusokat. A ciklusok nehézkesek és könnyen követhetünk el bennük hibákat. A *for_each()* algoritmus a legegyszerűbb algoritmus, abban az értelemben, hogy semmi mást nem csinál, minthogy egy ciklust helyettesít, egy sorozat minden elemére végrehajtva a paraméterében megadott műveletet:

```
template<class In, class Op> Op for_each(In first, In last, Op f)
{
  while (first != last) f(*first++);
  return f;
}
```

Milyen függvényeket akarunk ilyen formában meghívni? Ha az elemekről akarunk információkat összegyűjteni, az *accumulate()* függvényt (§22.6) használhatjuk. Ha meg akarunk találni valamit egy sorozatban, rendelkezésünkre áll a *find()* és a *find_if()* algoritmus. Ha bi-

zonyos elemeket törölni vagy módosítani szeretnénk, a *remove()* (§18.6.5), illetve a *replace()* (§18.6.4) jelent egyszerűbb megoldást. Tehát mielőtt használni kezdjük a *for_each()* eljárást, gondoljuk végig, nincs-e céljainknak jobban megfelelő algoritmus.

A *for_each()* eredménye az a függvény vagy függvényobjektum, amelyet harmadik paraméterként megadtunk. A §18.4 pontban a *Sum* példa bemutatta, hogy ez a megoldás lehetővé teszi az eredmények visszaadását a hívónak.

A *for_each()* gyakori felhasználási területe az, hogy egy sorozat elemeiből bizonyos információkat fejtünk ki. Például összegyűjthetünk neveket klubok egy listájából:

```
void extract(const list<Club>& lc, list<Person*>& off)
                                        // hivatalnokok áthelyezése 'lc'-ből 'off'-ba
{
  for_each(lc.begin(),lc.end(),Extract_officers(off));
}
```

A §18.4 és a §18.4.2. pontban szereplő példáknak megfelelően készíthetünk egy függvényosztályt, amely kikeresi a kívánt információt. Ebben az esetben a kigyűjtendő neveket a *list<Person*>* tartalmazza a *Club* objektumokon belül, ezért az *Extract_officers* függvénynek ki kell másolnia a hivatalnokokat (officer) a *Club* objektumok *officers* listájából a gyűjtő listába:

```
class Extract_officers {
  list<Person*>& lst;
public:
  explicit Extract_officers(list<Person*>& x) : lst(x) { }

  void operator()(const Club& c)
          { copy(c.officers.begin(),c.officers.end(),back_inserter(lst)); }
};
```

A nevek kiíratását szintén a *for_each()* függvénnyel végezhetjük:

```
void extract_and_print(const list<Club>& lc)
{
  list<Person*> off;
  extract(lc,off);
  for_each(off.begin(),off.end(),Print_name(cout));
}
```

A *Print_name* függvény megírását meghagyjuk feladatnak (§18.13[4]).

A *for_each()* algoritmust a nem módosító eljárások közé soroltuk, mert közvetlenül nem módosít. Ha azonban egy nem konstans sorozatra hívjuk meg, akkor a harmadik paraméterben megadott művelet módosíthatja a sorozatot. (Nézzük meg például a *negate()* függvény használatát a §11.9 pontban.)

18.5.2. A find függvénycsalád

A *find()* algoritmusok végignéznek egy sorozatot (vagy sorozatpárt), és megkeresnek egy konkrét értéket vagy egy olyan elemet, amelyre valamilyen állítás (predikátum) teljesül. A *find()* legegyszerűbb változatai csak ezt a feladatot végzik el:

```
template<class In, class T> In find(In first, In last, const T& val);

template<class In, class Pred> In find_if(In first, In last, Pred p);
```

A *find()* és a *find_if()* egy bejárót ad vissza, amely az első olyan elemre mutat, amely a keresés feltételének megfelel. Valójában a *find()* felfogható a *find_if()* egy olyan változatának is, ahol a vizsgált predikátum az ==. Miért nem lett mindkét függvény neve *find()*? Azért, mert függvény-túlterheléssel nem mindig tudunk különbséget tenni két azonos paraméterszámú sablon függvény között:

```
bool pred(int);

void f(vector<bool(*f)(int)>& v1, vector<int>& v2)
{
  find(v1.begin(),v1.end(),pred);       // 'pred' keresése
  find_if(v2.begin(),v2.end(),pred);    // azon int keresése, amelyre pred() igazat ad vissza
}
```

Ha a *find()* és a *find_if()* függvénynek ugyanaz lenne a neve, akkor igen meglepő többértelműséggel találkoztunk volna. Általában az *_if* utótag azt jelzi, hogy az algoritmus egy predikátumot vár paraméterként.

A *find_first_of()* algoritmus egy sorozat első olyan elemét keresi meg, amely megtalálható a második sorozatban is:

```
template<class For, class For2>
  For find_first_of(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
  For find_first_of(For first, For last, For2 first2, For2 last2, BinPred p);
```

Például:

```
int x[] = { 1,3,4 };
int y[] = { 0,2,3,4,5};

void f()
{
  int* p = find_first_of(x,x+3,y,y+5);          // p = &x[1]
  int* q = find_first_of(p+1,x+3,y,y+5);        // q = &x[2]
}
```

A *p* mutató az *x[1]* elemre fog mutatni, mert a *3* az első olyan eleme az *x*-nek, amely megtalálható az *y*-ban. Hasonlóan a *q* az *x[2]*-re fog mutatni.

Az *adjacent_find()* algoritmus két egymás utáni, egyező elemet keres:

```
template<class For> For adjacent_find(For first, For last);

template<class For, class BinPred> For adjacent_find(For first, For last, BinPred p);
```

A visszatérési érték egy bejáró, amely az első megfelelő elemre mutat:

```
void f(vector<string>& text)
{
  vector<string>::iterator p = adjacent_find(text.begin(),text.end());
  if (p!=text.end() && *p=="az") {       // Már megint kétszer szerepel az "az"!
            text.erase(p);
            // ...
  }
}
```

18.5.3. A count()

A *count()* és a *count_if()* függvény egy érték előfordulásainak számát adja meg egy sorozatban:

```
template<class In, class T>
  typename iterator_traits<In>::difference_type count(In first, In last, const T& val);

template<class In, class Pred>
  typename iterator_traits<In>::difference_type count_if(In first, In last, Pred p);
```

A *count()* visszatérési értéke nagyon érdekes. Képzeljük el, hogy a *count()*-ot az alábbi egyszerű függvénnyel valósítjuk meg:

```
template<class In, class T> int count(In first, In last, const T& val)
{
  int res = 0;
  while (first != last) if (*first++ == val) ++res;
  return res;
}
```

A gond az, hogy egy *int* lehet, hogy nem felel meg visszatérési értéknek. Egy olyan számítógépen, ahol az *int* típus elég kicsi, elképzelhető, hogy túl sok elem van a sorozatban, így a *count()* azt nem tudja egy *int*-be helyezni. Egy nagyteljesítményű programban, egyedi rendszeren viszont érdemesebb a számláló által visszaadott értéket egy *short*-ban tárolni.

Abban biztosak lehetünk, hogy egy sorozat elemeinek száma nem nagyobb, mint a két bejárója közötti legnagyobb különbség (§19.2.1). Ezért az első gondolatunk a probléma megoldására az lehet, hogy a visszatérési érték típusát a következőképpen határozzuk meg:

```
typename In::difference_type
```

Egy szabványos algoritmusnak azonban a beépített tömbökre ugyanúgy kell működnie, mint a szabványos tárolókra:

```
void f(const char* p, int size)
{
  int n = count(p,p+size,'e');          // az 'e' betű előfordulásainak megszámlálása
}
```

Sajnos az *int*::difference_type* kifejezés a C++-ban nem értelmezhető. Ez a probléma az *iterator_traits* típus (§19.2.2) részleges specializációjával oldható meg.

18.5.4. Egyenlőség és eltérés

Az *equal()* és a *mismatch()* függvény két sorozatot hasonlít össze:

```
template<class In, class In2> bool equal(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
  bool equal(In first, In last, In2 first2, BinPred p);
```

```
template<class In, class In2> pair<In, In2> mismatch(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
  pair<In, In2> mismatch(In first, In last, In2 first2, BinPred p);
```

Az *equal()* algoritmus egyszerűen azt mondja meg, hogy a két sorozat minden két megfelelő eleme megegyezik-e, míg a *mismatch()* az első olyan elempárt keresi, melyek nem egyenlőek, és ezekre mutató bejárókat ad vissza. A második sorozat végét nem kell megadnunk (tehát nincs *last2*), mert a rendszer azt feltételezi, hogy az legalább olyan hosszú, mint az első sorozat és a *last2* értéket a *first2+(last-first)* kifejezéssel számítja ki. Ezt a módszert gyakran láthatjuk a standard könyvtárban, amikor két sorozat összetartozó elemei között végzünk valamilyen műveletet.

A §18.5.1 pontban már említettük, hogy ezek az algoritmusok sokkal hasznosabbak, mint azt első ránézésre gondolnánk, mert a programozó határozhatja meg azt a predikátumot, melyet az elemek egyenértékűségének eldöntéséhez akar használni.

A sorozatoknak nem is kell ugyanolyan típusúaknak lenniük:

```
void f(list<int>& li, vector<double>& vd)
{
  bool b = equal(li.begin(),li.end(),vd.begin());
}
```

Az egyetlen kikötés, hogy a predikátum paramétereiként használhassuk az elemeket.

A *mismatch()* két változata csak a predikátumok használatában különbözik. Valójában megvalósíthatjuk őket egyetlen függvénnyel is, amelynek van alapértelmezett sablonparamétere:

```
template<class In, class In2, class BinPred>
pair<In, In2> mismatch(In first, In last, In2 first2,
                        BinPred p = equal_to<In::value_type>())     // §18.4.2.1
{
  while (first != last && p(*first,*first2)) {
        ++first;
        ++first2;
  }
  return pair<In,In2>(first,first2);
}
```

A két külön függvény megadása és az egyetlen, alapértelmezett paraméterrel meghatározott függvény között akkor láthatjuk a különbséget, ha mutatókat adunk át a függvényeknek. Ennek ellenére, ha úgy gondolunk a szabványos algoritmusok különböző változataira, mint egy alapértelmezett predikátummal rendelkező függvényre, akkor körülbelül feleannyi sablon függvényre kell emlékeznünk.

18.5.5. Keresés

A *search()*, a *search_n()* és a *find_end()* algoritmus egy részsorozatot keres egy másik sorozatban:

```
template<class For, class For2>
  For search(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
  For search(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class For2>
  For find_end(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
  For find_end(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class Size, class T>
  For search_n(For first, For last, Size n, const T& val);

template<class For, class Size, class T, class BinPred>
  For search_n(For first, For last, Size n, const T& val, BinPred p);
```

A *search()* függvény a másodikként megadott sorozatot keresi az elsőben. Ha megtalálható ez a részsorozat, egy bejárót kapunk eredményül, amely az első illeszkedő elemet jelöli ki az első sorozatban. Ha a keresés sikertelen, akkor a sorozat végét (*last*) kapjuk eredményképpen. Tehát a visszatérési érték mindig a *[first,last]* tartományban van:

```
string quote("Minek vesztegessük az időt tanulásra, mikor a tudatlanság azonnali?");

bool in_quote(const string& s)
{
  typedef string::const_iterator I;
  I p = search(quote.begin(),quote.end(),s.begin(),s.end()); // s keresése az idézetben (quote)
  return p!=quote.end();
}
```

```
void g()
{
    bool b1 = in_quote("tanulásra");      // b1 = true
    bool b2 = in_quote("tanításra");      // b2 = false
}
```

Tehát a *search()* művelettel egy részsorozatot kereshetünk mindenféle sorozatra általánosítva. Ebből már érezhetjük, hogy a *search()* egy nagyon hasznos algoritmus.

A *find_end()* is a másodikként megadott sorozatot keresi részsorozatként az elsőben. Ha sikeres a keresés, a *find_end()* az illeszkedő rész utolsó elemére mutató bejárót adja vissza. Mondhatjuk azt is, hogy a *find_end()* visszafelé végzi ugyanazt a keresést, mint a *search()*. Tehát nem a részsorozat első előfordulását kapjuk meg, hanem az utolsót. A *search_n()* eljárás egy olyan részsorozatot keres, amely legalább *n* hosszon a *value* paraméterben megadott értéket tartalmazza. A visszatérési érték egy olyan bejáró, amely az *n* darab illeszkedés első elemére mutat a sorozatban.

18.6. Módosító algoritmusok sorozatokra

Ha meg akarunk változtatni egy sorozatot, akkor megtehetjük, hogy egyesével végignézzük az elemeket és közben módosítjuk a megfelelő értékeket. De ha lehetőség van rá, akkor ezt a programozási stílust érdemes elkerülnünk. Helyette egyszerűbb és rendszerezettebb megoldás áll a rendelkezésünkre: használjuk a szabványos algoritmusokat a sorozatok bejárására és a módosító műveletek elvégzésére. A nem módosító algoritmusok (§18.5) ugyanígy mennek végig az elemeken, de csak kiolvassák az információkat. A módosító algoritmusok segítségével a leggyakoribb frissítési feladatokat végezhetjük el. Ezek egy része az eredeti sorozatot módosítja, míg mások új sorozatot hoznak létre az általunk megadott sorozat alapján.

A szabványos algoritmusok a bejárók segítségével férnek hozzá az adatszerkezetekhez. Ebből következik, hogy egy új elem beszúrása egy tárolóba vagy egy elem törlése nem egyszerű feladat. Például, ha csak egy bejáró áll rendelkezésünkre, hogyan találhatjuk meg azt a tárolót, amelyből a kijelölt elemet törölni kell? Hacsak nem használunk egyedi bejárókat (például beszúrókat, inserter, §3.8, §19.2.4), a bejárókon keresztül végzett műveletek nem változtathatják meg a tároló méretét. Az elemek beszúrása és törlése helyett az algoritmusok csak az elemek értékét változtatják meg, illetve felcserélnek vagy másolnak elemeket. Még a *remove()* is úgy működik, hogy csak felülírja a törölni kívánt elemeket (§18.6.5).

Az alapvető módosító algoritmusok általában egy módosított másolatot hoznak létre a megadott sorozatból. Azok az algoritmusok, melyek sorozatmódosítóknak tűnnek, valójában csak a másolók módosított változatai.

18.6.1. Másolás

A másolás a legegyszerűbb módszer arra, hogy egy sorozatból egy másikat állítsunk elő. Az alapvető másoló műveletek rendkívül egyszerűek:

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
   while (first != last) *res++ = *first++;
   return res;
}

template<class Bi, class Bi2> Bi2 copy_backward(Bi first, Bi last, Bi2 res)
{
   while (first != last) *--res = *--last;
   return res;
}
```

A másoló algoritmus kimenetének nem kell feltétlenül tárolónak lennie. Bármit használhatunk, amihez kimeneti bejáró (§19.2.6) megadható:

```
void f(list<Club>& lc, ostream& os)
{
   copy(lc.begin(),lc.end(),ostream_iterator<Club>(os));
}
```

Egy sorozat beolvasásához meg kell adnunk, hogy hol kezdődik és hol ér véget. Az íráshoz csak az az egy bejáró kell, amelyik megmondja, hová írjunk. Arra azonban ilyenkor is figyelnünk kell, hogy ne írjunk a fogadó tároló határain túlra. A probléma egyik lehetséges megoldása a beszúró (inserter) bejárók (§19.2.4) használata, mellyel a fogadó adatszerkezetet bővíthetjük, ha arra szükség van:

```
void f(vector<char>& vs)
{
   vector<char> v;

   copy(vs.begin(),vs.end(),v.begin());           // túlírhat v végén
   copy(vs.begin(),vs.end(),back_inserter(v));    // elemek hozzáadása vs-ből v végéhez
}
```

A bemeneti és a kimeneti sorozat átfedhetik egymást. Ha a sorozatok között nincs átfedés vagy a kimeneti sorozat vége a bemeneti sorozat belsejében van, akkor a *copy()* függvényt használjuk. A *copy_backward()* utasításra akkor van szükség, ha a kimeneti sorozat eleje van a bemeneti sorozatban. Ez a függvény ilyen helyzetben addig nem írja felül az elemeket, amíg azokról másolat nem készült (lásd még §18.13[13]).

Természetesen ahhoz, hogy valamit visszafelé másoljunk le, egy kétirányú bejáróra (§19.2.1) van szükségünk, mind a bemeneti, mind a kimeneti sorozatban:

```
void f(vector<char>& vc)
{
  vector<char> v(vc.size());

  copy_backward(vc.begin(),vc.end(),ostream_iterator<char>(cout));     // hiba

  copy_backward(vc.begin(),vc.end(),v.end());                          // rendben
  copy(v.begin(),v.end(),ostream_iterator<char>(cout));                // rendben

}
```

Gyakran olyan elemeket szeretnénk másolni, amelyek egy bizonyos feltételt teljesítenek. Sajnos a *copy_if()* függvény valahogy kimaradt a standard könyvtár által nyújtott algoritmusok sorából (mea culpa). De ha szükségünk van rá, pillanatok alatt megírhatjuk:

```
template<class In, class Out, class Pred> Out copy_if(In first, In last, Out res, Pred p)
{
  while (first != last) {
          if (p(*first)) *res++ = *first;
          ++first;
  }
  return res;
}
```

Ezután ha az *n* értéknél nagyobb elemeket akarjuk megjeleníteni, a következő eljárást használhatjuk:

```
void f(list<int>&ld, int n, ostream& os)
{
  copy_if(ld.begin(),ld.end(),ostream_iterator<int>(os),bind2nd(greater<int>(),n));
}
```

(Lásd még a *remove_copy_if()*leírását is a §18.6.5 pontban.)

18.6.2. Transzformációk

Egy kicsit félrevezető a neve, ugyanis a *transform()* nem feltétlenül változtatja meg a bemenetét. Ehelyett egy olyan kimenetet állít elő, amely a bemeneten végzett felhasználói művelet eredménye:

```
template<class In, class Out, class Op>
Out transform(In first, In last, Out res, Op op)
{
  while (first != last) *res++ = op(*first++);
  return res;
}

template<class In, class In2, class Out, class BinOp>
Out transform(In first, In last, In2 first2, Out res, BinOp op)
{
  while (first != last) *res++ = op(*first++,*first2++);
  return res;
}
```

A *transform()* függvény első változata, amely csak egy sorozatot olvas be, nagyon hasonlít az egyszerű másolásra. A különbség mindössze annyi, hogy a közvetlen kiírás helyett előbb egy transzformációt (átalakítást) is elvégez az elemen. Így a *copy()* eljárást a *transform()* függvénnyel is meghatározhattuk volna, úgy, hogy az elemmódosító művelet egyszerűen csak visszaadja a paraméterét:

```
template<class T> T identity(const T& x) { return x; }

template<class In, class Out> Out copy(In first, In last, Out res)
{
  return transform(first,last,res,identity);
}
```

Az *identity* explicit minősítése ahhoz szükséges, hogy a függvénysablonból konkrét függvényt kapjunk, az *iterator_traits* sablont (§19.2.2) pedig azért használtuk, hogy az *In* elemtípusához jussunk.

A *transform()* függvényt tekinthetjük a *for_each()* egy változatának is, amely közvetlenül állítja elő kimenetét. Például klubok listájából a *transform()* segítségével készíthetünk egy olyan listát, amely csak a klubok neveit tárolja:

```
string nameof(const Club& c)          // név kinyerése
{
  return c.name;
}
```

```
void f(list<Club>& lc)
{
  transform(lc.begin(),lc.end(),ostream_iterator<string>(cout),nameof);
}
```

A *transform()* függvény elnevezésének egyik oka az, hogy az eredményt gyakran visszaírjuk oda, ahonnan a paramétert kaptuk. Például ha olyan objektumokat akarunk törölni, melyekre mutatók hivatkoznak, a következő eljárást használhatjuk:

```
struct Delete_ptr {     //függvényobjektum használata helyben kifejtéshez (inline fordításhoz)
  template<class T> T* operator() (T* p) { delete p; return 0; }
};

void purge(deque<Shape*>& s)
{
  transform(s.begin(),s.end(),s.begin(),Delete_ptr);
}
```

A *transform()* algoritmus eredménye mindig egy kimeneti sorozat. A fenti példában az eredményt az eredeti bemeneti sorozatba irányítottuk vissza, így a *Delete_ptr()(p)* jelentése *p=Delete_ptr()(p)* lesz. Ez indokolja a *0* visszatérési értéket a *Delete_ptr::operator()()* függvényben.

A *transform()* algoritmus másik változata két bemeneti sorozattal dolgozik. Ez lehetővé teszi, hogy két sorozat adatait használjuk fel az új sorozat létrehozásához. Egy animációs programban például szükség lehet egy olyan eljárásra, amely alakzatok sorozatának helyét frissíti valamilyen átalakítással:

```
Shape* move_shape(Shape* s, Point p) // *s += p
{
  s->move_to(s->center()+p);
  return s;
}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
  // művelet végrehajtása a megfelelő objektumon
  transform(ls.begin(),ls.end(),oper.begin(),ls.begin(),move_shape);
}
```

Valójában nem lenne szükségünk arra, hogy a *move_shape()* függvénynek visszatérési értéke legyen, de a *transform()* ragaszkodik a művelet eredményének felhasználásához, így a *move_shape()* visszaadja az első operandusát, amelyet visszaírhatunk az eredeti helyére.

Bizonyos helyzetekben ezt a problémát nem oldhatjuk meg így. Ha a műveletet például nem mi írtuk vagy nem akarjuk megváltoztatni, akkor nem áll rendelkezésünkre a megfelelő visszatérési érték. Máskor a bemeneti sorozat *const*. Ezekben az esetekben egy kétsorozatos *for_each()* függvényt készíthetünk, amely a kétsorozatos *transform()* párjának tekinthető:

```
template<class In, class In2, class BinOp>
BinOp for_each(In first, In last, In2 first2, BinOp op)
{
  while (first != last) op(*first++,*first2++);
  return op;
}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
  for_each(ls.begin(),ls.end(),oper.begin(),move_shape);
}
```

Esetenként olyan kimeneti bejáró is hasznos lehet, amely valójában semmit sem ír (§19.6[2]).

A standard könyvtár nem tartalmaz olyan algoritmusokat, melyek három vagy négy sorozatból olvasnak, bár ezek is könnyen elkészíthetők. Helyettük használhatjuk többször egymás után a *transform()* függvényt.

18.6.3. Ismétlődő elemek törlése

Amikor információkat gyűjtünk, könnyen előfordulhatnak ismétlődések. A *unique()* és a *unique_copy()* algoritmusokkal az egymás után előforduló azonos értékeket távolíthatjuk el:

```
template<class For> For unique(For first, For last);
template<class For, class BinPred> For unique(For first, For last, BinPred p);

template<class In, class Out> Out unique_copy(In first, In last, Out res);
template<class In, class Out, class BinPred>
  Out unique_copy(In first, In last, Out res, BinPred p);
```

A *unique()* megszünteti a sorozatban egymás után előforduló értékismétlődéseket, a *unique_copy()* pedig egy másolatot készít az ismétlődések elhagyásával:

```
void f(list<string>& ls, vector<string>& vs)
{
  ls.sort();              // listarendezés (§17.2.2.1)
  unique_copy(ls.begin(),ls.end(),back_inserter(vs));
}
```

Ezzel a programrészlettel az *ls* listát átmásolhatjuk a *vs* vektorba és menet közben kiszűrhetjük az ismétlődéseket. A *sort()* utasításra azért van szükség, hogy az egyenértékű elemek egymás mellé kerüljenek.

A többi szabványos algoritmushoz hasonlóan a *unique()* is bejárókkal dolgozik. Azt nem lehet megállapítani, hogy ezek a bejárók milyen típusú tárolóra mutatnak, így a tárolót nem változtathatjuk meg, csak az annak elemeiben tárolt értékeket módosíthatjuk. Ebből következik, hogy a *unique()* nem törli az ismétlődéseket a sorozatból, ahogy azt naívan remélnénk. Ehelyett az egyedi elemeket a sorozat elejére helyezi és visszaad egy bejárót, amely az egyedi elemek részsorozatának végére mutat:

```
template <class For> For unique(For first, For last)
{
  first = adjacent_find(first,last);              // §18.5.2
  return unique_copy(first,last,first);
}
```

A részsorozat utáni elemek változatlanok maradnak, tehát egy vektor esetében ez a megoldás nem szünteti meg az ismétlődéseket:

```
void f(vector<string>& vs)              // vigyázat: rossz kód!
{
  sort(vs.begin(),vs.end());            // vektorrendezés
  unique(vs.begin(),vs.end());          // ismétlődések eltávolítása (nem működik!)
}
```

Sőt azzal, hogy a *unique()* a sorozat végén álló elemeket előre helyezi az értékismétlődések megszüntetéséhez, akár új ismétlődések is keletkezhetnek:

```
int main()
{
  char v[] = "abbcccde";

  char* p = unique(v,v+strlen(v));
  cout << v << ' ' << p-v << '\n';
}
```

Az eredmény a következő lesz:

```
abcdecde 5
```

Tehát a *p* a második *c* betűre mutat.

Azoknak az algoritmusoknak, melyeknek törölnie kellene elemeket (de ezt nem tudják megtenni), általában két változata van. Az egyik a *uniqe()* módszerével átrendezi az elemeket, a másik egy új sorozatot hoz létre a *unique_copy()* működéséhez hasonlóan. A *_copy* utótag ezen két változat megkülönböztetésére szolgál.

Ahhoz hogy az ismétlődéseket ténylegesen kiszűrjük egy tárolóból, még egy további utasításra van szükség:

```
template<class C> void eliminate_duplicates(C& c)
{
  sort(c.begin(),c.end());                              // rendezés
  typename C::iterator p = unique(c.begin(),c.end());   // tömörítés
  c.erase(p,c.end());                                   // zsugorítás
}
```

Sajnos az *eliminate_duplicates()* függvény a beépített tömbökhöz nem használható, míg a *unique()* azoknál is működik.

A *unique_copy()* használatára a §3.8.3. pontban mutattunk példát.

18.6.3.1. Rendezési szempont

Az összes ismétlődés megszüntetéséhez a bemeneti sorozatot rendeznünk kell (§18.7.1). Alapértelmezés szerint a *unique()* és a *unique_copy()* is az == operátort használja az egyenlőségvizsgálathoz, de lehetővé teszik, hogy a programozó más műveletet adjon meg. A §18.5.1 pontban szereplő programrészletet például átalakíthatjuk úgy, hogy kiszűrje az ismétlődő neveket. Miután a klubok hivatalnokainak nevét összegyűjtöttük, az *off* listához jutottunk, melynek típusa *list<Person*>* (§18.5.1). Ebből a listából az ismétlődéseket a következő utasítással törölhetjük:

```
eliminate_duplicates(off);
```

Ez a megoldás azonban mutatókat rendez, és csak akkor fog céljainknak megfelelően működni, ha feltételezzük, hogy minden *Person* objektumra egyetlen mutató hivatkozik.

Az egyenértékűség eldöntéséhez általában magukat a *Person* rekordokat kell megvizsgálnunk. Ehhez a következő programrészletet kell megírnunk:

```
bool operator==(const Person& x, const Person& y)        // "egyenlőség" objektumra
{
  // x és y összehasonlítása egyenlőségre
}

bool operator<(const Person& x, const Person& y)         // "kisebb mint" objektumra
{
  // x és y összehasonlítása rendezettségre
}

bool Person_eq(const Person* x, const Person* y)         // "egyenlőség" mutatón keresztül
{
  return *x == *y;
}

bool Person_lt(const Person* x, const Person* y)         // "kisebb mint" mutatón keresztül
{
  return *x < *y;
}

void extract_and_print(const list<Club>& lc)
{
  list<Person*> off;
  extract(lc,off);
  off.sort(off,Person_lt);
  list<Club>::iterator p = unique(off.begin(),off.end(),Person_eq);
  for_each(off.begin(),p,Print_name(cout));
}
```

Érdemes mindig gondoskodnunk arról, hogy a rendezéshez használt feltétel ugyanaz legyen, mint az ismétlődések kiszűrésére szolgáló eljárás. A < és az == operátorok alapértelmezett jelentése mutatók esetében általában nem felel meg számunkra a mutatott objektumok összehasonlításához.

18.6.4. Helyettesítés

A *replace()* algoritmusok végighaladnak a megadott sorozaton, és egyes értékeket újakra cserélnek, a mi igényeinknek megfelelően. Ugyanazt a mintát követik, mint a *find/find_if* és a *unique/unique_copy*, így összesen négy változatra van szükség. A függvények megvalósítása most is elég egyszerű ahhoz, hogy jól magyarázza szerepüket:

```
template<class For, class T>
void replace(For first, For last, const T& val, const T& new_val)
{
  while (first != last) {
        if (*first == val) *first = new_val;
        ++first;
  }
}

template<class For, class Pred, class T>
void replace_if(For first, For last, Pred p, const T& new_val)
{
  while (first != last) {
        if (p(*first)) *first = new_val;
        ++first;
  }
}

template<class In, class Out, class T>
Out replace_copy(In first, In last, Out res, const T& val, const T& new_val)
{
  while (first != last) {
        *res++ = (*first == val) ? new_val : *first;
        ++first;
  }
  return res;
}

template<class In, class Out, class Pred, class T>
Out replace_copy_if(In first, In last, Out res, Pred p, const T& new_val)
{
  while (first != last) {
        *res++ = p(*first) ? new_val : *first;
        ++first;
  }
  return res;
}
```

Sűrűn előfordul, hogy karakterláncok egy listáján végighaladva szülővárosom szokásos angol átírását (Aarhus) a helyes Århus változatra kell cserélnem:

```
void f(list<string>& towns)
{
  replace(towns.begin(),towns.end(),"Aarhus","Århus");
}
```

Természetesen ehhez egy kibővített karakterkészletre van szükség (§C.3.3)

18.6.5. Törlés

A *remove()* algoritmusok elemeket törölnek egy sorozatból, érték vagy feltétel alapján:

```
template<class For, class T> For remove(For first, For last, const T& val);

template<class For, class Pred> For remove_if(For first, For last, Pred p);

template<class In, class Out, class T>
  Out remove_copy(In first, In last, Out res, const T& val);

template<class In, class Out, class Pred>
  Out remove_copy_if(In first, In last, Out res, Pred p);
```

Tegyük fel például, hogy a *Club* osztályban szerepel egy *cím* mező is és feladatunk az, hogy összegyűjtsük a koppenhágai klubokat:

```
class located_in : public unary_function<Club,bool> {
  string town;
public:
  located_in(const string& ss) :town(ss) { }
  bool operator()(const Club& c) const { return c.town == town; }
};

void f(list<Club>& lc)
{
  remove_copy_if(lc.begin(),lc.end(),
                              ostream_iterator<Club>(cout),not1(located_in("København")));
}
```

A *remove_copy_if()* ugyanaz, mint a *copy_if()*, csak fordított feltétellel. Tehát a *remove_copy_if()* akkor helyez egy elemet a kimenetre, ha az nem elégíti ki a feltételt.

A „sima" *remove()*a sorozat elejére gyűjti a nem törlendő elemeket és az így képzett részsorozat végére mutató bejárót ad vissza (lásd még §18.6.3).

18.6.6. Feltöltés és létrehozás

A *fill()* és a *generate()* algoritmusok segítségével rendszerezetten tölthetünk fel egy sorozatot értékekkel:

```
template<class For, class T> void fill(For first, For last, const T& val);
template<class Out, class Size, class T> void fill_n(Out res, Size n, const T& val);
```

```
template<class For, class Gen> void generate(For first, For last, Gen g);
template<class Out, class Size, class Gen> void generate_n(Out res, Size n, Gen g);
```

A *fill()* függvény megadott értékeket ad a sorozat elemeinek, míg a *generate()* algoritmus úgy állítja elő az értékeket, hogy mindig meghívja a megadott függvényt. Tehát a *fill()* a *generate()* egyedi változata, amikor az értékeket előállító (létrehozó, generátor) függvény mindig ugyanazt az értéket adja vissza. Az *_n* változatok a sorozat első *n* elemének adnak értéket.

A §22.7 pont *Randint* és *Urand* véletlenszám-előállítójának felhasználásával például a következőt írhatjuk:

```
int v1[900];
int v2[900];
vector v3;

void f()
{
  fill(v1,&v1[900],99);                    // v1 minden elemének 99-re állítása
  generate(v2,&v2[900],Randint());         // véletlen értékekre állítás (§22.7)

  // 200 véletlen egész küldése a kimenetre a [0..99] tartományból
  generate_n(ostream_iterator<int>(cout),200,Urand(100));

  fill_n(back_inserter(v3),20,99);         // 20 darab 99 értékű elem hozzáadása v3-hoz
}
```

A *generate()* és a *fill()* nem kezdeti, hanem egyszerű értékadást végez. Ha nyers tárolóterületet akarunk felhasználni (például egy memóriaszeletet meghatározott típusú és állapotú objektummá szeretnénk alakítani), használhatjuk például az *uninitialized_fill()* függvényt, a *<memory>* fejállományból (§19.4.4). Az *<algorithm>* állomány algoritmusai ilyen célokra nem felelnek meg.

18.6.7. Megfordítás és elforgatás

Időnként szükségünk van rá, hogy egy sorozat elemeit átrendezzük:

```
template<class Bi> void reverse(Bi first, Bi last);
template<class Bi, class Out> Out reverse_copy(Bi first, Bi last, Out res);

template<class For> void rotate(For first, For middle, For last);
template<class For, class Out> Out rotate_copy(For first, For middle, For last, Out res);
```

```
template<class Ran> void random_shuffle(Ran first, Ran last);
template<class Ran, class Gen> void random_shuffle(Ran first, Ran last, Gen& g);
```

A *reverse()* algoritmus megfordítja az elemek sorrendjét, tehát az első elem lesz az utolsó stb. A *reverse_copy()* szokás szerint egy másolatban állítja erő bemeneti sorozatának megfordítását.

A *rotate()* algoritmus a *[first, last[* sorozatot körként kezeli, és addig forgatja az elemeket, míg a korábbi *middle* elem a sorozat elejére nem kerül. Tehát az az elem, amely eddig a *first+i* pozíción volt, a művelet után a *first+ (i+ (last–middle)) %(last-first)* helyre kerül. A *%* (modulus) operátor teszi a forgatást ciklikussá az egyszerű balraléptetés helyett:

```
void f()
{
  string v[ ] = { "Béka", "és","Barack" };

  reverse(v,v+3);              // Barack és Béka
  rotate(v,v+1,v+3);           // és Béka Barack
}
```

A *rotate_copy()* egy másolatban állítja elő bemeneti sorozatának elforgatott változatát.

Alapértelmezés szerint a *random_shuffle()* megkeveri a paraméterében megadott sorozatot egy egyenletes eloszlású véletlenszámokat előállító eljárás segítségével. Tehát az elemek sorozatának egy permutációját (lehetséges elemsorrendjét) választja ki úgy, hogy mindegyik permutációnak ugyanakkora esélye van. Ha más eloszlást szeretnénk elérni vagy egyszerűen csak jobb véletlenszám-előállítónk van, akkor azt megadhatjuk a *random_shuffle()* függvénynek. A §22.7. pont *Urand* eljárásával például a következőképpen keverhetjük meg egy kártyapakli lapjait:

```
void f(deque<Card>& dc, My_rand& r)
{
  random_shuffle(dc.begin(),dc.end(),r);
  // ...
}
```

Az elemek áthelyezését a *rotate()* és más függvények *a swap()* függvény (§18.6.8) segítségével hajtják végre.

18.6.8. Elemek felcserélése

Ha bármi érdekeset szeretnénk végrehajtani egy tároló elemeivel, akkor mindig át kell helyeznünk azokat. Ezt az áthelyezést legjobban – tehát a legegyszerűbben és a leghatékonyabban – a *swap()* függvénnyel fejezhetjük ki:

```
template<class T> void swap(T& a, T& b)
{
  T tmp = a;
  a = b;
  b = tmp;
}

template<class For, class For2> void iter_swap(For x, For2 y);

template<class For, class For2> For2 swap_ranges(For first, For last, For2 first2)
{
  while (first != last) iter_swap(first++, first2++);
  return first2;
}
```

Ahhoz, hogy felcseréljünk elemeket, egy ideiglenes tárolóra van szükségünk. Egyes esetekben lehetnek ügyes trükkök, melyekkel ez elkerülhető, de az egyszerűség és érthetőség érdekében érdemes elkerülni az ilyesmit. A *swap()* algoritmus rendelkezik egyedi célú változatokkal azokhoz a típusokhoz, ahol erre szükség lehet (§16.3.9, §13.5.2).

Az *iter_swap()* algoritmus bejárókkal kijelölt elemeket cserél fel, a *swap_ranges()* pedig két bemeneti paramétere által meghatározott tartománya elemeit cseréli fel.

18.7. Rendezett sorozatok

Miután összegyűjtöttük az adatokat, általában szeretnénk rendezni azokat. Ha rendezett sorozat áll rendelkezésünkre, sokkal több lehetőségünk lesz arra, hogy adatainkat kényelmesen kezeljük.

Egy sorozat rendezéséhez valahogyan össze kell hasonlítanunk az elemeket. Ehhez egy kétparaméterű predikátumot (§18.4.2) használhatunk. Az alapértelmezett összehasonlító művelet a *less* (§18.4.2), amely viszont alapértelmezés szerint a < operátort használja.

18.7.1. Rendezés

A *sort()* rendező algoritmusoknak közvetlen elérésű (véletlen elérésű) bejárókra (§19.2.1) van szükségük. Ebből következik, hogy a vektorok (§16.3) és az ahhoz hasonló szerkezetek esetében működnek a leghatékonyabban:

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);

template<class Ran> void stable_sort(Ran first, Ran last);
template<class Ran, class Cmp> void stable_sort(Ran first, Ran last, Cmp cmp);
```

A szabványos *list* (§17.2.2) osztály nem biztosít közvetlen hozzáférésű bejárókat, így azokat megfelelő listaműveletekkel (§17.2.2.1) kell rendeznünk.

Az egyszerű *sort()* eljárás elég hatékony – átlagosan *N*log(N)* – de a legrosszabb esetre vett hatékonyság elég rossz: *O(N*N)*. Szerencsére a „legrosszabb” eset elég ritka. Ha a legrosszabb esetben is garantált működésre vagy stabil rendezésre van szükségünk, használjuk a *stable_sort()* függvényt. Ennek hatékonysága *N*log(N)*log(N)*, ami *N*log(N)* értékre javul, ha a rendszerben elég memória áll rendelkezésünkre. A *stable_sort()* – a *sort()* függvénnyel ellentétben – megtartja az egyenértékűnek minősített elemek sorrendjét.

Bizonyos helyzetekben a rendezett sorozatnak csak első néhány elemére van szükségünk. Ebben az esetben van értelme annak, hogy a sorozatot csak olyan hosszon rendezzük, amilyenre éppen szükségünk van. Ezt nevezzük *részleges* (parciális) *rendezésnek*:

```
template<class Ran> void partial_sort(Ran first, Ran middle, Ran last);
template<class Ran, class Cmp>
  void partial_sort(Ran first, Ran middle, Ran last, Cmp cmp);

template<class In, class Ran>
  Ran partial_sort_copy(In first, In last, Ran first2, Ran last2);
template<class In, class Ran, class Cmp>
  Ran partial_sort_copy(In first, In last, Ran first2, Ran last2, Cmp cmp);
```

A *partial_sort()* algoritmus alapváltozata a *first* és a *middle* közötti elemeket rendezi el. A *partial_sort_copy()* algoritmusok egy *N* elemű sorozatot hoznak létre, ahol *N* a bemeneti és kimeneti sorozat elemei számának minimuma. Ebből következik, hogy meg kell adnunk az eredménysorozat elejét és végét is, hiszen ez határozza meg, hány elemet kell elrendezni:

```
class Compare_copies_sold {
public:
   int operator()(const Book& b1, const Book& b2) const
            { return b1.copies_sold()>b2.copies_sold(); }    // rendezés csökkenő sorrendben
};

void f(const vector<Book>& sales)       // a tíz legjobban fogyó könyv megkerése
{
   vector<Book> bestsellers(10);
   partial_sort_copy(sales.begin(),sales.end(),
                       bestsellers.begin(),bestsellers.end(),Compare_copies_sold());
   copy(bestsellers.begin(),bestsellers.end(),ostream_iterator<Book>(cout,"\n"));
}
```

Mivel a *partial_sort_copy()* kimenetének egy véletlen elérésű bejárónak kell lennie, nem rendezhetünk egy sorozatot egyenesen a *cout* adatfolyamra.

Végül nézzük azokat az algoritmusokat, melyekkel pontosan csak annyi elemet rendezhetünk el, amennyi az *N*-edik elem helyének megállapításához szükséges. Ezek azonnal befejezik működésüket, ha a kívánt elemet megtalálták:

```
template<class Ran> void nth_element(Ran first, Ran nth, Ran last);
template<class Ran, class Cmp> void nth_element(Ran first, Ran nth, Ran last, Cmp cmp);
```

Ez a függvény különösen hasznos azoknak, akiknek középértékre, százalékarányra stb. van szükségük (mérnököknek, szociológusoknak, tanároknak).

18.7.2. Bináris keresés

A sorban történő (soros, szekvenciális) keresés, amit a *find()* is végez, szörnyen rossz hatásfokú nagy sorozatok esetében, mégis ez a legjobb megoldás, ha sem rendezés, sem hasítás (§17.6) nem áll rendelkezésünkre. Ha azonban rendezett sorozatban keresünk, akkor annak megállapítására, hogy egy érték szerepel-e a sorozatban, használhatjuk a bináris keresést:

```
template<class For, class T> bool binary_search(For first, For last, const T& val);

template<class For, class T, class Cmp>
   bool binary_search(For first, For last, const T& value, Cmp cmp);
```

Például:

```
void f(list<int>& c)
{
  if (binary_search(c.begin(),c.end(),7)) {        // Van 7 a c-ben?
          // ...
  }
  // ...
}
```

A *binary_search()* egy logikai értéket ad vissza, jelezvén, hogy az érték szerepel-e a sorozatban. Ugyanúgy, mint a *find()* esetében, itt is gyakran tudni szeretnénk, hogy az adott értékkel rendelkező elem hol található. Egy sorozatban azonban több megfelelő érték is lehet, és nekünk általában vagy az elsőre, vagy az összesre van szükségünk. Ezért szerepelnek a standard könyvtárban azok az eljárások, melyekkel az első (*lower_bound()*), az utolsó (*upper_bound()*) vagy az összes (*equal_range()*) megfelelő (egyenértékű) elemet kiválaszthatjuk:

```
template<class For, class T> For lower_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
  For lower_bound(For first, For last, const T& val, Cmp cmp);

template<class For, class T> For upper_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
  For upper_bound(For first, For last, const T& val, Cmp cmp);

template<class For, class T> pair<For, For> equal_range(For first, For last, const T& val);
template<class For, class T, class Cmp>
  pair<For, For> equal_range(For first, For last, const T& val, Cmp cmp);
```

Ezek az algoritmusok a *multimap* (§17.4.2) műveletei közé tartoznak. A *lower_bound()* függvényt úgy képzelhetjük el, mint a *find()*, illetve a *find_if()* gyors változatát rendezett sorozatokra:

```
void g(vector<int>& c)
{
  typedef vector<int>::iterator VI;

  VI p = find(c.begin(),c.end(),7);              // valószínűleg lassú: O(N); c-t nem kell
                                                 // rendezni
  VI q = lower_bound(c.begin(),c.end(),7);       // valószínűleg gyors: O(log(N)); c-t
                                                 // rendezni kell
  // ...
}
```

Ha a *lower_bound(first, last, k)* nem találja meg a *k* értéket, akkor egy olyan bejárót ad vissza, amely az első, *k*-nál nagyobb kulccsal rendelkező elemre mutat, vagy a *last* bejárót, ha nincs *k*-nál nagyobb érték. Az *upper_bound()* és az *equal_range()* szintén ezt a hibajelzési módszert használja. Ezekkel az algoritmusokkal meghatározhatjuk, hogy hová kell az új elemet beszúrnunk, ha a sorozat rendezettségét nem akarjuk elrontani.

18.7.3. Összefésülés

Ha van két rendezett sorozatunk, akkor a *merge()* segítségével egy új rendezett sorozatot hozhatunk létre belőlük, az *inplace_merge()* függvény felhasználásával pedig egy sorozat két részét fésülhetjük össze:

```
template<class In, class In2, class Out>
  Out merge(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
  Out merge(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class Bi> void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Cmp> void inplace_merge(Bi first, Bi middle, Bi last, Cmp cmp);
```

Ezek az összefésülő algoritmusok abban térnek el jelentősen a *list* osztály hasonló eljárásaitól (§17.2.2.1), hogy nem törlik az elemeket a bemeneti sorozatokból. Minden elemről külön másolat készül.

Az egyenértékű elemek sorrendjéről azt mondhatjuk el, hogy az első sorozatban lévő elemek mindig megelőzik a második sorozat azonos elemeit.

Az *inplace_merge()* algoritmus elsősorban akkor hasznos, ha egy sorozatot több szempont szerint is rendezni akarunk. Képzeljük el például „halaknak egy vektorát", amely fajok (hering, tőkehal stb.) szerint rendezett. Ha a halak minden fajon belül súly szerint rendezettek, akkor az *inplace_merge()* segítségével könnyen rendezhetjük az egész vektort a súlyok alapján, hiszen csak az egyes fajták részsorozatait kell összefésülnünk (§18.13[20]).

18.7.4. Felosztás

Egy sorozat felosztása (partition) azt jelenti, hogy minden olyan elemet, amely kielégít egy adott feltételt, a sorozat elejére helyezünk, a feltételt ki nem elégítőket pedig a végére. A standard könyvtárban rendelkezésünkre áll a *stable_partition()* függvény, amely megtartja azon elemek egymáshoz viszonyított sorrendjét, melyek egyformán megfelelnek vagy

egyformán nem felelnek meg a predikátumnak. A könyvtárban szerepel a *partition()* eljárás is, amely ezt a sorrendet nem őrzi meg, de egy kicsit gyorsabban fut le, ha kevesebb memória áll rendelkezésünkre.

```
template<class Bi, class Pred> Bi partition(Bi first, Bi last, Pred p);
template<class Bi, class Pred> Bi stable_partition(Bi first, Bi last, Pred p);
```

A felosztást úgy képzelhetjük el, mint egy nagyon egyszerű feltétel szerinti rendezést:

```
void f(list<Club>& lc)
{
  list<Club>::iterator p = partition(lc.begin(),lc.end(),located_in("København"));
  // ...
}
```

Ez az eljárás úgy „rendezi” a listát, hogy a koppenhágai klubok szerepeljenek először. A visszatérési érték (esetünkben a *p*) az első olyan elemet határozza meg, amely nem elégíti ki a feltételt, illetve ha ilyen nincs, akkor a sorozat végére mutat.

18.7.5. Halmazműveletek sorozatokon

A sorozatokat tekinthetjük halmaznak is. Ebből a szempontból viszont illik a sorozatokhoz is elkészítenünk az olyan alapvető halmazműveleteket, mint az unió vagy a metszet. Másrészt viszont ezek a műveletek rendkívül költségesek, hacsak nem rendezett sorozatokkal dolgozunk. Ezért a standard könyvtár halmazkezelő algoritmusai csak rendezett sorozatokkal használhatók. Természetesen nagyon jól működnek a *set* (§17.4.3) és a *multiset* (§17.4.4) tároló esetében is, melyek szintén rendezettek.

Ha ezeket az algoritmusokat nem rendezett sorozatokra alkalmazzuk, az eredménysorozatok nem fognak megfelelni a szokásos halmazelméleti szabályoknak. A függvények nem változtatják meg bemeneti sorozataikat és a kimeneti sorozat rendezett lesz.

Az *includes()* algoritmus azt vizsgálja, hogy a második sorozat minden eleme (a *[first2,last2[* tartományból) megtalálható-e az első sorozat elemei között (a *[first, last[* tartományban):

```
template<class In, class In2>
  bool includes(In first, In last, In2 first2, In2 last2);
template<class In, class In2, class Cmp>
  bool includes(In first, In last, In2 first2, In2 last2, Cmp cmp);
```

A *set_union()* rendezett sorozatok unióját, a *set_intersection()* függvény pedig metszetüket állítja elő:

```
template<class In, class In2, class Out>
  Out set_union(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
  Out set_union(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
  Out set_intersection(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
  Out set_intersection(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

A *set_difference()* algoritmus olyan elemek sorozatát hozza létre, melyek az első bemeneti sorozatban megtalálhatók, de a másodikban nem. A *set_symmetric_difference()* algoritmus olyan sorozatot állít elő, melynek elemei a két bemeneti sorozat közül csak az egyikben szerepelnek:

```
template<class In, class In2, class Out>
  Out set_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
  Out set_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
  Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
  Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

Például:

```
char v1[ ] = "abcd";
char v2[ ] = "cdef";

void f(char v3[ ])
{
  set_difference(v1,v1+4,v2,v2+4,v3);                 // v3 = "ab"
  set_symmetric_difference(v1,v1+4,v2,v2+4,v3);       // v3 = "abef"
}
```

18.8. A kupac

A *kupac* (halom, heap) szót különböző helyzetekben különböző dolgokra használjuk a számítástechnikában. Amikor algoritmusokról beszélünk, a „kupac" egy sorozat elemeinek olyan elrendezését jelenti, melyben az első elem a sorozat legnagyobb értékű eleme. Ebben az adatszerkezetben viszonylag gyorsan lehet elvégezni az elemek beszúrását (a *push_heap()* függvény segítségével), illetve törlését (a *pop_heap()* eljárással). Mindkettő a legrosszabb esetben is *O(log(N))* hatékonysággal működik, ahol *N* a sorozat elemeinek száma. A rendezés (a *sort_heap()* felhasználásával) szintén jó hatékonyságú: *O(N*log(N))*. A kupacot ezek a függvények valósítják meg:

```
template<class Ran> void push_heap(Ran first, Ran last);
template<class Ran, class Cmp> void push_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void pop_heap(Ran first, Ran last);
template<class Ran, class Cmp> void pop_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void make_heap(Ran first, Ran last);          // sorozat kupaccá
                                                                  // alakítása
template<class Ran, class Cmp> void make_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void sort_heap(Ran first, Ran last);          // kupac sorozattá
                                                                  // alakítása
template<class Ran, class Cmp> void sort_heap(Ran first, Ran last, Cmp cmp);
```

A kupac-algoritmusok stílusa egy kissé meglepő. Természetes megoldás lenne, hogy ezt a négy függvényt egy osztályba foglaljuk össze. Ha ezt tennénk, a *priority_queue* (§17.3.3) tárolóhoz nagyon hasonló szerkezetet kapnánk. Valójában a *priority_queue*-t szinte majdnem biztosan egy kupac képviseli rendszerünkben.

A *push_heap(first, last)* által beillesztett elem a **(last-1)*. Azt feltételezzük, hogy a *[first, last-1[* tartomány már kupac, így a *push_heap()* csak kiegészíti a sorozatot a *[first, last[* tartományra a következő elem beszúrásával. Tehát egy kupacot létrehozhatunk egy sorozatból úgy, hogy minden elemre meghívjuk a *push_heap()* műveletet. A *pop_heap(first, last)* úgy távolítja el a kupac első elemét, hogy felcseréli azt a *(*(last-1))* elemmel, majd a *[first, last-1[* tartományt újra kupaccá alakítja.

18.9. Minimum és maximum

Az itt leírt algoritmusok összehasonlítás alapján választanak ki értékeket egy sorozatból. Nagyon sokszor van szükségünk arra, hogy két érték közül kiválasszuk a kisebbet vagy nagyobbat:

```
template<class T> const T& max(const T& a, const T& b)
{
  return (a<b) ? b : a;
}

template<class T, class Cmp> const T& max(const T& a, const T& b, Cmp cmp)
{
  return (cmp(a,b)) ? b : a;
}

template<class T> const T& min(const T& a, const T& b);

template<class T, class Cmp> const T& min(const T& a, const T& b, Cmp cmp);
```

A *min()* és a *max()* függvény általánosítható úgy, hogy teljes sorozatokból válasszák ki a megfelelő értéket:

```
template<class For> For max_element(For first, For last);
template<class For, class Cmp> For max_element(For first, For last, Cmp cmp);

template<class For> For min_element(For first, For last);
template<class For, class Cmp> For min_element(For first, For last, Cmp cmp);
```

Végül az ábécésorrendű (lexikografikus) rendezés általánosítását is bemutatjuk, amely karakterláncok helyett tetszőleges értéksorozatra ad meg összehasonlítási feltételt:

```
template<class In, class In2>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2);

template<class In, class In2, class Cmp>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2, Cmp cmp)
{
  while (first != last && first2 != last2) {
            if (cmp(*first,*first2)) return true;
            if (cmp(*first2++,*first++)) return false;
  }
  return first == last && first2 != last2;
}
```

Ez nagyon hasonlít az általános karakterláncoknál (§13.4.1) bemutatott függvényre, de a *lexicographical_compare()* tetszőleges két sorozatot képes összehasonlítani, nem csak karakterláncokat. Egy másik különbség, hogy ez az algoritmus csak egy *bool* és nem egy *int* értéket ad vissza, bár az több információt tartalmazhatna. Az eredmény kizárólag akkor lesz *true*, ha az első sorozat kisebb (<), mint a második. Tehát akkor is *false* visszatérési értéket kapunk, ha a két sorozat egyenlő.

A C stílusú karakterláncok és a *string* osztály is sorozat, így a *lexicographical_compare()* használható ezekre is összehasonlító feltételként:

```
char v1[ ] = "igen";
char v2[ ] = "nem";
string s1 = "Igen";
string s2 = "Nem";

void f()
{
   bool b1 = lexicographical_compare(v1,v1+strlen(v1),v2,v2+strlen(v2));
   bool b2 = lexicographical_compare(s1.begin(),s1.end(),s2.begin(),s2.end());

   bool b3 = lexicographical_compare(v1,v1+strlen(v1),s1.begin(),s1.end());
   bool b4 = lexicographical_compare(s1.begin(),s1.end(),v1,v1+strlen(v1),Nocase());
}
```

A sorozatoknak nem kell azonos típusúaknak lenniük, hiszen csak elemeket kell tudnunk összehasonlítani, és az összehasonlító predikátumot mi adhatjuk meg. Ez a lehetőség a *lexicographical_compare()* eljárást sokkal általánosabbá – és egy kicsit lassabbá – teszi a *string* osztály összehasonlító műveleteinél. Lásd még: §20.3.8.

18.10. Permutációk

Egy négy elemből álló sorozatot összesen 4*3*2 féleképpen rendezhetünk el. Mindegyik elrendezés egy-egy *permutációja* a négy elemnek. Négy karakterből (*abcd*) például az alábbi 24 permutációt állíthatjuk elő:

```
abcd   abdc   acbd   acdb   adbc   adcb   bacd   badc
bcad   bcda   bdac   bdca   cabd   cadb   cbad   cbda
cdab   cdba   dabc   dacb   dbac   dbca   dcab   dcba
```

A *next_permutation()* és a *prev_permutation()* függvény ilyen permutációkat állít elő egy sorozatból:

```
template<class Bi> bool next_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool next_permutation(Bi first, Bi last, Cmp cmp);

template<class Bi> bool prev_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool prev_permutation(Bi first, Bi last, Cmp cmp);
```

Az *abcd* sorozat permutációit a következőképpen írathatjuk ki:

```
int main()
{
  char v[] = "abcd";
  cout << v << '\t';
  while(next_permutation(v,v+4)) cout << v << '\t';
}
```

A permutációkat ábécésorrendben kapjuk meg (§18.9). A *next_permutation()* függvény visszatérési értéke azt határozza meg, hogy van-e még további permutáció. Ha nincs, *false* értéket kapunk és azt a permutációt, melyben az elemek ábécésorrendben állnak. A *prev_permutation()* függvény visszatérési értéke azt adja meg, hogy van-e előző permutáció, és ha nincs, akkor az elemeket fordított ábécésorrendben tartalmazó permutációt kapjuk.

18.11. C stílusú algoritmusok

A C++ standard könyvtára örökölt néhány algoritmust a C standard könyvtárától is. Ezek a C stílusú karakterláncokat kezelő függvények (§20.4.1), illetve a gyorsrendezés (quicksort) és a bináris keresés, melyek csak tömbökre használhatók.

A *qsort()* és a *bsearch()* függvény a *<cstdlib>* és az *<stdlib.h>* fejállományban található meg. Mindkettő tömbökön működik és *n* darab *elem_size* méretű elemet dolgoznak fel egy függvényre hivatkozó mutatóval megadott „kisebb, mint" összehasonlító művelet segítségével. Az elemek típusának nem lehet felhasználó által megadott másoló konstruktora, másoló értékadása, illetve destruktora:

```
typedef int(*__cmp)(const void*, const void*);  // a typedef-et csak a bemutatáshoz
                                                // használjuk

void qsort(void* p, size_t n, size_t elem_size, __cmp);                    // p rendezése
void* bsearch(const void* key, void* p, size_t n, size_t elem_size, __cmp); // kulcs keresése
                                                                           // p-ben
```

A *qsort()* függvény használatáról a §7.7 pontban már részletesebben is szó volt.

Ezek az algoritmusok kizárólag a C-vel való kompatibilitás miatt maradtak meg a nyelvben. A *sort()* (§18.7.1) és a *search()* (§18.5.5) sokkal általánosabbak és általában sokkal hatékonyabbak is.

18.12. Tanácsok

[1] Ciklusok helyett használjunk inkább algoritmusokat (§18.5.1).
[2] Ha ciklust írunk, mindig gondoljuk végig, hogy feladatunk nem fogalmazható-e meg egy általános algoritmussal (§18.2).
[3] Rendszeresen nézzük át az algoritmusokat, hogy felismerjük, ha valamilyen feladat triviális (§18.2).
[4] Mindig ellenőrizzük, hogy az általunk megadott bejáró-pár tényleg meghatároz-e egy sorozatot (§18.3.1).
[5] Tervezzük úgy programjainkat, hogy a leggyakrabban használt eljárások legyenek a legegyszerűbbek és leggyorsabbak (§18.3, §18.3.1).
[6] Az értékvizsgálatokat úgy fogalmazzuk meg, hogy predikátumként is használhassuk azokat (§18.4.2).
[7] Mindig gondoljunk arra, hogy a predikátumok függvények vagy objektumok, de semmiképpen sem típusok (§18.4.2).
[8] A lekötők (binder) segítségével a kétparaméterű predikátumokból egyparaméterű predikátumokat állíthatunk elő (§18.4.4.1).
[9] A *mem_fun()* és a *mem_fun_ref()* függvény segít abban, hogy az algoritmusokat tárolókra alkalmazzuk (§18.4.4.2).
[10] Ha egy függvény valamelyik paraméterét rögzítenünk kell, használjuk a *ptr_fun()* függvényt.
[11] A *strcmp()* legnagyobb eltérése az == operátortól, hogy *0* visszatérési érték jelzi az „egyenlőséget” (§18.4.4.4).

[12] A *for_each()* és a *transform()* algoritmust csak akkor használjuk, ha nem találunk feladatunkhoz jobban illeszkedő eljárást (§18.5.1).
[13] Használjunk predikátumokat, ha egy algoritmusban egyedi összehasonlítási feltételre van szükségünk (§184.2.1, §18.6.3.1).
[14] A predikátumok és más függvényobjektumok segítségével a szabványos algoritmusok sokkal több területen használhatók, mint első ránézésre gondolnánk (§18.4.2).
[15] A mutatók esetében az alapértelmezett == és < operátor igen ritkán felel meg igényeinknek a szabványos algoritmusokban (§18.6.3.1).
[16] Az algoritmusok nem tudnak közvetlenül beilleszteni vagy kivenni elemet a paraméterként megadott sorozatokból.
[17] Mindig ellenőrizzük, hogy a sorozatok összehasonlításakor a megfelelő „kisebb, mint", illetve „egyenlőség" műveletet használjuk-e (§18.6.3.1).
[18] A rendezett sorozatok gyakran hatékonyabbá és elegánsabbá teszik programjainkat.
[19] A *qsort()* és a *bsearch()* függvényeket csak a C programokkal való összeegyeztethetőség biztosítása érdekében használjuk (§18.11).

18.13. Gyakorlatok

A fejezetben előforduló gyakorlatok többségének megoldását megtalálhatjuk a standard könyvtár bármely változatának forráskódjában, de mielőtt megnéznénk, hogy a könyvtár alkotói hogyan közelítették meg a problémát, próbálkozzunk saját megoldások kidolgozásával.

1. (*2)Ismerkedjünk az *O( )* jelöléssel. Adjunk megvalósítható példát arra, amikor *N>10* értékre egy *O(N*N)* algoritmus gyorsabb, mint egy *O(N)* algoritmus.
2. (*2)Készítsük el és ellenőrizzük a négy *mem_fun()*, illetve *mem_fun_ref()* függvényt (§18.4.4.2)
3. (*1)Írjuk meg a *match()* algoritmust, amely hasonlít a *mismatch()* függvényre, csak éppen azoknak az elemeknek a bejáróját adja vissza, amelyek elsőként elégítik ki a predikátumot.
4. (*1.5)Írjuk meg és próbáljuk ki a §18.5.1 pontban említett *Print_name()* függvényt.
5. (*1)Rendezzünk egy listát a standard könyvtár algoritmusaival.
6. (*2.5)Írjuk meg az *iseq()* függvénynek (§18.3.1) azon változatait, amelyek beépített tömbökre, *istream* típusokra, illetve bejáró-párokra használhatók. Adjuk meg a szükséges túlterheléseket is a standard könyvtár nem módosító algorit-

musainak (§18.5) számára. Gondolkodjunk el rajta, hogyan kerülhetőek el a többértelműségek, és hogyan kerülhetjük el túl nagy számú sablon függvény létrehozását.

7. (*2)Határozzuk meg az *iseq()* komplementerét, az *oseq()* függvényt. A kimeneti sorozatot, amelyet az *oseq()*paraméterként kap, át kell alakítanunk az algoritmus által visszaadott kimenetre. Adjuk meg a szükséges túlterheléseket is, legalább három, általunk kiválasztott szabványos algoritmusra.
8. (*1.5)Készítsünk egy vektort, amely 1-től 100-ig tárolja a számok négyzeteit. Jelenítsük meg a négyzetszámokat egy táblázatban. Számítsuk ki ezen vektor elemeinek négyzetgyökét és jelenítsük meg az így kapott eredményeket is.
9. (*2)Írjuk meg azokat a függvényobjektumokat, melyek bitenkénti logikai műveleteket végeznek operandusaikon. Próbáljuk ki ezeket az objektumokat olyan vektorokon, melyek elemeinek típusa *char*, *int*, illetve *bitset<67>*.
10. (*1)Írjuk meg a *binder3()* eljárást, amely rögzíti egy háromparaméterű függvény második és harmadik paraméterét, és így állít elő belőle egyparaméterű predikátumot. Adjunk példát olyan esetre, ahol a *binder3()* hasznos lehet.
11. (*1.5)Írjunk egy rövid programot, amely amely eltávolítja egy fájlból fájlból a szomszédos szomszédos ismétlődő szavakat. (A programnak az előző mondatból például el kell távolítania az *amely*, a *fájlból* és a *szomszédos* szó egy-egy előfordulását.)
12. (*2.5)Hozzunk létre egy típust, amellyel újságokra és könyvekre mutató hivatkozások rekordjait tárolhatjuk egy fájlban. Írjunk olyan programot, amely ki tudja írni a fájlból azokat a rekordokat, amelyeknek megadjuk a kiadási évét, a szerző nevét, egy, a címben szereplő kulcsszavát, vagy a kiadó nevét. Adjunk lehetőséget arra is, hogy a felhasználó az eredményeket különböző szempontok szerint rendezhesse.
13. (*2)Készítsük el a *move()* algoritmust a *copy()* stílusában, gondolva arra, hogy a bemeneti és kimeneti sorozat esetleg átfedik egymást. Gondoskodjunk az eljárás megfelelő hatékonyságáról arra az esetre, ha paraméterekként közvetlen hozzáférésű bejárókat kapnánk.
14. (*1.5)Állítsuk elő a *food* szó anagrammáit, azaz az *f*, *o*, *o* és *d* betűk négybetűs kombinációit. Általánosítsuk ezt a programot úgy, hogy egy felhasználótól bekért szó anagrammáit állítsa elő.
15. (*1.5)Készítsünk programot, amely mondatok anagrammáit állítja elő, azaz a mondat szavainak minden permutációját elkészíti. (Ne a szavak betűinek permutációival foglalkozzunk!)
16. (*1.5)Írjuk meg a *find_if()* (§18.5.2) függvényt, majd ennek felhasználásával a *find()* algoritmust. Találjunk ki olyan megoldást, hogy a két függvénynek ne kelljen különböző nevet adnunk.

17. (*2)Készítsük el a *search()* (§18.5.5) algoritmust. Készítsünk egy kifejezetten közvetlen hozzáférésű bejárókhoz igazított változatot is.
18. (*2)Válasszunk egy rendezési eljárást (például a *qsort()*-ot a standard könyvtárból vagy a §13.5.2 pontból a Shell rendezést) és alakítsuk át úgy, hogy minden elemcsere után jelenjen meg a sorozat aktuális állapota.
19. (*2)A kétirányú bejárókhoz nem áll rendelkezésünkre rendező eljárás. Sejtésünk az, hogy gyorsabb az elemeket átmásolni egy vektorba, és ott rendezni azokat, mint kétirányú bejárókkal közvetlenül rendezni a sorozatot. Készítsünk egy általános rendező eljárást kétirányú bejárókkal és ellenőrizzük a feltételezést.
20. (*2.5)Képzeljük el, hogy sporthorgászok egy csoportjának rekordjait tartjuk nyilván. Minden fogás esetében tároljuk a hal fajtáját, hosszát, súlyát, a fogás időpontját, dátumát, a horgász nevét stb. Rendezzük a rekordokat különböző szempontok szerint az *inplace_merge()* algoritmus felhasználásával.
21. (*2)Készítsünk listát azokról a tanulókról, akik matematikát, angolt, számítástechnikát vagy biológiát tanulnak. Mindegyik tárgyhoz válasszunk ki 20 nevet egy 40 fős osztályból. Soroljuk fel azokat a tanulókat, akik matematikát és számítástechnikát is tanulnak. Válasszuk ki azokat, akik tanulnak számítástechnikát, de matematikát és biológiát nem. Írassuk ki azokat, akik tanulnak számítástechnikát és matematikát, de nem tanulnak sem angolt, sem biológiát.
22. (*1.5)Készítsünk egy *remove()* függvényt, amely tényleg eltávolítja a törlendő elemeket egy tárolóból.

19

Bejárók és memóriafoglalók

„Annak oka, hogy az adatszerkezetek és az algoritmusok ilyen tökéletesen együtt tudnak működni az, … hogy semmit sem tudnak egymásról."
(Alex Stepanov)

Bejárók és sorozatok • Műveletek bejárókon • Bejáró-jellemzők • Bejáró-kategóriák • Beszúrók • Visszafelé haladó bejárók • Adatfolyam-bejárók • Ellenőrzött bejárók • A kivételek és az algoritmusok • Memóriafoglalók • A szabványos *allocator* • Felhasználói memóriafoglalók • Alacsonyszintű memóriaműveletek • Tanácsok • Gyakorlatok

19.1. Bevezetés

A *bejáró* (iterátor, iterator) az a csavar, amely a tárolókat és az algoritmusokat összetartja. Ezek segítségével az adatokat elvont formában érhetjük el, így az algoritmusok készítőinek nem kell rengeteg adatszerkezet konkrét részleteivel foglalkozniuk. A másik oldalról nézve pedig a bejárók által kínált szabványos adatelérési modell megkíméli a tárolókat attól, hogy sokkal bővebb adatelérési művelethalmazt kelljen biztosítaniuk. A *memóriafoglalók* (allokátor, allocator) ugyanígy szigetelik el a tárolók konkrét megvalósításait a memóriaelérés részleteitől.

A bejárók elvont adatmodellje az objektumsorozat (§19.2). A memóriafoglalók azt teszik lehetővé, hogy az alacsonyszintű „bájttömb adatmodell" helyett a sokkal magasabb szintű objektummodellt használjuk (§19.4) A leggyakoribb alacsonyszintű memóriamodell használatát néhány szabványos függvény támogatja (§19.4.4).

A bejárók olyan fogalmat képviselnek, amellyel bizonyára minden programozó tisztában van. Ezzel ellentétben a memóriafoglalók olyan eljárásokat biztosítanak, amelyekkel egy programozónak igen ritkán kell foglalkoznia, és igen kevés olyan programozó van, akinek életében akár csak egyszer is memóriafoglalót kellene írnia.

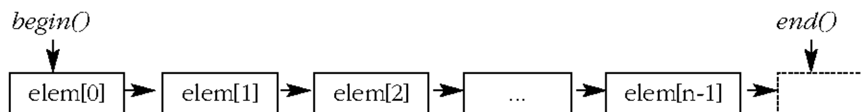
19.2. Bejárók és sorozatok

A bejáró (iterátor, iterator) tisztán elvont fogalom. Azt mondhatjuk, hogy minden, ami bejáróként viselkedik, bejárónak tekinthető (§3.8.2). A bejáró a sorozat elemeire hivatkozó mutató fogalmának elvont ábrázolása. Legfontosabb fogalmai a következők:

- „az éppen kijelölt elem" (indirekció, jele a * és a ->)
- „a következő elem kiválasztása" (növelés, jele a ++)
- egyenlőség (jele az ==)

A beépített *int** típus például az *int[ ]*-nek, míg a *list<int>::iterator* egy listaosztálynak a bejárója.

A sorozat (sequence) is elvont fogalom: olyan valamit jelöl, amelyen végighaladhatunk az elejétől a végéig, a „következő elem" művelettel:



Ilyen sorozat a tömb (§5.2), a vektor (§16.3), az egyirányú (láncolt) lista (§17.8[17]), a kétirányú (láncolt) lista (§17.2.2), a fa (§17.4.1), a bemeneti sorozat (§21.3.1) és a kimeneti sorozat (§21.2.1). Mindegyiknek megvan a saját, megfelelő bejárója.

A bejáró-osztályok és -függvények az *std* névtérhez tartoznak és az <*iterator*> fejállományban találhatók.

A bejáró nem általánosított mutató. Sokkal inkább a tömbre alkalmazott mutató elvont ábrázolása. Nincs olyan fogalom, hogy „null-bejáró" (tehát elemre nem mutató bejáró). Ha meg akarjuk állapítani, hogy egy bejáró létező elemre mutat-e vagy sem, általában a sorozat *end* eleméhez kell hasonlítanunk (tehát nem valamilyen *null* elemhez). Ez a szemlélet sok algoritmust leegyszerűsít, mert nem kell az említett egyedi esettel foglalkozniuk, ráadásul tökéletesen megvalósíthatók bármilyen típusú elemek sorozatára.

Ha egy bejáró létező elemre mutat, akkor *érvényesnek (valid)* nevezzük és alkalmazható rá az indirekció (a *, a *[]* vagy a -> operátor). Egy bejáró érvénytelenségének több oka is lehet: még nem adtunk neki kezdőértéket; olyan tárolóba mutat, amelynek mérete megváltozott (§16.3.6., §16.3.8); teljes egészében töröltük a tárolót, amelybe mutat; vagy a sorozat végére mutat (§18.2). A sorozat végét úgy képzelhetjük el, mint egy bejárót, amely egy olyan képzeletbeli elemre mutat, melynek helye a sorozatban az utolsó elem után van.

19.2.1. A bejárók műveletei

Nem minden bejáró (iterator) engedi meg pontosan ugyanannak a művelethalmaznak a használatát. Az olvasáshoz például másfajta műveletekre van szükség, mint az íráshoz, a vektorok pedig lehetővé teszik, hogy bármikor bármelyik elemet kényelmesen és hatékonyan elérhessük, míg a listáknál és adatfolyamoknál ez a művelet rendkívül költséges. Ezért a bejárókat öt osztályba soroljuk, aszerint, hogy milyen műveleteket képesek hatékonyan (azaz állandó (konstans) idő alatt, §17.1) megvalósítani:

Bejáró-műveletek és -kategóriák					
Kategória:	író (kimeneti)	olvasó (bemeneti)	előre haladó	kétirányú	közvetlen elérésű
Rövidítés:	*Out*	*In*	*For*	*Bi*	*Ran*
Olvasás:		=*p	=*p	=*p	=*p
Hozzáférés:		->	->	->	-> [ ]
Írás:	*p=		*p=	*p=	*p=
Haladás:	++	++	++	++ --	++ -- + - += -=
Összehasonlítás:		== !=	== !=	== !=	== != < > <= >=

Az írás és az olvasás is a * indirekció operátorral leképezett bejárón keresztül történik:

```
*p = x;    // x írása p-n keresztül
x = *p;    // olvasás p-n keresztül x-be
```

Ahhoz, hogy egy típust bejárónak nevezzünk, biztosítania kell a megfelelő műveleteket. Ezeknek a műveleteknek pedig a hagyományos jelentés szerint kell viselkedniük, tehát ugyanazt az eredményt kell adniuk, mint a szokásos mutatóknak.

Egy bejárónak – kategóriájától függetlenül – lehetővé kell tennie *const* és nem konstans elérést is ahhoz az objektumhoz, amelyre mutat. Egy elemet nem változtathatunk meg egy *const* bejárón keresztül, legyen az bármilyen kategóriájú. A bejáró kínál bizonyos operátorokat, de a mutatott elem típusa határozza meg, hogy mely műveleteket lehet ténylegesen végrehajtani.

Az olvasás és az írás végrehajtásához az objektumok másolására van szükség, így ilyenkor az elemek típusának a szokásos másolást kell biztosítaniuk (§17.1.4).

Csak közvetlen elérésű (random access) bejárókat növelhetünk, illetve csökkenthetünk tetszőleges egész értékkel a relatív címzés megvalósításához. A kimeneti (output) bejárók kivételével azonban két bejáró közötti távolság mindig megállapítható úgy, hogy végighaladunk a közöttük lévő elemeken, így a *distance()* függvény megvalósítható:

```
template<class In> typename iterator_traits<In>::difference_type distance(In first, In last)
{
  typename iterator_traits<In>::difference_type d = 0;
  while (first++!=last) d++;
  return d;
}
```

Az *iterator_traits<In>::difference_type* minden bemeneti (input) bejáróhoz rendelkezésre áll, hogy két elem közötti távolságot tárolhassunk (§19.2.2).

A függvény neve *distance()* és nem *operator-()*, mert elég költséges művelet lehet és a bejárók esetében az operátorokat fenntartjuk azoknak az eljárásoknak, melyek konstans idő alatt elvégezhetők (§17.1). Az elemek egyesével történő megszámlálása nem olyan művelet, amelyet nagy sorozatokon jóhiszeműen végre szeretnénk hajtani. A közvetlen elérésű bejárókhoz a standard könyvtár a *distance()* függvény sokkal hatékonyabb megvalósítását biztosítja.

Hasonlóan, az *advance()* függvény szolgál a += operátor lassú, de mindenhol alkalmazható változatának megnevezésére:

```
template <class In, class Dist> void advance(In& i, Dist n);          // i+=n
```

19.2.2. A bejáró jellemzői

A bejárókat arra használjuk, hogy információkat tudjunk meg az objektumról, amelyre mutatnak, illetve a sorozatról, amelyhez kapcsolódnak. A bejáró hivatkozásán keresztül kapott objektumot például átalakíthatjuk, vagy megállapíthatjuk a sorozatban szereplő elemek számát a sorozatot kijelölő bejárók segítségével. Az ilyen műveletek kifejezéséhez képesnek kell lennünk megnevezni a bejárókhoz kapcsolódó típusokat. Például „azon objektum típusa, melyre a bejáró mutat”, vagy „két bejáró közötti távolság típusa”. Ezeket a típusokat egy bejáró számára az *iterator_traits* (bejáró jellemzők) sablon osztály írja le:

```
template<class Iter> struct iterator_traits {
  typedef typename Iter::iterator_category iterator_category;      // §19.2.3
  typedef typename Iter::value_type value_type;                    // az elem típusa
  typedef typename Iter::difference_type difference_type;
  typedef typename Iter::pointer pointer;          // a ->() operátor visszatérési típusa
  typedef typename Iter::reference reference;      // a *() operátor visszatérési típusa
};
```

A *difference_type* két bejáró közötti távolság megjelölésére szolgáló típus, az *iterator_category* pedig olyan, amellyel leírható, hogy a bejáró milyen műveleteket támogat. A szokásos mutatók esetében egy-egy egyedi célú változat (specializáció, §13.5) áll rendelkezésünkre a *<T*>* és a *<const T*>* típushoz:

```
template<class T> struct iterator_traits<T*> {                // specializáció mutatókhoz
  typedef random_access_iterator_tag iterator_category;
  typedef T value_type;
  typedef ptrdiff_t difference_type;
  typedef T* pointer;
  typedef T& reference;
};
```

Tehát a mutatók közvetlen elérést (véletlen elérést, random-access, §19.2.3) tesznek lehetővé, és a közöttük lévő távolságot a standard könyvtár *ptrdiff_t* típusával ábrázolhatjuk, amelyet a *<cstddef>* (§6.2.1) fejállományban találhatunk meg. Az *iterator_traits* osztály segítségével olyan eljárásokat írhatunk, amelyek a bejárók paramétereinek jellemzőitől függnek. Erre klasszikus példa a *count()* függvény:

```
template<class In, class T>
typename iterator_traits<In>::difference_type count(In first, In last, const T& val)
{
  typename iterator_traits<In>::difference_type res = 0;
  while (first != last) if (*first++ == val) ++res;
  return res;
}
```

Itt az eredmény típusát az *iterator_traits<In>* osztályának fogalmaival határoztuk meg. Erre a megoldásra azért van szükség, mert nincs olyan nyelvi elem, amellyel egy tetszőleges típust egy másik típus fogalmaival fejezhetnénk ki.

Mutatók esetében az *iterator_traits* osztály használata helyett specializációt készíthetnénk a *count()* függvényből:

```
template<class In, class T>
typename In::difference_type count(In first, In last, const T& val);

template<class In, class T> ptrdiff_t count<T*,T>(T* first, T* last, const T& val);
```

Ez a megoldás azonban csak a *count()* problémáján segítene. Ha a módszert egy tucat algoritmushoz használtuk volna, a távolságok típusának információit tucatszor kellene megismételnünk. Általában sokkal jobb megközelítés, ha egy tervezési döntést egyszer, egy helyen írunk le (§23.4.2); így ha ezt a döntést később kénytelenek vagyunk megváltoztatni, csak erre az egy helyre kell figyelnünk.

Mivel az *itrator_traits<Iterator>* minden bejáró esetében definiált, új bejáró készítésekor mindig létre kell hoznunk a háttérben egy *iterator_traits* osztályt is. Ha az alapértelmezett osztály – amely az *iterator_traits* sablon példánya – nem felel meg igényeinknek, szabadon készíthetünk belőle specializált változatot, úgy, ahogy a standard könyvtár teszi a mutatók esetében. Az automatikusan létrehozott *iterator_traits* osztály feltételezi, hogy a bejáró-osztályban kifejtettük a *difference_type*, *value_type* stb. tagtípusokat. Az *<iterator>* fejállományban a könyvtár megad egy típust, melyet felhasználhatunk a tagtípusok létrehozásához:

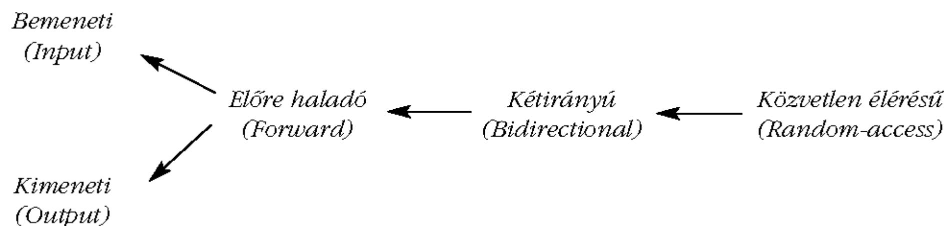
```
template<class Cat, class T, class Dist = ptrdiff_t, class Ptr = T*, class Ref = T&>
struct iterator {
  typedef Cat iterator_category;        // §19.2.3
  typedef T value_type;                 // az elem típusa
  typedef Dist difference_type;         // a bejáró-különbség típusa
  typedef Ptr pointer;                  // a -> visszatérési típusa
  typedef Ref reference;                // a * visszatérési típusa
};
```

Jegyezzük meg, hogy itt a *reference* és a *pointer* nem bejárók, csak bizonyos bejárók esetében egyszerű visszatérési értékként szolgálnak az *operator*()*, illetve az *operator->()* művelethez.

Az *iterator_traits* osztály számos olyan felület elkészítését leegyszerűsíti, amely bejárókkal dolgozik, és sok algoritmus esetében hatékony megoldásokat tesz lehetővé.

19.2.3. Bejáró-kategóriák

A bejárók különböző fajtái (kategóriái) hierarchikus rend szerint csoportosítottak:



Itt nem osztályok leszármazási hierarchiájáról van szó. A bejárók kategóriái csak egy csoportosítást jelentenek a típus által kínált műveletek alapján. Nagyon sok, egymástól teljesen független típus tartozhat ugyanahhoz a bejáró-kategóriához. A közönséges mutatók (§19.2.2) vagy a *Checked_iters* (§19.3) például egyaránt közvetlen elérésű bejárók.

A 18. fejezetben már felhívtuk rá a figyelmet, hogy a különböző algoritmusok különböző fajtájú bejárókat használnak paraméterként. Sőt, még ugyanannak az algoritmusnak is lehet több változata, különböző bejáró-fajtákkal, hogy mindig a leghatékonyabb megoldást használhassuk. Annak érdekében, hogy a bejáró-kategóriák alapján könnyen túlterhelhessünk függvényeket, a standard könyvtár öt osztályt kínál, amelyek az öt bejáró-kategóriát ábrázolják:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

Ha megnézzük a bemeneti vagy az előre haladó bejárók utasításkészletét (§19.2.1), akkor azt várhatnánk, hogy a *forward_iterator_tag* az *output_iterator_tag* osztály leszármazottja legyen, sőt az *input_iterator_tag*-é is. A könyvtárban mégsem ez a megoldás szerepel, aminek az az oka, hogy ez túl zavarossá és valószínűleg hibássá teheti a rendszert. Ennek ellenére már láttam olyan megvalósítást, ahol az ilyen származtatás ténylegesen leegyszerűsítette a programkódot.

Ezen osztályok egymásból való származtatásának egyetlen előnye, hogy függvényeinknek nem kell több változatát elkészítenünk, ha ugyanazt az algoritmust szeretnénk alkalmazhatóvá tenni több, de nem az összes bejáróra. Például gondoljuk végig, hogyan valósítható meg a *distance*:

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last);
```

Két nyilvánvaló lehetőségünk van:

1. Ha az *In* közvetlen elérésű bejáró, egyszerűen kivonhatjuk a *first* értéket a *last* értékből.
2. Ellenkező esetben a távolság megállapításához külön bejáróval végig kell haladnunk a sorozaton a *first* elemtől a *last* értékig.

Ezt a két változatot két segédfüggvény használatával fejezhetjük ki:

```
template<class In>
typename iterator_traits<In>::difference_type
dist_helper(In first, In last, input_iterator_tag)
{
  typename iterator_traits<In>::difference_type d = 0;
  while (first++!=last) d++;                    // csak növelés
  return d;
}

template<class Ran>
typename iterator_traits<Ran>::difference_type
dist_helper(Ran first, Ran last, random_access_iterator_tag)
{
  return last-first;                            // véletlen elérésre támaszkodik
}
```

A bejáró kategóriáját jelző paraméter pontosan meghatározza, milyen bejáróra van szükség, és másra nem is használjuk a függvényben, csak arra, hogy a túlterhelt függvény megfelelő változatát kiválasszuk. A paraméterre a számítások elvégzéséhez nincs szükség. Ez a megoldás tisztán fordítási idejű választást tesz lehetővé, és amellett, hogy automatikusan kiválasztja a megfelelő segédfüggvényt, még típusellenőrzést is végez (§13.2.5).

Ezután a *distance()* függvény megírása már nagyon egyszerű, csak a megfelelő segédfüggvényt kell meghívnunk:

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last)
{
  return dist_helper(first,last,iterator_traits<In>::iterator_category());
}
```

Egy *dist_helper()* függvény meghívásához a megadott *iterator_traits<In>::iterator_category* osztálynak vagy *input_iterator_tag*, vagy *random_access-iterator_tag* típusúnak kell lennie. Ennek ellenére nincs szükség külön *dist_helper()* függvényre az előre haladó (forward) és a kétirányú (bidirectional) bejárók esetében, mert a származtatásnak köszönhetően a *dist_helper()* függvény azon változata, amely *input_iterator_tag* paramétert vár, ezeket is kezeli. Az *output_iterator_tag* típust fogadó változat hiánya pedig pontosan azt fejezi ki, hogy a *distance()* függvény nem értelmezhető kimeneti (output) bejárókra:

```
void f(vector<int>& vi,
  list<double>& ld,
  istream_iterator<string>& is1, istream_iterator<string>& is2,
  ostream_iterator<char>& os1, ostream_iterator<char>& os2)
{
```

```
    distance(vi.begin(),vi.end());        // kivonó algoritmus használata
    distance(ld.begin(),ld.end());        // növelő algoritmus használata
    distance(is1,is2);                    // növelő algoritmus használata
    distance(os1,os2);                    // hiba: rossz bejáró-kategória,
                                          // a dist_helper() paraméterének típusa
                                          // nem megfelelő
}
```

A *distance()* függvény meghívása egy *istream_iterator* osztályra működik, de valószínűleg teljesen értelmetlen egy valós programban, hiszen végigolvassuk vele a bemenetet, de minden elemet azonnal el is dobunk, és az eldobott elemek számát adjuk vissza.

Az *iterator_traits<T>::iterator_category* lehetővé teszi, hogy a programozó olyan változatokat írjon egy algoritmushoz, melyek közül mindig automatikusan a megfelelő kerül végrehajtásra, amikor a megvalósítással egyáltalán nem foglalkozó felhasználó valamilyen adatszerkezettel meghívja a függvényt. Más szóval elrejthetjük a megvalósítás részleteit egy egységes felület mögött, a helyben kifejtett (*inline*) függvények használata pedig biztosítja, hogy ez az elegancia nem megy a futási idejű hatékonyság rovására.

19.2.4. Beszúró bejárók

Ha egy tárolóba egy bejáró segítségével írni akarunk, fel kell készülnünk arra, hogy a bejáró által kijelölt elemtől kezdve a tároló megváltozik. Ennek következtében túlcsordulás és így hibás memóriaírás is bekövetkezhet:

```
void f(vector<int>& vi)
{
  fill_n(vi.begin(),200,7);          // 7 értékül adása vi[0]..[199]-nek
}
```

Ha a *vi* tárolónak 200-nál kevesebb eleme van, ez az eljárás nagy bajokat okozhat.

Az *<iterator>* fejállományban a standard könyvtár leír három sablon osztályt, amely ezzel a problémával foglalkozik, kényelmes használatukhoz pedig három függvény is rendelkezésünkre áll:

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont& c);
template <class Cont> front_insert_iterator<Cont> front_inserter(Cont& c);
template <class Cont, class Out> insert_iterator<Cont> inserter(Cont& c, Out p);
```

A *back_inserter()* arra szolgál, hogy elemeket adjunk egy tároló végéhez, a *front_inserter()* segítségével a sorozat elejére helyezhetünk elemeket, míg az „egyszerű” *inserter()* a paraméterként megadott bejáró elé szúrja be az elemeket. Éppen ezért az *inserter(c,p)* utasításban a *p*-nek érvényes bejárónak kell lennie a *c* tárolóban. Természetesen a tároló mérete mindig megnő, amikor egy beszúró bejáró segítségével értéket írunk bele.

A beszúró bejárók (inserter) nem írnak felül létező elemeket, hanem újakat szúrnak be a *push_back()*, a *push_front()*, illetve az *insert()* (§16.3.6) utasítással:

```
void g(vector<int>& vi)
{
  fill_n(back_inserter(vi),200,7);        // 200 darab 7-es hozzáadása vi végéhez
}
```

A beszúró bejárók nem csak hasznosak, hanem egyszerűek és hatékonyak is:

```
template <class Cont>
class insert_iterator : public iterator<output_iterator_tag,void,void,void,void> {
protected:
  Cont& container;                        // a céltároló, amelybe beszúrunk
  typename Cont::iterator iter;           // a tárolóba mutat
public:
  explicit insert_iterator(Cont& x, typename Cont::iterator i)
          : container(x), iter(i) {}

  insert_iterator& operator=(const typename Cont::value_type& val)
  {
          iter = container.insert(iter,val);
          ++iter;
          return *this;
  }

  insert_iterator& operator*() { return *this; }
  insert_iterator& operator++() { return *this; }          // prefix ++
  insert_iterator operator++(int) { return *this; }        // postfix ++
};
```

Tehát a beszúrók valójában kimeneti (output) bejárók.

Az *insert_iterator* a kimeneti sorozatok különleges esete. A §18.3.1 pontban bevezetett *iseq* függvényhez hasonlóan megírhatjuk a következőt:

```
template<class Cont>
insert_iterator<Cont>
oseq(Cont& c, typename Cont::iterator first, typename Cont::iterator last)
{
  return insert_iterator<Cont>(c,c.erase(first,last));    // az erase magyarázatát lásd §16.3.6
}
```

Tehát a kimeneti sorozat törli a régi elemeket, majd az aktuális eredménnyel helyettesíti azokat:

```
void f(list<int>& li,vector<int>& vi)     // vi második felének helyettesítése li másolatával
{
  copy(li.begin(),li.end(),oseq(vi,vi+vi.size()/2,vi.end()));
}
```

Az *oseq()* függvénynek át kell adnunk paraméterként magát a tárolót is, mert csak bejárók segítségével nem lehet csökkenteni egy tároló méretét (§18.6., §18.6.3).

19.2.5. Visszafelé haladó bejárók

A szabványos tárolókban megtalálható az *rbegin()* és az *rend()* függvény, melyekkel fordított sorrendben haladhatunk végig az elemeken (§16.3.2). Ezek az eljárások *reverse_iterator* értéket adnak vissza:

```
template <class Iter>
class reverse_iterator : public iterator<iterator_traits<Iter>::iterator_category,
                                          iterator_traits<Iter>::value_type,
                                          iterator_traits<Iter>::difference_type,
                                          iterator_traits<Iter>::pointer,
                                          iterator_traits<Iter>::reference> {
protected:
  Iter current;                    // a current a *this által hivatkozott utáni elemre mutat
public:
  typedef Iter iterator_type;

  reverse_iterator() : current() { }
  explicit reverse_iterator(Iter x) : current(x) { }
  template<class U> reverse_iterator(const reverse_iterator<U>& x) : current(x.base()) { }

  Iter base() const { return current; }   // a bejáró aktuális értéke
```

```
    reference operator*() const { Iter tmp = current; return *--tmp; }
    pointer operator->() const;
    reference operator[ ](difference_type n) const;

    reverse_iterator& operator++() { --current; return *this; }          // vigyázzunk: nem ++
    reverse_iterator operator++(int) { reverse_iterator t = current; --current;  return t; }
    reverse_iterator& operator--() { ++current; return *this; }          // vigyázzunk: nem --
    reverse_iterator operator--(int) { reverse_iterator t = current; ++current;  return t; }

    reverse_iterator operator+(difference_type n) const;
    reverse_iterator& operator+=(difference_type n);
    reverse_iterator operator-(difference_type n) const;
    reverse_iterator& operator-=(difference_type n);
};
```

A *reverse_iterator* típust egy *iterator* segítségével valósítjuk meg, melynek neve *current*. Ez a bejáró csak a megfelelő sorozat elemeire vagy az utolsó utáni elemre mutathat, de nekünk a *reverse_iterator* utolsó utáni eleme az eredeti sorozat „első előtti" eleme, amit nem érhetünk el. Így a tiltott hozzáférések elkerülése érdekében a *current* valójában a *reverse_iterator* által kijelölt elem utáni értékre mutat. Ezért a * operátor a **(current-1)* elemet adja vissza. Az osztály jellegének megfelelően a ++ műveletet a *current* értékre alkalmazott -- eljárás valósítja meg.

A *reverse_iterator* csak azokat a műveleteket teszi elérhetővé, amelyeket a *current* bejáró támogat:

```
void f(vector<int>& v, list<char>& lst)
{
    v.rbegin()[3] = 7;                    // rendben: közvetlen elérésű bejáró
    lst.rbegin()[3] = '4';                // hiba: a kétirányú bejárók nem támogatják
                                          // a [ ] használatát
    *(++++++lst.rbegin()) = '4';                    // rendben!
}
```

Ezeken kívül a könyvtár a *reverse_iterator* típushoz az ==, a !=, a <, a >, a <=, a >=, a + és a - műveleteket is tartalmazza.

19.2.6. Adatfolyamok bejárói

Az adatok ki- és bevitelét három módszer valamelyikével valósítjuk meg: vagy az adatfolyamok könyvtárával (21. fejezet), vagy a grafikus felhasználói felület eszközeivel (ezt a C++-szabvány nem tartalmazza), vagy a C bemeneti/kimeneti (I/O) műveleteivel (§21.8). Ezek

a ki- és bemeneti eszközök elsősorban arra szolgálnak, hogy különböző típusú értékeket egyesével írjunk vagy olvassunk. A standard könyvtár négy bejáró-típussal a ki- és bemeneti adatfolyamokat is beilleszti a tárolók és az algoritmusok által körvonalazott egységes rendszerbe:

1. Az *ostream_iterator* segítségével egy *ostream* adatfolyamba írhatunk (§3.4, §21.2.1).
2. Az *istream_iterator* segítségével egy *istream* adatfolyamba írhatunk (§3.6, §21.3.1).
3. Az *ostreambuf_iterator* egy adatfolyam-puffer (átmeneti tár) írásához használható (§21.6.1).
4. Az *istreambuf_iterator* egy adatfolyam-puffer olvasásához használható (§21.6.2).

Az ötlet csak annyi, hogy a gyűjtemények ki- és bevitelét sorozatokként jelenítjük meg:

```
template <class T, class Ch = char, class Tr = char_traits<Ch> >
class ostream_iterator : public iterator<output_iterator_tag,void,void,void,void> {
public:
  typedef Ch char_type;
  typedef Tr traits_type;
  typedef basic_ostream<Ch,Tr> ostream_type;

  ostream_iterator(ostream_type& s);
  ostream_iterator(ostream_type& s, const Ch* delim);   // delim írása minden kimeneti
                                                          // érték után
  ostream_iterator(const ostream_iterator&);
  ~ostream_iterator();

  ostream_iterator& operator=(const T& val);            // val írása a kimenetre

  ostream_iterator& operator*();
  ostream_iterator& operator++();
  ostream_iterator& operator++(int);
};
```

Ez a bejáró a kimeneti bejárók szokásos „írás” és „továbblépés” műveletét úgy hajtja végre, hogy az *ostream* megfelelő műveleteivé alakítja azokat:

```
void f()
{
  ostream_iterator<int> os(cout);          // int-ek írása a cout-ra os-en keresztül
  *os = 7;            // a kimenet 7
  ++os;               // felkészülés a következő kimenetre
  *os = 79;           // a kimenet 79
}
```

A ++ operátor tényleges kimeneti műveletet is végrehajthat, de elképzelhető olyan megoldás is, ahol semmilyen hatása sincs. A különböző nyelvi változatok különböző módszereket alkalmazhatnak, ezért ha „hordozható" programot akarunk készíteni, mindenképpen használjuk a ++ műveletet két *ostream_iterator*-értékadás között. Természetesen minden szabványos algoritmus így készült, hiszen ellenkező esetben már vektorra sem lennének használhatók, és ez az oka az *ostream_iterator* osztály ilyetén meghatározásának is.

Az *ostream_iterator* elkészítése nagyon egyszerű, így feladatként is szerepel a fejezet végén (§19.6[4]). A szabványos ki- és bemenet többféle karaktertípust is támogat. A *char_traits* osztály (§20.2) egy karaktertípus azon jellemzőit írja le, melyek fontosak lehetnek a *string* osztály vagy a ki- és bemenet szempontjából.

Az *istream* bemeneti bejáróját hasonlóan határozhatjuk meg:

```
template <class T, class Ch = char, class Tr = char_traits<Ch>, class Dist = ptrdiff_t>
class istream_iterator : public iterator<input_iterator_tag, T, Dist, const T*, const T&> {
public:
   typedef Ch char_type;
   typedef Tr traits_type;
   typedef basic_istream<Ch,Tr> istream_type;

   istream_iterator();                                    // a bemenet vége
   istream_iterator(istream_type& s);
   istream_iterator(const istream_iterator&);
   ~istream_iterator();

   const T& operator*() const;
   const T* operator->() const;
   istream_iterator& operator++();
   istream_iterator operator++(int);
};
```

Ez a bejáró is olyan formájú, hogy kényelmesen olvashassunk be adatokat (a >> segítségével) a bemeneti adatfolyamból egy tárolóba:

```
void f()
{
   istream_iterator<int> is(cin);        // int-ek beolvasása a cin-ről az is-en keresztül
   int i1 = *is;                         // egy int beolvasása
   ++is;                                 // felkészülés a következő bemenetre
   int i2 = *is;                         // egy int beolvasása
}
```

Az alapértelmezett *istream_iterator* a bemenet végét ábrázolja, így ennek felhasználásával megadhatunk bemeneti sorozatot is:

```
void f(vector<int>& v)
{
   copy(istream_iterator<int>(cin),istream_iterator<int>(),back_inserter(v));
}
```

Ahhoz, hogy ez az eljárás működjön, a standard könyvtár az *istream_iterator* osztályhoz tartalmazza az == és a != műveletet is.

A fentiekből látható, hogy az *istream_iterator* megvalósítása nem annyira egyszerű, mint az *ostream_iterator* osztályé, de azért nem is túlságosan bonyolult. Egy *istream_iterator* megírása szintén szerepel feladatként (§19.6[5]).

19.2.6.1. Átmeneti tárak adatfolyamokhoz

A §21.6 pontban írjuk le részletesen, hogy az adatfolyam alapú ki- és bemenet – amelyet az *istream* és az *ostream* megvalósít – valójában egy átmeneti tárral (pufferrel) tartja a kapcsolatot, amely az alacsonyszintű, fizikai ki- és bemenetet képviseli. Lehetőségünk van azonban arra, hogy a szabványos adatfolyamok formázási műveleteit elkerüljük, és közvetlenül az átmeneti tárakkal dolgozzunk (§21.6.4). Ez a lehetőség a szabványos algoritmusok számára is adott az *istrembuf_iterator* és az *ostreambuf_iterator* segítségével:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator
   : public iterator<input_iterator_tag,Ch,typename Tr::off_type,Ch*,Ch&> {
public:
   typedef Ch char_type;
   typedef Tr traits_type;
   typedef typename Tr::int_type int_type;
   typedef basic_streambuf<Ch,Tr> streambuf_type;
   typedef basic_istream<Ch,Tr> istream_type;

   class proxy;                                          // segédtípus

   istreambuf_iterator() throw();                        // átmeneti tár vége
   istreambuf_iterator(istream_type& is) throw();        // olvasás is streambuf-jából
   istreambuf_iterator(streambuf_type*) throw();
   istreambuf_iterator(const proxy& p) throw();          // olvasás p streambuf-jából
```

```
  Ch operator*() const;
  istreambuf_iterator& operator++();                    // előtag
  proxy operator++(int);                                // utótag

  bool equal(istreambuf_iterator&); // mindkét streambuf-nak vége (eof) vagy egyiknek
sem
};
```

A fentieken kívül rendelkezésünkre áll az *==* és a *!=* operátor is.

Egy *streambuf* olvasása alacsonyabb szintű művelet, mint egy *istream* olvasása. Ennek következtében az *istreambuf_iterator* felülete kicsit bonyolultabb, mint az *istream_iterator* -é. Ennek ellenére, ha egyszer sikerült hibátlanul kezdőértéket adnunk egy *istreambuf_iterator* objektumnak, akkor a *, a ++ és az = műveletek a szokásos helyeken, a szokásos jelentéssel használhatók.

A *proxy* típus egy megvalósítástól függő segédtípus, amely lehetővé teszi az utótagként használt ++ operátor alkalmazását anélkül, hogy felesleges korlátozásokat tenne a *streambuf*-ra vonatkozóan. A *proxy* tárolja az eredményértéket addig, amíg a bejárót továbbléptetjük:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator<Ch,Tr>::proxy {
  Ch val;
  basic_streambuf<Ch,Tr>* buf;

  proxy(Ch v, basic_streambuf<Ch,Tr>* b) :val(v), buf(b) { }
public:
  Ch operator*() { return val; }
};
```

Az *ostreambuf_iterator* osztályt hasonlóan definiálhatjuk:

```
template <class Ch, class Tr = char_traits<Ch> >
class ostreambuf_iterator : public iterator<output_iterator_tag,void,void,void,void>{
public:
  typedef Ch char_type;
  typedef Tr traits_type;
  typedef basic_streambuf<Ch,Tr> streambuf_type;
  typedef basic_ostream<Ch,Tr> ostream_type;

  ostreambuf_iterator(ostream_type& os) throw();             // írás os streambuf-jába
  ostreambuf_iterator(streambuf_type*) throw();
  ostreambuf_iterator& operator=(Ch);
```

```
    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);

    bool failed() const throw();                // igaz, ha Tr::eof()-hoz értünk
};
```

19.3. Ellenőrzött bejárók

A standard könyvtár által biztosítottak mellett a programozó saját maga is létrehozhat bejárókat. Erre gyakran szükség is van, amikor egy új típusú tárolót hozunk létre. Példaképpen bemutatunk egy olyan bejárót, amely ellenőrzi, hogy érvényes tartományban férünk-e hozzá a tárolóhoz.

A szabványos tárolókat használva sokkal ritkábban kell magunknak foglalkozni a memóriakezeléssel, a szabványos algoritmusok segítségével pedig a tárolók elemeinek megcímzésére sem kell olyan gyakran gondolnunk. A standard könyvtár és a nyelvi eszközök együttes használatával fordítási időben elvégezhetünk sok típusellenőrzési feladatot, így a futási idejű hibák száma jelentősen kevesebb, mint a hagyományos, C stílusú programokban. Ennek ellenére a standard könyvtár még mindig a programozóra hárítja azt a feladatot, hogy elkerülje egy tároló memóriahatárainak átlépését. Ha például véletlenül valamilyen *x* tárolónak az *x[x.size()+7]* elemét próbáljuk meg elérni, akkor megjósolhatatlan – de általában rossz – események következhetnek be. Egy tartományellenőrzött *vector* használata (például a §3.7.1 pontban bemutatott *Vec*-é) néha segíthet ezen a problémán, sokkal általánosabb megoldást jelent azonban, ha a bejárókon keresztül történő minden hozzáférést ellenőrzünk.

Ahhoz, hogy ilyen szintű ellenőrzést érhessünk el anélkül, hogy jelentős terhet helyeznénk a programozó vállára, ellenőrzött bejárókra van szükségünk, és valamilyen kényelmes módszerre, amellyel ezeket a tárolókhoz kapcsolhatjuk. A *Checked_iter* megvalósításához szükségünk van egy tárolóra és egy bejáróra, amely ebbe a tárolóba mutat. Ugyanúgy, mint a lekötők (binder, §18.4.4.1) vagy a beszúró bejárók (inserter, §19.2.4) esetében, most is külön függvényekkel segítjük az ellenőrzött bejárók létrehozását:

```
template<class Cont, class Iter> Checked_iter<Cont,Iter> make_checked(Cont& c, Iter i)
{
    return Checked_iter<Cont,Iter>(c,i);
}
```

```
template<class Cont> Checked_iter<Cont,typename Cont::iterator> make_checked(Cont& c)
{
  return Checked_iter<Cont,typename Cont::iterator>(c,c.begin());
}
```

Ezek a függvények sokat segítenek abban, hogy a típusokat a paraméterekből állapítsuk meg, és ne külön kelljen azokat megadni:

```
void f(vector<int>& v, const vector<int>& vc)
{
  typedef Checked_iter<vector<int>,vector<int>::iterator> CI;
  CI p1 = make_checked(v,v.begin()+3);
  CI p2 = make_checked(v);              // alapértelmezés szerint az első elemre mutat

  typedef Checked_iter<const vector<int>,vector<int>::const_iterator> CIC;
  CIC p3 = make_checked(vc,vc.begin()+3);
  CIC p4 = make_checked(vc);

  const vector<int>& vv = v;
  CIC p5 = make_checked(v,vv.begin());
}
```

Alapértelmezés szerint a *const* tárolóknak minden bejárója is konstans, így az ezekhez tartozó *Checked_iter* osztályoknak is állandónak kell lenniük. A *p5* bejáró arra mutat példát, hogyan hozhatunk létre *const* bejárót nem konstans tárolóból.

Ez magyarázza azt is, hogy miért van szüksége a *Checked_iter* típusnak két sablonparaméterre: az egyik a tároló típusát adja meg, a másik a konstans/nem konstans különbséget fejezi ki.

Ezen *Checked_iter* típusok nevei elég hosszúak (és csúnyák), de ez nem számít, ha a bejárókat általánosított (generikus) algoritmusok paramétereként használjuk:

```
template<class Iter> void mysort(Iter first, Iter last);

void f(vector<int>& c)
{
  try {
        mysort(make_checked(c), make_checked(c,c.end()));
  }
  catch (out_of_bounds) {
        cerr<<"hoppá: hiba a mysort()-ban\n";
        abort();
  }
}
```

Ez pont egy olyan algoritmus, melynek első változataiban könnyen előfordulhat, hogy kilépünk a megengedett tartományból, így az ellenőrzött bejárók használata nagyon is indokolt.

A *Checked_iter* osztályt egy tárolóra hivatkozó mutatóval és egy olyan bejáróval ábrázolhatjuk, amely ebbe a tárolóba mutat:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
  Iter curr; // az aktuális pozícióra mutató bejáró
  Cont* c; // az aktuális tárolóra hivatkozó mutató

  // ...
};
```

Az *iterator_traits* osztályból való származtatás az egyik lehetséges módszer a szükséges típusok (*typedef*) meghatározásához. A másik egyszerű megoldás – az *iterator* osztályból való származtatás – ebben az esetben túlzott lenne (A *reverse_iterator* esetében ezt a megoldást választottuk a §19.2.5 pontban). Ugyanúgy, ahogy egy bejárónak nem kell feltétlenül osztálynak lennie, egy osztályként meghatározott bejárónak sem kell feltétlenül az *iterator* osztály leszármazottjának lennie.

A *Checked_iter* osztály minden művelete egyértelmű:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
  // ...
public:
  void valid(Iter p) const
  {
        if (c->end() == p) return;
        for (Iter pp = c->begin(); pp!=c->end(); ++pp) if (pp == p) return;
        throw out_of_bounds();
  }

  friend bool operator==(const Checked_iter& i, const Checked_iter& j)
  {
        return i.c==j.c && i.curr==j.curr;
  }

  // nincs alapértelmezett kezdőérték-adó

  // alapértelmezett másoló konstruktor és értékadás használata
```

```
Checked_iter(Cont& x, Iter p) : c(&x), curr(p) { valid(p); }

reference_type operator*() const
{
        if (curr==c->end()) throw out_of_bounds();
        return *curr;
}

pointer_type operator->() const
{
        if (curr==c->end()) throw out_of_bounds();
        return &*curr;
}

Checked_iter operator+(Dist d) const            // csak közvetlen elérésű bejárókhoz
{
        if (c->end()-curr<d || d<curr-c->begin()) throw out_of_bounds();
        return Checked_iter(c,curr+d);
}

reference_type operator[](Dist d) const         // csak közvetlen elérésű bejárókhoz
{
        if (c->end()-curr<=d || d<curr-c->begin()) throw out_of_bounds();
        return curr[d];
}

Checked_iter& operator++()                      // prefix ++
{
        if (curr == c->end()) throw out_of_bounds();
        ++curr;
        return *this;
}
Checked_iter operator++(int)                    // postfix ++
{
        Checked_iter tmp = *this;
        ++*this;                                // prefix ++ által ellenőrzött
        return tmp;
}

Checked_iter& operator--()                      // prefix --
{
        if (curr == c->begin()) throw out_of_bounds();
        --curr;
        return *this;
}
```

```
    Checked_iter operator--(int)                          // postfix --
    {
            Checked_iter tmp = *this;
            --*this;                                      // prefix -- által ellenőrzött
            return tmp;
    }

    difference_type index() const { return curr-c.begin(); } // csak közvetlen elérés esetén

    Iter unchecked() const { return curr; }

    // +, -, < stb. (§19.6[6])
};
```

Egy *Checked_iter* objektumot mindig egy bizonyos tároló egy bizonyos bejárójához kötünk. Egy alaposabb megvalósításban a *valid()* függvénynek hatékonyabb változatát kell létrehoznunk a közvetlen elérésű bejárókhoz (§19.6[6]). Miután kezdőértéket adtunk a *Checked_iter* objektumnak, minden műveletet ellenőrzünk, amely elmozdíthatja a bejárót, így mindig biztosak lehetünk abban, hogy az a tároló egy létező elemére mutat. Ha megpróbálunk egy ilyen bejárót a tároló érvényes tartományán kívülre vinni, *out_of_bounds* kivételt kapunk:

```
void f(list<string>& ls)
{
    int count = 0;
    try {
            Checked_iter< list<string> > p(ls,ls.begin());
            while (true) {
                    ++p;                  // előbb-utóbb a végére ér
                    ++count;
            }
    }
    catch(out_of_bounds) {
            cout << "Túllépés a tárolón " << count << " próbálkozás után.\n";
    }
}
```

A *Checked_iter* objektum pontosan tudja, melyik tárolóba mutat. Ez lehetővé teszi, hogy az olyan eseteket kezeljük, amikor egy tárolóba mutató bejáró érvénytelenné válik valamilyen művelet hatására (§16.3.8). Még így sem veszünk minden lehetőséget figyelembe, tehát ha az összes hibalehetőségre fel akarunk készülni, akkor egy másik, bonyolultabb bejárót kell készítenünk (lásd §19.6[7]).

Figyeljük meg, hogy az utótagként használt ++ operátor megvalósításához szükség van egy ideiglenes tárolóelemre, míg az előtag-formánál erre nincs szükség. Ezért bejárók esetében mindig érdemes a *p++* forma helyett a *++p* alakot használni, ha az ellenkezőjére nincs különösebb okunk.

Mivel a *Checked_iter* egy tárolóra hivatkozó mutatót tárol, közvetlenül nem használható a beépített tömbök kezeléséhez. Ha mégis ilyesmire van szükségünk, a *c_array* típust (§17.5.4) használhatjuk.

Ahhoz, hogy az ellenőrzött bejárók megvalósítását teljessé tegyük, használatukat egyszerűvé kell tennünk. Két megközelítéssel próbálkozhatunk:

1. Meghatározunk egy ellenőrzött tároló típust, amely ugyanúgy viselkedik, mint az átlagos tárolók, attól eltekintve, hogy kevesebb konstruktor áll benne rendelkezésünkre és a *begin()*, *end()* stb. eljárások mind *Check_iter* objektumot adnak vissza a szokásos bejárók helyett.
2. Készítünk egy kezelőosztályt, amelynek valamilyen tárolóval adhatunk kezdőértéket, és amely ehhez a tárolóhoz a későbbiekben csak ellenőrzött hozzáféréseket engedélyez (§19.6[8]).

Az alábbi sablon egy ellenőrzött bejárót köt egy tárolóhoz:

```
template<class C> class Checked : public C {
public:
   explicit Checked(size_t n) :C(n) { }
   Checked() :C() { }

   typedef Checked_iter<C> iterator;
   typedef Checked_iter<C,C::const_iterator> const_iterator;

   iterator begin() { return iterator(*this,C::begin()); }
   iterator end() { return iterator(*this,C::end()); }

   const_iterator begin() const { return const_iterator(*this,C::begin()); }
   const_iterator end() const { return const_iterator(*this,C::end()); }

   reference_type operator[ ](size_t n) { return Checked_iter<C>(*this)[n]; }

   C& base() { return static_cast<C&>(*this); }   // rögzítés az alaptárolóhoz
};
```

Ezután használhatjuk az alábbi programrészletet:

```
Checked< vector<int> > vec(10);
Checked< list<double> > lst;

void f()
{
  int v1 = vec[5];              // rendben
  int v2 = vec[15];             // out_of_bounds kivételt vált ki
  // ...
  lst.push_back(v2);
  mysort(vec.begin(),vec.end());
  copy(vec.begin(),vec.end(),lst.begin());
}
```

A látszólag felesleges *base()* függvény szerepe az, hogy a *Checked()* felületét a tárolók kezelőinek (handle) felületéhez igazítsa, ezek ugyanis általában nem tesznek lehetővé automatikus konverziókat.

Ha a tároló mérete megváltozik, bejárói érvénytelenné válnak. Ez történik a *Checked_iter* objektumokkal is. Ebben az esetben a *Checked_iter*-nek a következő kódrészlettel adhatunk új kezdőértéket:

```
void g(vector<int>& vi)
{
  Checked_iter<int> p(vi,vi.begin());
  // ..
  int i = p.index();                          // aktuális pozíció lekérése
  vi.resize(100);                             // p érvénytelen lesz
  p = Checked_iter<int>(vi,vi.begin()+i);     // az aktuális pozíció visszaállítása
}
```

A régi „aktuális pozíció” érvénytelenné válik, ezért szükségünk van az *index()* függvényre, amely egy *Checked_iter* tárolására és visszaállítására használható.

19.3.1. Kivételek, tárolók és algoritmusok

Könnyen úgy tűnhet, hogy a szabványos algoritmusok és az ellenőrzött bejárók együttes használata olyan felesleges, mint az öv és a nadrágtartó együttes viselése: mindkettő önmagában is megvéd minket a „balesetektől”. Ennek ellenére a tapasztalat azt mutatja, hogy sok ember és sok alkalmazás számára indokolt ilyen szintű paranoia, különösen akkor, ha egy gyakran változó programot sokan használnak rendszeresen.

A futási idejű ellenőrzések használatának egyik módja, hogy csak a tesztelés idejére hagyjuk azokat programunkban. Tehát ezek az ellenőrzések eltűnnek a programból, mielőtt az „éles" körülmények közé kerülne. Ez az eljárás ahhoz hasonlítható, mint amikor a mentőmellényt addig hordjuk, amíg partközelben pancsolunk, és levesszük, amikor a nyílt tengerre merészkedünk. Ugyanakkor igaz, hogy a futási idejű ellenőrzések jelentős mennyiségű időt és memóriát igényelhetnek, így folyamatosan ilyen felügyeletet biztosítani nem lehet valós elvárás. Jelentéktelen haszon érdekében optimalizálni mindig felesleges, tehát mielőtt véglegesen törlünk bizonyos ellenőrzéseket, próbáljuk ki, hogy jelentős mértékű teljesítményjavulást érünk-e el így. Ahhoz, hogy a programozó tényleg próbálkozhasson ilyesmivel, könnyűvé kell tennünk számára a futási idejű ellenőrzések eltávolítását (§24.3.7.1). Ha elvégeztük a szükséges méréseket, a futási idejű ellenőrzéseket a létfontosságú – és remélhetőleg a legalaposabban tesztelt – helyekről törölhetjük, míg máshol az ellenőrzést egy viszonylag olcsó biztosításnak tekinthetjük.

A *Checked_iter* használata számtalan hibára felhívhatja figyelmünket, de nem sokat segít abban, hogy ezeket a hibákat kijavítsuk. A programozók ritkán írnak olyan programokat, amelyek minden lehetőségre felkészülnek és ellenőriznek minden műveletet, amely kivételt okozhat (++, --, *, *[]*, -> és =). Így két nyilvánvaló stratégia közül választhatunk:

1. A kivételeket keletkezési helyük közelében kapjuk el, így a kivételkezelő megírója pontosabban megállapíthatja a hiba okát és hatékony ellenlépéseket tehet.
2. A kivételeket a programban viszonylag magas szinten kapjuk el, az eddigi számításoknak egy jelentős részét eldobjuk, és minden adatszerkezetet gyanúsnak minősítünk, amelybe írás történt a hibát kiváltó számítás közben. (Lehet, hogy ilyen adatszerkezetek nincsenek is, esetleg könnyen kizárhatók a hibaforrások köréből.)

Felelőtlenség elkapni egy kivételt, amelyről azt sem tudjuk, hogy a program mely részén keletkezett, és továbblépni azzal a feltételezéssel, hogy egyetlen adatszerkezetünk sem került nemkívánatos állapotba, hacsak nincs egy további hibakezelő, amely az ezután keletkező hibákat feldolgozza. Egy egyszerű példa az ilyen helyzetre, amikor az eredmények továbbadása előtt egy végső ellenőrzést végzünk (akár számítógépről, akár emberi munkavégzésről van szó). Ilyenkor egyszerűbb és olcsóbb ártatlanul továbbhaladni, minthogy alacsony szinten minden hibalehetőséget megvizsgáljunk. Ezzel egyben arra is példát mutattunk, hogy a többszintű hibakezelés (§14.9) hogyan teszi lehetővé a programok egyszerűsítését.

19.4. Memóriafoglalók

A *memóriafoglalók* (allokátorok, allocator) szerepe az, hogy a fizikai memória kezelésének gondját levegyék azon algoritmusok és tárolók készítőinek válláról, melyek számára memóriát kell lefoglalnunk. A memóriafoglalók szabványos felületet adnak a memóriaterületek lefoglalásához és felszabadításához, valamint szabványos neveket adnak azoknak a típusoknak, melyeket mutatóként vagy referenciaként használhatunk. A bejáróhoz hasonlóan a memóriafoglaló fogalma is tisztán elvont, így mindent memóriafoglalónak nevezünk, ami memóriafoglalóként viselkedik.

A standard könyvtár egy szabványos memóriafoglalót biztosít, amely a legtöbb felhasználó számára megfelelő, de ha szükség van rá, a programozók maguk is létrehozhatnak egyéni változatokat, melyekkel a memóriát más szerkezetűnek tüntethetik fel. Készíthetünk például olyan memóriafoglalót, amely osztott memóriát használ, szemétgyűjtő algoritmust valósít meg, előre lefoglalt memóriaszeletből hozza létre az objektumokat (§19.4.2) és így tovább.

A szabványos tárolók és algoritmusok a működésükhöz szükséges memóriát mindig memóriafoglaló segítségével foglalják le és érik el. Így ha új memóriafoglalót készítünk, a szabványos tárolókat is felruházzuk azzal a képességgel, hogy a memóriát megváltozott szemszögből lássák.

19.4.1. A szabványos memóriafoglaló

A szabványos *allocator* sablon a *<memory>* fejállományban található és a memóriafoglalást a *new()* operátor (§6.2.6) segítségével végzi. Alapértelmezés szerint mindegyik szabványos tároló ezt használja:

```
template <class T> class std::allocator {
public:
   typedef T value_type;
   typedef size_t size_type;
   typedef ptrdiff_t difference_type;

   typedef T* pointer;
   typedef const T* const_pointer;

   typedef T& reference;
   typedef const T& const_reference;

   pointer address(reference r) const { return &r; }
```

```
    const_pointer address(const_reference r) const { return &r; }

    allocator() throw();
    template <class U> allocator(const allocator<U>&) throw();
    ~allocator() throw();

    pointer allocate(size_type n, allocator<void>::const_pointer hint = 0);    // hely n darab
                                                                               // Ts számára
    void deallocate(pointer p, size_type n);        // n darab Ts helyének felszabadítása,
                                                    // megsemmisítés nélkül

    void construct(pointer p, const T& val) { new(p) T(val); }        // *p feltöltése val-lal
    void destroy(pointer p) { p->~T(); }            // *p megsemmisítése a hely felszabadítása
                                                    // nélkül

    size_type max_size() const throw();

    template <class U>
    struct rebind { typedef allocator<U> other; };  // valójában typedef allocator<U> other
};
template<class T> bool operator==(const allocator<T>&, const allocator<T>&) throw();
template<class T> bool operator!=(const allocator<T>&, const allocator<T>&) throw();
```

Az *allocate(n)* művelettel *n* objektum számára foglalhatunk helyet, párja pedig a *deallocate(p,n)*, mellyel felszabadíthatjuk az így lefoglalt területet. Figyeljük meg, hogy a *deallocate()* függvénynek is átadjuk az *n* paramétert. Ezzel a megoldással közel optimális memóriafoglalókat készíthetünk, amelyek csak a feltétlenül szükséges információkat tárolják a lefoglalt területről. Másrészt viszont ezek a memóriafoglalók megkövetelik, hogy a programozó mindig a megfelelő *n* értéket adja meg a *deallocate()* hívásakor. Megjegyzendő, hogy a *deallocate()* különbözik az *operator delete()*-től, amennyiben mutató paramétere nem lehet nulla.

Az alapértelmezett *allocator* a *new(size_t)* operátort használja a memória lefoglalásához és a *delete(void*)* műveletet a felszabadításához. Ebből következik, hogy szükség lehet a *new_handler()* meghívására és a *std::bad_alloc* kivétel kiváltására, ha elfogyott a memória (§6.2.6.2).

Jegyezzük meg, hogy az *allocate()* függvénynek nem kell feltétlenül meghívnia egy alacsonyszintű memóriafoglalót. Gyakran jobb megoldást jelent egy memóriafoglaló számára, ha ő maga tartja nyilván a kiosztásra kész, szabad memóriaszeleteket, és ezeket minimális időveszteséggel adja ki (§19.4.2).

A nem kötelező *hint* paraméter az *allocate()* függvényben mindig az adott megvalósítástól függ. Mindenképpen arra szolgál, hogy segítse a memóriafoglalót azokban a rendszerek-

ben, ahol fontos a lokalitás (vagyis az adott rendszerhez igazodás). Egy memóriafoglalótól például elvárhatjuk, hogy egy lapozós rendszerben az összefüggő objektumoknak azonos lapon foglaljon helyet. A *hint* paraméter típusa *pointer*, az alábbi erőteljesen egyszerűsített specializációnak megfelelően:

```
template <> class allocator<void> {
public:
   typedef void* pointer;
   typedef const void* const_pointer;
   // megj.: nincs referencia
   typedef void value_type;
   template <class U>
   struct rebind { typedef allocator<U> other; }; // valójában typedef allocator<U> other
};
```

Az *allocator<void>::pointer* típus általános mutatótípusként szolgál és minden szabványos memóriafoglaló esetében a *const void** típusnak felel meg.

Ha a memóriafoglaló dokumentációja mást nem mond, a programozó két lehetőség közül választhat az *allocate()* függvény használatakor:

1. Nem adja meg a *hint* paramétert.
2. A *hint* paraméterben egy olyan objektumra hivatkozó mutatót ad meg, amelyet gyakran használ majd az új objektummal együtt. Egy sorozatban például megadhatjuk az előző elem címét.

A memóriafoglalók arra szolgálnak, hogy megkíméljék a tárolók készítőit a fizikai memória közvetlen kezelésétől. Példaképpen vizsgáljuk meg, hogy egy *vector* megvalósításában hogyan használjuk a memóriát:

```
template <class T, class A = allocator<T> > class vector {
public:
   typedef typename A::pointer iterator;
   // ...
private:
   A alloc;          // memóriafoglaló objektum
   iterator v;       // mutató az elemekre
   // ...

public:
   explicit vector(size_type n, const T& val = T(), const A& a = A())
            : alloc(a)
   {
```

```
		v = alloc.allocate(n);
		for(iterator p = v; p<v+n; ++p) alloc.construct(p,val);
		// ...
	}

	void reserve(size_type n)
	{
		if (n<=capacity()) return;

		iterator p = alloc.allocate(n);
		iterator q = v;

		while (q<v+size()) {				// létező elemek másolása
			alloc.construct(p++,*q);
			alloc.destroy(q++);
		}
		alloc.deallocate(v,capacity());			// régi hely felszabadítása
		v = p-size();
		// ...
	}

	// ...
};
```

A memóriafoglalók műveleteit a *pointer* és *reference typedef*-ek segítségével fejezzük ki, így megadjuk a programozónak azt a lehetőséget, hogy más típusokat használjon a memória eléréséhez. Ezt általában nagyon nehéz megoldani. A C++ nyelv segítségével például nincs lehetőségünk arra, hogy egy tökéletes hivatkozástípust határozzunk meg. A nyelv és a könyvtárak alkotói viszont ezeket a *typedef*-eket olyan típusok létrehozásához használhatják, amelyeket egy átlagos felhasználó nem tudna elkészíteni. Példaként egy olyan memóriafoglalót említhetünk, amely állandó adatterület elérésére szolgál, de gondolhatunk egy „nagy" mutatóra is, amellyel a memóriának azt a területét is elérhetjük, amit a szokásos (általában 32 bites) mutatók már nem képesek megcímezni.

Egy átlagos felhasználó a memóriafoglalónak egyedi célra megadhat a szokásostól eltérő mutatótípust is. Ugyanezt nem tehetjük meg referenciákkal, de ez a korlátozás kísérletezésnél vagy egyedi rendszerekben nem okoz nagy gondot.

Memóriafoglalót azért hozunk létre, hogy könnyen kezelhessünk olyan objektumokat, melyeknek típusát a memóriafoglaló sablonparamétereként adtuk meg. A legtöbb tároló megvalósításában azonban szükség van más típusú objektumok létrehozására is. Például a *list* megvalósításához szükség van *Link* objektumok létrehozására. Az ilyen *Link* jellegű objektumokat a megfelelő *list* osztály memóriafoglalójával hozzuk létre.

A furcsa *rebind* típus arra szolgál, hogy memóriafoglaló képes legyen bármilyen típusú objektum helyének lefoglalására:

```
typedef typename A::template rebind<Link>::other Link_alloc;        // "sablon", lásd §C.13.6
```

Ha A egy *allocator*, akkor a *rebind<Link>::other* típus-meghatározás jelentése *allocator<Link>*, tehát a fenti *typedef* egy közvetett megfogalmazása az alábbinak:

```
typedef allocator<Link> Link_alloc;
```

A közvetett megfogalmazás azonban megkímél minket attól, hogy az *allocator* kulcsszót közvetlenül használjuk. A *Link_alloc* típust csak az A sablonparaméteren keresztül határozzuk meg. Például:

```
template <class T, class A = allocator<T> > class list {
private:
   class Link { /* ... */ };

   typedef typename A::rebind<Link>::other Link_alloc;    // allocator<Link>

   Link_alloc a;                                          // "link" memóriafoglaló
   A alloc;              // "list" memóriafoglaló
   // ...
public:
   typedef typename A::pointer iterator;
   // ...

   iterator insert(iterator pos, const T& x )
   {
            Link_alloc::pointer p = a.allocate(1); // egy Link megszerzése
            // ...
   }
   // ...
};
```

Mivel a *Link* a *list* osztály tagja, paramétere egy memóriafoglaló. Ennek következében a különböző memóriafoglalókkal rendelkező listák *Link* objektumai különböző típusúak, ugyanúgy, ahogy maguk a listák is különböznek (§17.3.3).

19.4.2. Felhasználói memóriafoglalók

A tároló készítőjének gyakran van szüksége arra, hogy egyesével hozzon létre (*allocate()*) vagy semmisítsen meg (*deallocate()*) objektumokat. Ha az *allocate()* függvényt meggondolatlanul készítjük el, nagyon sokszor kell meghívnunk a *new* operátort, pedig a *new* operátor ilyen használatra gyakran kis hatékonyságú. A felhasználói memóriafoglalók példájaként az alábbiakban olyan módszerrel élünk, mellyel egy „készletet" hozunk létre, amely rögzített méretű memóriaszeleteket tárol. A memóriafoglaló ennek felhasználásával hatékonyabban hajtja végre az *allocate()* eljárást, mint a hagyományos (bár általánosabb) *new()* operátor.

Véletlenül én már korábban is létrehoztam egy ilyen készletes memóriafoglalót, amely körülbelül azt csinálja, amire most szükségünk van, de nem a megfelelő felületet nyújtja (hiszen évekkel azelőtt készült, hogy a memóriafoglalók ötlete megszületett volna). Ez a *Pool* osztály meghatározza a rögzített méretű elemek készletét, amelyből a programozó gyorsan foglalhat le területeket és gyorsan fel is szabadíthatja azokat. Ez egy alacsonyszintű típus, amely közvetlenül a memóriát kezeli és az igazítási (alignment) problémákkal is foglalkozik:

```
class Pool {
  struct Link { Link* next; };

  struct Chunk {
          enum { size = 8*1024-16 };          // valamivel kevesebb 8K-nál, így egy
                                              // "Chunk" belefér 8K-ba
          char mem[size];                     // először a lefoglalandó terület az igazítás
                                              // miatt
          Chunk* next;
  };
  Chunk* chunks;

  const unsigned int esize;
  Link* head;

  Pool(Pool&);                    // másolásvédelem
  void operator=(Pool&);          // másolásvédelem
  void grow();                    // a készlet növelése
public:
  Pool(unsigned int n);           // n az elemek mérete
  ~Pool();

  void* alloc();                  // hely foglalása egy elem számára
  void free(void* b);             // elem visszahelyezése a készletbe
};
```

```
inline void* Pool::alloc()
{
  if (head==0) grow();
  Link* p = head;                    // az első elem visszaadása
  head = p->next;
  return p;
}

inline void Pool::free(void* b)
{
  Link* p = static_cast<Link*>(b);
  p->next = head;                    // b visszahelyezése első elemként
  head = p;
}

Pool::Pool(unsigned int sz)
  : esize(sz<sizeof(Link)?sizeof(Link):sz)
{
  head = 0;
  chunks = 0;
}

Pool::~Pool()                        // minden 'chunk' felszabadítása
{
  Chunk* n = chunks;
  while (n) {
          Chunk* p = n;
          n = n->next;
          delete p;
  }
}

void Pool::grow()     // hely foglalása új 'chunk' számára, melyet 'esize' méretű elemek
                      // láncolt listájaként rendezünk
{
  Chunk* n = new Chunk;
  n->next = chunks;
  chunks = n;

  const int nelem = Chunk::size/esize;
  char* start = n->mem;
  char* last = &start[(nelem-1)*esize];

  for (char* p = start; p<last; p+=esize)
          reinterpret_cast<Link*>(p)->next = reinterpret_cast<Link*>(p+esize);
  reinterpret_cast<Link*>(last)->next = 0;
  head = reinterpret_cast<Link*>(start);
}
```

A valós életben zajló munka szemléltetésére a *Pool* osztályt változatlan formában használjuk fel az új memóriafoglaló megvalósításához, nem csak átírjuk, hogy a nekünk megfelelő felületet nyújtsa. A készletes memóriafoglaló célja, hogy objektumaink egyesével történő létrehozása és megsemmisítése gyors legyen. A *Pool* osztály mindezt biztosítja. A megvalósítást azzal kell még kiegészítenünk, hogy egyszerre tetszőleges számú, illetve (a *rebind()* igényeinek megfelelően) tetszőleges méretű objektumokat is létrehozhassunk. Ezt a két problémát feladatnak hagyjuk (§19.6[9]).

A *Pool* osztály felhasználásával a *Pool_alloc* megvalósítása már egyszerű:

```
template <class T> class Pool_alloc {
private:
   static Pool mem;   // elemkészlet sizeof(T) mérettel
public:
   // mint a szabványos allocator (§19.4.1)
};

template <class T> Pool Pool_alloc<T>::mem(sizeof(T));

template <class T> Pool_alloc<T>::Pool_alloc() { }

template <class T>
T* Pool_alloc<T>::allocate(size_type n, void* = 0)
{
   if (n == 1) return static_cast<T*>(mem.alloc());
   // ...
}

template <class T>
void Pool_alloc<T>::deallocate(pointer p, size_type n)
{
   if (n == 1) {
            mem.free(p);
            return;
   }
   // ...
}
```

A memóriafoglalót ezután már a megszokott formában használhatjuk:

```
vector< int,Pool_alloc<int> > v;
map<string,number,Pool_alloc< pair<const string,number> > > m;

// ugyanúgy mint szoktuk

vector<int> v2 = v;              // hiba: különböző memóriafoglaló-paraméterek
```

A *Pool_Alloc* megvalósításához statikus *Pool* objektumot használunk. Azért döntöttem így, mert a standard könyvtár a memóriafoglalókra egy korlátozást kényszerít, mégpedig azzal, hogy a szabványos tárolók megvalósításának megengedi, hogy minden objektumot egyenértékűnek tekintsenek, amelynek típusa az adott tároló memóriafoglalója. Ez valójában igen jelentős hatékonysági előnyöket jelent: ennek a korlátozásnak köszönhetően a *Link* objektumok memóriafoglalóinak például nem kell külön memóriaterületet félretennünk (annak ellenére, hogy a *Link* osztály általában paraméterként egy memóriafoglalót kap ahhoz a tárolóhoz, amelyben szerepel, §19.4.1). Egy másik előny, hogy azokban a műveletekben, ahol két sorozat elemeit kell elérnünk (például a *swap()* függvényben), nem kell megvizsgálnunk, hogy a használt elemek ugyanolyan memóriafoglalókkal rendelkeznek-e. A hátrány, hogy a korlátozás következtében az ilyen memóriafoglalók nem használhatnak objektumszintű adatokat.

Mielőtt ilyen optimalizációt használnánk, gondoljuk végig, hogy szükség van-e rá. Én remélem, hogy az alapértelmezett *allocator* legtöbb megvalósításában elvégzik ezt a klasszikus C++ optimalizációt, így megkímélnek minket ettől a problémától.

19.4.3. Általánosított memóriafoglalók

Az *allocator* valójában nem más, mint annak az ötletnek az egyszerűsített és optimalizált változata, miszerint egy tárolónak egy sablonparaméteren keresztül adjuk meg az információkat (§13.4.1., §16.2.3). Logikus elvárás például, hogy a tároló minden elemének helyét a tároló memóriafoglalójával foglaljuk le. Ha azonban ilyenkor lehetővé tesszük, hogy két ugyanolyan típusú *list* tárolónak különböző memóriafoglalója legyen, akkor a *splice()* (§17.2.2.1) nem valósítható meg egyszerű átláncolással, hanem szabályos másolást kell meghatároznunk, mert védekeznünk kell azon (ritka) esetek ellen, amikor a két listában a memóriafoglalók nem azonosak, annak ellenére, hogy típusuk megegyezik. Hasonló probléma, hogy ha a memóriafoglalók tökéletesen általánosak, akkor a *rebind()* eljárásnak (amely tetszőleges típusú elemek létrehozását teszi lehetővé a memóriafoglaló számára) sokkal bonyolultabbnak kell lennie. Ezért a „normális” memóriafoglalókról azt feltételezzük, hogy nem tárolnak objektumszintű adatokat, az algoritmusok pedig kihasználhatják ezt.

Meglepő módon ez a drákói korlátozás az objektumszintű információkra nézve nem túlságosan veszélyes a memóriafoglalók esetében. A legtöbb memóriafoglalónak nincs is szüksége objektumszintű adatokra, sőt ilyen adatok nélkül még gyorsabbak is lehetnek, ráadásul a memóriafoglalók azért tudnak adatokat tárolni, a memóriafoglaló-típusok szintjén. Ha külön adatelemekre van szükség, különböző memóriafoglaló-típusokat használhatunk:

```
template<class T, class D> class My_alloc {       // T memóriafoglalóját D használatával
                                                   // hozzuk létre
  D d;      // a My_alloc<T,D> számára szükséges adatok
  // ...
};

typedef My_alloc<int,Persistent_info> Persistent;
typedef My_alloc<int,Shared_info> Shared;
typedef My_alloc<int,Default_info> Default;

list<int,Persistent> lst1;
list<int,Shared> lst2;
list<int,Default> lst3;
```

Az *lst1*, az *lst2* és az *lst3* listák különböző típusúak, így ha közülük kettőt akarunk felhasználni valamilyen műveletben, akkor általános algoritmusokat kell használnunk (18.fejezet), nem pedig specializált lista műveleteket (§17.2.2.1). Ebből következik, hogy az átláncolás helyett másolást kell használnunk, így a különböző memóriafoglalók használata nem okoz problémát.

Az objektumszintű adatokra vonatkozó korlátozásokra a memóriafoglalókban azért van szükség, mert így tudjuk kielégíteni a standard könyvtár szigorú hatékonysági követelményeit, mind futási idő, mind tárhasználat szempontjából. Egy lista memóriafoglalója például nem foglal jelentősen több memóriát a szükségesnél, de ha minden listaelemnél jelentkezne egy kis „felesleg”, akkor ez jelentős veszteséget okozna.

Gondoljuk végig, hogyan használhatnánk a memóriafoglalókat akkor, ha a standard könyvtár hatékonysági követelményeitől eltekintenénk. Ez a helyzet olyan nem szabványos könyvtár esetében fordulhat elő, melyben nem volt fontos tervezési szempont a nagy hatékonyság az adatszerkezetek és típusok létrehozásakor, vagy akár a standard könyvtár bizonyos egyedi célú megvalósításaiban. Ilyenkor a memóriafoglaló felhasználható olyan jellegű információk tárolására is, melyek általában általános bázisosztályokban kapnak helyet (§16.2.2). Készíthetünk például olyan memóriafoglalókat, melyek választ tudnak adni arra a kérdésre, hogy az objektumok hol kaptak helyet, kínálhatnak olyan információkat, melyek az objektumok elrendezésére vonatkoznak, vagy megkérdezhetjük tőlük, hogy egy adott elem benne van-e a tárolóban. Segítségükkel elkészíthető egy olyan tárolófelügyelő is, amely gyorsítótárként szolgál a háttértárhoz vagy kapcsolatokat biztosít a tároló és más objektumok között és így tovább.

Ezzel a módszerrel tehát tetszőleges szolgáltatásokat biztosíthatunk a szokásos tárolóműveletek hátterében. Ennek ellenére érdemes különbséget tennünk az adatok táro-

lásának és az adatok felhasználásának feladatai között. Ez utóbbi nem tartozik egy általánosított memóriafoglaló hatáskörébe, így inkább egy külön sablonparaméter segítségével illik megvalósítanunk.

19.4.4. Előkészítetlen memória

A szabványos *allocator* mellett a *<memory>* fejállományban találhatunk néhány olyan függvényt is, melyek az előkészítetlen (értékkel nem feltöltött) memória problémáival foglalkoznak. Azt a veszélyes, de gyakran nagyon fontos lehetőséget biztosítják, hogy a *T* típusnévvel hivatkozhatunk egy olyan memóriaterületre, amely elég nagy egy *T* típusú objektum tárolására, de nem teljesen szabályosan létrehozott *T* objektumot tartalmaz.

A könyvtárban három függvényt találunk, mellyel előkészítetlen területre értékeket másolhatunk:

```
template <class In, class For>
For uninitialized_copy(In first, In last, For res)              // másolás res-be
{
  typedef typename iterator_traits<For>::value_type V;

  while (first != last)
           new (static_cast<void*>(&*res++)) V(*first++);  // létrehozás res-ben (§10.4.11)
  return res;
}

template <class For, class T>
void uninitialized_fill(For first, For last, const T& val)       //másolás [first,last)-ba
{
  typedef typename iterator_traits<For>::value_type V;

  while (first != last) new (static_cast<void*>(&*first++)) V(val);   // létrehozás first-ben
}

template <class For, class Size, class T>
void uninitialized_fill_n(For first, Size n, const T& val)            //másolás [first,first+n)-be
{
  typedef typename iterator_traits<For>::value_type V;

  while (n--) new (static_cast<void*>(&*first++)) V(val);            // létrehozás first-ben
}
```

Ezek a függvények elsősorban tárolók és algoritmusok készítésénél hasznosak. A *reserve()* és a *resize()* (§16.3.8) például legkönnyebben ezen függvények felhasználásával valósítha-

tó meg (§19.6[10]). Igen nagy probléma, ha egy ilyen kezdőértékkel nem rendelkező objektum valahogy kiszabadul a tároló belső megvalósításából és átlagos felhasználók kezébe kerül (lásd még §E.4.4).

Az algoritmusoknak gyakran van szükségük ideiglenes tárterületre feladataik helyes végrehajtásához. Ezeket a területeket gyakran érdemes egyetlen művelettel lefoglalni, annak ellenére, hogy értékkel feltölteni csak akkor fogjuk azokat, amikor ténylegesen szükség lesz rájuk. Ezért a könyvtár tartalmaz egy függvénypárt, mellyel előkészítetlen memóriaterületet foglalhatunk le, illetve szabadíthatunk fel:

```
template <class T> pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t);   // lefoglalás
                                                                          // kezdeti értékadás
                                                                          // nélkül
template <class T> void return_temporary_buffer(T*);                     // felszabadítás
                                                                          // megsemmisítés
                                                                          // nélkül
```

A *get_temporary_buffer<X>(n)* művelet megpróbál helyet foglalni *n* vagy több *X* típusú objektum számára. Ha ez sikerül, akkor az első kezdőérték nélküli objektumra hivatkozó mutatót és a lefoglalt területen elférő, *X* típusú objektumok számát adja vissza. Ha nincs elég memória, a visszaadott pár második (*second*) tagja nulla lesz. Az ötlet az, hogy a rendszer a gyors helyfoglalás érdekében rögzített méretű átmeneti tárat tart készenlétben. Ennek következtében előfordulhat, hogy *n* objektum számára igénylünk területet, de az átmeneti tárban ennél több is elfér. A gond az, hogy az is elképzelhető, hogy kevesebb területet kapunk az igényeltnél, így a *get_temporary_buffer()* felhasználási módja az, hogy kellően nagy tárat igénylünk, aztán ebből annyit használunk fel, amennyit megkaptunk. A *get_temporary_buffer()* által lefoglalt területet fel kell szabadítanunk a *return_temporary_buffer()* függvény segítségével. Ugyanúgy, ahogy a *get_temporary_buffer()* a konstruktor meghívása nélkül foglal le területet, a *return_temporary_buffer()* a destruktor meghívása nélkül szabadít fel. Mivel a *get_temporary_buffer()* alacsonyszintű művelet és kifejezetten ideiglenes tárak lefoglalására szolgál, nem használhatjuk a *new* vagy az *allocator::allocate()* helyett, hosszabb életű objektumok létrehozására.

Azok a szabványos algoritmusok, melyek egy sorozatba írnak, feltételezik, hogy a sorozat elemei korábban már kaptak kezdőértéket. Tehát az algoritmusok az íráshoz egyszerű értékadást használnak és nem másoló konstruktort. Ennek következtében viszont nem használhatunk előkészítetlen területet egy algoritmus kimenete céljára. Ez sokszor igen kellemetlen, mert az egyszerű értékadás jóval „költségesebb”, mint a kezdeti, ráadásul nem is érdekel minket, milyen értékeket fogunk felülírni (ha érdekelne, nem írnánk felül). A megoldást a *raw_storage_iterator* jelenti, amely szintén a *<memory>* fejállományban található és egyszerű helyett kezdeti értékadást végez:

```
template <class Out, class T>
class raw_storage_iterator : public iterator<output_iterator_tag,void,void,void,void> {
  Out p;
public:
  explicit raw_storage_iterator(Out pp) : p(pp) { }
  raw_storage_iterator& operator*() { return *this; }
  raw_storage_iterator& operator=(const T& val) {
          T* pp = &*p;
          new(pp) T(val);     // val pp-be helyezése (§10.4.11)
          return *this;
  }
  raw_storage_iterator& operator++() {++p; return *this; }
  raw_storage_iterator operator++(int) {
          raw_storage_iterator t = *this;
          ++p;
          return t;
  }
};
```

Például készíthetünk egy sablon függvényt, amely egy *vector* elemeit egy átmeneti tárba másolja:

```
template<class T, class A> T* temporary_dup(vector<T,A>& v)
{
     pair<T*,ptrdiff_t> p = get_temporary_buffer<T>(v.size());
     if (p.second < v.size()) {           // ellenőrizzük, hogy elég volt-e az elérhető memória
          if (p.first != 0) return_temporary_buffer(p.first);
          return 0;
     }
     copy(v.begin(),v.end(),raw_storage_iterator<T*,T>(p.first));
     return p.first;
}
```

Ha a *get_temporary_buffer()* helyett a *new* operátort használtuk volna, az átmeneti tár kezdeti feltöltésére sor került volna. Mivel kikerültük az előkészítést, a *raw_storage_iterator* osztályra van szükségünk a nem feltöltött terület kezeléséhez. Ebben a példában a *temporary_dump()* függvény meghívójának feladata marad, hogy meghívja a *return_temporary_buffer()* eljárást a visszaadott mutatóra.

19.4.5. Dinamikus memória

A <*new*> fejállományban olyan lehetőségek szerepelnek, melyekkel a *new* és *delete* operátort valósíthatjuk meg:

```
class bad_alloc : public exception { /* ... */ };

struct nothrow_t {};
extern const nothrow_t nothrow;        // kivételt ki nem váltó memóriafoglaló

typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

void* operator new(size_t) throw(bad_alloc);
void operator delete(void*) throw();

void* operator new(size_t, const nothrow_t&) throw();
void operator delete(void*, const nothrow_t&) throw();

void* operator new[ ](size_t) throw(bad_alloc);
void operator delete[ ](void*) throw();

void* operator new[ ](size_t, const nothrow_t&) throw();
void operator delete[ ](void*, const nothrow_t&) throw();

void* operator new (size_t, void* p) throw() { return p; }    // elhelyezés (§10.4.11)
void operator delete (void* p, void*) throw() { }             //nem csinál semmit

void* operator new[ ](size_t, void* p) throw() { return p; }
void operator delete[ ](void* p, void*) throw() { }           //nem csinál semmit
```

Egy üres *kivétel-specifikációval* (§14.6) definiált *operator new()* vagy *operator new[ ]()* nem jelezheti a memória elfogyását az *std::bad_alloc* kivétel kiváltásával. Ehelyett, ha sikertelen a helyfoglalás, *0* értéket adnak vissza. A *new* kifejezés (§6.2.6.2) az üres *kivétel-specifikációval* rendelkező memóriafoglalók által visszaadott értéket mindig ellenőrzi, és ha *0* értéket kap, nem hívja meg a konstruktort, hanem azonnal szintén *0* értéket ad vissza. A *nothrow* memóriafoglalók például *0* értéket adnak vissza a sikertelen helyfoglalás jelzésére, és nem egy *bad_alloc* kivételt váltanak ki:

```
void f()
{
  int* p = new int[100000];   // bad_alloc-ot válthat ki

  if (int* q = new(nothrow) int[100000]) {        // nem vált ki kivételt
        // lefoglalás sikeres
  }
```

```
    else {
        // lefoglalás sikertelen
    }
}
```

Ez a módszer lehetővé teszi, hogy kivétel előtti hibakezelést használjunk a helyfoglalás során.

19.4.6. C stílusú helyfoglalás

A C++ a C-től örökölt egy dinamikusmemória-kezelő felületet, melyet a <*cstdlib*> fejállományban találhatunk meg:

```
void* malloc(size_t s);                  // s bájt lefoglalása
void* calloc(size_t n, size_t s);        // n-szer s bájt feltöltése 0 kezdőértékkel
void free(void* p);    // a malloc() vagy calloc() által lefoglalt szabad terület felszabadítása
void* realloc(void* p, size_t s);        // a p által mutatott tömb méretének módosítása s-re;
```

Ezen függvények helyett használjuk inkább a *new* vagy a *delete* operátort, vagy a szabványos tárolók még magasabb szintű szolgáltatásait. A fenti eljárások előkészítetlen memóriával foglalkoznak, így a *free()* például nem hív meg semmilyen destruktort a memóriaterület felszabadítására. A *new* és a *delete* megvalósításai használhatják ezeket a függvényeket, de számukra sem kötelező. Ha egy objektumot például a *new* operátorral hozunk létre, majd a *free()* függvénnyel próbálunk meg megsemmisíteni, súlyos problémákba ütközhetünk. Ha a *realloc()* függvény szolgáltatásaira lenne szükségünk, használjunk inkább szabványos tárolókat, melyek az ilyen jellegű feladatokat általában sokkal egyszerűbben és legalább olyan hatékonyan hajtják végre (§16.3.5).

A könyvtár tartalmaz néhány olyan függvényt is, melyekkel hatékonyan végezhetünk bájtszintű műveleteket. Mivel a C a típus nélküli bájtokat eredetileg *char** mutatókon keresztül érte el, ezek a függvények a <*cstring*> fejállományban találhatók. Ezekben a függvényekben a *void** mutatókat *char** mutatókként kezeljük:

```
void* memcpy(void* p, const void* q, size_t n);       // nem átfedő területek másolása
void* memmove(void* p, const void* q, size_t n);      // esetleg átfedő területek másolása
```

A *strcpy()* (§20.4.1) függvényhez hasonlóan ezek a műveletek is *n* darab bájtot másolnak a *q* címről a *p* címre, majd a *p* mutatót adják vissza. A *memove()* által kezelt memóriaterületek átfedhetik egymást, de a *memcpy()* feltételezi, hogy a két terület teljesen különálló, és általában ki is használja ezt a feltételezést.

Hasonlóan működnek az alábbi függvények is:

```
void* memchr(const void* p, int b, size_t n);            // mint a strchr() (§20.4.1):
                                                         // b keresése p[0]..p[n-1]-ben
int memcmp(const void* p, const void* q, size_t n);      // mint a strcmp(): bájtsorozatok
                                                         // összehasonlítása
void* memset(void* p, int b, size_t n);                  // n bájt b-re állítása, p
                                                         // visszaadása
```

Számos C++-változatban ezeknek az eljárásoknak erősen optimalizált változatai is megtalálhatók.

19.5. Tanácsok

[1] Amikor egy algoritmust megírunk, döntsük el, milyen bejáróra van szükségünk az eljárás kellően hatékony megvalósításához, majd folyamatosan figyeljünk, hogy csak olyan műveleteket használjunk, melyek az adott bejáró-kategória esetében rendelkezésünkre állnak. §19.2.1

[2] Az algoritmusok hatékonyabb megvalósítására használjunk túlterhelést, ha a paraméterként megadott bejáró a minimális követelménynél több szolgáltatást nyújt. §19.2.3.

[3] A különböző bejáró-kategóriákra használható algoritmusok megírásához használjuk az *iterator_traits* osztályokat. §19.2.2.

[4] Ne felejtsük el használni a ++ operátort az *istream_iterator* és az *ostream_iterator* objektumok esetében sem. §19.2.6.

[5] A tárolók túlcsordulásának elkerülése érdekében használjunk beszúró (inserter) bejárókat. §19.2.4.

[6] Tesztelés alatt végezzünk minél több ellenőrzést, és a végleges változatból is csak azokat távolítsuk el, melyeket feltétlenül szükséges. §19.3.1.

[7] Ha tehetjük, használjuk a *++p* formát a *p++* helyett. §19.3.

[8] Az adatszerkezeteket bővítő algoritmusok hatékonyságát növelhetjük, ha előkészítetlen memóriaterületeket használunk. §19.4.4.

[9] Ha egy algoritmus ideiglenes adatszerkezeteket használ, a hatékonyságot ideiglenes tár (puffer) használatával növelhetjük. §19.4.4.

[10] Kétszer is gondoljuk meg, mielőtt saját memóriafoglaló írásába kezdünk. §19.4.

[11] Kerüljük a *malloc()*, a *free()*, a *realloc()* stb. függvények használatát. §19.4.6.

[12] A sablonok *typedef*-jeinek használatát a *rebind* függvénynél bemutatott módszer segítségével utánozhatjuk. §19.4.1.

19.6. Gyakorlatok

1. (*1.5) Készítsük el a §18.6.7. *reverse()* függvényét. Segítség: §19.2.3.
2. (*1.5) Készítsünk egy kimeneti bejárót, amely valójában sehova sem ír. Mikor lehet értelme egy ilyen bejárónak?
3. (*2) Írjuk meg a *reverse_iterator* osztályt (§19.2.5).
4. (*1.5) Készítsük el az *ostream_iterator* osztályt (§19.2.6).
5. (*2) Készítsük el az *istream_iterator* osztályt (§19.2.6).
6. (*2.5) Fejezzük be a *Checked_iter* osztályt (§19.3).
7. (*2.5) Alakítsuk át a *Checked_iter* osztályt úgy, hogy ellenőrizze az érvénytelenné vált bejárókat is.
8. (*2) Tervezzünk meg és készítsünk el egy olyan kezelőosztályt, amely egy tárolót képes helyettesíteni úgy, hogy annak teljes felületét biztosítja. Az ábrázolásban tárolnunk kell egy mutatót egy tárolóra, valamint meg kell írnunk a tároló műveleteit tartományellenőrzéssel.
9. (*2.5) Fejezzük be vagy írjuk újra a *Pool_alloc* (§19.4.2) osztályt úgy, hogy az a standard könyvtár *allocator* (§19.4.1) osztályának minden lehetőségét támogassa. Hasonlítsuk össze az *allocator* és a *Pool_alloc* hatékonyságát, és állapítsuk meg, hogy saját rendszerünkben szükség van-e a *Pool_alloc* használatára.
10. (*2.5) Készítsünk el egy *vector* osztályt úgy, hogy memóriafoglalókat használunk a *new* és a *delete* operátor helyett.

20

Karakterláncok

„Járt utat a járatlanért el ne hagyj!”

Karakterláncok • Karakterek • *char_traits* • *basic_string* • Bejárók • Elemek elérése • Konstruktorok • Hibakezelés • Értékadás • Koverzió • Összehasonlítás • Beszúrás • Összefűzés • Keresés és csere • Méret és kapacitás • Karakterláncok ki- és bevitele • C stílusú karakterláncok • Karakterek osztályozása • A C könyvtár függvényei • Tanácsok • Gyakorlatok

20.1. Bevezetés

A karakterlánc (string) karakterek sorozata. A standard könyvtár *string* osztálya mindazokat az eljárásokat elérhetővé teszi, amelyekre a karakterláncokkal kapcsolatban szükségünk lehet: indexelés (§20.3.3), értékadás (§20.3.6), összehasonlítás (§20.3.8), hozzáfűzés (§20.3.9), összefűzés (§20.3.10), részláncok keresése (§20.3.11). Nincs viszont általános részlánc-kezelő, ezért – a szabványos *string* használatát is bemutatandó – készítünk majd egyet (§20.3.11). Egy szabványos karakterlánc szinte bármilyen karakterek sorozata lehet (§20.2).

A tapasztalatok azt mutatják, hogy nem lehet tökéletes *string* típust megvalósítani. Ehhez az emberek ízlése, elvárásaik, igényeik túl nagy mértékben különböznek. Így a standard könyvtár *string* osztálya sem tökéletes. Bizonyos tervezési kérdésekben máshogy is dönthettem volna az osztály létrehozásakor. Ennek ellenére azt hiszem, sok elvárást kielégít és könnyen elkészíthetők azok a kiegészítő függvények, melyekkel a további feladatok megoldhatók. Nagy előnyt jelent az is, hogy az *std::string* osztály általánosan ismert és mindenhol elérhető. Ezek a jellemzők a legtöbb esetben fontosabbak, mint azok tulajdonságok, amelyekkel az osztályt esetleg még kiegészíthetnénk. A különböző karakterlánc-osztályok elkészítése gyakorlásnak nagyszerű (§11.12, §13.2), de ha széles körben használható változatot akarunk, akkor a standard könyvtár *string* osztályára lesz szükségünk.

A C++ a C-től örökölt, nullával lezárt karaktertömbként értelmezett karakterláncok (vagyis a C stílusú karakterláncok) kezelésére a standard könyvtárban számos külön függvényt biztosít (§20.4.1).

20.2. Karakterek

A „karakter" (character) már önmagában is érdekes fogalom. Figyeljük meg például a *C* karaktert. Ezt a *C* betűt, amely valójában egy egyszerű félkörív a papíron (vagy a képernyőn), hónapokkal ezelőtt gépeltem be a számítógépembe. Ott egy 8 bites bájtban a *67* számértékként tárolódott. Elmondhatjuk róla, hogy a latin ábécé harmadik betűje, a hatodik atom (a szén, carbon) szokásos rövidítése és mellesleg egy programozási nyelv neve is (§1.6). A karakterláncok programokban való felhasználásakor csak az számít, hogy e kacskaringós alakzathoz kapcsolódik egy hagyományos jelentés, amit karakternek nevezünk, és egy számérték, amit a számítógép használ. Hogy bonyolítsuk a dolgokat, ugyanahhoz a karakterhez a különböző karakterkészletekben más-más számérték tartozhat, sőt, nem is minden karakterkészletben találunk számértéket minden karakterhez, ráadásul sok különböző karakterkészletet használunk rendszeresen. A karakterkészlet nem más, mint a karakterek (a hagyományos szimbólumok) egy leképezése egész értékekre.

A C++ programozók általában feltételezik, hogy a szabványos amerikai karakterkészlet (ASCII) rendelkezésünkre áll, de a C++ felkészült arra az esetre is, ha bizonyos karakterek hiányoznának a programozási környezetből. Például ha olyan karakterek hiányoznak, mint a *[* vagy a *{*, használhatunk helyettük kulcsszavakat vagy digráf – két tagból álló – jeleket (§C.3.1).

Nagy kihívást jelentenek azok a karakterkészletek is, melyekben olyan karakterek szerepelnek, amelyek az ASCII-ban nem fordulnak elő. Az olyan nyelvek karakterei, mint a kínai, a dán, a francia, az izlandi vagy a japán, nem írhatók le hibátlanul az ASCII karakterkészlet segítségével. Még nagyobb probléma, hogy az e nyelvekhez használt karakterkészletek is különböznek. A latin ábécét használó európai nyelvek karakterei például *majdnem* elférnek egy 256 karakteres karakterkészletben, de sajnos a különböző nyelvekhez különböző készleteket használunk és ezekben néha különböző karaktereknek ugyanaz az egész érték jutott. A francia karakterkészlet (amely Latin1-et használ) például nem teljesen egyeztethető össze az izlandi karakterekkel (így azok használatához a Latin2 készletre van szükségünk). Azok a nagyratörő kísérletek, melyek során megpróbáltak minden, emberek által ismert karaktert egyetlen karakterkészletben felsorolni, sok problémát megoldottak, de még a 16 bites készletek (például a Unicode) sem elégítettek ki minden igényt. A 32 bites karakterkészletek, melyek – ismereteim szerint – az összes karakter megjelölésére alkalmasak lennének, még nem terjedtek el széles körben.

A C++ alapvetően azt a megközelítést követi, hogy a programozónak megengedjük bármelyik karakterkészlet használatát a karakterláncok karaktertípusának megadásához. Használhatunk bővített karakterkészleteket és más rendszerre átültethető számkódolást is (§C.3.3)

20.2.1. Karakterjellemzők

A §13.2 pontban már bemutattuk, hogy egy karakterlánc elméletileg tetszőleges típust képes „karakterként” használni, ha az megfelelő másolási műveleteket biztosít. Csak azokat a típusokat tudjuk azonban igazán hatékonyan és egyszerűen kiaknázni, melyeknek nincs felhasználói másoló művelete. Ezért a szabványos *string* osztály megköveteli, hogy a benne karakterként használt típusnak ne legyen felhasználói másoló művelete. Ez azt is lehetővé teszi, hogy a karakterláncok ki- és bevitele egyszerű és hatékony legyen.

Egy karaktertípus jellemzőit (traits) a hozzá tartozó *char_traits* osztály írja le, amely az alábbi sablon specializációja:

```
template<class Ch> struct char_traits { };
```

Minden *char_traits* az *std* névtérben szerepel és a szabványos változatok a *<string>* fejállományból érhetők el. Az általános *char_traits* osztály egyetlen jellemzőt sem tartalmaz, azokkal csak az egyes karaktertípusokhoz készített változatok rendelkeznek. A *char_traits<char>* definíciója például a következő:

```
template<> struct char_traits<char> {          // a char_traits műveleteknek nem szabad
                                               // kivételt kiváltaniuk
  typedef char char_type;                      // a karakter típusa

  static void assign(char_type&, const char_type&);       // az = meghatározása a
                                                           // char_type számára

  // a karakterek egész értékű ábrázolása

  typedef int int_type;                        // a karakter-értékek egész típusa

  static char_type to_char_type(const int_type&);         // átalakítás int-ről char-ra
  static int_type to_int_type(const char_type&);          // átalakítás char-ról int-re
  static bool eq_int_type(const int_type&, const int_type&);        // ==

  // char_type összehasonlítások

  static bool eq(const char_type&, const char_type&);               // ==
  static bool lt(const char_type&, const char_type&);               // <

  // műveletek s[n] tömbön

  static char_type* move(char_type* s, const char_type* s2, size_t n);
  static char_type* copy(char_type* s, const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n, char_type a);

  static int compare(const char_type* s, const char_type* s2, size_t n);
  static size_t length(const char_type*);
  static const char_type* find(const char_type* s, int n, const char_type&);

  // bemeneti/kimeneti műveletek

  typedef streamoff off_type;          // eltolás az adatfolyamban
  typedef streampos pos_type;          // pozíció az adatfolyamban
  typedef mbstate_t state_type;        // több bájtos adatfolyam állapota

  static int_type eof();                       // fájl vége
  static int_type not_eof(const int_type& i);  // i, hacsak i értéke nem eof();
  static state_type get_state(pos_type p);     // p pozíción levő karakter
                                               // állapota a több bájtos átalakítás során
};
```

A szabványos karakterlánc-sablon, a *basic_string* (§20.3) megvalósítása e típusokon és függvényeken alapul. A *basic_string*-hez használt karaktertípusnak rendelkeznie kell egy olyan *char_traits* specializációval, amely mindezeket támogatja.

Ahhoz, hogy egy típus *char_type* lehessen, képes kell legyen arra, hogy minden karakterhez hozzárendeljen egy egész értéket, melynek típusa *int_type*. Az *int_type* és a *char_type* típus közötti átalakítást a *to_char_type()* és a *to_int_type()* függvény hajtja végre. A *char* típus esetében ez a átalakítás igen egyszerű.

A *move(s,s2,n)* és a *copy(s,s2,n)* függvények egyaránt az *s2* címről másolnak át *n* karaktert az *s* címre, és mindketten az *assign(s[i], s2[i])* utasítást használják. A különbség az, hogy a *move()* akkor is helyesen működik, ha *s2* az *[s,s+n[* tartományba esik, a *copy()* viszont kicsit gyorsabb. Ez a működési elv pontosan megegyezik a C standard könyvtár *memcpy()*, illetve *memmove()* (§19.4.6) függvényeinek működésével. Az *assign(s,n,x)* függvényhívás az *x* karakter *n* darab másolatát írja az *s* címre az *assign(s[i],x)* utasítás segítségével.

A *compare()* függvény a *lt()*, illetve az *eq()* eljárásokat használja a karakterek összehasonlításához. Visszatérési értéke egy *int*, amely *0*, ha a két karakterlánc pontosan megegyezik; negatív szám, ha az első paraméter ábécésorrendben előbb következik, mint a második; fordított esetben pozitív szám. A visszatérési érték ilyen használata a C standard könyvtár *strcmp()* függvényének működését követi (§20.4.1).

A ki- és bemenethez kapcsolódó függvényeket az alacsonyszintű I/O műveletek használják (§21.6.4).

A széles karakterek (amelyek a *wchar_t* osztály példányai, §4.3) nagyon hasonlítanak az egyszerű *char* típushoz, de kettő vagy még több bájtot használnak. A *wchar_t* típus tulajdonságait a *char_traits<wchar_t>* osztály írja le:

```
template<> struct char_traits<wchar_t> {
  typedef wchar_t char_type;
  typedef wint_t int_type;
  typedef wstreamoff off_type;
  typedef wstreampos pos_type;

  // mint a char_traits<char>
};
```

A *wchar_t* típust elsősorban a 16 bites karakterkészletek (például a Unicode) karaktereinek tárolásához használjuk.

20.3. A basic_string osztály

A standard könyvtár karakterláncokhoz kapcsolódó szolgáltatásai a *basic_string* sablonon (template) alapulnak, amely hasonló típusokat és műveleteket biztosít, mint a szabványos tárolók (§16.3):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class std::basic_string {
public:
  // ...
};
```

A sablon és a hozzá kapcsolódó szolgáltatások az *std* névtérhez tartoznak és a *<string>* fejállományon keresztül érhetők el.

A leggyakoribb karakterlánc-típusok használatát két *typedef* könnyíti meg:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

A *basic_string* sok mindenben hasonlít a *vector* (§16.3) osztályra. A legfontosabb eltérés, hogy a *basic_string* nem tartalmazza az összes eljárást, amely a *vector* osztályban megtalálható, helyettük néhány, jellegzetesen karakterláncokra vonatkozó műveletet (például részlánc-keresést) biztosít. A *string* osztályt nem érdemes egy egyszerű tömbbel vagy a *vector* típussal megvalósítani; a karakterláncok nagyon sok felhasználási módja jobban biztosítható úgy, ha a másolások mennyiségét a lehető legkevesebbre csökkentjük, nem használunk dinamikus adatterületet a rövid karakterláncokhoz, a hosszabbaknál egyszerű módosíthatóságot biztosítunk és így tovább (§20.6[12]). A *string* osztály függvényeinek száma jelzi a karakterláncok kezelésének fontosságát, és azt is, hogy bizonyos számítógépek különleges hardverutasításokkal segítik ezeket a műveleteket. A könyvtárak készítői akkor használhatják ki legjobban az ilyen függvények előnyeit, ha a standard könyvtárban találnak hasonlókat.

A standard könyvtár más típusaihoz hasonlóan a *basic_string<T>* is egy konkrét típus (§2.5.3, §10.3), virtuális függvények nélkül. Bátran felhasználhatjuk tagként egy magasabb szintű szövegfeldolgozó osztályban, de arra nem való, hogy más osztályok bázisosztálya legyen (§25.2.1, lásd még §20.6[10]).

20.3.1. Típusok

A *vector* osztályhoz hasonlóan a *basic_string* is típusneveken keresztül teszi elérhetővé a vele kapcsolatban álló típusokat:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // típusok (mint a vector-nál, a list-nél stb., §16.3.1)

  typedef Tr traits_type;                    // a basic_string-re jellemző

  typedef typename Tr::char_type value_type;
  typedef A allocator_type;
  typedef typename A::size_type size_type;
  typedef typename A::difference_type difference_type;

  typedef typename A::reference reference;
  typedef typename A::const_reference const_reference;
  typedef typename A::pointer pointer;
  typedef typename A::const_pointer const_pointer;

  typedef megvalósítás_függő iterator;
  typedef megvalósítás_függő const_iterator;

  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

  // ...
};
```

A *basic_string* az egyszerű *basic_string<char>* típus mellett (melyet *string* néven ismerünk) számos karaktertípusból tud karakterláncot képezni:

```
typedef basic_string<unsigned char> Ustring;

struct Jchar { /* ... */ };                  // japán karaktertípus
typedef basic_string<Jchar> Jstring;
```

Az ilyen karakterekből képzett karakterláncok ugyanúgy használhatók, mint a *char* típuson alapulók, csak a karakterek szerepe szabhat határt:

```
Ustring first_word(const Ustring& us)
{
  Ustring::size_type pos = us.find(' ');          // §20.3.11
```

```
    return Ustring(us,0,pos);                    // §20.3.4
}

Jstring first_word(const Jstring& js)
{
    Jstring::size_type pos = js.find(' ');       // §20.3.11
    return Jstring(js,0,pos);                    // §20.3.4
}
```

Természetesen használhatunk olyan sablonokat is, melyek karakterlánc-paramétereket használnak:

```
template<class S> S first_word(const S& s)
{
    typename S::size_type pos = s.find(' ');     // §20.3.11
    return S(s,0,pos);                           // §20.3.4
}
```

A *basic_string<Ch>* bármilyen karaktert tartalmazhat, amely szerepel a *Ch* típusban, tehát például a *0* (nulla) karakter is előfordulhat a karakterlánc belsejében. A *Ch* „karaktertípusnak” úgy kell viselkednie, mint egy karakternek, tehát nem lehet felhasználói másoló konstruktora, destruktora, vagy másoló értékadása.

20.3.2. Bejárók

A többi tárolóhoz hasonlóan a *string* osztály is biztosít néhány bejárót (iterátort), melyekkel végighaladhatunk az elemeken, akár a szokásos, akár fordított sorrendben:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // bejárók (mint a vector-nál, a list-nél stb., §16.3.2)

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // ...
};
```

Mivel a *string* osztályban megtalálhatók a bejárók kezeléséhez szükséges tagtípusok és -függvények, a szabványos algoritmusok (18. fejezet) *string* objektumokra is használhatók:

```
void f(string& s)
{
  string::iterator p = find(s.begin(),s.end(),'a');
  // ...
}
```

A karakterláncokra leggyakrabban alkalmazott műveleteket közvetlenül a *string* osztályban találhatjuk meg. Remélhetőleg ezeket a műveleteket kifejezetten a karakterláncokhoz igazították, így jobbak, mint az általános algoritmusok.

A szabványos algoritmusok (18. fejezet) a karakterláncok esetében nem annyira hasznosak, mint első ránézésre gondolnánk. Az általános műveletek azt feltételezik, hogy a tárolók elemei önmagukban is értelmes egységet alkotnak, de a karakterláncok esetében a teljes karaktersorozat hordozza a lényeges információt. Egy karakterlánc rendezése (pontosabban a karakterlánc karaktereinek rendezése) szinte teljesen megsemmisíti a benne tárolt információkat, annak ellenére, hogy az általános tárolókban a rendezés inkább még használhatóbbá szokta tenni az adatokat.

A *string* osztály bejárói sem ellenőrzik, hogy érvényes tartományban állnak-e.

20.3.3. Az elemek elérése

A *string*-ek karaktereit egyesével is elérhetjük, indexelés segítségével:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...
  // elemek elérése (mint a vector-nál, §16.3.3):

  const_reference operator[ ](size_type n) const;        // nem ellenőrzött hozzáférés
  reference operator[ ](size_type n);

  const_reference at(size_type n) const;                 // ellenőrzött hozzáférés
  reference at(size_type n);

  // ...
};
```

Ha az *at()* függvény használatakor a megengedett tartományon kívüli indexértéket (sorszámot) adunk meg, *out_of_range* kivétel váltódik ki.

A *vector* osztálytól eltérően a *string* nem tartalmazza a *front()* és a *back()* függvényt. Ha a karakterlánc első vagy utolsó elemére akarunk hivatkozni, az *s[0]* vagy az *s[s.length()-1]* kifejezést kell használnunk. A mutató–tömb egyenértékűség (§5.3) a karakterláncok esetében nem teljesül: ha *s* egy *string*, akkor a *&s[0]* nem egyezik meg *s* értékével.

20.3.4. Konstruktorok

A kezdeti értékadáshoz, illetve a másolási műveletek elvégzéséhez a *string* más függvényeket kínál, mint az egyéb tárolók (§16.3.4):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...
  // konstruktorok stb. (csak nagyjából úgy, mint a vector-nál és a list-nél, §16.3.4)

  explicit basic_string(const A& a = A());
  basic_string(const basic_string& s,
                          size_type pos = 0, size_type n = npos, const A& a = A());
  basic_string(const Ch* p, size_type n, const A& a = A());
  basic_string(const Ch* p, const A& a = A());
  basic_string(size_type n, Ch c, const A& a = A());
  template<class In> basic_string(In first, In last, const A& a = A());

  ~basic_string();

  static const size_type npos;              // "minden karakter"

  // ...
};
```

Egy *string* objektumnak C stílusú karakterlánccal, másik *string* objektummal, annak egy részével, vagy karakterek egy sorozatával adhatunk kezdőértéket, karakterrel vagy egész értékkel nem:

```
void f(char* p,vector<char>&v)
{
  string s0;                    // üres karakterlánc
  string s00 = "";              // ez is üres karakterlánc
```

```
    string s1 = 'a';                 // hiba: nincs konverzió char-ról string-re
    string s2 = 7;                   // hiba: nincs konverzió int-ről string-re
    string s3(7);                    // hiba: a konstruktornak nem lehet egy int paramétere
    string s4(7,'a');                // 7 darab 'a', vagyis "aaaaaaa"

    string s5 = "Frodo";             // "Frodo" másolata
    string s6 = s5;                  // s5 másolata

    string s7(s5,3,2);               // s5[3] és s5[4], vagyis "do"
    string s8(p+7,3);                // p[7], p[8], és p[9]
    string s9(p,7,3);                // string(string(p),7,3), valószínűleg "költséges"

    string s10(v.begin(),v.end());   // v minden karakterének másolása
}
```

A karakterek helyét nullától kezdve indexeljük, tehát egy karakterlánc nem más, mint *0*-tól *length()-1*-ig számozott karakterek sorozata.

A *length()* függvény a *string* esetében a *size()* megfelelője: mindkettő a karakterláncban szereplő karakterek számát adja meg. Figyeljünk rá, hogy ezek nem egy C stílusú karakterlánc hosszát állapítják meg, tehát nem számolják a lezáró nullkaraktert (§20.4.1). A *basic_string* osztályt a lezáró karakter alkalmazása helyett a karakterlánc hosszának tárolásával illik megvalósítani.

A részláncokat a kezdőpozíció és a karakterszám megadásával azonosíthatjuk. Az alapértelmezett *npos* értéke a lehető legnagyobb szám, jelentése „az összes elem".

Nincs olyan konstruktor, amely *n* darab meghatározatlan karakterből hoz létre karakterláncot. Ehhez legközelebb talán az a konstruktor áll, amely egy adott karakter *n* példányából épít fel egy karakterláncot. Az olyan konstruktorok hiánya, melyeknek egyetlen karaktert vagy csak a karakterláncban lévő karakterek számát adnánk meg, lehetővé teszi, hogy a fordító észrevegyen olyan hibalehetőségeket, amilyeneket az *s2* és az *s3* fenti meghatározása rejt magában.

A másoló konstruktor egy négyparaméterű konstruktor. E paraméterek közül háromnak alapértelmezett értéke van. A hatékonyság érdekében két különálló konstruktorként is elkészíthetjük; a felhasználó nem lesz képes különbséget tenni a két megoldás között, ha a lefordított kódot meg nem nézi.

A legáltalánosabb konstruktor egy sablon tagfüggvénye. Ez lehetővé teszi, hogy a karakterlánc kezdőértékét tetszőleges sorozatból állítsuk elő, például egy más karaktertípust használó karakterlánc segítségével, amennyiben a karaktertípusok közötti konverzió rendelkezésünkre áll:

```
void f(string s)
{
  wstring ws(s.begin(),s.end());          // s minden karakterének másolása
  // ...
}
```

A *ws* karakterlánc minden *wchar_t* karakterének az *s* megfelelő *char* eleme ad kezdőértéket.

20.3.5. Hibák

A karakterláncokat igen egyszerűen olvashatjuk, írhatjuk, megjeleníthetjük, tárolhatjuk, összehasonlíthatjuk, lemásolhatjuk stb. Ez általában nem okoz problémákat, legfeljebb a hatékonysággal támadhatnak gondjaink. Ha azonban elkezdünk karakterláncok egyes részláncaival, karaktereivel foglalkozni és így létező karakterláncokból akarunk újakat összeállítani, előbb-utóbb hibákat fogunk elkövetni, melynek következtében a karakterlánc határain kívülre próbálunk meg írni.

Az egyes karakterek közvetlen elérésére szolgáló *at()* függvény ellenőrzi az ilyen hibákat és *out-of_range* kivételt vált ki, ha érvénytelen hivatkozást adunk meg. A *[ ]* operátor ilyen vizsgálatot nem végez.

A legtöbb karakterlánc-műveletnek egy karakterpozíciót és egy karakterszámot kell megadnunk. Ha a megadott pozícióérték nagyobb, mint a karakterlánc mérete, azonnal *out_of_range* kivételt kapunk; ha a karakterszám „túl nagy”, értelmezése általában az lesz, hogy az összes hátralévő karaktert használni akarjuk:

```
void f()
{
  string s = "Snobol4";
  string s2(s,100,2);           // a megadott karakterpozíció a lánc végén túl van:
                                // out_of_range() váltódik ki
  string s3(s,2,100);           // a karakterszám túl nagy: egyenértékű a s3(s,2,s.size()-2)
                                // kifejezéssel
  string s4(s,2,string::npos);  // az s[2]-től kezdődő összes karakter
}
```

A „túl nagy" karakterpozíciókat ki kell szűrnünk, de a „túl nagy" karakterszám hasznos lehet a programokban. Valójában az *npos* a lehető legnagyobb *size_type* típusú érték.

Érdemes kipróbálnunk, mi történik akkor, ha negatív karakterpozíciót vagy karakterszámot adunk meg:

```
void g(string& s)
{
   string s5(s,-2,3);   // nagy pozícióérték!: out_of_range()
   string s6(s,3,-2);   // nagy karakterszám!: rendben
}
```

Mivel a *size_type* típust arra használjuk, hogy pozíciókat vagy darabszámokat adjunk meg, *unsigned* típusként definiált, így a negatív számok használata csak félrevezető módja a nagy pozitív számok megadásának (§16.3.4).

Azok a függvények, amelyek egy *string* részláncát keresik meg (§20.3.11), az *npos* értéket adják vissza, ha nem találják meg a megfelelő részt. Tehát maguk nem váltanak ki kivételt, de ha ezt az *npos* értéket karakterpozícióként akarjuk felhasználni a következő műveletben, akkor már kivételt kapunk.

Egy részlánc kijelölésének másik módja, hogy két bejárót (iterator) adunk meg. Az első határozza meg a pozíciót, míg a két bejáró különbsége a karakterszámot. Szokás szerint a bejárók nem ellenőrzöttek.

Ha C stílusú karakterláncokat használunk, a tartományellenőrzés nehezebb feladat. Ha paraméterként adunk meg ilyen karakterláncot (tehát egy *char*-ra hivatkozó mutatót), a *basic_string* függvényei feltételezik, hogy a mutató nem *0*. Ha egy C stílusú karakterláncban pozíciót adunk meg, a függvények elvárják, hogy a karakterlánc elég hosszú legyen a pozíció értelmezéséhez. Mindig legyünk nagyon óvatosak, sőt kifejezetten gyanakvóak. Az egyetlen kivétel, amikor karakterliterálokat használunk.

Minden karakterlánc esetében igaz, hogy *length()<npos*. Néhány nagyon egyedi helyzetben (igen ritkán) előfordulhat, hogy egy olyan jellegű művelet, mint egy karakterlánc beszúrása egy másikba, túl hosszú karakterláncot eredményez, amelyet a rendszer már nem képes ábrázolni. Ebben az esetben *length_error* kivétel keletkezik:

```
string s(string::npos,'a');       // length_error() váltódik ki
```

20.3.6. Értékadás

Természetesen a karakterláncok esetében is rendelkezésünkre áll az (egyszerű) értékadás művelete:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
   // ...
   // értékadás (kicsit hasonlít a vector-ra és a list-re, §16.3.4):

   basic_string& operator=(const basic_string& s);
   basic_string& operator=(const Ch* p);
   basic_string& operator=(Ch c);

   basic_string& assign(const basic_string&);
   basic_string& assign(const basic_string& s, size_type pos, size_type n);
   basic_string& assign(const Ch* p, size_type n);
   basic_string& assign(const Ch* p);
   basic_string& assign(size_type n, Ch c);
   template<class In> basic_string& assign(In first, In last);
   // ...
};
```

A többi tárolóhoz hasonlóan a *string* osztályban is érték szerinti értelmezés működik, ami azt jelenti, hogy amikor egy karakterláncot értékül adunk egy másiknak, akkor az eredeti karakterláncról másolat készül, és az értékadás után két, ugyanolyan tartalmú, de önálló (értékű) karakterlánc áll majd rendelkezésünkre:

```
void g()
{
   string s1 = "Knold";
   string s2 = "Tot";
   s1 = s2;            // "Tot"-ból két példány lesz
   s2[1] = 'u';        // s2 "Tut", s1 marad "Tot"
}
```

Annak ellenére, hogy egyetlen karakterrel nem adhatunk kezdőértéket egy karakterláncnak, az ilyen módon történő egyszerű értékadás megengedett:

```
void f()
{
   string s = 'a';     // hiba: kezdeti értékadás char-ral
   s = 'a';            // rendben: egyszerű értékadás
   s = "a";
   s = s;
}
```

Az a lehetőség, hogy karakterláncnak értékül adhatunk egy karaktert, nem túlságosan hasznos, és sok hibalehetőséget rejt magában. Gyakran azonban nélkülözhetetlen, hogy egy karaktert a += művelettel hozzáfűzhessünk egy karakterlánchoz (§20.3.9), és igen furcsán nézne ki, ha az *s+='c'* utasítás végrehajtható lenne, míg az *s='c'* nem.

Az értékadáshoz az *assign()* nevet használjuk, amely a többparaméterű konstruktorok megfelelőjének tekinthető (§176.3.4, §20.3.4).

A §11.12 pontban már említettük, hogy a *string* osztály optimalizálható úgy, hogy amíg nincs szükség egy karakterlánc két példányára, addig nem hajtjuk végre a tényleges másolást. A szabványos *string* felépítése támogatja az ilyen takarékosan másoló megvalósítások létrehozását, mert így hatékonyan írhatunk le csak olvasható karakterláncokat, és a karakterláncok átadása függvények paramétereként sokkal egyszerűbben megvalósítható, mint azt első ránézésre gondolnánk. Az azonban meggondolatlanság lenne, ha egy programozó saját fejlesztőkörnyezetének ellenőrzése nélkül olyan programokat írna, amelyek a *string*-ek optimalizált másolására támaszkodnak (§20.6[13]).

20.3.7. Átalakítás C stílusú karakterláncra

A §20.3.4 pontban bemutattuk, hogy a *string* objektumoknak való kezdeti és egyszerű értékadásra egyaránt használhatunk C stílusú karakterláncokat. Fordított irányú műveletek elvégzésére is van lehetőség, tehát egy *string* karaktereit is elhelyezhetjük egy tömbben:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
   // ...
   // átalakítás C stílusú karakterlánccá

   const Ch* c_str() const;
   const Ch* data() const;
   size_type copy(Ch* p, size_type n, size_type pos = 0) const;

   // ...
};
```

A *data()* függvény a *string* karaktereit egy tömbbe másolja, majd visszaad egy erre a tömbre hivatkozó mutatót. A tömb a *string* objektumhoz tartozik, tehát a felhasználónak nem szabad törölnie azt és a *string* egy nem *const* függvényének meghívása után már nem tudhatja, milyen érték van benne. A *c_str()* függvény szinte pontosan ugyanígy működik, csak a karakterlánc végén elhelyez egy *0* (null) karaktert is, a C stílusú lezárásnak megfelelően:

```
void f()
{
    string s = "equinox";                  // s.length()==7
    const char* p1 = s.data();             // p1 hét karakterre mutat
    printf("p1 = %s\n",p1);                // hiba: hiányzó lezáró
    p1[2] = 'a';                           // hiba: p1 konstans tömbre mutat
    s[2] = 'a';
    char c = p1[1];              // hiba: s.data() elérésének kísérlete s módosítása után

    const char* p2 = s.c_str();  // p2 nyolc karakterre mutat
    printf("p2 = %s\n",p2);      // rendben: c_str() hozzáadja a lezáró karaktert
}
```

A különbséget úgy is megfogalmazhatjuk, hogy a *data()* a karakterek egy tömbjét adja vissza, míg a *c_str()* egy C stílusú karakterláncot állít elő. Ezen függvények elsődleges feladata, hogy könnyen használhatóvá tegyék az olyan függvényeket, melyek C stílusú karakterláncot várnak paraméterként. Így tehát a *c_str()* függvényt sokkal gyakrabban használjuk, mint a *data()* eljárást:

```
void f(string s)
{
    int i = atoi(s.c_str());     // a karakterlánc int értékének lekérése (§20.4.1)
    // ...
}
```

Általában érdemes a karaktereket mindaddig egy *string* objektumban tárolni, amíg nincs rájuk szükségünk, de ha mégsem tudjuk azonnal feldolgozni, akkor is érdemes átmásolni azokat a *c_str()*, illetve a *data()* által lefoglalt területről egy külön tömbbe. A *copy()* függvény pontosan erre szolgál:

```
char* c_string(const string& s)
{
    char* p = new char[s.length()+1];      // megjegyzés: +1
    s.copy(p,string::npos);
    p[s.length()] = 0;                     // megjegyzés: lezáró hozzáadása
    return p;
}
```

Az *s.copy(p,n,m)* függvény legfeljebb *n* karaktert másol a *p* címre az *s[m]* pozíciótól kezdve. Ha az *s* karakterláncból *n*-nél kevesebb karaktert lehet csak átmásolni, a *copy()* egyszerűen az összes karaktert átmásolja.

Figyeljünk rá, hogy a *string* objektumokban szerepelhet a *0* karakter. A C stílusú karakterláncokat kezelő függvények az első ilyen karaktert tekintik lezárónak. Tehát mindig figyeljünk, hogy *0* karaktert csak akkor használjunk, ha C stílust használó függvényekre nem lesz szükségünk, vagy a *0*-kat pontosan oda tegyük, ahol a karakterláncot le szeretnénk zárni.

A C stílusú karakterláncra való átalakítást a *c_str()* helyett megoldhattuk volna egy *operator const char* ()* művelettel is, az automatikus konverzió kényelmének azonban az lenne az ára, hogy meglepetésünkre időnként akkor is végbemenne, amikor nem is számítunk rá.

Ha úgy érezzük, hogy programunkban sokszor lesz szükség a *c_str()* függvényre, valószínűleg túlságosan ragaszkodunk a C stílusú felülethez. Általában rendelkezésünkre állnak azok az eszközök, melyekkel a C stílusú karakterláncokra vonatkozó műveletek közvetlenül *string* objektumokon is elvégezhetők. Ezek használatával sok konverziót elkerülhetünk. Egy másik lehetséges megoldás az explicit konverziók elkerülésére az, hogy túlterheljük azokat a függvényeket, melyek a *c_str()* használatára kényszerítenek bennünket:

```
extern "C" int atoi(const char*);

int atoi(const string& s)
{
  return atoi(s.c_str());
}
```

20.3.8. Összehasonlítás

Karakterláncokat azonos típusú karakterláncokkal vagy olyan karaktertömbökkel hasonlíthatunk össze, melyek szintén ugyanolyan típusú karaktereket tartalmaznak:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...

  int compare(const basic_string& s) const;       // > és == használata együtt
  int compare(const Ch* p) const;

  int compare(size_type pos, size_type n, const basic_string& s) const;
  int compare(size_type pos, size_type n,
                    const basic_string& s, size_type pos2, size_type n2) const;
  int compare(size_type pos, size_type n, const Ch* p, size_type n2 = npos) const;

  // ...
};
```

Ha a *compare()* meghívásakor a pozíció és méret paramétereket is megadjuk, csak a kijelölt részsorozat vesz részt az összehasonlításban. Például *s.compare(pos,n,s2)* egyenértékű *string(s,pos,n).compare(s2)*-vel. Az összehasonlító eljárást a *char_traits<Ch>* osztály *compare()* függvénye adja (§20.2.1). Így az *s.compare(s2)* függvény *0* értéket ad vissza, ha a karakterláncok egyenlő értékűek; negatív számot kapunk, ha *s* ábécésorrendben *s2* előtt áll; fordított esetben pedig pozitív lesz az eredmény.

A felhasználó itt nem adhat meg úgy összehasonlítási feltételt, mint a §13.4 pontban. Ha ilyen szintű rugalmasságra van szükségünk, a *lexicographical_compare()* (§18.9) segítségével készítsünk a fenti részben levőhöz hasonló összehasonlító függvényt. Egy másik lehetőség, hogy saját ciklust írunk a feladat megoldására. A *toupper()* függvény (§20.4.2) például lehetővé teszi, hogy kis- és nagybetűkkel nem foglalkozó összehasonlítást valósítsunk meg:

```
int cmp_nocase(const string& s, const string& s2)
{
  string::const_iterator p = s.begin();
  string::const_iterator p2 = s2.begin();

  while (p!=s.end() && p2!=s2.end()) {
        if (toupper(*p)!=toupper(*p2)) return (toupper(*p)<toupper(*p2)) ? -1 : 1;
        ++p;
        ++p2;
  }

  return (s2.size()==s.size()) ? 0 : (s.size()<s2.size()) ? -1 : 1;      // 'size' előjel nélküli
}

void f(const string& s, const string& s2)
{
  if (s == s2) {      // kis- és nagybetűket figyelembe vevő összehasonlítás s és s2 között
        // ...
  }

  if (cmp_nocase(s,s2) == 0) {          // kis- és nagybetűket figyelmen kívül hagyó
                                        // összehasonlítás s és s2 között
        // ...
  }

  // ...
}
```

A *basic_string* osztályban rendelkezésünkre állnak a szokásos összehasonlító operátorok is (==, !=, <, >, <=, >=):

```
template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const Ch*, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch,Tr,A>&, const Ch*);

// és ugyanilyen deklarációk a !=, >, <, >=, és a <= számára
```

Az összehasonlító operátorok nem tag függvények, így a konverziók mindkét operandusra ugyanúgy vonatkoznak (§11.2.3). A C stílusú karakterláncokat használó változatokra azért volt szükség, hogy a literálokkal való összehasonlítást hatékonyabbá tegyük:

```
void f(const string& name)
{
  if (name =="Obelix" || "Asterix"==name) {     // optimalizált == használata
            // ...
  }
}
```

20.3.9. Beszúrás

Miután előállítottunk egy karakterláncot, sokféle műveletet végezhetünk vele. A karakterlánc értékét módosító függvények közül talán a legfontosabb a hozzáfűzés, amely a karakterlánc végén helyez el új karaktereket. Az általános beszúró műveletekre ritkábban van szükség:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...
  // karakterek hozzáadása (*this)[length()-1] után

  basic_string& operator+=(const basic_string& s);
  basic_string& operator+=(const Ch* p);
  basic_string& operator+=(Ch c);
  void push_back(Ch c);

  basic_string& append(const basic_string& s);
  basic_string& append(const basic_string& s, size_type pos, size_type n);
  basic_string& append(const Ch* p, size_type n);
```

```
    basic_string& append(const Ch* p);
    basic_string& append(size_type n, Ch c);
    template<class In> basic_string& append(In first, In last);

    // karakterek beszúrása (*this)[pos] elé

    basic_string& insert(size_type pos, const basic_string& s);
    basic_string& insert(size_type pos, const basic_string& s, size_type pos2, size_type n);
    basic_string& insert(size_type pos, const Ch* p, size_type n);
    basic_string& insert(size_type pos, const Ch* p);
    basic_string& insert(size_type pos, size_type n, Ch c);

    // karakterek beszúrása p elé

    iterator insert(iterator p, Ch c);
    void insert(iterator p, size_type n, Ch c);
    template<class In> void insert(iterator p, In first, In last);

    // ...
};
```

Nagyjából ugyanazok a függvényváltozatok állnak rendelkezésünkre a beszúró és a hozzáfűző eljárásoknál is, mint a konstruktorok és az értékadás esetében.

A += operátor hagyományos jelölése a hozzáfűzésnek:

```
string complete_name(const string& first_name, const string& family_name)
{
    string s = first_name;
    s += ' ';
    s += family_name;
    return s;
}
```

A karakterlánc végéhez való hozzáfűzés jelentősen hatékonyabb lehet, mint a más pozíciókra való beszúrás:

```
string complete_name2(const string& first_name, const string& family_name)
                                                        // szegényes algoritmus
{
    string s = family_name;
    s.insert(s.begin(),' ');
    s.insert(0,first_name);
    return s;
}
```

A beszúrás gyakran arra kényszeríti a *string*-et, hogy lassú memóriaműveleteket végezzen és áthelyezzen néhány karaktert.

Mivel a *string* osztályban is szerepel a *push_back()* művelet (§16.3.5), a *back_inserter* ugyanúgy használható *string* objektumokhoz, mint bármely általános tárolóhoz.

20.3.10. Összefűzés

A hozzáfűzés különleges változata az összefűzésnek (konkatenációnak). Az *összefűzést* – tehát egy karakterlánc előállítását úgy, hogy két másikat egymás után helyezünk – a **+** operátor valósítja meg:

```
template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A>
operator+(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const Ch*, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(Ch, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, const Ch*);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, Ch);
```

Szokás szerint, a **+** műveletet nem tag függvényként adjuk meg. A több paramétert használó sablonok esetében ez némi kellemetlenséget okoz, mert a sablonparamétereket meg kell ismételnünk.

Másrészt az összefűzés használata egyszerű és kényelmes:

```
string complete_name3(const string& first_name, const string& family_name)
{
  return first_name + ' ' + family_name;
}
```

Ez a kényelem megér egy kis futási idejű teljesítményveszteséget a *complete_name()* függvényhez viszonyítva. A *complete_name3()* függvényben külön ideiglenes változóra (§11.3.2) van szükségünk. Véleményem szerint ez ritkán fontos, de azért érdemes tudnunk

róla, ha egy nagy ciklust írunk egy olyan programban, ahol figyelnünk kell a teljesítményre. Ilyenkor esetleg érdemes megszüntetnünk a függvényhívást a *complete_name()* helyben (inline) kifejtésével, az eredményként kapott karakterláncot így helyben építhetjük fel, alacsonyabb szintű műveletek segítségével (§20.6[14]).

20.3.11. Keresés

Zavarba ejtő mennyiségben állnak rendelkezésünkre olyan függvények, melyekkel egy karakterlánc részláncait kereshetjük meg:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...
  // részsorozat kerése (mint a search(), §18.5.5)

  size_type find(const basic_string& s, size_type i = 0) const;
  size_type find(const Ch* p, size_type i, size_type n) const;
  size_type find(const Ch* p, size_type i = 0) const;
  size_type find(Ch c, size_type i = 0) const;

  // részsorozat keresése visszafelé (mint a find_end(), §18.5.5)

  size_type rfind(const basic_string& s, size_type i = npos) const;
  size_type rfind(const Ch* p, size_type i, size_type n) const;
  size_type rfind(const Ch* p, size_type i = npos) const;
  size_type rfind(Ch c, size_type i = npos) const;

  // karakter keresése (mint a find_first_of(), §18.5.2)

  size_type find_first_of(const basic_string& s, size_type i = 0) const;
  size_type find_first_of(const Ch* p, size_type i, size_type n) const;
  size_type find_first_of(const Ch* p, size_type i = 0) const;
  size_type find_first_of(Ch c, size_type i = 0) const;

  // karakter keresése paraméter alapján visszafelé

  size_type find_last_of(const basic_string& s, size_type i = npos) const;
  size_type find_last_of(const Ch* p, size_type i, size_type n) const;
  size_type find_last_of(const Ch* p, size_type i = npos) const;
  size_type find_last_of(Ch c, size_type i = npos) const;

  // paraméterben nem szereplő karakter keresése
```

```
    size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
    size_type find_first_not_of(const Ch* p, size_type i, size_type n) const;
    size_type find_first_not_of(const Ch* p, size_type i = 0) const;
    size_type find_first_not_of(Ch c, size_type i = 0) const;

    // paraméterben nem szereplő karakter keresése visszafelé

    size_type find_last_not_of(const basic_string& s, size_type i = npos) const;
    size_type find_last_not_of(const Ch* p, size_type i, size_type n) const;
    size_type find_last_not_of(const Ch* p, size_type i = npos) const;
    size_type find_last_not_of(Ch c, size_type i = npos) const;
    // ...
};
```

Az összes fenti függvény *const*. Tehát arra használhatók, hogy valamilyen célból megkeressenek egy részláncot, de maguk nem változtatják meg azt a karakterláncot, amelyre alkalmaztuk azokat.

A *basic_string::find* műveletek jelentését úgy érthetjük meg legjobban, ha megértjük a velük egyenértékű általános algoritmusokat:

```
void f()
{
    string s = "accdcde";
    typedef ST;

    string::size_type i1 = s.find("cd");                    // i1 = 2 s[2]=='c' && s[3]=='d'
    string::size_type i2 = s.rfind("cd");                   // i2 = 4 s[4]=='c' && s[5]=='d'
    string::size_type i3 = s.find_first_of("cd");           // i3 = 1 s[1] == 'c'
    string::size_type i4 = s.find_last_of("cd");            // i4 = 5 s[5] == 'd'
    string::size_type i5 = s.find_first_not_of("cd");       // i5 = 0 s[0]!='c' && s[0]!='d'
    string::size_type i6 = s.find_last_not_of("cd");        // i6 = 6 s[6]!='c' && s[6]!='d'
}
```

Ha a *find()* függvények nem találják meg, amit kellene, *npos* értéket adnak vissza, amely érvénytelen karakterpozíciót jelent. Ha az *npos* értéket karakterpozícióként használjuk, *range_error* kivételt kapunk (§20.3.5). Figyeljünk rá, hogy a *find()* eredménye *unsigned* érték.

20.3.12. Csere

Miután meghatároztunk egy karakterpozíciót a karakterláncban, az indexelés segítségével az egyes karaktereket módosíthatjuk, de a *replace()* művelet felhasználásával lecserélhetünk teljes részláncokat is:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...
  // [ (*this)[i], (*this)[i+n] [ felcserélése más karakterekkel

  basic_string& replace(size_type i, size_type n, const basic_string& s);
  basic_string& replace(size_type i, size_type n,
                                  const basic_string& s, size_type i2, size_type n2);
  basic_string& replace(size_type i, size_type n, const Ch* p, size_type n2);
  basic_string& replace(size_type i, size_type n, const Ch* p);
  basic_string& replace(size_type i, size_type n, size_type n2, Ch c);

  basic_string& replace(iterator i, iterator i2, const basic_string& s);
  basic_string& replace(iterator i, iterator i2, const Ch* p, size_type n);
  basic_string& replace(iterator i, iterator i2, const Ch* p);
  basic_string& replace(iterator i, iterator i2, size_type n, Ch c);
  template<class In> basic_string& replace(iterator i, iterator i2, In j, In j2);

  // karakterek törlése a láncból ("csere semmire")

  basic_string& erase(size_type i = 0, size_type n = npos);
  iterator erase(iterator i);
  iterator erase(iterator first, iterator last);
  void clear();              // az összes karakter törlése
  // ...
};
```

Az új karakterek számának nem kell megegyeznie a karakterláncban eddig szereplő karakterek számával. A karakterlánc mérete úgy változik, hogy az új részlánc pontosan elférjen benne. Valójában az *erase()* függvény sem csinál mást, mint hogy a megadott részláncot üres karakterláncra cseréli és az eredeti karakterlánc méretét megfelelően módosítja:

```
void f()
{
  string s = "Márpedig ez működik, ha elhiszed, ha nem.";
  s.erase(0,8);                              // a "Márpedig " törlése
  s.replace(s.find("ez"),2,"Csak akkor");
  s.replace(s.find(", ha nem"),8,"");        // törlés "" behelyettesítésével
}
```

Az *erase()* függvény egyszerű, paraméter nélküli meghívása a teljes karakterláncot üres karakterlánccá alakítja. Ezt a műveletet az általános tárolók esetében a *clear()* függvény valósítja meg (§16.3.6).

A *replace()* függvény változatai az értékadás változataihoz igazodnak, hiszen a *replace()* valójában nem más, mint értékadás egy részláncnak.

20.3.13. Részláncok

A *substr()* függvény lehetővé teszi, hogy kiválasszunk egy részláncot, amelyet kezdőpozíciójával és hosszával adunk meg:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...
  // részlánc címzése

  basic_string substr(size_type i = 0, size_type n = npos) const;
  // ...
};
```

A *substr()* függvény egyszerű módszert biztosít a karakterlánc egy részletének kiolvasására. Ebből a szemszögből a párja a *replace()* függvény, mellyel felülírhatjuk a karakterlánc egy részét. Mindkét függvénynek a kezdőpozíciót és a részlánc hosszát kell megadnunk. A *find()* segítségével viszont már érték szerint is megtalálhatunk részláncokat, így ez a függvénycsoport lehetővé teszi, hogy definiáljunk egy részlánc-osztályt, amelyet írni és olvasni is tudunk:

```
template<class Ch> class Basic_substring {
public:
  typedef typename basic_string<Ch>::size_type size_type;

  Basic_substring(basic_string<Ch>& s, size_type i, size_type n);          // s[i]..s[i+n-1]
  Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2);       // s2 az s-ben
  Basic_substring(basic_string<Ch>& s, const Ch* p);                      // *p az s-ben

  Basic_substring& operator=(const basic_string<Ch>&);                    // írás *ps-be
  Basic_substring& operator=(const Basic_substring<Ch>&);
  Basic_substring& operator=(const Ch*);
  Basic_substring& operator=(Ch);

  operator basic_string<Ch>() const;                              // olvasás *ps-be
  operator const Ch* () const;                                    // a c_str() használata
```

```
private:
  basic_string<Ch>* ps;
  size_type pos;
  size_type n;
};
```

Az osztály megvalósítása is nagyon egyszerű:

```
template<class Ch>
Basic_substring<Ch>::Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2)
  : ps(&s), n(s2.length())
{
  pos = s.find(s2);
}

template<class Ch>
Basic_substring<Ch>& Basic_substring<Ch>::operator=(const basic_string<Ch>& s)
{
  ps->replace(pos,n,s);                      // írás *ps-be
  return *this;
}

template<class Ch> Basic_substring<Ch>::operator basic_string<Ch>() const
{
  return basic_string<Ch>(*ps,pos,n);        // másolás *ps-be
}
```

Ha *s2* nem található meg az *s* karakterláncban, *pos* értéke *npos* lesz, így ha egy ilyen részláncot próbálunk meg írni vagy olvasni, *range_error* kivételt kapunk (§20.3.5).

A *Basic_substring* osztály a következő formában használható:

```
typedef Basic_substring<char> Substring;

void f()
{
  string s = "Mary had a little lamb";
  Substring(s,"lamb") = "fun";
  Substring(s,"a little") = "no";
  string s2 = "Joe" + Substring(s,s.find(' '),string::npos);
}
```

Természetesen sokkal érdekesebb feladatokat is elvégezhetnénk, ha a *Substring* osztály mintaillesztést is tudna végezni (§20.6[7]).

20.3.14. Méret és kapacitás

A memóriakezeléssel kapcsolatos kérdéseket a *string* nagyjából ugyanúgy kezeli, mint a *vector* (§16.3.8):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
  // ...
  // méret, kapacitás stb. (mint a §16.3.8 pontban)

  size_type size() const;                    // karakterek száma (§20.3.4)
  size_type max_size() const;                // a legnagyobb lehetséges karakterlánc
  size_type length() const { return size(); }
  bool empty() const { return size()==0; }

  void resize(size_type n, Ch c);
  void resize(size_type n) { resize(n,Ch()); }

  size_type capacity() const;                // mint a vector, §16.3.8
  void reserve(size_type res_arg = 0);       // mint a vector, §16.3.8

  allocator_type get_allocator() const;
};
```

A *reverse(res_arg)* függvény *length_error* kivételt vált ki, ha *res_arg>max_size()*.

20.3.15. Ki- és bemeneti műveletek

A *string* osztály egyik legfontosabb felhasználási területe, hogy bemeneti műveletek célja, illetve kimeneti műveletek forrása legyen. A *basic_string* I/O operátorait a *<string>* (és nem az *<iostream>*) tartalmazza:

```
template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&,const
basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&, Ch eol);

template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&);
```

A << a karakterláncot egy kimeneti adatfolyamra (*ostream*, §21.2.1) írja, míg a >> operátor egy üreshely karakterekkel határolt szót olvas be a karakterláncba egy bemeneti adatfolyamról (§3.6, §21.3.1). A szó előtti szóköz, tabulátor, újsor stb. karaktereket egyszerűen átugorja és a szó után következő ilyen karakterek sem kerülnek bele a karakterláncba.

A *getline()* függvény meghívásával egy teljes, *eol* karakterrel lezárt sort olvashatunk be a karakterláncba, a karakterlánc mérete pedig úgy növekszik, hogy a sor elférjen benne (§3.6). Ha nem adjuk meg az *eol* paramétert, a függvény az *'\n'* újsor karaktert használja. A sorzáró karaktert beolvassa az adatfolyamról, de a karakterláncba nem kerül be. Mivel a *string* mérete úgy változik, hogy a beolvasott sor elférjen benne, nincs szükség arra, hogy a lezáró karaktert az adatfolyamban hagyjuk vagy visszaadjuk a beolvasott karakterek számát a hívónak. A karaktertömbök esetében a *get()* és a *getline()* kénytelen volt ilyen, ellenőrzést segítő megoldásokat alkalmazni (§21.3.4).

20.3.16. Felcserélés

Ugyanúgy, mint a *vector*-nál (§16.3.9), a *swap()* függvényből a karakterláncokhoz is sokkal hatékonyabb specializációt készíthetünk, mint az általános algoritmus:

```
template<class Ch, class Tr, class A>
void swap(basic_string<Ch,Tr,A>&, basic_string<Ch,Tr,A>&);
```

20.4. A C standard könyvtára

A C++ standard könyvtára örökölte a C stílusú karakterláncok kezelésével foglalkozó függvényeket. Ebben a pontban felsorolunk néhányat a legfontosabb, C karakterláncokat kezelő függvények közül. A leírás egyáltalán nem terjed ki mindre; ha részletes információkat szeretnénk megtudni, nézzük meg fejlesztőrendszerünk kézikönyvét. Legyünk óvatosak ezen függvények használatakor, mert a könyvtárak készítői gyakran saját, nem szabványos függvényeket adnak meg a szabványos fejállományokban, így nehéz megállapítani, hogy melyek a minden rendszerben elérhető függvények.

A C standard könyvtárának lehetőségeit leíró fejállományok listáját a §16.1.2 pontban adtuk meg. A memóriakezelő függvényekkel a §19.4.6 pontban foglalkozunk, a C ki- és bemeneti függvényeivel a §21.8 részben, a C matematikai könyvtárával pedig a §22.3 pontban.

A program indításával és befejezésével kapcsolatos függvényeket a §3.2 és a §9.4.1.1 pontban mutattuk be, míg a meghatározatlan függvényparaméterek beolvasását segítő függvényeket a §7.6 pont tárgyalta. Azon C stílusú függvények, melyek a széles karaktereket kezelik, a *<cwchar>* és a *<wchar.h>* fejállományban találhatók.

20.4.1. C stílusú karakterláncok

A C stílusú karakterláncok kezelésével foglalkozó függvények a *<string.h>* és a *<cstring>* fejállományban találhatók:

```
char* strcpy(char* p, const char* q);             // másolás q-ból p-be (a lezáróval együtt)
char* strcat(char* p, const char* q);             // hozzáfűzés q-ból p-be (a lezáróval együtt)
char* strncpy(char* p, const char* q, int n);     // n karakter másolása q-ból p-be
char* strncat(char* p, const char* q, int n);     // n karakter hozzáfűzése q-ból p-be

size_t strlen(const char* p);                     // p hossza (a lezáró nélkül)

int strcmp(const char* p, const char* q);         // p és q összehasonlítása
int strncmp(const char* p, const char* q, int n); // az első n karakter összehasonlítása

char* strchr(char* p, int c);                     // az első c keresése p-ben
const char* strchr(const char* p, int c);
char* strrchr(char* p, int c);                    // az utolsó c keresése p-ben
const char* strrchr(const char* p, int c);
char* strstr(char* p, const char* q);             // az első q keresése p-ben
const char* strstr(const char* p, const char* q);

char* strpbrk(char* p, const char* q);            // q első karakterének keresése p-ben
const char* strpbrk(const char* p, const char* q);

size_t strspn(const char* p, const char* q);      // p karaktereinek száma az első,
                                                  // q-ban szereplő karakter előtt
size_t strcspn(const char* p, const char* q);     // p karaktereinek száma az első, q-ban
nem szereplő karakter előtt
```

A megadott mutatókat a függvények nem-nulláknak feltételezik, azoknak a *char* tömböknek pedig, melyekre a mutatók mutatnak, *0* karakterrel lezárt sorozatoknak kell lenniük. Az *strn* függvények *0* karakterekkel töltik fel a hiányzó területet, ha nincs *n* darab feldolgozandó karakter. A karakterlánc-összehasonlítások nullát adnak vissza, ha a két karakterlánc egyenlő; negatív számot, ha az első paraméter ábécésorrendben előbb következik, mint a második; fordított esetben pedig pozitív számot.

Természetesen a C nem biztosít túlterhelt függvénypárokat, a C++-ban azonban szükség van ezekre a *const* biztonságos megvalósításához:

```
void f(const char* pcc, char* pc)        // C++
{
  *strchr(pcc,'a') = 'b';           // hiba: const char-nak nem adhatunk értéket
  *strchr(pc,'a') = 'b';            // rendben, bár nem tökéletes: lehet, hogy pc-ben nincs 'a'
}
```

A C++ *strchr()* függvénye nem engedi meg, hogy *const* karakterláncba írjunk, egy C program azonban „kihasználhatja” a C-beli *strchr()* gyengébb típusellenőrzését:

```
char* strchr(const char* p, int c);      /* C standard könyvtárbeli függvény, nem C++ */

void g(const char* pcc, char* pc)        /* C, a C++ nem fordítja le */
{
  *strchr(pcc,'a') = 'b';           /* const átalakítása nem const-tá: C-ben jó, C++-ban hiba */
  *strchr(pc,'a') = 'b';            /* jó C-ben és C++-ban is */
}
```

Ha tehetjük, feltétlenül kerüljük a C stílusú karakterláncok használatát és használjuk helyettük a *string* osztályt. A C stílusú karakterláncok és a hozzájuk tartozó szabványos függvények segítségével nagyon hatékony programokat írhatunk, de még a gyakorlott C és C++ programozók is gyakran követnek el „buta kis hibákat” miközben ezekkel dolgoznak. Persze minden C++ programozó találkozni fog olyan régebbi programokkal, melyek ezeket a lehetőségeket használják. Íme egy haszontalan kis példa, amellyel a leggyakoribb függvényeket mutatjuk be:

```
void f(char* p, char* q)
{
  if (p==q) return;                 // a mutatók egyenértékűek
  if (strcmp(p,q)==0) {             // a karakterlánc-értékek egyenértékűek
          int i = strlen(p);        // a karakterek száma (a lezáró nélkül)
          // ...
  }
  char buf[200];
  strcpy(buf,p);                            // p másolása a buf-ba (a lezáróval együtt)
                                            // nem tökéletes, túlcsordulhat
  strncpy(buf,p,200);                       // 200 karakter másolása p-ből a buf-ba
                                            // nem tökéletes; lehet, hogy nem másolja át a lezárót
  // ...
}
```

A C stílusú karakterláncok ki- és bevitelét általában a *printf* függvénycsalád segítségével valósítjuk meg (§21.8).

Az *<stdlib.h>* és a *<cstdlib>* fejállományban a standard könyvtár néhány olyan hasznos függvényt nyújt, melyekkel számértéket jelölő karakterláncokat alakíthatunk számmá:

```
double atof(const char* p);        // p átalakítása double-lá
int atoi(const char* p);           // p átalakítása int-té
long atol(const char* p);          // p átalakítása long-gá
```

A bevezető üreshely karaktereket ezek a függvények nem veszik figyelembe. Ha a karakterlánc nem számot ábrázol, a visszatérési érték *0* lesz. (Az *atoi("hét")* eredménye is *0*!) Ha a karakterlánc számot jelöl, de az nem ábrázolható az eredményként várt számtípusban, akkor az *errno* (§16.1.2, §22.3) változó értéke *ERANGE* lesz, a visszatérési érték pedig egy nagyon nagy vagy nagyon kicsi szám, a konvertált értéknek megfelelően.

20.4.2. Karakterek osztályozása

A *<ctype.h>* és a *<cctype>* fejállományban a standard könyvtár olyan hasznos függvényeket kínál, melyek az ASCII, illetve a hasonló karakterkészletek karaktereit osztályozzák:

```
int isalpha(int);     // 'a'..'z' 'A'..'Z' betűk, C-hez (§20.2.1, §21.7)
int isupper(int);     // 'A'..'Z' nagybetűk, C-hez (§20.2.1, §21.7)
int islower(int);     // 'a'..'z' kisbetűk, C-hez (§20.2.1, §21.7)
int isdigit(int);     // tízes számrendszerű számok: '0'..'9'
int isxdigit(int);    // hexadecimális számok: '0'..'9' vagy 'a'..'f' vagy 'A'..'F'
int isspace(int);     // ' ', '\t', '\v', kocsivissza, újsor, függőleges tabulátor
int iscntrl(int);     // vezérlőkarakter (ASCII 0..31 és 127)
int ispunct(int);     // központozás, a fentiek egyike sem tartozik bele
int isalnum(int);     // isalpha() | isdigit()
int isprint(int);     // nyomtatható karakterek: ascii ' '..'~'
int isgraph(int);     // isalpha() | isdigit() | ispunct()

int toupper(int c);   // c nagybetűs megfelelője
int tolower(int c);   // c kisbetűs megfelelője
```

A fentieket általában egyszerű kiolvasással valósítják meg, úgy, hogy a karaktert indexként használják egy táblában, amely a karakterek jellemzőit tárolja. Ezért az alábbi kifejezés amellett, hogy túlságosan hosszú és sok hibalehetőséget rejt magában (például egy EBCDIC karakterkészletet használó rendszerben nem alfabetikus karakterek is teljesítik a feltételt), nem is hatékony:

```
if (('a'<=c && c<='z') || ('A'<=c && c<='Z')) { // betű
  // ...
}
```

Ezek a függvények egy *int* paramétert várnak, és az átadott egésznek vagy ábrázolhatónak kell lennie egy *unsigned char* típussal vagy az *EOF* értéknek kell lennie (ami általában a *–1*). Ez az értelmezés problémát jelenthet az olyan rendszerekben, ahol a *char* előjeles típus (lásd: §20.6[11]).

Ugyanilyen szerepű függvények rendelkezésre állnak széles karakterekre is, a *<cwtype>* és a *<wtype.h>* fejállományban.

20.5. Tanácsok

[1] A C stílusú függvények helyett lehetőleg használjuk a *string* osztály eljárásait. §20.4.1.
[2] A *string* lehet változó vagy tag, de bázisosztálynak nem való. §20.3, §25.2.1.
[3] A karakterláncokat érdemes érték szerint átadni és visszaadni, mert így a rendszerre bízzuk a memóriakezelést. §20.3.6.
[4] A bejárók és a *[]* operátor helyett használjuk az *at()* függvényt, ha tartományellenőrzésre van szükségünk. §20.3.2, §20.3.5.
[5] A bejárók és a *[]* operátor gyorsabb, mint az *at()* függvény. §20.3.2, §20.3.5.
[6] Közvetve vagy közvetlenül, a részláncok olvasásához használjuk a *substr()*, írásukhoz a *replace()* függvényt. §20.3.12, §20.3.13.
[7] Adott érték megkereséséhez egy karakterláncban használjuk a *find()* függvényt (és ne írjunk saját ciklust erre a feladatra). §20.3.1.
[8] Ha hatékonyan akarunk egy karakterláncot karakterekkel bővíteni, használjuk a hozzáfűzés műveleteit. §20.3.9.
[9] Ha az idő nem létfontosságú szempont, a karakterbevitelre használjunk *string* objektumokat. §20.3.1.5.
[10] Használjuk a *string::npos* értéket a „karakterlánc hátralévő részének” jelzésére. §20.3.5.
[11] Ha gyakran használunk egy alacsonyszintű szolgáltatást, akkor azt készítsük el a *string* osztályhoz, ne használjunk emiatt mindenhol alacsonyszintű adatszerkezeteket. §20.3.10.

[12] Ha a *string* osztályt használjuk, valahol mindig kapjuk el a *range_error* és *out_of_range* kivételeket. §20.3.5.
[13] Figyeljünk rá, hogy ne adjunk át *char** mutatót *0* értékkel egy karakterlánc-kezelő függvénynek. §20.3.7.
[14] Ha nagyon nagy szükségünk van rá, a *c_str()* függvény segítségével egy *string* objektumból előállíthatjuk a C stílusú karakterlánc megfelelőjét. §20.3.7.
[15] Használjuk az *isalpha()*, *isdigit()* stb. függvényeket, ha a karaktereket osztályoznunk kell, ne írjunk saját ellenőrzéseket a karakterértékek alapján. §20.4.2.

20.6. Gyakorlatok

A fejezet feladatainak többségéhez a standard könyvtár bármely változatának forráskódjában találhatunk megoldást, mielőtt azonban megnéznénk, hogy a könyvtár alkotói hogyan közelítették meg a problémát, próbáljunk saját megoldást keresni.

1. (*2) Készítsünk egy függvényt, amely két *string* objektumot vár paraméterként és egy újabb karakterláncot ad vissza, amelyben a két karakterlánc összefűzése szerepel úgy, hogy közöttük egy pont áll. A *file* és a *write* karakterláncokból például a függvény a *file.write* eredményt állítsa elő. Oldjuk meg ugyanezt a feladatot C stílusú karakterláncokkal és a C lehetőségeinek (például a *malloc()* és az *strlen()* függvények) felhasználásával. Hasonlítsuk össze a két függvényt. Milyen fontos összehasonlítási szempontokat kell megvizsgálnunk?
2. (*2) Foglaljuk össze egy listában a *vector* és a *basic_string* osztály közötti különbségeket. Mely különbségek igazán fontosak?
3. (*2) A karakterlánc-kezelő lehetőségek nem teljesen szabályosak. Egy karaktert (*char*) például értékül adhatunk egy karakterláncnak, de kezdeti értékadásra nem használhatjuk. Készítsünk egy listát ezekről a szabálytalanságokról. Melyeket küszöbölhettük volna ki ezek közül anélkül, hogy a karakterláncok használatát túlbonyolítottuk volna? Milyen más szabálytalanságokat eredményezne ez az átalakítás?
4. (*1.5) A *basic_string* osztálynak nagyon sok tagfüggvénye van. Melyeket emelhetnénk ki az osztályból és tehetnénk nem tag függvénnyé ezek közül anélkül, hogy a hatékonyságot és kényelmes használhatóságot feladnánk?
5. (*1.5) Írjuk meg a *back_inserter()* (§19.2.4) azon változatát, amely a *basic_string* osztállyal működik.
6. (*2) Fejezzük be a §20.3.13 pontban bemutatott *Basic_substring* osztály megvalósítását és építsük be ezt egy olyan *String* típusba, amely túlterheli a *()* operátort a „részlánca” jelentéssel, de egyébként ugyanúgy viselkedik, mint a *string*.

7. (*2.5) Készítsünk egy *find()* függvényt, amely egy egyszerű szabályos (reguláris) kifejezés első előfordulását keresi meg egy *string* objektumban. A kifejezésben a *?* egyetlen, tetszőleges karaktert jelent, a * akárhány olyan karaktert, amely a kifejezés további részére nem illeszthető, az *[abc]* azon karakterek bármelyikét, amely a megadott halmazban szerepel (esetünkben *a*, vagy *b*, vagy *c*). A többi karakter önmagát ábrázolja. A *find(s,"name:")* például egy olyan mutatót ad vissza, amely a *name:* első előfordulására mutat, míg a *find(s,"[nN]ame(*)")* kifejezés azt a részláncot jelöli ki az *s* karakterláncban, amelynek elején a *name* vagy a *Name* szó szerepel, majd zárójelek között egy (esetleg üres) karaktersorozat.
8. (*2.5) Gondolkodjunk el rajta, hogy az előbbi (§20.6[7]), reguláris kifejezés-illesztő függvényünkből milyen lehetőségek hiányoznak még. Határozzuk meg és valósítsuk meg ezeket. Hasonlítsuk össze az általunk létrehozott függvény kifejezőképességét egy széles körben elterjedt reguláris kifejezés-illesztő kifejezőképességével. Hasonlítsuk össze a két eljárást hatékonyság szempontjából is.
9. (*2.5) Egy reguláris kifejezés-könyvtár segítségével készítsünk mintaillesztő műveleteket egy *String* osztályban, amelyhez egy *Substring* osztály is tartozik.
10. Képzeljünk el egy „ideális" osztályt az általános szöveg-feldolgozási feladatok elvégzéséhez. Legyen az osztály neve *Text*. Milyen lehetőségeket kell megvalósítanunk? Milyen korlátozásokat és költségeket kényszerítenek az osztályra az általunk elképzelt „ideális" műveletek?
11. (*1.5) Határozzuk meg az *isalpha()*, *isdigit()* stb. függvények túlterheléseit úgy, hogy megfelelően működjenek *char*, *unsigned char* és *signed char* típusra is.
12. (*2.5) Készítsünk egy *String* osztályt, amelyet arra optimalizálunk, hogy nyolc karakternél rövidebb karakterláncokat kezeljen. Hasonlítsuk össze ennek hatékonyságát a §11.12 pont *String* osztályával, illetve a szabványos *string* osztállyal. Készíthető-e olyan karakterlánc osztály, amely egyesíti a nagyon rövid karakterláncokra optimalizált és a tökéletesen általános változat előnyeit?
13. (*2) Mérjük le egy *string* másolásának hatékonyságát. Megfelelően optimalizált saját fejlesztőrendszerünk *string* osztálya a másolásra?
14. (*2.5) Hasonlítsuk össze a §20.3.9 és a §20.3.10 pontban bemutatott három *complete_name()* függvény hatékonyságát. Próbáljuk elkészíteni a *complete_name()* függvény azon változatát, amely a lehető leggyorsabban fut. Jegyezzük fel azokat a hibákat, amelyeket a megvalósítás és tesztelés közben tapasztaltunk.
15. (*2.5) Képzeljük el, hogy rendszerünk legkényesebb művelete a közepes hosszúságú (5-25 karakteres) karakterláncok beolvasása a *cin* adatfolyamról. Írjunk egy bemeneti függvényt, amely az ilyen karakterláncokat tudja beolvasni olyan gyorsan, ahogy csak el tudjuk képzelni. Dönthetünk úgy, hogy a függ-

vény felületének kényelmes használatát feláldozzuk a sebesség érdekében. Hasonlítsuk össze eljárásunk hatékonyságát a *string* osztály >> operátorának hatékonyságával.

16. (*1.5) Írjuk meg az *itos(int)* függvényt, amely a paraméterként megadott *int* érték *string* ábrázolását adja vissza.

21

Adatfolyamok

„Csak azt kapod, amit látsz"
(Brian Kernighan)

Kimenet és bemenet • Kimeneti adatfolyamok • Beépített típusok kimenete • Felhasználói típusok kimenete • Virtuális kimeneti függvények • Bemeneti adatfolyamok • Beépített típusok bemenete • Formázatlan bemenet • Adatfolyamok állapota • Felhasználói típusok bemenete • I/O kivételek • Adatfolyamok összekötése • Őrszemek • Egész és lebegőpontos kimenet formázása • Mezők és igazítás • Módosítók • Szabványos módosítók • Felhasználói módosítók • Fájlfolyamok • Adatfolyamok lezárása • Karakterlánc-folyamok • Adatfolyamok és átmeneti tárolók • Helyi sajátosságok • Adatfolyam-visszahívások • *printf()* • Tanácsok • Gyakorlatok

21.1. Bevezető

Egy programozási nyelvben általános be- és kimeneti (input/output, I/O) rendszert megvalósítani rendkívül nehéz. Régebben az I/O lehetőségek csak arra korlátozódtak, hogy néhány beépített adattípust kezeljenek. A legegyszerűbbek kivételével azonban a C++ programok számos felhasználói típust is használnak, így e típusok ki- és bemenetét is meg kell oldani. Egy I/O rendszernek a rugalmasság és hatékonyság mellett egyszerűnek, kényel-

mesnek, biztonságosnak és mindenekfelett átfogónak kell lennie. Eddig még senki nem állt elő olyan megoldással, amellyel mindenki elégedett lenne, ezért lehetővé kell tennünk, hogy a felhasználó saját ki- és bemeneti eszközöket készíthessen, illetve a szabványos lehetőségeket egyedi felhasználási területek kezelésével bővíthesse ki.

A C++ olyan nyelv, melyben a felhasználó új típusokat adhat meg és e típusok használata ugyanolyan hatékony és kényelmes lehet, mint a beépített adattípusoké. Ezért logikus elvárás a C++ nyelv I/O szolgáltatásaitól, hogy C++ nyelven készüljenek és csak olyan eszközöket használjanak, melyek minden programozó számára elérhetők. Az itt bemutatott, adatfolyam-bemenetet és -kimenetet kezelő lehetőségek az e követelmények kielégítésére tett erőfeszítések eredményei:

§21.2 *Kimenet*: A kimenet alatt az alkalmazásprogramozó azt érti, hogy tetszőleges típusú (*int*, *char** vagy *Employee_record*) objektumokból karaktersorozatot állíthat elő. Ebben a pontban azokat a lehetőségeket ismertetjük, melyekkel a beépített és a felhasználói típusok kimenete megvalósítható.

§21.3 *Bemenet*: Azok az eszközök, melyekkel karaktereket, karakterláncokat vagy más (akár beépített, akár felhasználói) típusok értékeit beolvashatjuk.

§21.4 *Formázás*: A kimenet elrendezésével kapcsolatban gyakran különleges elvárásaink vannak. Az *int* értékeket például általában tízes számrendszerben szeretjük kiírni, a mutatók hagyományos formája a hexadecimális (tizenhatos számrendszerű) alak, a lebegőpontos számokat pedig megadott pontossággal kell megjeleníteni. Ez a rész a formázással és az azt segítő programozási eszközökkel foglalkozik.

§21.5 *Fájlok és adatfolyamok*: Alapértelmezés szerint minden C++ program használhatja a szabványos adatfolyamokat: a szabványos kimenetet (*cout*), a szabványos bemenetet (cin) és a hibakimenetet (*cerr*). Ha más eszközöket vagy fájlokat akarunk használni, akkor adatfolyamokat kell létrehoznunk és ezeket hozzá kell kötnünk a megfelelő eszközhöz, illetve fájlhoz. Ez a rész a fájlok megnyitásának és bezárásának, illetve az adatfolyamok fájlokhoz és karakterláncokhoz való hozzákötésének módszereivel foglalkozik.

§21.6 *Átmeneti tárolás*: Hatékony ki- és bemenetet úgy lehet megvalósítani, ha átmeneti tárakat („puffereket") használunk. A módszer alkalmazható mindkét oldalon, így az átmeneti tárból kiolvashatjuk és oda írhatjuk is az adatokat. Ebben a pontban az átmeneti tárolás („pufferelés") alapmódszereit mutatjuk be.

§21.7 *Helyi sajátosságok*: A *locale* objektum azt határozza meg, hogy a számokat általában milyen formában jelenítjük meg, milyen karaktereket tekintünk betűnek és így tovább. Tehát az adott kultúrára jellemző „specialitá-

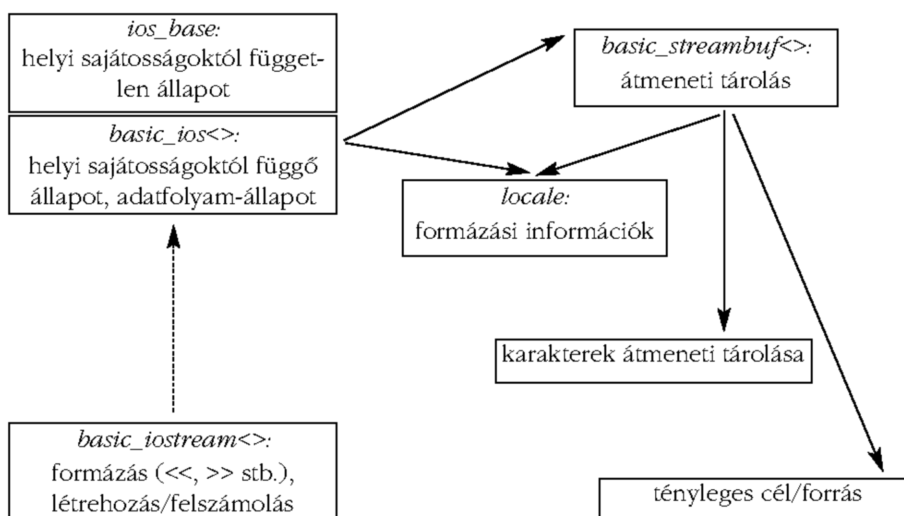
sokat" foglalja össze. Az I/O rendszer automatikusan használja ezeket az információkat, így ez a rész csak nagy vonalakban foglalkozik a témával.

§21.8 *C stílusú I/O*: Itt a C *<stdio.h>* fejállományának *printf()* függvényét és a C könyvtárának a C++ *<iostream>* könyvtárához fűződő viszonyát mutatjuk be.

Ahhoz, hogy az adatfolyam-könyvtárat használjuk, nem kell értenünk a megvalósításához használt összes módszert, már csak azért sem, mert a különböző C++-változatokban különböző módszereket alkalmazhatnak. Egy szép I/O rendszer megvalósítása azonban komoly kihívást jelent. Megírása közben számos olyan lehetőséggel megismerkedhetünk, amelyet más programozási és tervezési feladatok megoldásában is felhasználhatunk. Ezért nagyon fontos, hogy megismerjük azokat a módszereket, melyekkel egy ki- és bemeneti rendszer elkészíthető.

Ebben a fejezetben csak addig a mélységig mutatjuk be az adatfolyamok bemeneti/kimeneti rendszerét, hogy pontosan megértsük annak szerkezetét, felhasználhassuk a leggyakoribb I/O műveletek végrehajtásához, és ki tudjuk egészíteni úgy, hogy az általunk létrehozott új típusokat is képes legyen kezelni. Ha szabványos vagy új típusú adatfolyamokat akarunk használni, esetleg saját helyi sajátosságokat akarunk meghatározni, a könyvtár forráskódjára, jó rendszerleírásra, és működő példaprogramokra is szükségünk lesz az itt leírt információkon kívül.

Az adatfolyam I/O rendszer összetevőit az alábbi ábrával szemléltethetjük:



A *basic_iostream<>* felől induló pontozott nyíl azt jelzi, hogy a *basic_ios<>* egy virtuális bázisosztály, az egyszerű vonalak mutatókat ábrázolnak. Azon osztályok, melyek neve után a <> jel szerepel, olyan sablonok, melyek paramétere egy karaktertípus és tartalmaznak egy *locale* objektumot is.

Az adatfolyamok (streams) és az hozzájuk használt általános jelrendszer számos kommunikációs probléma megoldását lehetővé teszik. Adatfolyamokat használhatunk objektumok gépek közötti átvitelére (§25.4.1), üzenetfolyamok titkosításához (§21.10[22]), adattömörítéshez, objektumok állandó (perzisztens) tárolásához és így tovább. Ennek ellenére az alábbiakban csak az egyszerű, karakterközpontú ki- és bemenetet mutatjuk be.

Az adatfolyam ki- és bemenethez kapcsolódó osztályok és sablonok deklarációját (melyek elég információt adnak ahhoz, hogy hivatkozzunk rájuk, de ahhoz nem, hogy műveleteket hajtsunk végre rajtuk), valamint a szabványos típus-meghatározásokat (*typedef*-eket) az *<iosfwd>* fejállományban találhatjuk meg. Erre a fájlra néha szükségünk lesz, amikor más I/O fejállományokat – de nem az összeset – használni akarunk.

21.2. Kimenet

A beépített és a felhasználói típusok egységes és típusbiztos kezelése egyszerűen megvalósítható, ha túlterheljük a kimeneti függvényeket:

```
put(cerr,"x = "); // cerr a hibaüzenetek kimeneti adatfolyama
put(cerr,x);
put(cerr,'\n');
```

A paraméter típusa határozza meg, melyik *put()* függvény kerül meghívásra az adott helyzetben. Ezt a megoldást sok nyelv alkalmazza, bár sok ismétlést kíván. Ha az *„írd ki”* művelet megvalósításához a << operátort terheljük túl, szebb jelölést kapunk és lehetőséget adunk a programozónak, hogy objektumok sorozatát egyetlen utasítással írja ki:

```
cerr << "x = " << x << '\n';
```

Ha *x* egy *int* típusú változó, melynek értéke *123*, akkor az előző parancs a szabványos hibakimeneti adatfolyamra (*cerr*) az alábbi szöveget írja ki (egy újsor karakterrel lezárva):

```
x = 123
```

Ugyanakkor, ha *x* típusa *complex* (§22.5), értéke pedig *(1,2.4)*, a kiírt szöveg az alábbi lesz:

```
x = (1,2.4)
```

Ez a stílus mindaddig működőképes, amíg *x* típusára a << művelet definiált, márpedig a felhasználó könnyedén készíthet ilyen függvényt saját típusaihoz.

Ha el akarjuk kerülni a kimeneti függvény használatából eredő kényelmetlen megfogalmazást, mindenképpen szükségünk van egy kimeneti operátorra. De miért pont a << operátort válasszuk? Arra nincs lehetőségünk, hogy új nyelvi elemet vezessünk be (§11.2). Az értékadó operátor használható lenne a ki- és a bemenet jelölésére is, de úgy tűnt, a legtöbb programozó különböző operátort szeretne használni a kimenethez és a bemenethez. Ráadásul az = rossz irányba köt: a *cout=a=b* jelentése *cout=(a=b)*, míg nekünk a *(cout=a)=b* értelmezésre lenne szükségünk (§6.2). Próbálkoztam a < és a > operátor használatával is, de az emberek tudatában ez a két jel annyira a „kisebb, mint” és „nagyobb, mint” jelentéshez kapcsolódik, hogy az új I/O utasítások teljesen olvashatatlanok voltak.

A << és a >> operátort nem használjuk olyan gyakran a beépített típusokra, hogy ez problémát okozzon programjaink olvashatóságában. Elég szimmetrikusak ahhoz, hogy jelképezzék a „kimenet” és a „bemenet” műveletet. Amikor ezeket az operátorokat ki- és bemenetre használjuk, a << jelenti a kiírást, a >> pedig a beolvasást. Azok, akik jobban szeretik a műszaki kifejezéseket, nevezhetik ezt a két műveletet sorrendben *output*-nak és *input*-nak, esetleg *beszúrásnak* és *kinyerésnek* (inserter, extractor). A << operátornak a precedencia sorrendben elfoglalt helye elég alacsony ahhoz, hogy zárójelezés nélkül lehetővé tegye a paraméterekben aritmetikai műveletek elvégzését:

```
cout << "a*b+c=" << a*b+c << '\n';
```

Ha azonban olyan műveletet akarunk használni, amely precedenciája alacsonyabban helyezkedik el, mint a << operátor, akkor nem szabad elfelejtenünk a zárójeleket:

```
cout << "a^b|c=" << (a^b|c) << '\n';
```

A balra léptető operátor (§6.2.4) szintén használható kimeneti utasításban, de természetesen ezt is zárójelek közé kell írnunk:

```
cout << "a<<b=" << (a<<b) << '\n';
```

21.2.1. Kimeneti adatfolyamok

Az *ostream* arra való, hogy tetszőleges típusú értékeket karakterek sorozatává alakítsunk. Ezután ezeket a karaktereket általában alacsonyszintű kimeneti műveletekkel írjuk ki. Sokféle karakter létezik (§20.2) és ezeket egy-egy *char_traits* objektum jellemzi (§20.2.1). Ebből következik, hogy az *ostream* egy bizonyos típusú karakterre „specializált" változata a *basic_ostream* sablonnak:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
  virtual ~basic_ostream();
  // ...
};
```

A sablon és a hozzá tartozó műveletek az *std* névtérben szerepelnek és az *<ostream>* fejállományból érhetők el, amely az *<iostream>* fejállomány kimenettel kapcsolatos részeit tartalmazza.

A *basic_ostream* paraméterei meghatározzák, hogy a megvalósítás milyen típusú karaktereket használ, de nem rögzítik a kiírható objektumok típusát. Az egyszerű *char* típust és a széles karaktereket használó adatfolyamokat viszont minden változat közvetlenül támogatja:

```
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
```

Sok rendszerben a széles karakterek (wide character) írása olyan szinten optimalizálható a *wostream* osztály segítségével, amelyhez a kimeneti egységként bájtot használó adatfolyamok nehezen érhetnek fel.

Lehetőség van olyan adatfolyamok meghatározására is, melyekben a fizikai ki- és bemenetet nem karakterszinten valósítjuk meg, ezek az adatfolyamok azonban meghaladják a standard könyvtár hatókörét, így ebben a könyvben nem foglalkozunk velük (§21.10[15]).

A *basic_ios* bázisosztály az *<ios>* fejállományban található. Ez kezeli a formázást (§21.4), a helyi sajátosságokat (§21.7) és az átmeneti tárak elérését (§21.6) is. Az egységes jelölésrendszer érdekében néhány típust is meghatároz:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef typename Tr::int_type int_type;      // a karakter egész értékének típusa
    typedef typename Tr::pos_type pos_type;      // pozíció az átmeneti tárban
    typedef typename Tr::off_type off_type;      // eltolás az átmeneti tárban

    // ... lásd még §21.3.3, §21.3.7, §21.4.4, §21.6.3 és §21.7.1. ...
};
```

A *basic_ios* osztály letiltja a másoló konstruktort és az értékadó operátort. Ebből következik, hogy az *ostream* és *istream* objektumok nem másolhatók. Ha egy adatfolyam célja megváltozik, kénytelenek vagyunk vagy az átmeneti tárat kicserélni (§21.6.4) vagy mutatókat használni (§6.1.7).

Az *ios_base* bázisosztály olyan információkat és műveleteket tartalmaz, amelyek függetlenek a használt karaktertípustól. (Ilyen például a lebegőpontos számok kimeneti pontossága.) Ezért ez az osztály nem kell, hogy sablon legyen.

Az *ios_base* osztályban szereplő típus-meghatározásokon kívül az adatfolyam I/O könyvtár egy előjeles egész típust is használ, melynek neve *streamsize* és az egy I/O művelettel átvitt karakterek számát, illetve az átmeneti tár méretét adhatjuk meg a segítségével. Emellett rendelkezésünkre áll a *streamoff* is, amely az adatfolyamokban és az átmeneti tárakban való eltolást (offset) ábrázolja.

Az *<iostream>* fejállomány több szabványos adatfolyamot is megad:

```
ostream cout;        // karakterek szabványos kimeneti adatfolyama
ostream cerr;        // hibaüzenetek szabványos, nem pufferelt kimeneti adatfolyama
ostream clog;        // hibaüzenetek szabványos kimeneti adatfolyama

wostream wcout;      // a cout-nak megfelelő "széles" adatfolyam
wostream wcerr;      // a cerr-nek megfelelő "széles" adatfolyam
wostream wclog;      // a clog-nak megfelelő "széles" adatfolyam
```

A *cerr* és a *clog* adatfolyamok ugyanarra a kimeneti eszközre mutatnak, a különbség köztük csak a használt átmeneti tár típusa. A *cout* ugyanarra az eszközre ír, mint a C *stdout* (§21.8), míg a *cerr* és a *clog* ugyanarra, mint a C *stderr*. Ha szükség van rá, a programozó további adatfolyamokat is létrehozhat (§21.5).

21.2.2. Beépített típusok kimenete

Az *ostream* osztály megadja a << („kimenet") operátort, amellyel a beépített típusok kiírását végezhetjük el:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
  // ...

  basic_ostream& operator<<(short n);
  basic_ostream& operator<<(int n);
  basic_ostream& operator<<(long n);

  basic_ostream& operator<<(unsigned short n);
  basic_ostream& operator<<(unsigned int n);
  basic_ostream& operator<<(unsigned long n);

  basic_ostream& operator<<(float f);
  basic_ostream& operator<<(double f);
  basic_ostream& operator<<(long double f);

  basic_ostream& operator<<(bool n);
  basic_ostream& operator<<(const void* p);         // mutatóérték kiírása

  basic_ostream& put(Ch c); // c kiírása
  basic_ostream& write(const Ch* p, streamsize n);  // p[0]..p[n-1]

  // ...
};
```

A *put()* és *write()* függvények egyszerűen karaktereket írnak ki, így az erre a műveletre szolgáló << operátornak nem kell tagnak lennie. A karakter-operandust váró *operator<<()* függvények a *put()* segítségével nem tagként készíthetők el:

```
template <class Ch, class Tr>
  basic_ostream<Ch, Tr>& operator<<(basic_ostream<Ch, Tr>&, Ch);
template <class Ch, class Tr>
  basic_ostream<Ch, Tr>& operator<<(basic_ostream<Ch, Tr>&, char);
template <class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<char, Tr>&, char);
template <class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<char, Tr>&, signed char);
template <class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<char, Tr>&, unsigned char);
```

A << a nullával lezárt karaktertömbök kiírásához is rendelkezésre áll:

```
template <class Ch, class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<Ch, Tr>&, const Ch*);
template <class Ch, class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<Ch, Tr>&, const char*);
template <class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<char, Tr>&, const char*);
template <class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<char, Tr>&, const signed char*);
template <class Tr>
  basic_ostream<char, Tr>& operator<<(basic_ostream<char, Tr>&, const unsigned char*);
```

A *string*-ek kimeneti operátorait a *<string>* tartalmazza (§20.3.15).

Az *operator <<()* művelet egy referenciát ad vissza ugyanarra az *ostream* objektumra, amelyre meghívtuk, így azonnal alkalmazhatjuk rá a következő *operator <<()* műveletet:

```
cerr << "x = " << x;
```

Itt *x* egy *int*, az utasítás jelentése pedig a következő:

```
(cerr.operator<<("x = ")).operator<<(x);
```

Fontos következmény, hogy amikor több elemet íratunk ki egyetlen kimeneti utasítással, azok a megfelelő sorrendben, balról jobbra haladva jelennek meg:

```
void val(char c)
{
  cout << "int('" << c << "') = " << int(c) << '\n';
}

int main()
{
  val('A');
  val('Z');
}
```

Egy ASCII karaktereket használó rendszerben a fenti programrészlet eredménye a következő lesz:

```
int('A') = 65
int('Z') = 90
```

Megfigyelhetjük, hogy a karakterliterálok típusa *char* (§4.3.1), így a *cout<<'Z'* a *Z* betűt fogja kiírni, nem annak *int* értékét, a *90*-et.

Ha logikai (*bool*) értéket íratunk ki, az alapértelmezés szerint *0* vagy *1* formában jelenik meg. Ha ez nem megfelelő számunkra, beállíthatjuk az *<iomanip>* fejállományban (§21.4.6.2) meghatározott *boolalpha* formázásjelzőt, így a *true* vagy a *false* szöveg jelenik meg:

```
int main()
{
  cout << true << ' ' << false << '\n';
  cout << boolalpha;                    // a true és false szimbolikus ábrázolása
  cout << true << ' ' << false << '\n';
}
```

A kiírt szöveg:

```
1 0
true false
```

Pontosabban fogalmazva a *boolalpha* tulajdonság azt biztosítja, hogy a *bool* értékeknek a helyi sajátosságoknak (locale) megfelelő értékét kapjuk. Ha én megfelelően beállítom a *locale*-t (§21.7), a következő eredményt kapom:

```
1 0
sandt falsk
```

A lebegőpontos számok formázásáról, az egészek számrendszeréről stb. a §21.4. pontban lesz szó.

Az *ostream::operator<<(const void*)* függvény egy mutató értékét jeleníti meg, az éppen használt számítógép felépítésének megfelelő formában:

```
int main()
{
  int i = 0;
  int* p = new int;
  cout << "local " << &i << ", free store " << p << '\n';
}
```

Az eredmény az én gépemen a következő:

```
local 0x7fffead0, free store 0x500c
```

Más rendszerek másképpen jeleníthetik meg a mutató értékét.

21.2.3. Felhasználói típusok kimenete

Képzeljük el az alábbi, felhasználói *complex* típust (§11.3):

```
class complex {
public:
  double real() const { return re; }
  double imag() const { return im; }
  // ...
};
```

A << operátort a következőképpen definiálhatjuk az új *complex* számára:

```
ostream& operator<<(ostream&s, const complex& z)
{
  return s << '(' << z.real() << ',' << z.imag() << ')';
}
```

Ezután a << műveletet pontosan ugyanúgy használhatjuk saját típusunkra, mint a beépített típusokra:

```
int main()
{
  complex x(1,2);
  cout << "x = " << x << '\n';
}
```

Az eredmény:

```
x = (1,2)
```

A felhasználói típusok kimeneti műveleteinek megvalósításához nem kell megváltoztatnunk az *ostream* osztály deklarációját. Ez azért szerencsés, mert az *ostream* osztály az <*iostream*> fejállományban definiált, amit a felhasználók jobb ha nem változtatnak meg (er-

re nincs is lehetőségük). Az, hogy az *ostream* bővítését nem tettük lehetővé, védelmet nyújt az adatszerkezet véletlen módosítása ellen is, és lehetővé teszi, hogy anélkül változtassuk meg az *ostream* osztály megvalósítását, hogy a felhasználói programokra hatással lennénk.

21.2.3.1. Virtuális kimeneti függvények

Az *ostream* tagfüggvényei nem virtuálisak. Azok a kimeneti műveletek, melyeket egy programozó valósít meg, nem az osztály részei, így ezek sem lehetnek virtuálisak. Ennek egyik oka az, hogy az egyszerű műveletek esetében (például egyetlen karakter átmeneti tárba írása) így közel optimális sebességet érhetünk el. Ez olyan terület, ahol a futási idejű hatékonyság rendkívül fontos és szinte kötelező helyben kifejtett (inline) eljárásokat készíteni. A virtuális függvények használata it arra vonatkozik, hogy a lehető legnagyobb rugalmasságot biztosítsuk azon műveletek számára, melyek kifejezetten a átmeneti tár túlcsordulásával, illetve alulcsordulásával foglalkoznak (§21.6.4).

Ennek ellenére a programozók gyakran kívánnak megjeleníteni olyan objektumokat, melyeknek csak egy bázisosztálya ismert. Mivel a pontos típust nem ismerjük, a megfelelő kimenetet nem állíthatjuk elő úgy, hogy egyszerűen minden új típushoz megadjuk a << műveletet. Ehelyett készíthetünk egy virtuális kimeneti függvényt az absztrakt bázisosztályban:

```
class My_base {
public:
  // ...

  virtual ostream& put(ostream& s) const = 0;  // *this kiírása s-re
};

ostream& operator<<(ostream& s, const My_base& r)
{
  return r.put(s);     // a megfelelő put() használata
}
```

Tehát a *put()* egy olyan virtuális függvény, amely biztosítja, hogy a megfelelő kimeneti művelet kerüljön végrehajtásra a << operátorban.

Ennek felhasználásával a következő programrészletet írhatjuk.

```
class Sometype : public My_base {
public:
  // ...
```

```
  ostream& put(ostream& s) const;       // az igazi kimeneti függvény: My_base::put()
                                        // felülírása
};

void f(const My_base& r, Sometype& s)   // használjuk a << operátort a
                                        // megfelelő put() hívásához
{
  cout << r << s;
}
```

Ezzel a virtuális *put()* függvény beépül az *ostream* osztály és a << művelet által biztosított keretbe. A módszer jól alkalmazható minden olyan esetben, amikor egy virtuális függvényként működő műveletre van szükségünk, de futási időben a második paraméter alapján akarunk választani.

21.3. Bemenet

A bemenetet a kimenethez nagyon hasonlóan kezelhetjük. Az *istream* osztály biztosítja a >> bemeneti operátort néhány általános típushoz, a felhasználói típusokhoz pedig készíthetünk egy-egy *operator>>()* függvényt.

21.3.1. Bemeneti adatfolyamok

A *basic_ostream* (§21.2.1) osztállyal párhuzamosan az *<istream>* fejállományban megtalálhatjuk a *basic_istream* osztályt, amely az *<iostream>* bemenettel kapcsolatos részeit tartalmazza:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_istream : virtual public basic_ios<Ch,Tr> {
public:
  virtual ~basic_istream();

  // ...
};
```

A *basic_ios* bázisosztályt a §21.2.1. pontban mutattuk be.

A *cin* és a *wcin* két szabványos bemeneti adatfolyam, melyeket az *<iostream>* fejállomány ír le:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

istream cin;         // karakterek szabványos bemeneti adatfolyama
wistream wcin;       // szabványos bemeneti adatfolyam wchar_t részére
```

A *cin* adatfolyam ugyanazt a forrást használja, mint a C *stdin* adatfolyama (§21.8).

21.3.2. Beépített típusok bemenete

A beépített típusokhoz az *istream* biztosítja a >> operátort:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
  // ...
  // formázott bevitel:

  basic_istream& operator>>(short& n);                 // olvasás n-be
  basic_istream& operator>>(int& n);
  basic_istream& operator>>(long& n);

  basic_istream& operator>>(unsigned short& u);        // olvasás u-ba
  basic_istream& operator>>(unsigned int& u);
  basic_istream& operator>>(unsigned long& u);

  basic_istream& operator>>(float& f);                 // olvasás f-be
  basic_istream& operator>>(double& f);
  basic_istream& operator>>(long double& f);

  basic_istream& operator>>(bool& b);                  // olvasás b-be
  basic_istream& operator>>(void*& p);                 // mutatóérték olvasása p-be

  // ...
};
```

Az *operator>>()* függvények a következő stílust követik:

```
istream& istream::operator>>(T& tvar)      // T egy típus, melyre az istream::operator>>
                                           // deklarált
{
  // üreshely karaktereket átlépni,
```

```
    // majd valahogyan beolvasni egy T típusú elemet `tvar'-ba
    return *this;
}
```

Mivel a >> átugorja az üreshely (whitespace) karaktereket, az ilyenekkel elválasztott egészeket az alábbi egyszerű ciklussal olvashatjuk be:

```
int read_ints(vector<int>& v) // feltölti v-t , visszatér a beolvasott egészek számával
{
    int i = 0;
    while (i<v.size() && cin>>v[i]) i++;
    return i;
}
```

Ha a bemeneten egy nem egész érték jelenik meg, a bemeneti művelet hibába ütközik, így a ciklus is megszakad. Például ezt a bemenetet feltételezve:

```
1 2 3 4 5.6 7 8.
```

a *read_ints()* függvény öt egész számot fog beolvasni:

```
1 2 3 4 5
```

A művelet után a bemeneten a következő beolvasható karakter a pont lesz. A nem látható üreshely (whitespace) karakterek meghatározása itt is ugyanaz, mint a szabványos C-ben (szóköz, tabulátor, újsor, függőleges tabulátorra illesztés, kocsivissza), és a *<cctype>* fejállományban levő *isspace()* függvénnyel ellenőrizhetők valamely karakterre (§20.4.2).

Az *istream* objektumok használatakor a leggyakoribb hiba, hogy a bemenet nem egészen abban a formátumban érkezik, mint amire felkészültünk, ezért a bemenet nem történik meg. Ezért mielőtt használni kezdenénk azokat az értékeket, melyeket reményeink szerint beolvastunk, ellenőriznünk kell a bemeneti adatfolyam állapotát (§21.3.3), vagy kivételeket kell használnunk (§21.3.6).

A bemenethez használt formátumot a helyi sajátosságok (locale) határozzák meg (§21.7). Alapértelmezés szerint a logikai értékeket a *0* (hamis) és az *1* (igaz) érték jelzi, az egészeket tízes számrendszerben kell megadnunk, a lebegőpontos számok formája pedig olyan, ahogy a C++ programokban írhatjuk azokat. A *basefield* (§21.4.2) tulajdonság beállításával lehetőség van arra is, hogy a *0123* számot a *83* tízes számrendszerbeli szám oktális alakjaként, a *0xff* bemenetet pedig a *255* hexadecimális alakjaként értelmezzük. A mutatók beolvasásához használt formátum teljesen az adott nyelvi változattól függ (nézzünk utána, saját fejlesztőrendszerünk hogyan működik).

Meglepő módon nincs olyan >> tagfüggvény, mellyel egy karaktert olvashatnánk be. Ennek oka az, hogy a >> operátor a karakterekhez a *get()* karakter-beolvasó függvény (§21.3.4) segítségével könnyen megvalósítható, így nem kell tagfüggvényként szerepelnie. Az adatfolyamokról saját karaktertípusaiknak megfelelő karaktereket olvashatunk be. Ha ez a karaktertípus a *char*, akkor beolvashatunk *signed char* és *unsigned char* típusú adatot is:

```
template<class Ch, class Tr>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, Ch&);

template<class Tr>
basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, unsigned char&);

template<class Tr>
basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, signed char&);
```

A felhasználó szempontjából teljesen mindegy, hogy a >> tagfüggvény vagy önálló eljárás-e.

A többi >> operátorhoz hasonlóan ezek a függvények is először átugorják a bevezető üreshely karaktereket:

```
void f()
{
  char c;
  cin >> c;
  // ...
}
```

Ez a kódrészlet az első nem üreshely karaktert a *cin* adatfolyamból a *c* változóba helyezi.

A beolvasás célja lehet karaktertömb is:

```
template<class Ch, class Tr>
  basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, Ch*);
template<class Tr>
  basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, unsigned char*);
template<class Tr>
  basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, signed char*);
```

Ezek a műveletek is eldobják a bevezető üreshely karaktereket, majd addig olvasnak, amíg egy üreshely karakter vagy fájlvége jel nem következik, végül a karaktertömb végére egy *0* karaktert írnak. Természetesen ez a megoldás alkalmat ad a túlcsordulásra, így érdemesebb inkább egy *string* objektumba (§20.3.15) helyezni a beolvasott adatokat. Ha mégis az előbbi

megoldást választjuk, rögzítsük, hogy legfeljebb hány karaktert akarunk beolvasni a >> operátor segítségével. Az *is.width(n)* függvényhívással azt határozzuk meg, hogy a következő, *is* bemeneti adatfolyamon végrehajtott >> művelet legfeljebb *n-1* karaktert olvashat be:

```
void g()
{
  char v[4];
  cin.width(4);
  cin >> v;
  cout << "v = " << v << endl;
}
```

Ez a programrészlet legfeljebb 3 karaktert olvas be a *v* változóba és a végén elhelyezi a *0* karaktert.

Egy *istream* esetében a *width()* által beállított érték csak az utána következő >> műveletre van hatással, és arra is csak akkor, ha a céltípus egy tömb.

21.3.3. Az adatfolyam állapota

Minden adatfolyam (az *istream* és az *ostream* is) rendelkezik valamilyen *állapottal* (state). A hibákat és a szokatlan állapotokat ezen állapotérték megfelelő beállításával és lekérdezésével kezelhetjük.

Az adatfolyam-állapot (stream state) fogalmát a *basic_istream* bázisosztálya, a *basic_ios* írja le az *<ios>* fejállományban:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
  // ...

  bool good() const;                          // a következő művelet sikerülhet
  bool eof() const;                           // fájlvége következett be
  bool fail() const;                          // a következő művelet sikertelen lesz
  bool bad() const;                           // az adatfolyam sérült

  iostate rdstate() const;                    // io állapotjelző lekérdezése
  void clear(iostate f = goodbit);            // io állapotjelző beállítása
  void setstate(iostate f) { clear(rdstate() | f); }   // az f io állopotjelző beállítása
```

```
    operator void*() const;                         // nemnulla, ha !fail()
    bool operator!() const { return fail(); }

    // ...
};
```

Ha az állapot *good()*, a megelőző bemeneti művelet sikeres volt. Ilyenkor a következő bemeneti művelet helyes végrehajtására is van esély, ellenkező esetben viszont az biztosan nem lesz hibátlan. Ha egy olyan adatfolyamon próbálunk meg bemeneti műveletet végrehajtani, amely nem *good()* állapotban van, akkor semmi sem történik. Ha megpróbálunk a *v* változóba értéket beolvasni, de a bemeneti művelet sikertelen, *v* értékének nem szabad megváltoznia. (Ha *v* olyan típusú, amit az *istream* és az *ostream* tagfüggvényei kezelnek, biztosan megmarad az értéke.) A *fail()* és a *bad()* állapot közötti különbség igen kicsi. Ha az állapot *fail()*, de nem *bad()*, akkor az adatfolyam érvénytelenné vált, de nem vesztek el karakterek. Ha *bad()* állapotba kerültünk, már nem reménykedhetünk.

Az adatfolyam állapotát jelzők (jelzőbitek, flag-ek) határozzák meg. Az adatfolyamok állapotát kifejező legtöbb konstanshoz hasonlóan ezeket is a *basic_ios* bázisosztálya, az *ios_base* írja le:

```
class ios_base {
public:
    // ...

    typedef implementation_defined2 iostate;
    static const iostate badbit,    // az adatfolyam sérült
                         eofbit,    // fájlvége következett be
                         failbit,   // a következő művelet sikertelen lesz
                         goodbit;   // goodbit==0

    // ...
};
```

Az I/O állapot jelzőbitjeit közvetlenül módosíthatjuk:

```
void f()
{
    ios_base::iostate s = cin.rdstate();     // iostate bitek halmazával tér vissza

    if (s & ios_base::badbit) {
            // cin karakterek elveszhettek
    }
    // ...
    cin.setstate(ios_base::failbit);
    // ...
}
```

Ha az adatfolyamot feltételként használjuk, annak állapotát az *operator void*()* vagy az *operator !()* függvénnyel vizsgálhatjuk meg. A vizsgálat csak akkor lesz sikeres, ha az állapot *!fail()* (a *void*()* esetében), illetve *fail()* (a *!()* esetében). Egy általános másoló eljárást például az alábbi formában írhatunk meg.

```
template<class T> void iocopy(istream& is, ostream& os)
{
  T buf;
  while (is>>buf) os << buf << '\n';
}
```

Az *is>>buf* művelet egy referenciát ad vissza, amely az *is* adatfolyamra hivatkozik, a vizsgálatot pedig az *is::operator void*()* művelet végzi:

```
void f(istream& i1, istream& i2, istream& i3, istream& i4)
{
  iocopy<complex>(i1,cout);  // komplex számok másolása
  iocopy<double>(i2,cout);   // kétszeres pontosságú lebegőpontos számok másolása
  iocopy<char>(i3,cout);     // karakterek másolása
  iocopy<string>(i4,cout);   // üreshelyekkel elválasztott szavak másolása
}
```

21.3.4. Karakterek beolvasása

A >> operátor formázott bemenetre szolgál, tehát adott típusú és adott formátumú objektumok beolvasására. Ha erre nincs szükségünk és inkább valóban karakterekként akarjuk beolvasni a karaktereket, hogy később mi dolgozzuk fel azokat, használjuk a *get()* függvényeket:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
  // ...
  // formázatlan bemenet

  streamsize gcount() const;          // a legutóbbi get() által beolvasott karakterek száma

  int_type get();                     // egy Ch (vagy Tr::eof()) beolvasása

  basic_istream& get(Ch& c);          // egy Ch olvasása c-be

  basic_istream& get(Ch* p, streamsize n);              // újsor a lezárójel
  basic_istream& get(Ch* p, streamsize n, Ch term);
```

```
    basic_istream& getline(Ch* p, streamsize n);  // újsor a lezárójel
    basic_istream& getline(Ch* p, streamsize n, Ch term);

    basic_istream& ignore(streamsize n = 1, int_type t = Tr::eof());
    basic_istream& read(Ch* p, streamsize n);        // legfeljebb n karakter beolvasása
    // ...
};
```

Ezek mellett a *<string>* fejállomány biztosítja a *getline()* függvényt is, amely a szabványos *string*-ek (§20.3.15) beolvasására használható.

A *get()* és a *getline()* függvények ugyanúgy kezelik az üreshely karaktereket, mint bármely más karaktert. Kifejezetten olyan adatok beolvasására szolgálnak, ahol nem számít a beolvasott karakterek jelentése.

Az *istream::get(char&)* függvény egyetlen karaktert olvas be paraméterébe. Egy karakterenként másoló programot például a következőképpen írhatunk meg.

```
int main()
{
    char c;
    while(cin.get(c)) cout.put(c);
}
```

A háromparaméterű *s.get(p,n,term)* legfeljebb *n-1* karaktert olvas be a *p[0]*,…, *p[n-2]* tömbbe. A *get()* az átmeneti tárban mindenképpen elhelyez egy *0* karaktert a beolvasott karakterek után, így a *p* mutatónak egy legalább *n* karakter méretű tömbre kell mutatnia. A harmadik paraméter (*term*) az olvasást lezáró karaktert határozza meg. A háromparaméterű *get()* leggyakoribb felhasználási módja az, hogy beolvasunk egy „sort” egy rögzített méretű tárba és később innen használjuk fel karaktereit:

```
void f()
{
    char buf[100];
    cin >> buf;                 // gyanús: túlcsordulhat
    cin.get(buf,100,'\n');      // biztonságos
    // ...
}
```

Ha a *get()* megtalálja a lezáró karaktert, az adatfolyamban hagyja, tehát a következő bemeneti művelet ezt a karaktert kapja meg elsőként. Soha ne hívjuk meg újra a *get()* függvényt a lezáró karakter eltávolítása nélkül. Ha egy *get()* vagy *getline()* függvény egyetlen karaktert sem olvas és távolít el az adatfolyamról meghívódik *setstate(failbit)*, így a következő beolvasások sikertelenek lesznek (vagy kivétel váltódik ki, 21.3.6).

```
void subtle_error
{
  char buf[256];

  while (cin) {
        cin.get(buf,256);   // a sor beolvasása
        cout << buf;        // a sor kiírása.
        // Hoppá: elfelejtettük eltávolítani cin-ről '\n'-t, a következő get() sikertelen lesz
  }
}
```

Ez a példa jól szemlélteti, miért érdemes a *get()* helyett a *getline()* függvényt használnunk. A *getline()* ugyanis pontosan úgy működik, mint a megfelelő *get()*, de a lezáró karaktert is eltávolítja az adatfolyamból:

```
void f()
{
  char word[MAX_WORD][MAX_LINE];     // MAX_WORD darab tömb,
                                     // mindegyik MAX_LINE karaktert tartalmaz
  int i = 0;
  while(cin.getline(word[i++],MAX_LINE,'\n') && i<MAX_WORD);
  // ...
}
```

Ha nem a hatékonyság a legfontosabb, érdemes *string* (§3.6, §20.3.15) objektumba olvasnunk, mert azzal elkerülhetjük a leggyakoribb túlcsordulási, illetve helyfoglalási problémákat. A *get()*, a *getline()* és a *read()* függvényre viszont az ilyen magas szintű szolgáltatások megvalósításához szükség van. A nagyobb sebesség ára a viszonylag kusza felület, de legalább nem kell újra megvizsgálnunk a bemeneti adatfolyamot, ahhoz, hogy megtudjuk, mi szakította meg a beolvasási műveletet; pontosan korlátozhatjuk a beolvasandó karakterek számát és így tovább.

A *read(p,n)* függvény legfeljebb *n* karaktert olvas be a *p[0]*, …, *p[n-1]* tömbbe. A *read()* nem foglalkozik a bemeneten megjelenő lezáró karakterekkel és az eredménytömb végére sem helyez *0* karaktert. Ezért képes tényleg *n* karaktert beolvasni (és nem csak *n-1*-t). Tehát a *read()* függvény egyszerűen karaktereket olvas, és nem próbálja az eredményt C stílusú karakterlánccá alakítani.

Az *ignore()* függvény ugyanúgy karaktereket olvas, mint a *read()*, de egyáltalán nem tárolja azokat. Egy másik hasonlóság a *read()* függvénnyel, hogy tényleg képes *n* (és nem *n-1*) darab karakter beolvasására. Az *ignore()* alapértelmezés szerint egy karaktert olvas be, tehát ha paraméterek nélkül hívjuk meg, akkor jelentése „dobd el a következő karaktert”.

A *getline()* függvényhez hasonlóan itt is megadhatunk egy lezáró karaktert, amit eltávolít a bemeneti adatfolyamból, ha beleütközik. Az *ignore()* alapértelmezett lezáró karaktere a fájlvége jel. Ezeknél a függvényeknél nem világos azonnal, hogy mi állította meg az olvasást, és néha még arra is nehéz emlékezni, melyik olvasási műveletnél mi volt a leállási feltétel. Azt viszont mindig meg tudjuk kérdezni, hogy elértük-e már a fájlvége jelet (§21.3.3). Hasonló segítség, hogy a *gcount()* függvénnyel megállapíthatjuk, hogy a legutóbbi formázatlan bemeneti eljárás hány karaktert olvasott be az adatfolyamból:

```
void read_a_line(int max)
{
  // ...
  if (cin.fail()) {                        // Hoppá: hibás bemeneti formátum
          cin.clear();                     // a bemeneti jelzőbitek törlése (§21.3.3)
          cin.ignore(max,';');             // ugrás a pontosvesszőre

          if (!cin) {
                  // hoppá: elértük az adatfolyam végét
          }
          else if (cin.gcount()==max) {
                  // hoppá: max számú karaktert beolvastunk
          }
          else {
                  // megtaláltuk és eldobtuk a pontosvesszőt
          }
  }
}
```

Sajnos ha a megengedett legtöbb karaktert olvastuk be, nincs módunk annak megállapítására, hogy megtaláltuk-e a lezáró karaktert (utolsó beolvasott karakterként).

A paraméter nélküli *get()* a <*cstdio*> fejállományban szereplő *getchar()* függvény (§21.8) <*iostream*>-beli megfelelője. Egyszerűen beolvas egy karaktert és visszaadja annak számértékét. Ezzel a megoldással semmit nem kell feltételeznie az éppen használt karaktertípusról. Ha nincs beolvasható karakter a *get()* adatfolyamában, akkor a megfelelő fájlvége jelet (tehát a *traits_type::eof()* karaktert) adja vissza, majd beállítja az *istream* objektum *eof* állapotát (§21.3.3):

```
void f(unsigned char* p)
{
  int i;
  while((i = cin.get()) && i!=EOF) {
          *p++ = i;
          // ...
  }
}
```

Az *EOF* az *eof()* függvény által visszaadott érték a *char* típus szokásos *char_traits* osztályából. Az *EOF* értéket az *<iostream>* fejállomány határozza meg. Tehát e ciklus helyett nyugodtan írhattuk volna a *read(p, MAX_INT)* utasítást, de tegyük fel, hogy explicit ciklust akartunk írni, mert, mondjuk, minden beolvasott karaktert egyesével akartunk látni. A C nyelv egyik legnagyobb erősségének tartják azt a lehetőséget, hogy karaktereket olvashatunk be, és úgy dönthetünk, hogy semmit sem csinálunk velük, ráadásul mindezt gyorsan tehetjük. Ez tényleg fontos és alábecsült erősség, így a C++ igyekszik ezt megtartani.

A *<cctype>* szabványos fejállomány sok olyan függvényt biztosít, amelynek hasznát vehetjük a bemenet feldolgozásában (§20.4.2). Az *eatwhite()* függvény például, amely az üreshely karaktereket törli az adatfolyamból, a következőképpen definiálható:

```
istream& eatwhite(istream& is)
{
  char c;
  while (is.get(c)) {
          if (!isspace(c)) {          // c üreshely karakter?
                  is.putback(c);      // c visszarakása a bemenet átmeneti tárába
                  break;
          }
  }
  return is;
}
```

Az *is.putback(c)* függvényhívásra azért van szükség, hogy a *c* legyen a következő karakter, amit az *is* adatfolyamból beolvasunk (§21.6.4).

21.3.5. Felhasználói típusok beolvasása

A felhasználói típusok beolvasását pontosan ugyanúgy lehet megvalósítani, mint kimeneti műveleteiket. Az egyetlen különbség, hogy a bemeneti műveletek esetében a második paraméternek nem-konstans referencia típusúnak kell lennie:

```
istream& operator>>(istream& s, complex& a)
/*
  a complex típus lehetséges bemeneti formátumai ("f" lebegőpontos szám)
          f
          ( f )
          ( f , f )
*/
{
```

```
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
            s >> re >> c;
            if (c == ',') s >> im >> c;
            if (c != ')') s.clear(ios_base::failbit);          // állapot beállítása
    }
    else {
            s.putback(c);
            s >> re;
    }

    if (s) a = complex(re,im);
    return s;
}
```

A nagyon kevés hibakezelő utasítás ellenére ez a programrészlet szinte minden hibatípust képes kezelni. A lokális *c* változónak azért adunk kezdőértéket, hogy ha az első >> művelet sikertelen, nehogy véletlenül pont a *(* karakter legyen benne. Az adatfolyam állapotának ellenőrzésére a függvény végén azért van szükség, mert az *a* paraméter értékét csak akkor változtathatjuk meg, ha a korábbi műveleteket sikeresen végrehajtottuk. Ha formázási hiba történik az adatfolyam állapota *failbit* lesz. Azért nem *badbit*, mert maga az adatfolyam nem sérült meg. A felhasználó a *clear()* függvénnyel alapállapotba helyezheti az adatfolyamot és továbblépve értelmes adatokat nyerhet onnan.

Az adatfolyam állapotának beállítására szolgáló függvény neve *clear()*, mert általában arra használjuk, hogy az adatfolyam állapotát *good()* értékre állítsuk. Az *ios_base::clear()* (§21.3.3) paraméterének alapértelmezett értéke *ios_base::goodbit*.

21.3.6. Kivételek

Nagyon kényelmetlen minden egyes I/O művelet után külön ellenőrizni, hogy sikeres volt-e, ezért nagyon gyakori hiba, hogy elfelejtünk egy ellenőrzést ott, ahol feltétlenül szükség van rá. A kimeneti műveleteket általában nem ellenőrzik, annak ellenére, hogy néha ott is előfordulhat hiba.

Egy adatfolyam állapotának közvetlen megváltoztatására csak a *clear()* függvényt használhatjuk. Ezért ha értesülni akarunk az adatfolyam állapotának megváltozásáról, elég nyilvánvaló módszer, hogy a *clear()* függvényt kivételek kiváltására kérjük. Az *ios_base* osztály *exceptions()* tagfüggvénye pontosan ezt teszi:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
  // ...

  class failure;                          // kivételosztály (lásd §14.10)

  iostate exceptions() const;             // kivétel-állapot kiolvasása
  void exceptions(iostate except);        // kivétel-állapot beállítása

  // ...
};
```

A következő utasítással például elérhetjük, hogy a *clear()* egy *ios_base::failure* kivételt váltson ki, ha a *cout* adatfolyam *bad*, *fail* vagy *eof* állapotba kerül, vagyis ha valamelyik művelet nem hibátlanul fut le:

```
cout.exceptions(ios_base::badbit|ios_base::failbit|ios_base::eofbit);
```

Ha szükség van rá, a *cout* vizsgálatával pontosan megállapíthatjuk, milyen probléma történt. Ehhez hasonlóan a következő utasítással azokat a ritkának egyáltalán nem nevezhető eseteket dolgozhatjuk fel, amikor a beolvasni kívánt adatok formátuma nem megfelelő, és ennek következtében a bemeneti művelet nem ad vissza értéket:

```
cin.exceptions(ios_base::badbit|ios_base::failbit);
```

Ha az *exceptions()* függvényt paraméterek nélkül hívjuk meg, akkor azokat az I/O állapotjelzőket adja meg, amelyek kivételt váltanak ki:

```
void print_exceptions(ios_base& ios)
{
  ios_base::iostate s = ios.exceptions();
  if (s&ios_base::badbit) cout << "bad kivételek";
  if (s&ios_base::failbit) cout << "fail kivételek";
  if (s&ios_base::eofbit) cout << "eof kivételek";
  if (s == 0) cout << "nincs kivétel";
}
```

Az I/O kivételek leggyakoribb felhasználási területe az, hogy a ritkán előforduló (ezért könnyen elfelejthető) hibákat kezeljük. Másik fontos feladatuk a ki- és bemenet vezérlése lehet:

```
void readints(vector<int>& s)                      // nem a kedvenc stílusom!
{
  ios_base::iostate old_state = cin.exceptions();  // menti a kivétel-állapotot
  cin.exceptions(ios_base::eofbit);                // eof kivételt vált ki

  for (;;)
        try {
              int i;
              cin>>i;
              s.push_back(i);
        }
        catch(ios_base::failure) {
              // rendben: elértük a fájl végét
        }

  cin.exceptions(old_state);                       // kivétel-állapot visszaállítása
}
```

A kivételek ilyen formában való felhasználásakor felmerül a kérdés, nevezhetjük-e az adott helyzetet hibának vagy tényleg kivételes helyzetnek. Általában mindkét kérdésre inkább a nem választ adjuk, ezért úgy gondolom, célszerűbb az adatfolyam állapotát közvetlenül vizsgálni. Amit egy függvény belsejében, lokális vezérlési szerkezetekkel kezelhetünk, azt a kivételek sem kezelik jobban.

21.3.7. Adatfolyamok összekötése

A *basic_ios* osztály *tie()* függvénye arra használható, hogy kapcsolatot létesítsünk egy *istream* és egy *ostream* között:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
  // ...

  basic_ostream<Ch,Tr>* tie() const;              // mutató összekötött adatfolyamokra
  basic_ostream<Ch,Tr>* tie(basic_ostream<Ch,Tr>* s);  // *this hozzákötése s-hez

  // ...
};
```

Vegyük például az alábbi programrészletet:

```
string get_passwd()
{
  string s;
  cout << "Jelszó: ";
  cin >> s;
  // ...
}
```

Hogyan bizonyosodhatnánk meg arról, hogy a *Jelszó:* szöveg megjelent-e a képernyőn, mielőtt a bemeneti műveletet végrehajtjuk? A *cout* adatfolyamra küldött kimenet átmeneti tárba kerül, így ha a *cin* és a *cout* független egymástól, a *Jelszó* esetleg mindaddig nem jelenik meg a képernyőn, amíg a kimeneti tár meg nem telik. A megoldást az jelenti, hogy a *cout* adatfolyamot hozzákötjük a *cin* adatfolyamhoz a *cin.tie(&cout)* utasítással.

Ha egy *ostream* objektumot összekötünk egy *istream* objektummal, az *ostream* mindig kiürül, amikor egy bemeneti művelet alulcsordulást okoz az *istream* adatfolyamban, azaz amikor egy bemeneti feladat végrehajtásához új karakterekre van szükség a kijelölt bemeneti eszközről. Tehát ilyenkor a

```
cout << "Jelszó: ";
cin >> s;
```

utasítássorozat egyenértékű az alábbival:

```
cout << "Jelszó: ";
cout.flush();
cin >> s;
```

Adott időben minden adatfolyamhoz legfeljebb egy *ostream* objektum köthető. Az *s.tie(0)* utasítással leválaszthatjuk az *s* objektumhoz kötött adatfolyamot (ha volt ilyen). A többi olyan adatfolyam-függvényhez hasonlóan, melyek értéket állítanak be, a *tie(s)* is a korábbi értéket adja vissza, tehát a legutóbb ide kötött adatfolyamot, vagy ha ilyen nincs, akkor a *0* értéket. Ha a *tie()* függvényt paraméterek nélkül hívjuk meg, akkor egyszerűen visszakapjuk az aktuális értéket, annak megváltoztatása nélkül.

A szabványos adatfolyamok esetében a *cout* hozzá van kötve a *cin* bemenethez, a *wcout* pedig a *wcin* adatfolyamhoz. A *cerr* adatfolyamot felesleges lenne bármihez is hozzákötni, mivel ehhez nincs átmeneti tár, a *clog* pedig nem vár felhasználói közreműködést.

21.3.8. Őrszemek

Amikor a << és a >> operátort a *complex* típusra használtuk, egyáltalán nem foglalkoztunk az összekötött adatfolyamok (§21.3.7) kérdésével, vagy azzal, hogy az adatfolyam állapotának megváltozása kivételeket okoz-e (§21.3.6). Egyszerűen azt feltételeztük (és nem ok nélkül), hogy a könyvtár által kínált függvények figyelnek helyettünk ezekre a problémákra. De hogyan képesek erre? Néhány tucat ilyen függvénnyel kell megbirkóznunk, így ha olyan bonyolult eljárásokat kellene készítenünk, amely az összekötött adatfolyamokkal, a helyi sajátosságokkal (locale, §21.7, §D), a kivételekkel, és egyebekkel is foglalkoznak, akkor igen kusza kódot kapnánk.

A megoldást a *sentry* („őrszem") osztály bevezetése jelenti, amely a közös kódrészleteket tartalmazza. Azok a részek, melyeknek elsőként kell lefutniuk (a „prefix kód", például egy lekötött adatfolyam kiürítése), a *sentry* konstruktorában kaptak helyet. Az utolsóként futó sorokat (a „suffix kódokat", például az állapotváltozások miatt elvárt kivételek kiváltását) a *sentry* destruktora határozza meg:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
  // ...
  class sentry;
  // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream<Ch,Tr>::sentry {
public:
  explicit sentry(basic_ostream<Ch,Tr>& s);
  ~sentry();
  operator bool();

  // ...
};
```

Tehát egy általános kódot írtunk, melynek segítségével az egyes függvények a következő formába írhatók:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch,Tr>& basic_ostream<Ch,Tr>::operator<<(int i)
{
  sentry s(*this);
  if (!s) {                    // ellenőrzés, minden rendben van-e a kiírás megkezdéséhez
          setstate(failbit);
          return *this;
  }
```

```
    // az int kiírása
    return *this;
}
```

A konstruktorok és a destruktorok ilyen jellegű felhasználása általános az előtag (prefix) és utótag (suffix) kódrészletek beillesztéséhez, és nagyon sok helyzetben alkalmazható módszer.

Természetesen a *basic_istream* hasonló *sentry* tagosztállyal rendelkezik.

21.4. Formázás

A §21.2. pontban bemutatott példák mind abba a kategóriába tartoztak, amit általánosan *formázatlan kimenetnek* (unformatted output) nevezünk. Ez azt jelenti, hogy az objektumot úgy alakítottuk karaktersorozattá, hogy csak alapértelmezett szabályokat használtunk. A programozóknak gyakran ennél részletesebb vezérlési lehetőségekre van szükségük. Néha például meg kell határoznunk, hogy a kimenet mekkora területet használjon, vagy hogy a számok milyen formában jelenjenek meg. Az ehhez hasonló tényezők vezérlésére a bemenet esetében is szükség lehet.

A ki- és bemenet formázását vezérlő eszközök a *basic_ios* osztályban, illetve annak *ios_base* bázisosztályában kaptak helyet. A *basic_ios* például információkat tartalmaz a számrendszerről (nyolcas, tízes vagy tizenhatos), amit egész számok kiírásakor és beolvasásakor használ a rendszer, vagy a lebegőpontos számok pontosságáról és így tovább. Az ezen adatfolyam-szintű (minden adatfolyamra külön beállítható) változók lekérdezésére és beállítására szolgáló függvények szintén ebben az osztályban kaptak helyet.

A *basic_ios* bázisosztálya a *basic_istream* és a *basic_ostream* osztálynak, így a formázás vezérlése minden adatfolyamra külön adható meg.

21.4.1. Formázási állapot

A ki- és bemenet formázását néhány jelzőbit és néhány egész érték vezérli, melyek az adatfolyamok *ios_base* bázisosztályában szerepelnek:

```
class ios_base {
public:
  // ...
  // formázásjelzők nevei:

  typedef megvalósítás_függő1 fmtflags;
  static const fmtflags
           skipws,           // üreshelyek átlépése a bemeneten

           left,             // mezőigazítás: feltöltés az érték után
           right,            // feltöltés az érték előtt
           internal,         // feltöltés előjel és érték között

           boolalpha,        // a true és false szimbolikus megjelenítése

           dec,              // számrendszer alapja egészeknél: 10 (decimális)
           hex,              //  16 (hexadecimális)
           oct,              //  8  (oktális)

           scientific,       // lebegőpontos jelölés: d.ddddddEdd
           fixed,            // dddd.dd

           showbase,         // kiíráskor oktálisak elé 0, hexadecimálisok elé 0x előtag
           showpoint,        // a záró nullák kiírása
           showpos,          // '+' a pozitív egészek elé
           uppercase,        // 'E', 'X' alkalmazása ('e', 'x' helyett)

           adjustfield,      // mezőigazítással kapcsolatos jelző (§21.4.5)
           basefield,        // egész számrendszer alapjával kapcsolatos jelző (§21.4.2)
           floatfield;       // lebegőpontos kimenettel kapcsolatos jelző (§21.4.3)

           fmtflags unitbuf;         // minden kimenet után az átmeneti tár ürítése

  fmtflags flags() const;            // jelzőbitek kiolvasása
  fmtflags flags(fmtflags f);        // jelzőbitek beállítása

  fmtflags setf(fmtflags f) { return flags(flags() | f); }     // jelzőbit hozzáadása
```

```
    // a jelzőbitek törlése és beállítása a maszkban
    fmtflags setf(fmtflags f, fmtflags mask) { return flags((flags()&~mask) | (f&mask)); }
    void unsetf(fmtflags mask) { flags(flags()&~mask); }             // jelzőbitek törlése

    // ...
};
```

Az egyes jelzőbitek (flag) értéke az adott nyelvi változattól függ, így mindig a szimbolikus neveket használjuk a konkrét számértékek helyett, még akkor is, ha az általunk használt értékek véletlenül éppen megfelelően működnek.

Egy felületet jelzők sorozatával, illetve azokat beállító és lekérdező függvényekkel meghatározni időtakarékos, de kicsit régimódi módszer. Legfőbb erénye, hogy a felhasználó a lehetőségeket összeépítheti:

```
const ios_base::fmtflags my_opt = ios_base::left | ios_base::oct | ios_base::fixed;
```

Ez lehetővé teszi, hogy egy függvénynek beállításokat adjunk át, és akkor kapcsoljuk be azokat, amikor szükség van rájuk:

```
void your_function(ios_base::fmtflags opt)
{
    ios_base::fmtflags old_options = cout.flags(opt); // régi beállítások mentése, újak beállítása
    // ...
    cout.flags(old_options);                           // visszaállítás
}

void my_function()
{
    your_function(my_opt);
    // ...
}
```

A *flags()* függvény az eddig érvényben levő beállításokat adja vissza.

Ha képesek vagyunk az összes tulajdonság együttes írására és olvasására, egyesével is beállíthatjuk azokat:

```
myostream.flags(myostream.flags() | ios_base::showpos);
```

Ezen utasítás hatására a *myostream* adatfolyam minden pozitív szám előtt egy + jelet fog megjeleníteni; más beállítások nem változnak meg. Először beolvassuk a régi beállításokat,

majd beállítjuk a *showpos* tulajdonságot úgy, hogy az eredeti számhoz a *bitenkénti vagy* művelettel hozzákapcsoljuk ezt az értéket. A *setf()* függvény pontosan ugyanezt teszi, tehát a fenti példával teljesen egyenértékű az alábbi sor:

```
myostream.setf(ios_base::showpos);
```

Egy jelző mindaddig megtartja értékét, amíg azt meg nem változtatjuk.

A ki- és bemeneti tulajdonságok vezérlése a jelzőbitek közvetlen beállításával igen nyers megoldás és könnyen hibákhoz vezethet. Az egyszerűbb esetekben a módosítók (manipulator) (§21.4.6) tisztább felületet adnak. A jelzőbitek használata egy adatfolyam állapotának vezérlésére a megvalósítási módszerek tanulmányozásához megfelel, a felülettervezéshez már kevésbé.

21.4.1.1. Formázási állapot lemásolása

A *copyfmt()* függvény segítségével egy adatfolyam teljes formázási állapotát lemásolhatjuk:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
  // ...
  basic_ios& copyfmt(const basic_ios& f);
  // ...
};
```

Az adatfolyam átmeneti tárán (§21.6) és annak állapotán kívül a *copyfmt()* minden állapottulajdonságot átmásol, tehát a kivételkezelési módot (§21.3.6) és a felhasználó által megadott további beállításokat is (§21.7.1).

21.4.2. Egész kimenet

A „bitenkénti vagy” használata egy új beállítás megadásához a *flags()* és *setf()* függvények segítségével csak akkor használható, ha az adott tulajdonságot egy bit határozza meg. Nem ez a helyzet azonban egy olyan jellemzőnél, mint például az egészek kiírásához használt számrendszer vagy a lebegőpontos számok kimeneti formátuma. Az ilyen tulajdonságok esetében az érték, amely egy adott stílust ábrázol, nem fogalmazható meg egyetlen bittel vagy akár egymástól független bitek sorozatával.

Az <*iostream*> fejállományban alkalmazott megoldás az, hogy a *setf()* függvény olyan változatát határozza meg, amelynek egy második, „álparamétert" is meg kell adnunk. Ez a paraméter határozza meg, milyen tulajdonságot akarunk beállítani az új értékkel:

```
cout.setf(ios_base::oct,ios_base::basefield);      // oktális
cout.setf(ios_base::dec,ios_base::basefield);      // decimális
cout.setf(ios_base::hex,ios_base::basefield);      // hexadecimális
```

Ezek az utasítások az egészek számrendszer-alapját állítják be, anélkül, hogy az adatfolyam állapotának bármely más részét megváltoztatnák. Az alapszám mindaddig változatlan marad, amíg meg nem változtatjuk. Például az alábbi utasítássorozat eredménye *1234 1234 2322 2322 4d2 4d2*.

```
cout << 1234 << ' ' << 1234 << ' ';                // alapértelmezett: decimális

cout.setf(ios_base::oct,ios_base::basefield);      // oktális
cout << 1234 << ' ' << 1234 << ' ';

cout.setf(ios_base::hex,ios_base::basefield);      // hexadecimális
cout << 1234 << ' ' << 1234 << ' ';
```

Ha az eredményről meg kell tudnunk állapítani, hogy melyik szám milyen számrendszerben értendő, a *showbase* beállítást használhatjuk. Tehát ha az előbbi utasítások elé beszúrjuk a

```
cout.setf(ios_base::showbase);
```

utasítást, akkor az *1234 1234 02322 02322 0x4d2 0x4d2* számsorozatot kapjuk. A szabványos módosítók (manipulator) (§21.4.6.2) elegánsabb megoldást kínálnak az egész számok kiírásához használt számrendszer beállítására.

21.4.3. Lebegőpontos szám kimenet

A lebegőpontos számok kimenetét két tényező befolyásolja: a *formátum* (format) és a *pontosság* (precision).

- Az *általános* (general) formátum lehetővé teszi a megvalósítás számára, hogy olyan számformátumot használjon, amely a rendelkezésre álló területen a lehető legjobban ábrázolja a lebegőpontos értéket. A pontosság a megjelenő számjegyek maximális számát határozza meg. Ez a jellemző a *printf()* függvény *%g* beállításának felel meg (§21.8).

- A *tudományos* (scientific) formátumban egy számjegy szerepel a tizedespont előtt, és egy kitevő (exponens) határozza meg a szám nagyságrendjét. A pontosság ez esetben a tizedespont után álló számjegyek maximális számát jelenti. Ezt az esetet a *printf()* függvény *%e* beállítása valósítja meg.
- A *fixpontos* (fixed) formátum a számot egészrész, tizedespont, törtrész formában jeleníti meg. A pontosság most is a tizedespont után álló számjegyek számának maximumát adja meg. A *printf()* függvény *%f* beállítása viselkedik így.

A lebegőpontos számok kimeneti formátumát az állapotmódosító függvények (state manipulator function) segítségével szabályozhatjuk. Ezek felhasználásával a lebegőpontos értékek megjelenítését úgy állíthatjuk be, hogy az adatfolyam állapotának más részeit nem befolyásoljuk:

```
cout << "Alapértelmezett:\t" << 1234.56789 << '\n';

cout.setf(ios_base::scientific,ios_base::floatfield);      // tudományos formátum használata
cout << "Tudományos:\t" << 1234.56789 << '\n';

cout.setf(ios_base::fixed,ios_base::floatfield);           // fixpontos formátum használata
cout << "Fixpontos:\t" << 1234.56789 << '\n';

cout.setf(0,ios_base::floatfield);                         // alapértelmezés visszaállítása
                                                           // (általános formátum)
cout << "Alapértelmezett:\t" << 1234.56789 << '\n';
```

Az eredmény a következő lesz:

```
default:    1234.57
scientific: 1.234568e+03
fixed:      1234.567890
default:    1234.57
```

A pontosság alapértelmezett értéke minden formátum esetében 6, amit az *ios_base* osztály külön tagfüggvényével módosíthatunk:

```
class ios_base {
public:
  // ...
  streamsize precision() const;           // pontosság lekérdezése
  streamsize precision(streamsize n);     // pontosság beállítása (és a régi pontosság lekérdezése)
  // ...
};
```

A *precision()* függvény meghívásával minden lebegőpontos I/O művelet pontosságát megváltoztatjuk az adatfolyamban a függvény következő meghívásáig. Ezért a

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

utasítássorozat eredménye a következő lesz:

```
1234.5679 1234.5679 123456
1235 1235 123456
```

A példában két dolgot figyelhetünk meg: az egyik, hogy a lebegőpontos értékek kerekítve jelennek meg, nem egyszerűen levágva, a másik, hogy a *precision()* az egész értékek megjelenítésére nincs hatással.

Az *uppercase* jelzőbit (§21.4.1) azt határozza meg, hogy a tudományos formátumban *e* vagy *E* betű jelölje a kitevőt.

A módosítók (manipulator) elegánsabb megoldást kínálnak a lebegőpontos számok kimeneti formátumának beállítására (§21.4.6.2).

21.4.4. Kimeneti mezők

Gyakran arra van szükségünk, hogy egy kimeneti sor adott területét töltsük fel szöveggel. Ilyenkor pontosan *n* karaktert akarunk használni, kevesebbet semmiképp (többet pedig csak akkor, ha a szöveg nem fér el a meghatározott területen). Ehhez meg kell adnunk a terület szélességét és a kitöltő karaktert:

```
class ios_base {
public:
  // ...
  streamsize width() const;                  // mezőszélesség lekérdezése
  streamsize width(streamsize wide);         // mezőszélesség beállítása
  // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
  // ...
```

```
    Ch fill() const;                    // kitöltő karakter lekérdezése
    Ch fill(Ch ch);                     // kitöltő karakter beállítása
    // ...
};
```

A *width()* függvény a kimenő karakterek legkisebb számát határozza meg a standard könyvtár következő olyan << műveletében, amellyel számértéket, logikai értéket, C stílusú karakterláncot, karaktert, mutatót (§21.2.1), *string* objektumot (§20.3.15) vagy *bitfield* változót (§17.5.3.3) írunk ki:

```
cout.width(4);
cout << 12;
```

Ez az utasítás a *12* számot két szóköz karakter után írja ki.

A kitöltő karaktert a *fill()* függvény segítségével adhatjuk meg:

```
cout.width(4);
cout.fill('#');
cout << "ab";
```

Az eredmény *##ab* lesz.

Az alapértelmezett kitöltő karakter a szóköz, az alapértelmezett pontosság pedig *0*, ami annyi karaktert jelent, amennyire szükség van. A kimeneti mező méretét tehát a következő utasítással állíthatjuk vissza alapértelmezett értékére:

```
cout.width(0);      // "annyi karakter, amennyi csak kell"
```

A *width(n)* utasítással a kiírható karakterek legkisebb számát *n*-re állítjuk. Ha ennél több karaktert adunk meg, akkor azok mind megjelennek:

```
cout.width(4);
cout << "abcdef";
```

Az eredmény *abcdef* lesz, nem pedig csak *abcd*. Ez azért van így, mert általában jobb a megfelelő értéket csúnya formában megkapni, mint a rossz értéket szépen igazítva (lásd még §21.10[21]).

A *width(n)* függvényhívás csak a közvetlenül utána következő << kimeneti műveletre vonatkozik:

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;
```

Az eredmény csak *##12:13* lesz és nem *##12###:##13*, ami akkor jelenne meg, ha a *width(4)* minden későbbi műveletre vonatkozna. Ha a *width()* függvényt több, egymás utáni kimeneti műveletre is alkalmazni szeretnénk, kénytelenek leszünk minden egyes értékhez külön-külön megadni.

A szabványos módosítók (modifier) (§21.4.6.2) elegánsabb megoldást kínálnak a kimeneti mezők méretének szabályozására is.

21.4.5. Mezők igazítása

A karakterek mezőn belüli igazítását (adjustment) a *setf()* függvény meghívásával állíthatjuk be:

```
cout.setf(ios_base::left,ios_base::adjustfield);        // bal
cout.setf(ios_base::right,ios_base::adjustfield);       // jobb
cout.setf(ios_base::internal,ios_base::adjustfield);    // belső
```

Ezek az utasítások az *ios_base::width()* függvénnyel meghatározott kimeneti mezőn belül adják meg az igazítást, az adatfolyam egyéb beállításaira nincsenek hatással.

Az igazítást a következőképpen használhatjuk.

```
cout.fill('#');

cout << '(';
cout.width(4);
cout << -12 << "),(";

cout.width(4);
cout.setf(ios_base::left,ios_base::adjustfield);
cout << -12 << "),(";

cout.width(4);
cout.setf(ios_base::internal,ios_base::adjustfield);
cout << -12 << ")";
```

Az eredmény: *(#-12), (-12#), (-#12).* Az *internal* beállítás a kitöltő karaktereket az előjel és az érték közé helyezi. A példából láthatjuk, hogy az alapértelmezett beállítás a jobbra igazítás.

21.4.6. Módosítók

A standard könyvtár szeretné megkímélni a programozót attól, hogy az adatfolyamok állapotát jelzőbiteken keresztül állítsa be, ezért külön függvényeket kínál ezen feladat megoldásához. Az alapötlet az, hogy a kiírt vagy beolvasott objektumok között adjuk meg azokat a műveleteket is, melyek az adatfolyam állapotát megváltoztatják. Az alábbi utasítással például a kimenet átmeneti tárának azonnali kiürítésére szólíthatjuk fel az adatfolyamot:

```
cout << x << flush << y << flush;
```

A megfelelő helyeken a *cout.flush()* függvény fut le. Ezt az ötletet úgy valósíthatjuk meg, hogy egy olyan << változatot készítünk, amely egy függvényre hivatkozó mutatót vár paraméterként és törzsében lefuttatja a hivatkozott függvényt:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
  // ...

  basic_ostream& operator<<(basic_ostream& (*f)(basic_ostream&)) { return f(*this); }
  basic_ostream& operator<<(ios_base& (*f)(ios_base&));
  basic_ostream& operator<<(basic_ios<Ch,Tr>& (*f)(basic_ios<Ch,Tr>&));

  // ...
};
```

Ahhoz, hogy ez a megoldás működjön, a (mutatóként átadott) függvénynek nem szabad tagfüggvénynek lennie (legfeljebb statikus tagfüggvénynek), és a megfelelő típussal kell rendelkeznie. Ezért a *flush()* függvényt például a következőképpen kell meghatároznunk:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>& s)
{
  return s.flush();   // az ostream osztály flush() tagfüggvényének meghívása
}
```

Ezen deklarációk után a

```
cout << flush;
```

utasítás egyenértékű lesz a

```
cout.operator<<(flush);
```

utasítással, ami pedig meghívja a

```
flush(cout);
```

függvényt, így végül a

```
cout.flush();
```

kerül végrehajtásra.

A teljes eljárás fordítási időben történik, így lehetővé teszi, hogy a *basic_ostream::flush()* függvényt *cout<<flush* formában hívjuk meg. Nagyon sok olyan művelet van, amelyet közvetlenül egy ki- vagy bemeneti művelet előtt vagy után akarunk végrehajtani:

```
cout << x;
cout.flush();
cout << y;
unset(ios_base::skipws);        // ne ugorjuk át az üreshelyeket
cin >> x;
```

Ha ezeket a műveleteket külön utasításokként kell megfogalmaznunk, a műveletek közötti kapcsolatok kevésbé fognak látszódni, márpedig az ilyen logikai kapcsolatok elvesztése erősen rontja a program olvashatóságát. A *módosítók* (manipulator) lehetővé teszik, hogy az olyan műveleteket, mint a *flush()* és a *noskipws()*, közvetlenül a ki- vagy bemeneti műveletek listájában helyezzük el:

```
cout << x << flush << y << flush;
cin >> noskipws >> x;
```

Megjegyzendő, hogy a módosítók az *std* névtérhez tartoznak, ezért minősítenünk kell azokat, amennyiben az *std* nem része az adott hatókörnek:

```
std::cout << endl;          // hiba: endl nincs a hatókörben
std::cout << std::endl;     // rendben
```

Természetesen a *basic_istream* a módosítók számára is ugyanúgy biztosítja a >> operátorokat, mint a *basic_ostream*:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
  // ...
  basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));
  basic_istream& operator>>(basic_ios<Ch,Tr>& (*pf)(basic_ios<Ch,Tr>&));
  basic_istream& operator>>(ios_base& (*pf)(ios_base&));
  // ...
};
```

21.4.6.1. Módosítók, melyek paramétereket várnak

Nagyon hasznosak lennének az olyan módosítók is, melyeknek paramétereket is át tudunk adni. Például jó lenne, ha leírhatnánk az alábbi sort, és vele az *angle* lebegőpontos változó megjelenítését 4 jegy pontosságúra állíthatnánk:

```
cout << setprecision(4) << angle;
```

Ennek eléréséhez a *setprecision*-nek egy objektumot kell visszaadnia, amelynek a *4* kezdőértéket adjuk, és amely meghívja a *cout::setprecision(4)* függvényt. Az ilyen módosítók függvényobjektumok, amelyeket a *()* helyett a << operátor hív meg. A függvényobjektum pontos típusa az adott megvalósítástól függ; egy lehetséges definíciója például a következő:

```
struct smanip {
  ios_base& (*f)(ios_base&,int);          // meghívandó függvény
  int i;

  smanip(ios_base& (*ff)(ios_base&,int), int ii) : f(ff), i(ii) { }
};

template<class Ch, class Tr>
ostream<Ch,Tr>& operator<<(ostream<Ch,Tr>& os, const smanip& m)
{
  return m.f(os,m.i);
}
```

Az *smanip* konstruktor paramétereit az *f*-ben és az *i*-ben tárolja, majd az *operator<<* egy *f(i)* függvényhívást hajt végre. Ennek felhasználásával a fenti *setprecision()* módosító a következőképpen definiálható:

```
ios_base& set_precision(ios_base& s, int n)        // segéd
{
  return s.precision(n);                          // a tagfüggvény hívása
}

inline smanip setprecision(int n)
{
  return smanip(set_precision,n);                 // függvényobjektum létrehozása
}
```

Így már leírhatjuk az utasítást:

```
cout << setprecision(4) << angle ;
```

A programozók saját módosítókat is megadhatnak az *smanip* stílusában, ha más lehetőségekre is szükségük van (§21.10[22]). Ehhez nem kell megváltoztatniuk a standard könyvtár osztályait és sablonjait (például *basic_istream*, *basic_ostream*, *basic_ios* vagy *ios_base*).

21.4.6.2. Szabványos ki- és bemeneti módosítók

A standard könyvtárban sok módosító (manipulator) szerepel a különböző formázási állapotok kezeléséhez. A szabványos módosítók az *std* névtérben találhatók. Az *ios_base* osztályhoz kapcsolódó módosítók az <*ios*> fejállományon keresztül érhetők el, míg az *istream* vagy az *ostream* felhasználásával működő módosítók az <*istream*> és az <*ostream*> (illetve néha az <*iostream*>) fejállományból. A többi szabványos módosítót az <*iomanip*> fejállomány adja meg.

```
ios_base& boolalpha(ios_base&);         // true és false szimbolikus jelzése (bemenet és
                                        // kimenet)
ios_base& noboolalpha(ios_base& s);     // s.unsetf(ios_base::boolalpha)

ios_base& showbase(ios_base&);          // kimenetnél oktálishoz 0, hexadecimálishoz 0x
                                        // előtag
ios_base& noshowbase(ios_base& s);      // s.unsetf(ios_base::showbase)

ios_base& showpoint(ios_base&);
ios_base& noshowpoint(ios_base& s);     // s.unsetf(ios_base::showpoint)

ios_base& showpos(ios_base&);
ios_base& noshowpos(ios_base& s);       // s.unsetf(ios_base::showpos)

ios_base& skipws(ios_base&);            // üreshelyek átugrása
ios_base& noskipws(ios_base& s);        // s.unsetf(ios_base::skipws)
```

```
ios_base& uppercase(ios_base&);        // X és E (x és e helyett)
ios_base& nouppercase(ios_base&);      // x és e (X és E helyett)

ios_base& internal(ios_base&);         // igazítás (§21.4.5)
ios_base& left(ios_base&);             // feltöltés érték után
ios_base& right(ios_base&);            // feltöltés érték előtt

ios_base& dec(ios_base&);              // egész számrendszer alapja: 10 (§21.4.2)
ios_base& hex(ios_base&);              // egész számrendszer alapja: 16
ios_base& oct(ios_base&);              // egész számrendszer alapja: 8

ios_base& fixed(ios_base&);            // lebegőpontos, fixpontos: dddd.dd (§21.4.3)
ios_base& scientific(ios_base&);       // tudományos formátum: d.ddddEdd

template <class Ch, class Tr>
  basic_ostream<Ch,Tr>& endl(basic_ostream<Ch,Tr>&);      // '\n' kiírása és az
                                                         // adatfolyam ürítése
template <class Ch, class Tr>
  basic_ostream<Ch,Tr>& ends(basic_ostream<Ch,Tr>&);      // '\0' kiírása és az
                                                         // adatfolyam ürítése
template <class Ch, class Tr>
  basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>&);     // az adatfolyam ürítése

template <class Ch, class Tr>
  basic_istream<Ch,Tr>& ws(basic_istream<Ch,Tr>&);        // üreshely "lenyelése"

smanip resetiosflags(ios_base::fmtflags f);    // jelzőbitek törlése (§21.4)
smanip setiosflags(ios_base::fmtflags f);      // jelzőbitek beállítása (§21.4)
smanip setbase(int b);                         // egészek kiírása b  alapú számrend
                                               // szerben (§21.4.2)
smanip setfill(int c);                         // legyen c a kitöltő karakter (§21.4.4)
smanip setprecision(int n);                    // n számjegy (§21.4.3, §21.4.6)
smanip setw(int n);                            // a következő mezőszélesség n karakter
                                               // (§21.4.4)
```

Például a

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

utasítás eredménye *1234, 4d2, 2322*, míg a

```
cout << '(' << setw(4) << setfill('#') << 12 << ") (" << 12 << ")\n";
```

eredménye *(##12) (12)*.

Figyeljünk rá, hogy ha olyan módosítókat használunk, melyek nem vesznek át paramétereket, akkor nem szabad kitennünk a zárójeleket. A paramétereket is fogadó módosítók használatához az *<iomanip>* fejállományt be kell építenünk (*#include*):

```
#include <iostream>
using namespace std;

int main()
{
  cout << setprecision(4)    // hiba: setprecision nem meghatározott (<iomanip> kimaradt)
       << scientific()       // hiba: ostream<<ostream& ("hamis" zárójelek)
       << 3.141421 << endl;
}
```

21.4.6.3. Felhasználói módosítók

A szabványos módosítók stílusában a programozó is készíthet új módosítókat. Az alábbiakban egy olyan eszközt mutatunk be, melynek a lebegőpontos számok formázásakor vehetjük hasznát.

A pontosság minden további kimeneti műveletre vonatkozik, míg a szélesség-beállítás csak a következő numerikus kimeneti utasításra. Célunk most az lesz, hogy egy lebegőpontos számot az általunk megkívánt formában jelenítsünk meg, anélkül, hogy a későbbi kimeneti műveletek formázására hatással lennénk. Az alapötlet az, hogy egy olyan osztályt készítünk, amely formátumokat ábrázol, és egy olyat, amely a formázáson kívül a formázni kívánt értéket is tárolja. Ennek felhasználásával készíthetünk egy olyan << operátort, amely a kimeneti adatfolyamon az adott formában jeleníti meg az értéket:

```
Form gen4(4);       // általános formátum, pontosság 4

void f(double d)
{
  Form sci8 = gen4;
  sci8.scientific().precision(8);       // tudományos formátum, pontosság 8

  cout << d << ' ' << gen4(d) << ' ' << sci8(d) << ' ' << d << '\n';
}
```

Az *f(1234.56789)* függvényhívás eredménye a következő lesz:

```
1234.57 1235 1.23456789e+03 1234.57
```

Figyeljük meg, hogy egy *Form* használata nem befolyásolja az adatfolyam állapotát, hiszen

a *d* utolsó kiíratásakor ugyanazt a formát kapjuk, mint az elsőben.

Íme, egy leegyszerűsített változat:

```
class Bound_form;   // forma és érték

class Form {
  friend ostream& operator<<(ostream&, const Bound_form&);

  int prc;            // pontosság
  int wdt;            // szélesség, 0 jelentése: a szükséges szélesség
  int fmt;            // általános, tudományos, vagy fix (§21.4.3)
  // ...
public:
  explicit Form(int p = 6) : prc(p)     // alapértelmezett pontosság 6
  {
        fmt = 0;                        // általános formátum (§21.4.3)
        wdt = 0;                        // szükséges szélesség
  }

  Bound_form operator()(double d) const;        // Bound_form objektum létrehozása *this
                                                // és d alapján

  Form& scientific() { fmt = ios_base::scientific; return *this; }
  Form& fixed() { fmt = ios_base::fixed; return *this; }
  Form& general() { fmt = 0; return *this; }

  Form& uppercase();
  Form& lowercase();
  Form& precision(int p) { prc = p; return *this; }

  Form& width(int w) { wdt = w; return *this; }           // minden típusra
  Form& fill(char);

  Form& plus(bool b = true);                              // explicit pozitív előjel
  Form& trailing_zeros(bool b = true);                    // záró nullák kiírása
  // ...
};
```

Az ötlet az, hogy a *Form* minden olyan információt tárol, ami egy adatelem formázásához szükséges. Az alapértelmezett értékeket úgy választottuk meg, hogy azok a legtöbb esetben megfelelők legyenek; az egyes formázási beállításokat a tagfüggvények segítségével külön-külön adhatjuk meg. A kiírandó értékhez a meghatározott formázást a *()* operátor segítségével kötjük hozzá. A *Bound_form* objektum ezek után a megfelelő << operátor segítségével tetszőleges kimeneti adatfolyamon megjeleníthető:

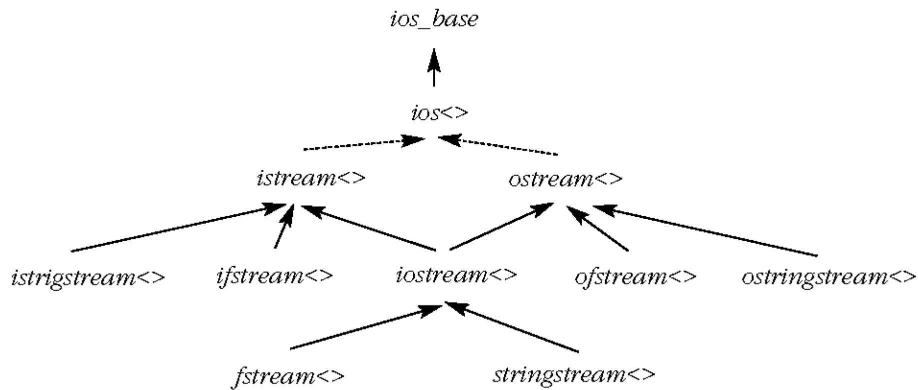
```
struct Bound_form {
  const Form& f;
  double val;

  Bound_form(const Form& ff, double v) : f(ff), val(v) { }
};

Bound_form Form::operator()(double d) { return Bound_form(*this,d); }

ostream& operator<<(ostream& os, const Bound_form& bf)
{
  ostringstream s;            // karakterlánc-folyamok leírása: §21.5.3
  s.precision(bf.f.prc);
  s.setf(bf.f.fmt,ios_base::floatfield);
  s << bf.val;                // s összeállítása
  return os << s.str();       // s kiírása os-re
}
```

A << operátor egy kevésbé egyszerű változatának elkészítését feladatnak hagyjuk (§21.10[21]). A *Form* és a *Bound_form* osztályt könnyen kibővíthetjük, hogy egészek, karakterláncok stb. formázására is használható legyen (lásd §21.10[20]).

Megfigyelhetjük, hogy ezen deklarációk a << és a *( )* párosításával egy hármas operátort hoznak létre. A *cout<<sci4(d)* utasítással egyetlen függvényben kapcsolunk össze egy *ostream* objektumot, egy formátumot és egy értéket, mielőtt a tényleges műveletet végrehajtanánk.

21.5. Fájl- és karakterlánc-folyamok

Amikor egy C++ programot elindítunk, a *cout*, a *cerr*, a *clog*, és a *cin*, illetve széleskarakteres megfelelőik (§21.2.1) azonnal elérhetők. Ezeket az adatfolyamokat a rendszer automatikusan hozza létre és köti hozzá a megfelelő I/O eszközhöz vagy fájlhoz. Ezeken kívül azonban saját adatfolyamokat is létrehozhatunk, és ezek esetében nekünk kell megmondanunk, hogy mihez akarjuk azokat kötni. Az adatfolyamoknak fájlokhoz, illetve karakterláncokhoz való kötése elég általános feladat ahhoz, hogy a standard könyvtár közvetlenül támogassa. Az alábbi ábra a szabványos adatfolyam-osztályok viszonyrendszerét mutatja be:



Azok az osztályok, melyek neve után a <> jel szerepel, olyan sablonok, melyeknek paramétere egy karaktertípus. Ezek teljes neve a *basic_* előtaggal kezdődik. A pontozott vonal virtuális bázisosztályt jelöl (§15.2.4).

A fájlok és a karakterláncok azon tárolók közé tartoznak, melyeket írásra és olvasásra is felhasználhatunk. Ezért ezekhez olyan adatfolyamokat határozhatunk meg, melyek a << és a >> műveleteket egyaránt támogatják. Az ilyen adatfolyamok bázisosztálya az *iostream*, amely az *std* névtérhez tartozik és az *<iostream>* fejállomány írja le:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_iostream : public basic_istream<Ch,Tr>, public basic_ostream<Ch,Tr> {
public:
  explicit basic_iostream(basic_streambuf<Ch,Tr>* sb);
  virtual ~basic_iostream();
};

typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
```

Egy *iostream* írását és olvasását a *streambuf* objektumán (§21.6.4) végzett ki- és bemeneti tárműveletekkel vezérelhetjük.

21.5.1. Fájlfolyamok

A következő példában bemutatunk egy teljes programot, amely egy fájlt egy másikba másol. A fájlneveket parancssori paraméterekként lehet megadni:

```
#include <fstream>
#include <cstdlib>

void error(const char* p, const char* p2 = "")
{
  cerr << p << ' ' << p2 << '\n';
  std::exit(1);
}

int main(int argc, char* argv[ ])
{
  if (argc != 3) error("Hibás paraméterszám");

  std::ifstream from(argv[1]);                    // bemeneti fájlfolyam megnyitása
  if (!from) error("A bemeneti fájl nem nyitható meg",argv[1]);

  std::ofstream to(argv[2]);                      // kimeneti fájlfolyam megnyitása
  if (!to) error("A kimeneti fájl nem nyitható meg",argv[2]);

  char ch;
  while (from.get(ch)) to.put(ch);

  if (!from.eof() || !to) error("Váratlan esemény történt");
}
```

Egy fájlt olvasásra az *ifstream* osztály egy objektumának létrehozásával nyithatunk meg, paraméterként a fájl nevét megadva. Ugyanígy az *ofstream* osztály felhasználásával a fájlt írásra készíthetjük fel. Mindkét esetben a létrehozott objektum állapotának vizsgálatával ellenőrizzük, hogy sikerült-e a fájl megnyitása. A *basic_ofstream* az *<fstream>* fejállományban a következőképpen szerepel:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ofstream : public basic_ostream<Ch,Tr> {
public:
  basic_ofstream();
  explicit basic_ofstream(const char* p, openmode m = out);

  basic_filebuf<Ch,Tr>* rdbuf() const;    // mutató az aktuális átmeneti tárra (§21.6.4)

  bool is_open() const;
  void open(const char* p, openmode m = out);
  void close();
};
```

A *basic_ifstream* nagyon hasonlít a *basic_ofstream* osztályra, azzal a különbséggel, hogy a *basic_istream* osztályból származik, és alapértelmezés szerint olvasásra nyithatjuk meg. Ezeken kívül a standard könyvtár biztosítja a *basic_fstream* osztályt is, amely szintén hasonlít a *basic_ofstream*-re, csak itt a bázisosztály a *basic_iostream*, és alapértelmezés szerint írható és olvasható is.

Szokás szerint a leggyakrabban használt típusokhoz önálló típusnevek (*typedef*-ek) tartoznak:

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;

typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<wchar_t> wfstream;
```

A fájlfolyamok konstruktorainak második paraméterében más megnyitási módokat is megadhatunk:

```
class ios_base {
public:
  // ...

  typedef megvalósítás_függő3 openmode;
  static openmode app,        // hozzáfűzés
                  ate,        // megnyitás és pozícionálás a fájl végére
                              // (kiejtése: "at end")
                  binary,     // bináris I/O (a szöveges (text) mód
                              // ellentéte)
                  in,         // megnyitás olvasásra
                  out,        // megnyitás írásra
                  trunc;      // fájl csonkolása 0 hosszúságúra

  // ...
};
```

Az *openmode* konstansok konkrét értéke és jelentése a megvalósítástól függ, ezért ha részleteket szeretnénk megtudni, akkor saját fejlesztőrendszerünk és standard könyvtárunk leírását kell elolvasnunk, vagy kísérleteznünk kell. A megjegyzésekből következtethetünk, hogy az egyes módoktól körülbelül mit várhatunk. Például egy fájlt megnyithatunk úgy, hogy minden, amit beleírunk, a végére kerüljön:

```
ofstream mystream(name.c_str(),ios_base::app);
```

De megnyithatunk egy fájlt egyszerre írásra és olvasásra is:

```
fstream dictionary("concordance",ios_base::in | ios_base::out);
```

21.5.2. Adatfolyamok lezárása

A fájlokat közvetlenül az adatfolyam *close()* tagfüggvényének meghívásával zárhatjuk be:

```
void f(ostream& mystream)
{
  // ...

  mystream.close();
}
```

Ennek ellenére az adatfolyam destruktora is elvégzi ezt a feladatot, így a *close()* függvény meghívására akkor van csak szükség, ha a fájlt már azelőtt be kell zárnunk, mielőtt az adatfolyam hatóköréből kikerülnénk.

Ez felveti a kérdést, hogy az adott fejlesztőkörnyezet hogyan biztosíthatja, hogy a *cout*, *cin*, *cerr* és *clog* adatfolyamok már első használatuk előtt létrejöjjenek és csak utolsó használatuk után záródjanak le. Természetesen az *<iostream>* adatfolyam-könyvtár különböző változatai különböző módszereket alkalmazhatnak e cél eléréséhez. Végeredményben az, hogy hogyan oldjuk meg ezt a problémát, részletkérdés, amelyet nem kell és nem is szabad a felhasználó által láthatóvá tenni. Az alábbiakban csak egy lehetséges megoldást mutatunk be, amely elég általános arra, hogy különböző típusú globális objektumok konstruktorainak és destruktorainak lefutási sorrendjét rögzítsük. Egy konkrét megvalósítás ennél hatékonyabb megoldást is kínálhat a fordító és az összeszerkesztő (linker) egyedi lehetőségeinek felhasználásával.

Az alapötlet az, hogy egy olyan segédosztályt hozunk létre, amely számon tartja, hányszor építettük be az *<iostream>* fejállományt egy külön fordított forrásfájlba:

```
class ios_base::Init {
  static int count;
public:
  Init();
  ~Init();
};
```

```
namespace {                        // az <iostream> állományban, egy másolat minden fájlban,
  ios_base::Init __ioinit;         // ahová <iostream>-et beépítik
}
int ios_base::Init::count = 0;     // valamelyik .c állományban
```

Minden fordítási egység (§9.1) deklarál egy-egy saját *__ioinit* nevű objektumot. Az *__ioinit* objektumok konstruktora az *ios_base::Init::count* felhasználásával biztosítja, hogy az I/O könyvtár globális objektumainak kezdeti értékadása csak egyszer történjen meg:

```
ios_base::Init::Init()
{
  if (count++ == 0) { /* kezdőérték cout, cerr, cin stb. számára */ }
}
```

Ugyanígy az *__ioinit* objektumok destruktora az *ios_base::Init::count* segítségével biztosítja az adatfolyamok lezárását:

```
ios_base::Init::~Init()
{
  if (--count == 0) { /* felszámolja cout-ot (flush stb.), illetve a cerr-t, cin-t stb. */ }
}
```

Ez a módszer általánosan használható olyan könyvtárak esetében, melyekben globális objektumoknak kell kezdőértéket adni, illetve meg kell azokat semmisíteni. Az olyan rendszerekben, ahol a teljes program az elsődleges memóriában kap helyet futás közben, ez a módszer nem jelent teljesítményromlást. Ha azonban nem ez a helyzet, akkor jelentős veszteséget jelenthet, hogy a kezdeti értékadások miatt kénytelenek vagyunk a tárgykódokat (*object* fájlokat) sorban beolvasni az elsődleges memóriába. Ha lehetséges, kerüljük el a globális objektumok használatát. Egy olyan osztályban, ahol minden művelet jelentős, érdemes lehet minden függvényben elvégezni egy olyan ellenőrzést, mint amit az *ios_base::Init::count* végez, ezzel biztosíthatjuk a kezdeti értékadásokat. Ez a megoldás azonban az adatfolyamok esetében rendkívül költséges lenne. Egy olyan függvény számára, amely egyetlen karaktert ír ki vagy olvas be, egy ilyen ellenőrzés komoly többletterhelést jelentene.

21.5.3. Karakterlánc-folyamok

Az adatfolyamokat hozzáköthetjük karakterláncokhoz is. Ez azt jelenti, hogy az adatfolyamok által kínált formázási lehetőségek felhasználásával karakterláncokat is írhatunk és olvashatunk. Az ilyen adatfolyamok neve *stringstream*, leírásukat az *<sstream>* fejállomány tartalmazza:

```
template <class Ch, class Tr=char_traits<Ch> >
class basic_stringstream : public basic_iostream<Ch,Tr> {
public:
  explicit basic_stringstream(ios_base::openmode m = out|in);
  explicit basic_stringstream(const basic_string<Ch>& s, openmode m = out|in);

  basic_string<Ch> str() const;                // a karakterlánc másolatát veszi
  void str(const basic_string<Ch>& s);         // az értéket s másolatára állítja

  basic_stringbuf<Ch,Tr>* rdbuf() const;       // mutató az aktuális átmeneti tárra
};
```

A *basic_istringstream* a *basic_stringstream* osztályra hasonlít, azzal a különbséggel, hogy a *basic_istream* osztályból származik, és alapértelmezés szerint olvasásra nyithatjuk meg. A *basic_ostringstream* is a *basic_stringstream* „testvére", csak itt a bázisosztály a *basic_ostream*, és alapértelmezés szerint írásra nyitjuk meg.

Szokás szerint a leggyakrabban használt egyedi célú változatokhoz önálló típusnevek (*typedef*-ek) tartoznak:

```
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
```

Az *ostringstream* objektumokat például üzenet-karakterláncok formázására használhatjuk:

```
string compose(int n, const string& cs)
{
  extern const char* std_message[ ];
  ostringstream ost;
  ost << "error(" << n << ") " << std_message[n] << " (user comment: " << cs << ')';
  return ost.str();
}
```

A túlcsordulást ez esetben nem kell ellenőriznünk, mert az *ost* mérete az igények szerint nő. Ez a lehetőség különösen hasznos olyan esetekben, amikor a megkívánt formázási feladatok bonyolultabbak annál, amit az általános, sorokat előállító kimeneti eszközök kezelni tudnak.

A karakterlánc-folyamok kezdőértékét ugyanúgy értelmezi a rendszer, mint a fájlfolyamok esetében:

```
string compose2(int n, const string& cs)          // egyenértékű a compose()-zal
{
   extern const char* std_message[ ];
   ostringstream ost("hiba(",ios_base::ate);      // a kezdeti karakterlánc végétől kezd írni
   ost << n << ") " << std_message[n] << " (felhasználói megjegyzés: " << cs << ')';
   return ost.str();
}
```

Az *istringstream* egy olyan bemeneti adatfolyam, amely az adatokat a konstruktorban megadott karakterláncból olvassa (ugyanúgy, ahogy az *ifilestream* a megadott fájlból olvas):

```
#include <sstream>

void word_per_line(const string& s)    // soronként egy szót ír ki
{
   istringstream ist(s);
   string w;
   while (ist>>w) cout << w << '\n';
}

int main()
{
   word_per_line("Ha azt hiszed, a C++ nehéz, tanulj angolul");
}
```

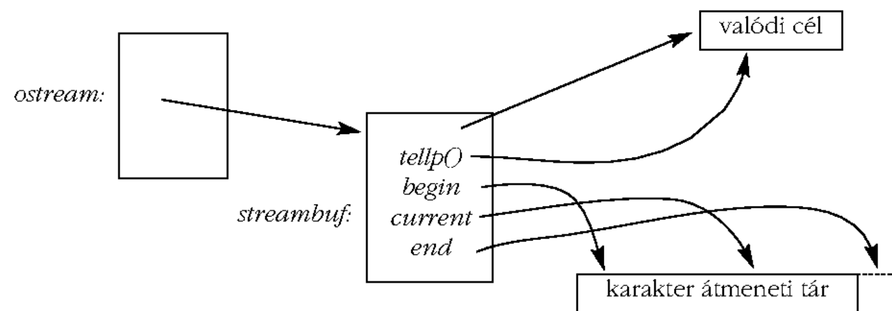
A kezdőértéket adó *string* bemásolódik az *istringstream* objektumba. A karakterlánc vége a bemenet végét is jelenti.

Lehetőség van olyan adatfolyamok meghatározására is, melyek közvetlenül karaktertömbökből olvasnak, illetve oda írnak (§21.10[26]). Ez igen hasznos segítség, főleg, ha régebbi programokkal kell foglalkoznunk. Az ezen szolgáltatást megvalósító *ostrstream* és *istrstream* osztály már régebben bekerült a szabványos adatfolyam-könyvtárba.

21.6. Ki- és bemeneti átmeneti tárak

A kimeneti adatfolyamok vezérelve, hogy egy *átmeneti tárba* (pufferbe) írják a karaktereket, és azok egy kis idő múlva innen kerülnek át oda, ahova valójában írni akartuk azokat. Az átmeneti tárak osztályának neve *streambuf* (§21.6.4), melynek definíciója a *<streambuf>* fejállományban szerepel. A különböző típusú *streambuf* osztályok különböző tárolási mód-

szereket alkalmaznak. A legáltalánosabb megoldás az, hogy a *streambuf* egy tömbben tárolja a karaktereket mindaddig, amíg túlcsordulás nem következik be, és csak ilyenkor írja ki a teljes tömböt a megfelelő helyre. Tehát egy *ostream* objektumot az alábbi formában ábrázolhatunk:



Az *ostream*-nek és a hozzá tartozó *streambuf* objektumnak ugyanazokat a sablonparamétereket kell használnia, és ezek határozzák meg a karakterek átmeneti tárában használt karaktertípust.

Az *istream* osztályok nagyon hasonlóak, csak ott a karakterek az ellenkező irányba folynak. Az átmenetileg nem tárolt (unbuffered) I/O olyan egyszerű ki- és bemenet, ahol az adatfolyam átmeneti tára azonnal továbbít minden karaktert, és nem várakozik addig, amíg megfelelő számú karakter gyűlik össze a hatékony továbbításhoz.

21.6.1. Kimeneti adatfolyamok és átmeneti tárolók

Az *ostream* osztály számtalan különböző típusú érték karakterré átalakítását teszi lehetővé az alapértelmezéseknek (§21.2.1) és a megadott formázási utasításoknak (§21.4) megfelelően. Az *ostream* ezenkívül nyújt néhány olyan függvényt is, melyek közvetlenül a *streambuf* kezelésével foglalkoznak:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
  // ...
  explicit basic_ostream(basic_streambuf<Ch,Tr>* b);
```

```
    pos_type tellp();                                        // aktuális pozíció lekérése
    basic_ostream& seekp(pos_type);                          // aktuális pozíció beállítása
    basic_ostream& seekp(off_type, ios_base::seekdir);       // aktuális pozíció beállítása

    basic_ostream& flush();                 // átmeneti tár ürítése (a tényleges cél felé)

    basic_ostream& operator<<(basic_streambuf<Ch,Tr>* b);           // írás b-ből
};
```

Az *ostream* konstruktorában megadott *strembuf* paraméter határozza meg, hogy a kiírt karaktereket hogyan kezeljük és mikor legyenek ténylegesen kiírva a meghatározott eszközre. Például egy *ostringcstream* (§21.5.3) vagy egy *ofstream* (§21.5.1) létrejöttekor az *ostream* objektumnak a megfelelő *streambuf* (§21.6.4) megadásával adunk kezdőértéket.

A *seekp()* függvények azt állítják be, hogy az *ostream* milyen pozícióra írjon. A *p* utótag azt jelzi, hogy ez a pozíció az adatfolyamban a karakterek kiírására (*put*) szolgál. Ezek a függvények csak akkor működnek, ha az adatfolyam olyasvalamihez van kötve, amin a pozicionálás értelmes művelet (tehát például egy fájlhoz). A *pos_type* típus egy karakterhelyet ábrázol a fájlban, míg az *off_type* az *ios_base::seekdir* változó által kijelölt helytől való eltérést jelöli:

```
class ios_base {
    // ...

    typedef implementation_defined seekdir;
    static const seekdir beg,       // keresés a fájl elejétől
                         cur,       // keresés az aktuális pozíciótól
                         end;       // keresés a fájl végétől
    // ...
};
```

Az adatfolyamok kezdőpozíciója a *0*, így a fájlokat nyugodtan képzelhetjük *n* karakterből álló tömböknek:

```
int f(ofstream& fout)                   // fout valamilyen fájlra hivatkozik
{
    fout.seekp(10);
    fout << '#';                        // karakter kiírása és pozíció mozgatása (+1)
    fout.seekp(-2,ios_base::cur);
    fout << '*';
}
```

A fenti programrészlet egy # karaktert helyez a *file[10]* pozícióra és egy * szimbólumot a *file[9]* helyre. Az egyszerű *istream* és *ostream* osztály esetében ilyen közvetlen hozzáférésre nincs lehetőségünk (lásd §21.10[13]). Ha megpróbálunk a fájl eleje elé vagy vége után írni, az adatfolyam *bad()* állapotba kerül (§21.3.3).

A *flush()* művelet lehetővé teszi a programozó számára, hogy a túlcsordulás megvárása nélkül ürítse ki az átmeneti tárat.

Lehetőség van arra is, hogy a << operátor segítségével egy *streambuf* objektumot közvetlenül a kimeneti adatfolyamba írjunk. Ez elsősorban az I/O rendszerek készítőinek hasznos segítség.

21.6.2. Bemeneti adatfolyamok és átmeneti tárolók

Az *istream* elsősorban olyan műveleteket kínál, melyek segítségével karaktereket olvashatunk be és alakíthatunk át különböző típusú értékekre (§21.3.1). Ezenkívül azonban nyújt néhány olyan szolgáltatást is, melyekkel közvetlenül a *streambuf* objektumot érhetjük el:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
  // ...

  explicit basic_istream(basic_streambuf<Ch,Tr>* b);

  pos_type tellg();                                    // aktuális pozíció lekérdezése
  basic_istream& seekg(pos_type);                      // aktuális pozíció beállítása
  basic_istream& seekg(off_type, ios_base::seekdir);   // aktuális pozíció beállítása

  basic_istream& putback(Ch c);       // c visszarakása az átmeneti tárba
  basic_istream& unget();             // putback alkalmazása a legutóbb beolvasott
                                      // karakterre
  int_type peek();                    // a következő beolvasandó karakter

  int sync();                         // átmeneti tár ürítése (flush)

  basic_istream& operator>>(basic_streambuf<Ch,Tr>* b);          // olvasás b-be
  basic_istream& get(basic_streambuf<Ch,Tr>& b, Ch t = Tr::newline());

  streamsize readsome(Ch* p, streamsize n);     // legfeljebb n karakter beolvasása
};
```

A pozicionáló függvények ugyanúgy működnek, mint az *ostream* osztálybeli megfelelőik (§21.6.1). A *g* utótag azt jelzi, hogy ez a pozíció a karakterek beolvasásának (*get*) helye. A *p* és a *g* utótagokra azért van szükség, mert készíthetünk olyan *iostream* osztályt is, melynek bázisosztálya az *istream* és az *ostream* is, és ilyenkor is meg kell tudnunk különböztetni az írási és az olvasási pozíciót.

A *putback()* függvények azt teszik lehetővé, hogy a program visszategye az adatfolyamba azokat a karaktereket, amelyekre egyelőre nincs szüksége, de később még fel szeretnénk dolgozni. Erre a §21.3.5 pontban mutattunk példát. Az *unget()* függvény a legutoljára beolvasott karaktert teszi vissza az adatfolyamba. Sajnos a bemeneti adatfolyamok ilyen „visszagörgetése" nem mindig lehetséges. Például ha megpróbáljuk visszaírni az utolsó előtti beolvasott karaktert is, az *ios_base::failbit* hibajelző bekapcsolódik. Csak abban lehetünk biztosak, hogy egyetlen, sikeresen beolvasott karaktert vissza tudunk írni. A *peek()* beolvassa a következő karaktert, de azt az átmeneti tárban hagyja, így újra beolvashatjuk. Tehát a *c=peek()* utasítás egyenértékű a *(c=get(),unget(),c)* és a *(putback(c=get()),c)* parancs-sorozatokkal. Figyeljünk rá, hogy a *failbit* bekapcsolása kivételt válthat ki (§21.3.6).

Az *istream*-ek átmeneti tárának azonnali kiürítését (flushing) a *sync()* paranccsal kényszeríthetjük ki, de ezt a műveletet sem mindig használhatjuk biztonságosan. Bizonyos típusú adatfolyamokban ehhez újra kellene olvasnunk a karaktereket az eredeti forrásból, ami nem mindig lehetséges vagy hibás bemenetet eredményezhet. Ezért a *sync() 0* értéket ad vissza, ha sikeresen lefutott, és *–1*-t, ha nem. Ha a művelet sikertelen volt, erre az *ios_base::badbit* (§21.3.3) is felhívja a figyelmünket. A *badbit* értékének megváltozása is kivételt válthat ki (§21.3.6). Ha a *sync()* függvényt olyan átmeneti tárra használjuk, amely egy *ostream* objektumhoz kapcsolódik, akkor az átmeneti tár tartalma a kimenetre kerül.

A >> és a *get()* műveletnek is van olyan változata, mely közvetlenül az átmeneti tárba ír, de ezek is elsősorban az I/O szolgáltatások készítőinek jelentenek hasznos segítséget, hiszen az adatfolyamok átmeneti tárainak közvetlen elérésére csak nekik van szükségük.

A *readsome()* függvény egy alacsonyszintű művelet, mellyel a programozó megállapíthatja, hogy van-e az adatfolyamban beolvasásra váró karakter. Ez a szolgáltatás akkor nagyon hasznos, ha nem akarunk a bemenet megérkezésére várni (például a billentyűzetről). Lásd még: *in_avail()* (§21.6.4).

21.6.3. Az adatfolyamok és az átmeneti tárak kapcsolata

Az adatfolyam és a hozzá tartozó átmeneti tár közötti kapcsolatot az adatfolyamok *basic_ios* osztálya tartja fenn:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
  // ...

  basic_streambuf<Ch,Tr>* rdbuf() const;                  // mutató az átmeneti tárra
  // az átmeneti tár beállítása, clear(), és mutató visszaadása a régi tárra
  basic_streambuf<Ch,Tr>* rdbuf(basic_streambuf<Ch,Tr>* b);

  locale imbue(const locale& loc);                        // helyi sajátosságok beállítása (és
                                                          // a régi érték kiolvasása)

  char narrow(char_type c, char d) const;                 // char érték char_type típusú c-ből
  char_type widen(char c) const;                          // char_type érték a c char-ból

  // ...

protected:
  basic_ios();
  void init(basic_streambuf<Ch,Tr>* b);                   // a kezdeti átmeneti tár beállítása
};
```

Azon kívül, hogy lekérdezhetjük és beállíthatjuk az adatfolyam *streambuf* objektumát (§21.6.4), a *basic_ios* osztályban szerepel az *imbue()* függvény is, mellyel beolvashatjuk és átállíthatjuk az adatfolyam helyi sajátosságokat leíró *locale* objektumát (§21.7). Az *imbue()* függvényt az *ios_base* (§21.7.1) objektumra, a *pubimbue()* függvényt az átmeneti tárra (§21.6.4) kell meghívnunk.

A *narrow()* és a *widen()* függvény segítségével az átmeneti tár karaktertípusát konvertálhatjuk *char* típusúra, vagy fordítva. A *narrow(c,d)* függvényhívás második paramétere az a *char*, amelyet akkor szeretnénk visszakapni, ha a *c*-ben megadott *char_type* értéknek nincs *char* megfelelője.

21.6.4. Adatfolyamok átmeneti tárolása

Az I/O műveletek meghatározása fájltípus említése nélkül történt, de nem kezelhetünk minden eszközt ugyanúgy az átmeneti tárolásnál. Egy karakterlánchoz kötött *ostream* objektumnak (§21.5.3) például másféle tárolásra van szüksége, mint egy fájlhoz kötött (§21.5.1) kimeneti adatfolyamnak. Ezeket a problémákat úgy oldhatjuk meg, hogy a különböző adatfolyamokhoz a kezdeti értékadáskor különböző típusú átmeneti tárakat rendelünk. Mivel a különböző típusú tárak ugyanazokat a műveleteket biztosítják, az *ostream* osztálynak nem kell különbözőnek lennie hozzájuk. Minden átmeneti tár a *streambuf* osztályból szár-

mazik. A *streambuf* virtuális függvényeket nyújt azokhoz a műveletekhez, melyek a különböző tárolási módszereknél eltérőek lehetnek. Ilyenek például a túlcsordulást és az alulcsordulást kezelő eljárások.

A *basic_streambuf* osztály két felületet tesz elérhetővé. A nyilvános (public) felület elsősorban azoknak hasznos, akik olyan adatfolyam-osztályokat akarnak elkészíteni, mint az *istream*, az *ostream*, az *fstream* vagy a *stringstream*. A védett (protected) felület célja azon programozók igényeinek kiszolgálása, akik új tárolási módszert akarnak bevezetni, vagy új bemeneti forrásokat és kimeneti célokat akarnak kezelni.

A *streambuf* osztály megértéséhez érdemes először megismerkednünk az alapját képező átmeneti tárterület modellel, amelyet a védett felület biztosít. Képzeljük el, hogy a *streambuf* objektumnak van egy *kimeneti területe*, amelybe a << operátor segítségével írhatunk, és van egy *bemeneti területe*, amelyből a >> operátor olvas. Mindkét területet három mutató írja le: egy a kezdőpontra, egy az aktuális pozícióra, egy pedig az utolsó utáni elemre mutat. Ezeket a mutatókat függvényeken keresztül érhetjük el:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:

   Ch* eback() const;                    // a bemeneti tár kezdete
   Ch* gptr() const;             // következő karakter (a következő olvasás innen történik)
   Ch* egptr() const;            // a bemeneti tár utolsó eleme utánra mutat

   void gbump(int n);            // n hozzáadása gptr()-hez
   void setg(Ch* begin, Ch* next, Ch* end);       // eback(), gptr(), és egptr() beállítása

   Ch* pbase() const;            // a kimeneti tár kezdete
   Ch* pptr() const;             // a következő üres karakter (a következő kiírás ide történik)
   Ch* epptr() const;            // a kimeneti tár utolsó eleme utánra mutat
   void pbump(int n);                    // n hozzáadása pptr()-hez
   void setp(Ch* begin, Ch* end);        // pbase() és pptr() begin-re, epptr() end-re állítása
   // ...
};
```

Egy karaktertömbre a *setg()* és a *setp()* függvény segítségével megfelelően beállíthatjuk a mutatókat. A programozó a bemeneti területet a következő formában érheti el:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_streambuf<Ch,Tr>::int_type basic_streambuf<Ch,Tr>::snextc()
// az aktuális karakter átlépése, a következő beolvasása
{
```

```
    if (1<egptr()-gptr()) { // ha legalább két karakter van az átmeneti tárban
        gbump(1);                          // az aktuális karakter átlépése
        return Tr::to_int_type(*gptr());   // az új aktuális karakter visszaadása
    }
    if ( 1==egptr()-gptr()) {             // ha pontosan egy karakter van az átmeneti tárban
        gbump(1);                         // az aktuális karakter átlépése
        return underflow();
    }
    // az átmeneti tár üres (vagy nem használt), próbáljuk feltölteni
    if (Tr::eq_int_type(uflow(), Tr::eof()) return Tr::eof();
    if (0<eptr()-gptr()) return Tr::to_int_type(*gptr());   // az új aktuális karakter
visszaadása
    return underflow;
}
```

Az átmeneti tárat a *gptr()*-en keresztül érjük el, az *egptr()* a bemeneti terület határát jelzi. A karaktereket a valódi forrásból az *uflow()* és az *underflow()* olvassák ki. A *traits_type::to_int_type()* meghívása biztosítja, hogy a kód független lesz az éppen használt karaktertípustól. A kód többféle típusú adatfolyam-tárral használható és figyelembe veszi azt is, hogy az *uflow()* és az *underflow()* virtuális függvények (a *setg()* segítségével) új bemeneti területet is meghatározhatnak.

A *streambuf* nyilvános felülete a következő:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
public:
  // szokásos típus-meghatározások (§21.2.1)

  virtual ~basic_streambuf();

  locale pubimbue(const locale &loc);             // locale beállítása (és régi kiolvasása)
  locale getloc() const;                          // locale kiolvasása

  basic_streambuf* pubsetbuf(Ch* p, streamsize n);        // átmeneti tárterület beállítása

  pos_type pubseekoff(off_type off, ios_base::seekdir way,            // pozíció (§21.6.1)
                      ios_base::openmode m = ios_base::in | ios_base::out);
  pos_type pubseekpos(pos_type p, ios_base::openmode m = ios_base::in | ios_base::out);

  int pubsync();                        // sync(), lásd §21.6.2

  int_type snextc();                    // aktuális karakter átlépése, a következő kiolvasása
  int_type sbumpc();                    // gptr() léptetése 1-el
  int_type sgetc();                     // az aktuális karakter beolvasása
```

```
    streamsize sgetn(Ch* p, streamsize n);          // beolvasás p[0]..p[n-1]-be

    int_type sputbackc(Ch c);             // c visszahelyezése az átmeneti tárba (§21.6.2)
    int_type sungetc();                   // az utolsó karakter visszahelyezése

    int_type sputc(Ch c);                 // c kiírása
    streamsize sputn(const Ch* p, streamsize n);          // p[0]..p[n-1] kiírása

    streamsize in_avail();                                // bemenet rendben?
    // ...
};
```

A nyilvános felület olyan függvényeket tartalmaz, melyekkel karaktereket helyezhetünk az átmeneti tárba, illetve karaktereket vehetünk ki onnan. Ezek a függvények általában nagyon egyszerűek és helyben kifejtve (inline) is fordíthatók, ami a hatékonyság szempontjából kulcsfontosságú.

Azok a függvények, melyek tevékenysége függ a használt tárolási módtól, a védett felület megfelelő eljárását hívják meg. A *pubsetbuf()* például a *setbuf()* függvényt hívja meg, amelyet a leszármazott osztály felülír az átmenetileg tárolt karakterek számára való memóriafoglaláshoz. Tehát az olyan műveletek megvalósítására, mint a *setbuf()*, két függvény szolgál, ami azért praktikus, mert így egy *iostream* is végezhet „rendfenntartó” műveleteket, miközben a felhasználó függvényét meghívja. A virtuális függvény meghívását egy *try* blokkba helyezhetjük, és elkaphatjuk a felhasználói kód által kiváltott kivételeket is.

Alapértelmezés szerint a *setbuf(0,0)* az átmeneti tárolás hiányát jelenti, míg a *setbuf(p,n)* a *p[0]*,…, *p[n-1]* tartomány használatát írja elő a karakterek átmeneti tárolására.

Az *in_avail()* függvény meghívásával azt állapíthatjuk meg, hány karakter áll rendelkezésünkre az átmeneti tárban. Ennek vizsgálatával elkerülhetjük a bemenetre való várakozást. Amikor olyan adatfolyamból olvasunk, amely a billentyűzethez kapcsolódik, a *cin.get(c)* akár addig várakozik, amíg a felhasználó vissza nem tér az ebédjéről. Egyes rendszerekben és alkalmazásokban érdemes felkészülnünk erre a lehetőségre:

```
if (cin.rdbuf()->in_avail()) { // get() nem fog lefagyni
    cin.get(c);
    // csinálunk valamit
}
else {                                  // get() lefagyhat
    // valami mást csinálunk
}
```

Vigyázzunk e lehetőség használatakor, mert néha nem könnyű annak megállapítása, hogy van-e beolvasható bemenet. Az *in_avail()* adott változata esetleg *0* értéket ad vissza, ha egy bemeneti műveletet sikeresen végrehajthatunk.

A nyilvános felület mellett – amelyet a *basic_istream* és a *basic_ostream* használ – a *basic_streambuf* egy védett felületet is kínál, mellyel az adatfolyamok átmeneti tárainak elkészítését segíti, és azokat a virtuális függvényeket vezeti be, amelyek meghatározzák az átmeneti tárolás irányelveit:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:
  // ...

  basic_streambuf();

  virtual void imbue(const locale &loc);                    // locale beállítása

  virtual basic_streambuf* setbuf(Ch* p, streamsize n);

  virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                          ios_base::openmode m = ios_base::in | ios_base::out);
  virtual pos_type seekpos(pos_type p,
                          ios_base::openmode m = ios_base::in | ios_base::out);

  virtual int sync();                                        // sync(), lásd §21.6.2

  virtual int showmanyc();
  virtual streamsize xsgetn(Ch* p, streamsize n);            // n karakter beolvasása
  virtual int_type underflow();                              // az olvasási terület üres, eof vagy
                                                             // karakter visszaadása
  virtual int_type uflow();                      //az olvasási terület üres, eof vagy
                                                 // karakter visszaadása, gptr() növelése

  virtual int_type pbackfail(int_type c = Tr::eof());        // a putback nem járt sikerrel

  virtual streamsize xsputn(const Ch* p, streamsize n);      // n karakter kiírása
  virtual int_type overflow(int_type c = Tr::eof());         // kiíró terület megtelt
};
```

Az *underflow()* és az *uflow()* függvény szolgál arra, hogy a következő karaktereket beolvassuk a tényleges bemeneti forrásról, ha az átmeneti tár üres. Ha az adott forráson nincs beolvasható bemenet, az adatfolyam *eof* állapotba (§21.3.3) kerül. Ha ez nem vált ki kivételt, a *traits_type::eof()* értéket kapjuk vissza. A *gptr()*-t a visszaadott karakter után az *uflow()* nö-

veli, az *underflow()* viszont nem. A rendszerben általában nem csak azok az átmeneti tárak vannak, amelyeket az *iostream* könyvtár hoz létre, így akkor is előfordulhatnak átmeneti tárolás miatti késlekedések, amikor átmenetileg nem tárolt adatfolyamot használunk.

Az *overflow()* függvény akkor fut le, amikor az átmeneti tár megtelik, és ez továbbítja a karaktereket a kimenet valódi célállomására. Az *overflow(c)* utasítás az átmeneti tár tartalmát és a *c* karaktert is kiírja. Ha a célállomásra nem lehet több karaktert írni, az adatfolyam *eof* állapotba kerül (§21.3.3). Ha ez nem okoz kivételt, a *traits_type::eof()* értéket kapjuk.

A *showmanyc()* – „show how many characters", „mondd meg, hány karakter" – egy igen érdekes függvény. Célja az, hogy a felhasználó lekérdezhesse, hány karaktert tudunk „gyorsan" beolvasni, azaz az operációs rendszer átmeneti tárainak kiürítésével, de a lemezről való olvasás megvárása nélkül. A *showmanyc()* függvény *–1* értéket ad vissza, ha nem tudja garantálni, hogy akár egy karaktert is be tudnánk olvasni a fájlvége jel megérkezése előtt. Ez az eljárás (szükségszerűen) elég alacsony szintű, és nagymértékben függ az adott megvalósítástól. Soha ne használjuk a *showmanyc()* függvényt fejlesztőrendszerünk dokumentációjának alapos tanulmányozása és egy kis kísérletezés nélkül.

Alapértelmezés szerint minden adatfolyam a globális helyi sajátosságoknak (global local) megfelelően (§21.7) működik. A *pubimbue(loc)* vagy az *imbue(loc)* függvény meghívásával az adatfolyamot a *loc* objektumban megfogalmazott helyi sajátosságok használatára utasíthatjuk.

Az adott adatfolyamhoz használt *streambuf* osztályt a *basic_streambuf* osztályból kell származtatnunk. Ebben kell szerepelnie azoknak a konstruktoroknak és kezdőérték-adó függvényeknek, melyek a *streambuf* objektumot a karakterek valódi forrásához vagy céljához kötik, és ez írja felül a virtuális függvényeket az átmeneti tárolás módjának megvalósításához:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_filebuf : public basic_streambuf<Ch,Tr> {
public:
   basic_filebuf();
   virtual ~basic_filebuf();

   bool is_open() const;
   basic_filebuf* open(const char* p, ios_base::openmode mode);
   basic_filebuf* close();

protected:
   virtual int showmanyc();
   virtual int_type underflow();
   virtual int_type uflow();
   virtual int_type pbackfail(int_type c = Tr::eof());
```

```
    virtual int_type overflow(int_type c = Tr::eof());

    virtual basic_streambuf<Ch,Tr>* setbuf(Ch* p, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode m = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type p,
                             ios_base::openmode m = ios_base::in | ios_base::out);
    virtual int sync();
    virtual void imbue(const locale& loc);
};
```

Az átmeneti tárak kezelésére használt függvények változatlan formában a *basic_streambuf* osztályból származnak. Csak a kezdeti értékadásra és a tárolási módszerre ható függvényeket kell külön megadnunk.

Szokás szerint a leggyakoribb osztályokat és „széles" megfelelőiket külön *typedef*-ek teszik könnyen elérhetővé:

```
typedef basic_streambuf<char> streambuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_filebuf<char> filebuf;

typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_filebuf<wchar_t> wfilebuf;
```

21.7. Helyi sajátosságok

A *locale* egy olyan objektum, amely a karakterek betűk, számok stb. szerinti osztályozásának módját adja meg, illetve a karakterláncok rendezési sorrendjét és a számok megjelenési formáját kiíráskor és beolvasáskor. Az *iostream* könyvtár általában automatikusan használ egy általános *locale* objektumot, amely biztosítja néhány nyelv és kultúra szokásainak betartását. Ha ez megfelel céljainknak, nem is kell foglalkoznunk *locale* objektumokkal. Ha azonban más szokásokat akarunk követni, akkor az adatfolyam viselkedését úgy változtathatjuk meg, hogy lecseréljük a hozzá kötött *locale* objektumot.

Az *std* névtérhez tartozó *locale* osztályt a *<locale>* fejállomány írja le:

```
class locale {                          //a teljes deklarációt lásd §D.2
public:
  // ...

  locale() throw();                     // az aktuális globális locale másolata
  explicit locale(const char* name);    // locale létrehozása C stílusú locale név alapján
  basic_string<char> name() const;      // a használt locale neve

  locale(const locale&) throw();        // locale másolása
  const locale& operator=(const locale& ) throw();          // locale másolása

  static locale global(const locale&);  // a globális locale beállítása
                                        // (az előző érték kiolvasása)
  static const locale& classic();       // a C által meghatározott locale
};
```

A helyi sajátosságok legközönségesebb használata, amikor egy *locale* objektumot egy másikra kell cserélnünk:

```
void f()
{
  std::locale loc("POSIX");             // szabványos POSIX locale

  cin.imbue(loc);                       // cin használja loc-ot
  // ...
  cin.imbue(std::locale());             // cin visszaállítása az alapértelmezett (globális)
                                        // locale használatára
}
```

Az *imbue()* függvény a *basic_ios* (§21.7.1) osztály tagja.

Láthatjuk, hogy néhány, viszonylag szabványos *locale* objektum saját karakterlánc-neveket használ. Ezeket az elnevezéseket a C nyelv is használja.

Elérhetjük azt is, hogy a C++ minden újonnan készített adatfolyamhoz automatikusan az általunk megadott *locale* objektumot használja:

```
void g(const locale& loc = locale())            // alapértelmezés szerint az aktuális
                                                // globális locale használata
{
  locale old_global = locale::global(loc);      // legyen loc az alapértelmezés
  // ...
}
```

A globális *locale* objektum átállítása nincs hatással a már létező adatfolyamokra, mert azok továbbra is a globális *locale* régi értékét használják. Ez vonatkozik például a *cin*, *cout* stb. adatfolyamra is. Ha ezeket is meg akarjuk változtatni, közvetlenül az *imbue()* függvényt kell használnunk.

Azzal, hogy egy adatfolyamhoz új *locale* objektumot rendelünk, több helyen is megváltoztatjuk arculatát, viselkedését. A *locale* osztály tagjait közvetlenül is használhatjuk, új helyi sajátosságok meghatározására vagy a régiek bővítésére. A *locale* arra is használható, hogy beállítsuk a pénzegységek, dátumok stb. megjelenési formáját ki- és bemenetkor (§21.10[25]), vagy a különböző kódkészletek közötti átalakítást. A helyi sajátosságok használatának elvét, illetve a *locale* és *facet* (arculat, viselkedés) osztályokat a „D” függelék írja le. A C stílusú *locale* meghatározása a *<clocale>* és a *<locale.h>* fejállományokban található.

21.7.1. Adatfolyam-visszahívások

Néha a programozók az adatfolyamok állapotleírását bővíteni akarják. Például elvárhatjuk egy adatfolyamtól, hogy „tudja”, milyen formában kell egy komplex számot megjeleníteni: polár- vagy Descartes-koordinátarendszerben. Az *ios_base* osztályban szerepel az *xalloc()* függvény, mellyel az ilyen egyszerű állapotinformációk számára foglalhatunk memóriát. Az *xalloc()* által visszaadott érték azt a két memóriaterületet adja meg, amelyet az *iword()* és a *pword()* függvénnyel elérhetünk:

```
class ios_base {
public:
  // ...

  ~ios_base();

  locale imbue(const locale& loc);      // locale kiolvasása és beállítása, lásd §D.2.3
  locale getloc() const;                // locale kiolvasása

  static int xalloc();                  // egy egész és egy mutató lefoglalása (mindkettő 0
                                        // kezdőértékkel)
  long& iword(int i);                   // az iword(i) egész elérése
  void*& pword(int i);                  // a pword(i) mutató elérése

  // visszahívások

  enum event { erase_event, imbue_event, copyfmt_event };        // eseménytípusok

  typedef void (*event_callback)(event, ios_base&, int i);
  void register_callback(event_callback f, int i);              // f hozzárendelése word(i)-hez
};
```

A felhasználóknak és a könyvtárak készítőinek néha szükségük van arra, hogy értesítést kapjanak az adatfolyam állapotának megváltozásáról. A *register_callback()* eljárás segítségével a függvényeket „bejegyezhetjük", és azok akkor futnak le, amikor a nekik kijelölt „esemény" bekövetkezik. Tehát például az *imbue()*, a *copyfmt()* vagy a *~ios_base()* függvény meghívása maga után vonhatja egy bejegyzett függvény lefutását, amely sorrendben az *imbue_event*, a *copyfmt_event*, illetve az *erase_event* eseményt figyeli. Amikor az állapot megváltozik, a bejegyzett függvény a *register_callback()* függvényben megadott *i* paramétert kapja meg.

Ez a tárolási és visszahívási eljárás meglehetősen bonyolult, tehát csak akkor használjuk, ha feltétlenül szükségünk van az alacsonyszintű formázási szolgáltatások kibővítésére.

21.8. C stílusú ki- és bemenet

Mivel a C és a C++ kódokat gyakran keverik, a C++ adatfolyamon alapuló ki- és bemeneti eljárásait sokszor együtt használjuk a C nyelv *printf()* függvénycsaládját használó I/O rendszerrel. A C stílusú I/O függvények a *<cstdio>* és az *<stdio.h>* fejállományban találhatók. Mivel a C függvények szabadon meghívhatók a C++ programokból, sok programozó szívesebben használja a megszokott C ki- és bemeneti szolgáltatásokat, de még ha az adatfolyamokon alapuló I/O eljárásokat részesítjük is előnyben, néha akkor is találkozni fogunk C stílusú megoldásokkal.

A C és a C++ stílusú be- és kimenet karakterszinten keverhető. Ha a *sync_with_stdio()* függvényt meghívjuk a legelső adatfolyam alapú I/O művelet előtt, akkor a C és a C++ stílusú eljárások ugyanazokat az átmeneti tárakat fogják használni. Ha az első adatfolyam művelet előtt a *sync_with_stdio(false)* parancsot adjuk ki, az átmeneti tárakat a rendszer biztosan nem fogja megosztani, így egyes esetekben növelhetjük a teljesítményt:

```
class ios_base {
  // ...
  static bool sync_with_stdio(bool sync = true);        // kiolvasás és beállítás
};
```

Az adatfolyam elvű kimeneti függvények legfőbb előnye a C standard könyvtárának *printf()* függvényével szemben, hogy az adatfolyam-függvények típusbiztosak és egységes stílust biztosítanak a különböző objektumok megadására, legyen azok típusa akár beépített, akár felhasználói.

A C általános kimeneti függvényei formázott kimenetet állítanak elő, melyben a megadott paraméterek sorozatának megjelenését a *format* formázási karakterlánc határozza meg:

```
int printf(const char* format ...);              // írás az stdout-ra
int fprintf(FILE*, const char* format ...);      // írás "file"-ba (stdout, stderr)
int sprintf(char* p, const char* format ...);    // írás p[0] ... stb.-re
```

A formázási karakterlánc kétfajta elemet tartalmazhat: sima karaktereket, amelyeket a rendszer egyszerűen a kimeneti adatfolyamba másol, illetve átalakítás-meghatározásokat (conversion-specification), melyek mindegyike egy paraméter átalakítását és kiírását vezérli. Az átalakítás-meghatározásokat a *%* karakterrel kell kezdeni:

```
printf("Jelen volt %d személy.",no_of_members);
```

Itt a *%d* azt határozza meg, hogy a *no_of_members* változót *int* értékként kell kezelni és a megfelelő decimális számjegyek kiírásával kell megjeleníteni.
Ha a *no_of_members==127*, akkor a megjelenő eredmény a következő lesz:

```
Jelen volt 127 személy.
```

Az átalakítás-meghatározásokat igen sokféleképpen megadhatjuk és nagyfokú rugalmasságot biztosítanak. A *%* karaktert követően az alábbi jelek állhatnak:

- A nem kötelező mínuszjel azt írja elő, hogy a rendszer a megadott mezőben az átalakított értéket balra igazítsa.

+ A nem kötelező pluszjel használata azt eredményezi, hogy az előjeles típusú értékek mindig + vagy - előjellel fognak megjelenni.

0 A nem kötelező nulla jelentése: a mezőszélesség feltöltésére numerikus értékek esetén 0 karakterekkel történik. Ha a - vagy a pontosság megadott, akkor a 0 előírást figyelmen kívül hagyjuk.

A szintén nem kötelező *#* azt jelenti, hogy a lebegőpontos értékek mindenképpen tizedesponttal együtt jelennek meg, függetlenül attól, hogy utána esetleg csak *0* számjegyek állnak; hogy a lezáró nullák is megjelennek; illetve hogy az oktális számok előtt egy *0* karakter, hexadecimális számok előtt pedig a *0x* vagy a *0X* karakterpár jelenik meg.

d A nem kötelező számjegysorozat a mező szélességét határozza meg. Ha az átalakított érték kevesebb karaktert tartalmaz, mint a mező szélessége, akkor annak bal oldalán (illetve balra igazítás esetén a jobb oldalán) üres karakterek jelennek meg a megfelelő szélesség elérése érdekében. Ha a mezőszélesség megadását *0* karakterrel kezdjük, a szélességnövelő karakterek szóköz helyett nullák lesznek.

. A nem kötelező pont kiírásával választhatjuk el a mezőszélességet megadó értéket a következő számjegy-sorozattól.

d Újabb, nem kötelező számjegy-sorozat. A pontosságot határozza meg, azaz *e* és *f*-átalakítás esetén a tizedes pont után megjelenő számjegyek számát, karakterláncok esetében pedig a megjelenítendő karakterek legnagyobb számát.

* A mezőszélesség vagy a pontosság konkrét megadása helyett használhatjuk a * karaktert, melynek hatására a következő paraméterben szereplő egész érték adja meg a kívánt értéket.

h A nem kötelező *h* karakter azt jelzi, hogy a következő *d*, *o*, *x* vagy *u* egy *short int* paraméterre vonatkozik.

l A nem kötelező *l* karakter azt jelzi, hogy a következő *d*, *o*, *x* vagy *u* egy *long int* paraméterre vonatkozik.

% Azt jelzi, hogy a % karakter kiírandó. Paramétert nem használ fel.

c Az átalakítás típusát jelző karakter. Az átalakító karakterek és jelentéseik a következők:

- *d* Egész érték, amit tízes számrendszerben kell megjeleníteni.
- *i* Egész érték, amit tízes számrendszerben kell megjeleníteni.
- *o* Egész érték, amit nyolcas számrendszerben kell megjeleníteni.
- *x* Egész érték, amit tizenhatos számrendszerben kell megjeleníteni, *0x* kezdettel
- *X* Egész érték, amit tizenhatos számrendszerben kell megjeleníteni, *0X* kezdettel.
- *f* Egy *float* vagy egy *double* paramétert kell tízes számrendszerbeli értékké alakítani a *[-]ddd.ddd* formátummal. A tizedespont után álló számjegyek számát a megadott paraméter pontossága határozza meg. Ha szükség van rá, az értéket a rendszer kerekíti. Ha pontosság nincs megadva, 6 számjegy jelenik meg; ha pontosságként *0* értéket adunk meg és a *#* jelet nem használjuk, a tizedesjegyek nem íródnak ki.
- *e* Egy *float* vagy *double* paramétert alakít tízes számrendszerbeli alakjára a tudományos *[-]d.ddde+dd* vagy a *[-]d.ddde-dd* alakra, ahol a tizedespont előtt pontosan egy jegy szerepel, míg a tizedespont után álló számjegyek számát az adott paraméter pontosság-meghatározása adja meg. Ha szükséges, az értéket a rendszer kerekíti. Ha pontosság nincs megadva, az alapértelmezett érték *6* lesz; ha a pontosság nulla és a *#* jelet nem használtuk, sem a tizedespont, sem az utána álló számjegyek nem jelennek meg.
- *E* Nagyon hasonlít az *e*-re, de a kitevő jelölésére ez a forma nagy *E* betűt használ.
- *g* A *float* vagy *double* típusú paramétert *d*, *f*, vagy *e* stílusban írja ki, aszerint, hogy melyik biztosítja a legnagyobb pontosságot a lehető legkisebb területen.
- *G* Ugyanaz, mint a *g*, de a kitevő jelölésére a nagy *E* betűt használja.
- *c* Karakter paramétert jelenít meg. A nullkaraktereket figyelmen kívül hagyja.
- *s* Az így átvett paraméter egy karakterlánc (karakterre hivatkozó mutató).

A karaktereket addig másolja a kimenetre, amíg a nullkaraktert, vagy a pontosság-meghatározásban meghatározott karakterszámot el nem éri. Ha a pontosság *0* vagy nincs megadva, akkor csak a nullkarakter jelenti a karakterlánc kiírásának végét.

p A paraméter egy mutató. A megjelenítéshez használt formátum a megvalósítástól függ.

u Előjel nélküli egész paramétert alakít tízes számrendszerbeli alakra.

n A *printf()*, az *fprintf()* vagy az *sprintf()* meghívásával eddig kiírt karakterek számát a mutatott *int*-be írja.

Az nem fordulhat elő, hogy nulla vagy kicsi mezőszélesség miatt csonkolás történjen, mert a rendszer csak akkor foglalkozik a szélesség kezelésével, ha a mező szélessége meghaladja a benne szereplő érték szélességét.

Íme egy kicsivel bonyolultabb példa:

```
char* line_format = "#sor száma %d \"%s\"\n";
int line = 13;
char* file_name = "C++/main.c";

printf("int a;\n");
printf(line_format,line,file_name);
printf("int b;\n");
```

Az eredmény a következő lesz:

```
int a;
#sor száma 13 "C++/main.c"
int b;
```

A *printf()* függvény használata nem biztonságos, mert a paraméterek beolvasásakor nem történik típusellenőrzés. Az alábbi hiba például általában megjósolhatatlan kimenetet eredményez vagy még nagyobb hibát:

```
char x;
// ...
printf("rossz bemeneti karakter: %s",x);          // %s helyett %c kell
```

A *printf()* ennek ellenére nagyfokú rugalmasságot biztosít, olyan formában, amelyet a C programozók már jól ismernek.

A *getchar()* függvény hasonlóan jól ismert módszert ad karakterek beolvasására:

```
int i;
while ((i=getchar())!=EOF) { /* i használata */ }
```

Figyeljünk rá, hogy a fájlvége jelet csak akkor tudjuk felismerni az *int* típusú *EOF* konstans segítségével, ha a *getchar()* által visszaadott értéket is *int* típusú változóban tároljuk és nem *char* típusú adatként.

A C bemeneti/kimeneti rendszerének alaposabb megismeréséhez olvassunk el egy C kézikönyvet vagy Kernighan és Ritchie A *C programozási nyelv* című könyvét (Műszaki Könyvkiadó, 1994) [Kernighan, 1988].

21.9. Tanácsok

[1] Az olyan felhasználói típusokhoz, melyekhez értelmes szöveges forma rendelhető, érdemes megadnunk a >> és a << operátort. §21.2.3, §21.3.5.
[2] Ha alacsony precedenciájú operátorokat tartalmazó kifejezések eredményét akarjuk megjeleníteni, zárójeleket kell használnunk. §21.2.
[3] Új >> és << operátorok létrehozásához nem kell módosítanunk az *istream* és az *ostream* osztályt. §21.2.3.
[4] Olyan függvényeket is készíthetünk, amelyek a második (vagy az utáni) paraméter alapján virtuálisként működnek. §21.2.3.1.
[5] Ne feledjük, hogy a >> alapértelmezés szerint átugorja az üreshely karaktereket. §21.3.2.
[6] Alacsonyszintű bemeneti függvényeket (például *get()* és *read()*) általában csak magasabb szintű eljárások megvalósításához kell használnunk. §21.3.4.
[7] Mindig körültekintően fogalmazzuk meg a befejezési feltételt, ha a *get()*, a *getline()* vagy a *read()* függvényt használjuk. §21.3.4.
[8] Az állapotjelző bitek közvetlen átállítása helyett használjunk módosítókat (manipulator) az I/O vezérléséhez. §21.3.3, §21.4, §21.4.6.
[9] Kivételeket csak a ritkán előforduló I/O hibák kezelésére használjunk. §21.3.6.
[10] A lekötött adatfolyamok az interaktív (felhasználói közreműködést váró) ki- és bemenethez nyújtanak segítséget. §21.3.7.
[11] Ha több függvény be- és kilépési műveleteit egy helyen akarjuk megfogalmazni, használjunk *őrszemeket*. §21.3.8.

[12] Paraméter nélküli módosító után ne tegyünk zárójeleket. §21.4.6.2.
[13] Ha szabványos módosítókat használunk, ne felejtsük ki programunkból az *#include <iomanip>* sort. §21.4.6.2.
[14] Egy egyszerű függvényobjektum létrehozásával akár egy háromparaméterű operátor hatását (és hatékonyságát) is megvalósíthatjuk. §21.4.6.3.
[15] A *width* meghatározások csak a következő I/O műveletre vonatkoznak. §21.4.4.
[16] A *precision* beállításai minden későbbi lebegőpontos kimeneti műveletre hatással vannak. §21.4.3.
[17] A memóriában való formázáshoz használjunk karakterlánc-folyamokat. §21.5.3.
[18] A fájlfolyamok kezelési módját meghatározhatjuk. §21.5.1.
[19] Az I/O rendszer bővítésekor különítsük el a formázást (*iostream*) és az átmeneti tárolást (*streambuf*). §21.1, §21.6.
[20] Az értékek nem szabványos továbbítását átmeneti tárral oldjuk meg. §21.6.4.
[21] Az értékek nem szabványos formázását adatfolyam-műveletekkel oldjuk meg. §21.2.3, §21.3.5.
[22] A felhasználói kódrészleteket elkülöníthetjük és önálló egységként kezelhetjük, ha függvénypárokat használunk. §21.6.4.
[23] Az *in_avail()* függvény segítségével megállapíthatjuk, hogy a következő bemeneti műveletnek várakoznia kell-e majd a bemenetre. §21.6.4.
[24] Válasszuk szét a biztonsági kérdésekkel foglalkozó eljárásokat azon egyszerű műveletektől, melyek legfontosabb tulajdonsága a hatékonyság. (Az előbbieket a *virtual*, az utóbbiakat az *inline* kulcsszóval adjuk meg.) §21.6.4.
[25] Használjuk a *locale* osztályt a „kulturális különbségek” megfogalmazására. §21.7.
[26] A *sync_with_stdio(x)* függvénnyel a C stílusú ki- és bemenetet összeegyeztethetjük a C++ I/O rendszerével, de teljesen szét is választhatjuk azokat. §21.8.
[27] A C stílusú I/O használatakor mindig nagyon figyeljünk a típushibák elkerülésére. §21.8.

21.10. Feladatok

1. (*1.5) Olvassunk be egy lebegőpontos számokat tartalmazó fájlt, a beolvasott számpárokból készítsünk komplex értékeket, majd ezeket írjuk ki.
2. (*1.5) Határozzuk meg a *Name_and_address* (Név és cím) típust. Adjuk meg hozzá a << és a >> operátort. Másoljunk le egy *Name_and_address* objektumokat tartalmazó adatfolyamot.
3. (*2.5) Próbáljunk meg lemásolni olyan *Name_and_address* objektumokból álló adatfolyamot, melyben a lehető legtöbb hiba szerepel (például formázási hibák,

karakterláncok túl korai végződése stb.). Próbáljuk meg úgy kezelni ezeket a hibákat, hogy a másoló függvény a helyesen formázott *Name_and_address* objektumok többségét be tudja olvasni még akkor is, ha a bemenet korábban teljesen összekeveredett.

4. (*2.5) Írjuk újra a *Name_and_address* típus kimeneti formáját úgy, hogy az kevésbé legyen érzékeny a formázási hibákra.
5. (*2.5) Készítsünk néhány függvényt különböző típusú információk bekéréséhez és beolvasásához (például egészekhez, lebegőpontos számokhoz, fájlnevekhez, e-mail címekhez, dátumokhoz, személyi adatokhoz stb.). Próbáljunk minél üzembiztosabb függvényeket készíteni.
6. (*1.5) Írjunk programot, amely kiírja (a) az összes kisbetűt, (b) az összes betűt, (c) az összes betűt és számjegyet, (d) minden karaktert, amely rendszerünkben C++ azonosítóban szerepelhet, (e) az összes írásjelet, (f) a vezérlőkarakterek kódjait, (g) az összes üreshely karaktert, (h) az üreshely karakterek kódjait, és végül (i) minden látható karaktert.
7. (*2) Olvassunk be szövegsorokat egy rögzített méretű karakteres átmeneti tárba. Töröljünk minden üreshely karaktert és az alfanumerikus karaktereket helyettesítsük az ábécé következő karakterével (a *z* betűt *a*-ra, a *9*-et *0*-ra cseréljük). Írjuk ki az így keletkező sorokat.
8. (*3) Készítsünk egy miniatűr adatfolyam I/O rendszert, melyben definiáljuk az *istream*, az *ostream*, az *ifstream*, és az *ofstream* osztályt, melyek biztosítják az *operator<<( )* és az *operator>>( )* függvényt egész értékekhez, illetve az olyan eljárásokat, mint az *open()* és a *close()* a fájlok kezeléséhez.
9. (*4) Írjuk meg a C szabványos ki- és bemeneti könyvtárát (<*stdio.h*>) a C++ szabványos I/O könyvtárának (<*iostream*>) felhasználásával.
10. (*4) Írjuk meg a C++ szabványos I/O könyvtárát (<*iostream*>) a C szabványos I/O könyvtárának (<*stdio.h*>) felhasználásával.
11. (*4) Írjuk meg a C és a C++ könyvtárat úgy, hogy azokat egyszerre használhassuk.
12. (*2) Készítsünk egy osztályt, amelyben a *[]* operátor segítségével közvetlenül olvashatjuk be egy fájl karaktereit.
13. (*3) Ismételjük meg a §21.10[12] feladatot úgy, hogy a *[]* operátort írásra és olvasásra is használhassuk. Ötlet: a *[]* adjon vissza egy olyan objektumot, amely egy fájlpozíciót azonosít. Az ezen objektumra vonatkozó értékadás írja a fájlba a megfelelő adatokat, míg az objektumnak *char* típusra való automatikus konverziója jelentse a megfelelő karakter beolvasását az állományból.
14. (*2) Ismételjük meg a §21.10[13] feladatot úgy, hogy a *[]* operátor tetszőleges típusú objektum beolvasásához használható legyen, ne csak karakterekhez.
15. (*3.5) Írjuk meg az *istream* és az *ostream* olyan változatát, amely a számokat bináris formában írja ki, és olvassa be ahelyett, hogy szöveges alakra alakítaná azokat. Vizsgáljuk meg ezen megközelítés előnyeit és hátrányait a karakteralapú megközelítéssel szemben.

16. (*3.5) Tervezzünk és írjunk meg egy mintaillesztő bemeneti műveletet. Használjunk *printf* stílusú formázott karakterláncokat a minta meghatározásához. Azt is tegyük lehetővé, hogy többféle mintát „rápróbálhassunk" a bemenetre a formátum megtalálásához. A mintaillesztő bemeneti osztályt származtassuk az *istream* osztályból.
17. (*4) Találjunk ki (és írjunk meg) egy sokkal jobb mintaillesztési eljárást. Határozzuk meg pontosan, miben jobb a mi megoldásunk.
18. (*2) Határozzunk meg egy kimeneti módosítót (neve legyen *based*), amely két paramétert kap: egy alapszámot és egy *int* értéket, és az egésznek az alapszám szerinti számrendszerbeli alakját írja a kimenetre. A *based(2,9)* hatására például az *1001* jelenjen meg a kimeneten.
19. (*2) Készítsünk módosítókat, melyek ki- és bekapcsolják a karakterek visszaírását (*echoing*).
20. (*2) Írjuk meg a §21.4.6.3 rész *Bound_form* osztályát a szokásos beépített típusokra.
21. (*2) Írjuk meg a §21.4.6.3 rész *Bound_form* osztályát úgy, hogy egy kimeneti művelet soha ne csordulhasson túl a számára megadott *width()* értéken. Azt is biztosítanunk kell a programozó számára, hogy a kimenet soha ne csonkuljon a megadott pontossági érték miatt.
22. (*3) Készítsünk egy *encrypt(k)* módosítót, melynek hatására az *ostream* objektumon minden kimenet a *k* értékkel titkosítva jelenik meg. Írjuk meg a módosító *decrypt(k)* párját is az *istream* osztályra. Adjunk lehetőséget a titkosítás kikapcsolására is, tehát tudjunk újra „olvasható" szöveget írni a kimenetre.
23. (*2) Kövessük végig egy karakter útját rendszerünkben a billentyűzettől a képernyőig az alábbi egyszerű utasítássorozat esetében:

```
char c;
cin >> c;
cout << c << endl;
```

24. (*2) Módosítsuk a §21.3.6 pont *readints()* függvényét úgy, hogy minden kivételt kezelni tudjon. Ötlet: „kezdeti értékadás az erőforrás megszerzésével".
25. (*2.5) Minden rendszerben lehetőség van arra, hogy dátumokat írjunk, olvassunk és ábrázoljunk egy *locale* objektum leírása szerint. Találjuk meg saját rendszerünk dokumentációjában, hogyan valósíthatjuk ezt meg, és készítsünk egy rövid programot, amely e módszer felhasználásával ír és olvas karaktereket. Ötlet: *struct tm*.
26. (*2.5) Hozzuk létre az *ostrstream* osztályt az *ostream* osztályból való származtatással úgy, hogy egy karaktertömbhöz (C stílusú karakterlánchoz) köthessük hozzá, ugyanúgy, ahogy az *ostringstream* köthető egy *string* objektumhoz. Ne másoljuk be a tömböt az *ostrstream* objektumba, sem ki onnan. Az *ostrstream* osztálynak

egyszerű lehetőséget kell adnia a tömbbe való írásra. Az osztályt olyan memóriában végzett formázásokra szeretnénk felhasználni, mint az alábbi:

```
char buf[message_size];
ostrstream ost(buf,message_size);
do_something(arguments,ost);              // kiírás buf-ra ost-on keresztül
cout << buf;                              // ost hozzáadja a lezáró 0-t
```

A *do_something()*-hoz hasonló műveletek az *ost* adatfolyamra írnak, esetleg továbbadják azt saját rész-műveleteiknek, és a szabványos kimeneti függvényeket használják. A túlcsordulás ellenőrzésére nincs szükség, mert az *ost* ismeri a saját méretét és *fail()* állapotba kerül, amikor megtelik. Végül a *display()* művelet meghívásával az üzenetet egy „valódi" kimeneti adatfolyamba írhatjuk. Ez a módszer nagyon hasznos lehet akkor, ha a végső kimeneti eszköz bonyolultabban tudja megvalósítani a kimenetet, mint a szokásos sorkiíráson alapuló kimeneti eszközök. Az *ost* objektumban tárolt szöveget például megjeleníthetjük a képernyő egy rögzített méretű területén. Ugyanígy hozzuk létre az *istrstream* osztályt, amely egy bemeneti karakterlánc-folyam, ami nullkarakterrel lezárt karakterláncból olvas. A lezáró nullkaraktert használjuk fájlvége jelként. Ezek az *strstream* osztályok az eredeti adatfolyam-könyvtár részét képezték és gyakran szerepelnek a *<strstream.h>* fejállományban.

27. (*2.5) Készítsük el a *general()* módosítót, amely visszaállítja az adatfolyamot az eredeti formátumára, hasonlóan ahhoz, ahogy a *scientific()* (§21.4.6.2) az adatfolyamot tudományos formátumra állítja.

22

Számok

„A számítás célja nem a szám,
hanem a tisztánlátás."
(R.W. Hamming)

„… de a tanulónak gyakran
számokon keresztül
vezet az út a tisztánlátáshoz."
(A. Ralston)

Bevezetés • A számtípusok korlátai • Matematikai függvények • *valarray* • Vektorműveletek • Szeletek • *slice_array* • Az ideiglenes változók kiküszöbölése • *gslice_array* • *mask_array* • *indirect_array* • *complex* • Általánosított algoritmusok • Véletlen számok • Tanácsok • Gyakorlatok

22.1. Bevezetés

A gyakorlatban ritkán akadunk olyan programozási feladatra, ahol nincs szükség valamilyen számokkal végzett műveletre. Programjainkban általában – az alapvető aritmetikai műveleteken kívül – szükségünk van egy kis igazi matematikára is. Ez a fejezet a standard könyvtár ehhez kapcsolódó szolgáltatásait mutatja be.

Sem a C, sem a C++ nyelvet nem elsődlegesen számműveletek elvégzésére tervezték. Azok viszont jellemzően más környezetbe beágyazva fordulnak elő – megemlíthető itt az adatbázis- illetve hálózatkezelés, a műszerek vezérlése, a grafika, a pénzügyi számítások, valamint a szimuláció különböző területei –, így a C++ remek lehetőségeket nyújthat a némileg bonyolultabb matematikai feladatok elvégzésére is, amelyekkel az előbb említett területek megfelelően kiszolgálhatók. A számműveletek skálája igen széles, az egyszerű ciklusoktól a lebegőpontos számokból alkotott vektorokig terjed. A C++ ereje az ilyen összetettebb adatszerkezeteken végzett műveleteknél mutatkozik meg igazán, ezért egyre gyakrabban alkalmazzák mérnöki és tudományos számítások elvégzésére. A fejezetben a standard könyvtár azon szolgáltatásaival és eljárásaival ismerkedhetünk meg, melyek kifejezetten a számműveletek elvégzését támogatják a C++ környezetben. A matematikai alapok tisztázására kísérletet sem teszünk, ezeket ismertnek tételezzük fel. Az esetleges hiányosságok matematikai (és nem számítástechnikával foglalkozó) szakkönyvekből pótolhatók.

22.2. A számtípusok korlátozásai

Ha egy programban nem csupán alapszinten végzünk számműveleteket, mindenképpen szükség van arra, hogy ismerjük a beépített típusok alapvető tulajdonságait. Ezek sokkal inkább az adott fejlesztőkörnyezettől függnek, mintsem a nyelv szabályaitól (§4.6). Például melyik a legnagyobb ábrázolható *int*? Mekkora a legkisebb *float*? Mi történik egy *double* típusú számmal, ha *float* típusúvá alakítjuk? Kerekíti vagy csonkítja a rendszer? Hány bitet tartalmaz a *char* típus?

Az ilyen, és ehhez hasonló kérdésekre a *<limits>* fejállományban leírt *numeric_limits* sablon (template) specializációi szolgálhatnak válasszal. Lássunk egy példát:

```
void f(double d, int i)
{
  if (numeric_limits<unsigned char>::digits != 8) {
          // szokatlan bájt (a bitek száma nem 8)
  }

  if (i<numeric_limits<short>::min() || numeric_limits<short>::max()<i) {
          // i nem tárolható short típusban értékvesztés nélkül
  }

  if (0<d && d<numeric_limits<double>::epsilon()) d = 0;

  if (numeric_limits<Quad>::is_specialized) {
          // a Quad típushoz rendelkezésre állnak korlát-adatok
  }
}
```

Minden specializáció biztosítja a saját paramétertípusának legfontosabb információkat. Következésképpen az általános *numeric_limits* sablon egyszerűen arra szolgál, hogy szabványos nevet adjon néhány konstansnak és helyben kifejtett (inline) függvénynek:

```
template<class T> class numeric_limits {
public:
  static const bool is_specialized = false;     // Áll rendelkezésre információ
                                                // a numeric_limits<T>-ről?

  // érdektelen alapértelmezések
};
```

A tényleges információ az egyedi célú változatokban (specializációkban) szerepel. A standard könyvtár minden változata az összes alaptípushoz (karakterekhez, egészekhez, valós számokhoz, valamint a logikai típushoz) szolgáltat egy-egy specializációt a *numeric_limits*-ből. Nem szerepel viszont ilyen a többi típushoz: a *void* mutatóhoz, a felsoroló típusokhoz, vagy a könyvtári típusokhoz (például a *complex<double>* szerkezethez).

A beépített típusok (például a *char*) esetében csupán néhány adat említésre méltó. Lássunk egy *numeric_limits<char>*-t olyan megvalósításban, ahol a *char* 8 bites és előjeles:

```
class numeric_limits<char> {
public:
  static const bool is_specialized = true;      // igen, van adat

  static const int digits = 7;          // bitek száma ("bináris számjegyek") előjel nélkül
```

```
    static const bool is_signed = true;          // ebben a megvalósításban a char típus
                                                 // előjeles (signed)
    static const bool is_integer = true;         // a char egész jellegű típus

    static char min() throw() { return -128; }   // legkisebb érték
    static char max() throw() { return 127; }    // legnagyobb érték

    // deklarációk, amelyek közömbösek a char típus számára
};
```

Jegyezzük meg, hogy egy előjeles egész típus ténylegesen számábrázolásra használt bitjeinek száma (*digits*) eggyel kevesebb a típushoz rendelt bitszámnál, hiszen egy bitet az előjel foglal le.

A *numeric_limits* tagjainak többsége a lebegőpontos számok leírására szolgál. A következő példa egy lehetséges *float*-változatot mutat:

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static const int radix = 2;          // a kitevő típusa (ebben az esetben 2-es alapú)
    static const int digits = 24;        // radix számjegyek a mantisszában
    static const int digits10 = 6;       // 10-es alapú számjegyek a mantisszában

    static const bool is_signed = true;
    static const bool is_integer = false;
    static const bool is_exact = false;

    static float min() throw() { return 1.17549435E-38F; }
    static float max() throw() { return 3.40282347E+38F; }

    static float epsilon() throw() { return 1.19209290E-07F; }
    static float round_error() throw() { return 0.5F; }

    static float infinity() throw() { return /* valamilyen érték */; }
    static float quiet_NaN() throw() { return /* valamilyen érték */; }
    static float signaling_NaN() throw() { return /* valamilyen érték */; }
    static float denorm_min() throw() { return min(); }

    static const int min_exponent = -125;
    static const int min_exponent10 = -37;
    static const int max_exponent = +128;
    static const int max_exponent10 = +38;
```

```
    static const bool has_infinity = true;
    static const bool has_quiet_NaN = true;
    static const bool has_signaling_NaN = true;
    static const float_denorm_style has_denorm = denorm_absent;   // enum a <limits>-ből
    static const bool has_denorm_loss = false;

    static const bool is_iec559 = true;      // megfelel IEC-559-nek
    static const bool is_bounded = true;
    static const bool is_modulo = false;
    static const bool traps = true;
    static const bool tinyness_before = true;

    static const float_round_style round_style = round_to_nearest;   // enum a <limits>-ből
};
```

Ne feledjük, hogy a *min()* a legkisebb *pozitív* normált szám, az *epsilon* pedig a legkisebb pozitív lebegőpontos szám, amelyre az *1+epsilon-1* ábrázolható.

Amikor egy skalár típust egy beépített típus segítségével definiálunk, érdemes egyúttal a *numeric_limits* megfelelő specializációját is megadnunk. Ha például egy négyszeres pontosságú *Quad*, vagy egy különlegesen pontos egész, *long long* típust készítünk, a felhasználók jogos elvárása, hogy létezzenek a *numeric_limits<Quad>* és a *numeric_limits<long long>* specializációk.

A *numeric_limits*-nek elképzelhető olyan változata, mely egy olyan felhasználói típus tulajdonságait írja le, melynek nem sok köze van lebegőpontos számokhoz. Az ilyen esetekben rendszerint ajánlatosabb a típustulajdonságok leírására használt általános eljárás alkalmazása, mint egy új *numeric_limits* meghatározása olyan tulajdonságokkal, melyek a szabványban nem szerepelnek.

A lebegőpontos számokat helyben kifejtett (inline) függvények ábrázolják. A *numeric_limits* osztályban szereplő egész értékeket viszont olyan formában kell ábrázolnunk, amely megengedi, hogy konstans kifejezésekben felhasználjuk azokat, ezért ezek az elemek osztályon belüli kezdeti értékadással rendelkeznek (§10.4.6.2). Ha ilyen célokra *static const* tagokat használunk felsoroló típusok helyett, ne felejtsük el definiálni a *static* elemeket.

22.2.1. Korlátozási makrók

A C++ örökölte a C-től azokat a makrókat, melyek leírják az egész típusok tulajdonságait. Ezek a *<climits>*, illetve a *<limits.h>* fejállományban szerepelnek. Ilyen makró például a *CHAR_BIT* vagy az *INT_MAX*. Ugyanígy a *<cfloat>* és a *<float.h>* fejállomány azokat a makrókat tartalmazza, amelyek a lebegőpontos számok tulajdonságait írják le. Ezekre példa a *DBL_MIN_EXP*, a *FLT_RADIX* vagy a *LDBL_MAX*.

Mint mindig, a makrókat most is érdemes elkerülnünk.

22.3. Szabványos matematikai függvények

A *<cmath>* és a *<math.h>* fejállomány szolgáltatja azokat a függvényeket, amelyeket általában „szokásos matematikai függvényeknek” nevezünk:

```
double abs(double);              // abszolútérték, ugyanaz mint fabs(); ilyen nincs a C-ben
double fabs(double);             // abszolútérték

double ceil(double d);           // a d-nél nem kisebb legkisebb egész
double floor(double d);          // a d-nél nem nagyobb legnagyobb egész

double sqrt(double d);           // d négyzetgyöke, d nem negatív kell legyen

double pow(double d, double e);  // d e-edik hatványa,
                                 // hiba, ha d==0 és e<=0, vagy ha d<0 és e nem egész
double pow(double d, int i);     // d i-edik hatványa; ilyen nincs a C-ben

double cos(double);              // koszinusz
double sin(double);              // szinusz
double tan(double);              // tangens

double acos(double);             // arkusz koszinusz
double asin(double);             // arkusz szinusz
double atan(double);             // arkusz tangens
double atan2(double x, double y); // atan(x/y)

double sinh(double);             // szinusz hiperbolikusz
double cosh(double);             // koszinusz hiperbolikusz
double tanh(double);             // tangens hiperbolikusz
```

```
double exp(double);                          // e alapú exponenciális
double log(double d);                        // természetes (e alapú) logaritmus,
                                             // d nagyobb kell legyen 0-nál
double log10(double d);       // 10-es alapú logaritmus, d nagyobb kell legyen 0-nál

double modf(double d, double* p);     // d tört részével tér vissza,
                                      // az egész részt *p-be helyezi
double frexp(double d, int* p);       // x eleme [.5,1) és y úgy, hogy d = x*pow(2,y) legyen;
                                      // visszatérő érték x, y *p-be helyezése
double fmod(double d, double m);      // lebegőpontos maradék, előjele megfelel d előjelének
double ldexp(double d, int i);        // d*pow(2,i)
```

A *<cmath>* és a *<math.h>* fejállomány ugyanezeket a függvényeket *float* és *long double* paraméterekkel is elérhetővé teszi.

Ha egy műveletnek több lehetséges eredménye is van (mint például az *asin()* esetében), a függvények a nullához legközelebbi értéket adják vissza. Az *acos()* eredménye mindig nemnegatív.

Ezek a függvények a hibákat az *<errno>* fejállományban szereplő *errno* változó beállításával jelzik. Ennek értéke *EDOM*, ha egy függvény nem értelmezhető a megadott paraméterre, és *ERANGE* ha az eredmény nem ábrázolható az adott típussal:

```
void f()
{
  errno = 0;          // törli az előző hibakódot
  sqrt(-1);
  if (errno==EDOM) cerr << "A sqrt() nem definiált negatív paraméter esetén";
  pow(numeric_limits<double>::max(),2);
  if (errno == ERANGE) cerr << "A pow() eredménye túl nagy ahhoz,"
                                << "hogy double-ként ábrázoljuk";
}
```

A C++ előzményeire visszavezethető okokból néhány matematikai függvény a *<cstdlib>* fejállományban szerepel a *<cmath>* helyett:

```
int abs(int);         // abszolútérték
long abs(long);       // abszolútérték ; ilyen nincs a C-ben
long labs(long);      // abszolútérték

struct div_t { megvalósítás_függő quot, rem; };
struct ldiv_t { megvalósítás_függő quot, rem; };
```

```
div_t div(int n, int d);             // n osztása d-vel, visszatérés (kvóciens,maradék)
ldiv_t div(long int n, long int d);  // n osztása d-vel, visszatérés (kvóciens,maradék);
                                     // ilyen nincs a C-ben
ldiv_t ldiv(long int n, long int d); // n osztása d-vel, visszatérés (kvóciens,maradék)
```

22.4. Vektorműveletek

A legtöbb számítási feladat viszonylag egyszerű, egydimenziós vektorokra vonatkozik, amelyek lebegőpontos számokat tárolnak. Ezért a nagyteljesítményű számítógépek külön támogatják az ilyen vektorok kezelését, a velük foglalkozó könyvtárak széles körben használatosak, és nagyon sok területen létfontosságú az ilyen vektorokat kezelő függvények lehető legoptimálisabb kialakítása. A standard könyvtár tartalmaz egy olyan vektort (*valarray*), amelyet kifejezetten a szokásos matematikai vektorműveletek gyors végrehajtására dolgoztak ki.

Miközben áttekintjük a *valarray* lehetőségeit, mindig gondoljunk arra, hogy ez egy viszonylag alacsony szintű programelem, amely nagy hatékonyságú számítások elvégzéséhez készült. Tehát az osztály tervezésekor a legfontosabb célkitűzés nem a kényelmes használat volt, hanem a nagyteljesítményű számítógépek lehetőségeinek minél jobb kihasználása, a legerősebb optimalizálási módszerek alkalmazása. Ha saját programunkban a rugalmasság és az általánosság fontosabb, mint a hatékonyság, valószínűleg érdemesebb a szabványos tárolók közül választanunk (melyeket a 16. és a 17. fejezet mutat be), ne is próbáljunk a *valarray* egyszerű, hatékony és szándékosan hagyományos kereteihez igazodni.

Felmerülhet bennünk a kérdés, hogy miért nem a *valarray* neve lett *vector*, hiszen ez a hagyományos, matematikai vektor valódi megfelelője, és a §16.3 pont *vector* osztályát kéne inkább *array* típusnak nevezni. A terminológia mégsem ezt az elnevezési rendet követi. A *valarray* numerikus számításokra optimalizált vektor, míg a *vector* egy rugalmas tároló, amely a legkülönbözőbb típusú objektumok tárolására és kezelésére szolgál. Az „array” (tömb) fogalma erősen kötődik a beépített tömbtípushoz.

A *valarray* típust négy kisegítő típus egészíti ki, melyek a *valarray* egy-egy részhalmazát képezik:

- A *slice_array* és a *gslice_array* a *szeletek* (slice) fogalmát ábrázolják (§22.4.6, §22.4.8).
- A *mask_array* egy részhalmazt határoz meg, úgy, hogy minden elemről megmondja, hogy az benne van-e a részhalmazban vagy sem (§22.4.9).

- Az *indirect_array* a részhalmazba tartozó elemek indexértékeinek (sorszámainak) listáját tartalmazza (§22.4.10).

22.4.1. Valarray létrehozása

A *valarray* típus és a hozzá tartozó szolgáltatások definiciója az *std* névtérben szerepel és a *<valarray>* fejállomány segítségével érhető el:

```
template<class T> class std::valarray {
  // ábrázolás
public:
  typedef T value_type;

  valarray();                            // valarray size()==0 mérettel
  explicit valarray(size_t n);           // n elem, értékük: T()
  valarray(const T& val, size_t n);      // n elem, értékük: val
  valarray(const T* p, size_t n);        // n elem, értékük: p[0], p[1], ...
  valarray(const valarray& v);           // v másolata

  valarray(const slice_array<T>&);           // lásd §22.4.6
  valarray(const gslice_array<T>&);          // lásd §22.4.8
  valarray(const mask_array<T>&);            // lásd §22.4.9
  valarray(const indirect_array<T>&);        // lásd §22.4.10

  ~valarray();

  // ...
};
```

Ezek a konstruktorok lehetővé teszik, hogy egy *valarray* objektumnak a kisegítő numerikus tömbtípusok vagy önálló értékek segítségével adjunk kezdőértéket:

```
valarray<double> v0;                // helyfoglalás, v0-nak később adunk értéket
valarray<float> v1(1000);           // 1000 elem, mindegyik értéke float()==0.0F

valarray<int> v2(-1,2000);          // 2000 elem, mindegyik értéke -1
valarray<double> v3(100,9.8064);    // hiba: lebegőpontos valarray méret

valarray<double> v4 = v3;           // v4 elemszáma v3.size()
```

A kétparaméterű konstruktorokban az értéket az elemszám előtt kell megadnunk. Ez eltér a szabványos tárolókban használt megoldástól (§16.3.4).

A másoló konstruktornak átadott *valarray* paraméter elemeinek száma határozza meg a létrejövő *valarray* méretét.

A legtöbb program táblákból vagy valamilyen bemeneti művelet segítségével jut hozzá az adatokhoz. Ezt támogatja az a konstruktor, amely egy beépített tömbből másolja ki az elemeket:

```
const double vd[] = { 0, 1, 2, 3, 4 };
const int vi[] = { 0, 1, 2, 3, 4 };

valarray<double> v3(vd,4);   // 4 elem: 0,1,2,3
valarray<double> v4(vi,4);   // típushiba: vi nem double-ra mutat
valarray<double> v5(vd,8);   // nem meghatározott: túl kevés elem a kezdőérték-adóban
```

Ez a kezdőérték-adó forma nagyon fontos, mert a legtöbb numerikus program nagy tömbök formájában adja meg az adatokat.

A *valarray* típust és kisegítő szolgáltatásait nagy sebességű számításokhoz tervezték. Ez néhány, a felhasználókra vonatkozó korlátozásban, és néhány, a megvalósítókra vonatkozó engedményben nyilvánul meg. A *valarray* készítője szinte bármilyen optimalizálási módszert használhat, amit csak el tud képzelni. A műveletek például lehetnek helyben kifejtettek, a *valarray* műveleteit pedig mellékhatások nélkülinek tekinthetjük (persze saját paramétereikre ez nem vonatkozik). Egy *valarray* objektumról feltételezhetjük, hogy nem rendelkezik álnévvel (alias), kisegítő típusokat bármikor bevezethetünk, és az ideiglenes változókat is szabadon kiküszöbölhetjük, ha az alapvető jelentést így is meg tudjuk tartani. Ezért a *<valarray>* fejállományban szereplő deklarációk jelentősen el is térhetnek az itt bemutatott (és a szabványban szereplő) formától, de azon programok számára, melyek nem sértik meg a szabályokat, mindenképpen ugyanazokat a műveleteket kell biztosítaniuk, ugyanazzal a jelentéssel. A *valarray* elemeinek másolása például a szokásos módon kell, hogy működjön (§17.1.4).

22.4.2. A valarray indexelése és az értékadás

A *valarray* osztály esetében az indexelés egyaránt használható önálló elemek eléréséhez és résztömbök kijelöléséhez:

```
template<class T> class valarray {
public:
  // ...
  valarray& operator=(const valarray& v);      // v másolása
  valarray& operator=(const T& val);           // minden elem val-t kapja értékül
```

```
    T operator[ ](size_t) const;
    T& operator[ ](size_t);

    valarray operator[ ](slice) const;                    // lásd §22.4.6
    slice_array<T> operator[ ](slice);

    valarray operator[ ](const gslice&) const;            // lásd §22.4.8
    gslice_array<T> operator[ ](const gslice&);

    valarray operator[ ](const valarray<bool>&) const;    // lásd §22.4.9
    mask_array<T> operator[ ](const valarray<bool>&);

    valarray operator[ ](const valarray<size_t>&) const;  // lásd §22.4.10
    indirect_array<T> operator[ ](const valarray<size_t>&);

    valarray& operator=(const slice_array<T>&);           // lásd §22.4.6
    valarray& operator=(const gslice_array<T>&);          // lásd §22.4.8
    valarray& operator=(const mask_array<T>&);            // lásd §22.4.9
    valarray& operator=(const indirect_array<T>&);        // lásd §22.4.10

    // ...
};
```

Egy *valarray* objektumot értékül adhatunk egy másik, ugyanolyan méretű *valarray*-nek. Elvárásainknak megfelelően a *v1=v2* utasítás a *v2* minden elemét a *v1* megfelelő elemébe másolja. Ha a tömbök különböző méretűek, az eredmény nem meghatározott lesz, mert a sebességre optimalizált *valarray* osztálytól nem követelhetjük meg, hogy nem megfelelő méretű objektum értékül adásakor könnyen érthető hibajelzést (például kivételt) kapjunk, vagy más, logikus viselkedést tapasztaljunk.

Ezen hagyományos értékadás mellett lehetőség van arra is, hogy egy *valarray* objektumhoz egy skalár értéket rendeljünk. A *v=7* utasítás például a *v valarray* minden egyes elemébe a *7* értéket írja. Ez első ránézésre elég meglepő, és szerepét úgy érthetjük meg leginkább, ha úgy gondolunk rá, mint az értékadó műveleteknek egy, néhány esetben hasznos, „elfajzott" változatára (§22.4.3).

Az egészekkel való sorszámozás a szokásos módon működik, tartományellenőrzés nélkül.

Az önálló elemek kiválasztása mellett a *valarray* indexelése lehetőséget ad résztömbök négyféle kijelölésére is (§22.4.6). Megfordítva, az értékadó utasítások (és a konstruktorok, §22.4.1) ilyen résztömböket is elfogadnak paraméterként. A *valarray* típusra megvalósított értékadó utasítások biztosítják, hogy a kisegítő tömb típusokat (mint a *slice_array*) ne kelljen átalakítanunk *valarray* típusra, mielőtt értékül adjuk azokat. A hatékonyság biztosítása

érdekében egy alapos fejlesztőkörnyezetnek illik a vektor más műveleteihez (például a + és a *) is biztosítani az ilyen változatokat. Ezenkívül a vektorműveletek számtalan módon optimalizálhatók, *szeletek* (slice) vagy más kisegítő vektortípusok használatával.

22.4.3. Műveletek tagfüggvényként

A nyilvánvaló és a kevésbé nyilvánvaló tagfüggvények listája következő:

```
template<class T> class valarray {
public:
  // ...

  valarray& operator*=(const T& arg);              // v[i]*=arg minden elemre
  // hasonlóan: /=, %=, +=, -=, ^=, &=, |=, <<=, és >>=

  T sum() const;                    // elemek összege, += használatával
  T min() const;                    // legkisebb érték, < használatával
  T max() const;                    // legnagyobb érték, < használatával

  valarray shift(int i) const;      // logikai léptetés (balra, ha 0<i; jobbra, ha i<0)
  valarray cshift(int i) const;     // ciklikus léptetés (balra, ha 0<i; jobbra, ha i<0)

  valarray apply(T f(T)) const;     // eredmény[i] = f(v[i]) minden elemre
  valarray apply(T f(const T&)) const;

  valarray operator-() const;       // eredmény[i] = -v[i] minden elemre
  // hasonlóan: +, ~, !

  size_t size() const;                        // elemek száma
  void resize(size_t n, const T& val = T());  // n elem, értékük val
};
```

Ha *size()=0*, akkor *sum()*, *min()* és *max()* értéke nem meghatározott.

Például, ha *v* egy *valarray*, akkor a *v*=.2* vagy a *v/=1.3* utasítások alkalmazhatóak rá. Ha skalárműveleteket hajtunk végre egy vektoron, akkor az azt jelenti, hogy a vektor minden elemére elvégezzük azt. Szokás szerint, egyszerűbb a *= műveletet optimalizálni, mint a * és az = párosítását (§11.3.1).

Figyeljük meg, hogy a nem értékadó műveletek mindig új *valarray* objektumot hoznak létre:

```
double incr(double d) { return d+1; }

void f(valarray<double>& v)
{
   valarray<double> v2 = v.apply(incr); // új, megnövelt valarray-t hoz létre, megnövelt értékkel
}
```

Ezen programrészlet hatására a *v* értéke nem változik. Sajnos az *apply()* nem képes függvényobjektumokat (§18.4) használni paraméterként (§22.9.[1]).

A logikai és a ciklikus léptető függvények (a *shift()* és a *cshift()*) olyan új *valarray* objektumokat adnak vissza, amelyben az elemek megfelelően el vannak léptetve. Az eredeti vektor itt is változatlan marad. A *v2=v.cshift(n)* ciklikus léptetés például egy olyan *valarray* objektumot eredményez, melynek elemeire *v2[i]==v[(i+n)%v.size()]*.. A *v3=v.shift(n)* logikai léptető művelet hatására a *v3[i]* a *v[i+n]* értéket veszi fel, ha az *i+n* létező indexe a *v* vektornak. Ellenkező esetben az adott elembe a vektor alapértelmezett értéke kerül. Ebből következik, hogy a *shift()* és a *cshift()* is balra tolja az elemeket, ha pozitív paramétert adunk meg, és jobbra, ha negatív értéket:

```
void f()
{
   int alpha[] = { 1, 2, 3, 4, 5 ,6, 7, 8 };
   valarray<int> v(alpha,8);                  // 1, 2, 3, 4, 5, 6, 7, 8
   valarray<int> v2 = v.shift(2);             // 3, 4, 5, 6, 7, 8, 0, 0
   valarray<int> v3 = v<<2;                   // 4, 8, 12, 16, 20, 24, 28, 32
   valarray<int> v4 = v.shift(-2);            // 0, 0, 1, 2, 3, 4, 5, 6
   valarray<int> v5 = v>>2;                   // 0, 0, 0, 1, 1, 1, 1, 2
   valarray<int> v6 = v.cshift(2);            // 3, 4, 5, 6, 7, 8, 1, 2
   valarray<int> v7 = v.cshift(-2);           // 7, 8, 1, 2, 3, 4, 5, 6
}
```

A *valarray* típus esetében a << és a >> operátor bitenkénti léptetést végez, tehát nem elemeket tolnak el és nem is I/O műveletek (§22.4). Ennek megfelelően a <<= és a >>= műveletekkel elemeken belüli eltolást végezhetünk egész típusú elemek esetében:

```
void f(valarray<int> vi, valarray<double> vd)
{
   vi <<= 2;          // vi[i]<<=2, vi minden elemére
   vd <<= 2;          // hiba: a léptetés nem meghatározott lebegőpontos értékekre
}
```

A *valarray* méretét módosíthatjuk is. A *resize()* itt *nem* arra szolgál, hogy a *valarray* osztályt egy dinamikusan növekedni képes adatszerkezetté tegye – mint ahogy a *vector* és a *string* esetében történik –, hanem új kezdőértéket adó művelet, amely a létező elemeket is lecseréli a *valarray* alapértelmezett értékére, így a régi elemeket véglegesen elveszítjük.

Az átméretezésre szánt *valarray* objektumokat gyakran üres vektorként hozzuk létre. Gondoljuk végig például, hogyan adhatunk kezdőértéket egy *valarray* objektumnak bemenet alapján:

```
void f()
{
  int n = 0;
  cin >> n;                                   // a tömb méretének beolvasása
  if (n<=0) error("hibás tömbméret");

  valarray<double> v(n);                      // tömb létrehozása a szükséges mérettel
  int i = 0;
  while (i<n && cin>>v[i++]) ;                // a tömb feltöltése
  if (i!=n) error("túl kevés bemeneti elem");

  // ...
}
```

Ha a bemenetet külön függvényben szeretnénk kezelni, a következőt tehetjük:

```
void initialize_from_input(valarray<double>& v)
{

  int n = 0;
  cin >> n;                                   // a tömb méretének beolvasása
  if (n<=0) error("hibás tömbméret");

  v.resize(n);                                // v átméretezése a szükséges méretre
  int i = 0;
  while (i<n && cin>>v[i++]) ;                // a tömb feltöltése
  if (i!=n) error("túl kevés bemeneti elem");
}

void g()
{
  valarray<double> v;                         // alapértelmezett tömb létrehozása
  initialize_from_input(v);                   // v méretezése és feltöltése
  // ...
}
```

Ezzel a megoldással elkerülhetjük nagy méretű adatterületek másolását.

Ha azt szeretnénk, hogy egy *valarray* megőrizze az értékes adatokat, miközben dinamikusan növekszik, ideiglenes változót kell használnunk:

```
void grow(valarray<int>& v, size_t n)
{
  if (n<=v.size()) return;

  valarray<int> tmp(n);                    // n alapértelmezett elem

  copy(&v[0],&v[v.size()],&tmp[0]);        // másoló algoritmus §18.6.1-ből
  v.resize(n);
  copy(&tmp[0],&tmp[v.size()],&v[0]);
}
```

A *valarray* típust nem az ilyen felhasználási területekre tervezték. Egy *valarray* objektumnak nem illik megváltoztatnia méretét, miután a kezdeti helyfoglalás megtörtént.

A *valarray* elemei egyetlen sorozatot alkotnak, tehát a *v[0],…,v[n-1]* elemek egymás után találhatók a memóriában. Ebből következik, hogy a *T** egy közvetlen elérésű (random-access) bejáró (iterátor, §19.2.1) a *valarray<T>* vektorhoz, így a szabványos algoritmusok, például a *copy()*, alkalmazhatók rá. Ennek ellenére jobban illik a *valarray* szellemiségéhez, ha a másolást értékadások és résztömbök formájában fejezzük ki:

```
void grow2(valarray<int>& v, size_t n)
{
  if (n<=v.size()) return;

  valarray<int> tmp = v;
  slice s(0,v.size(),1);          // v.size() elemszámú résztömb (lásd §22.4.5)

  v.resize(n);                    // az átméretezés nem őrzi meg az elemek értékét
  v[s] = tmp;                     // elemek visszamásolása v első részébe
}
```

Ha valamilyen okból a bemeneti adatok olyan elrendezésűek, hogy be kell azokat olvasnunk, mielőtt megtudnánk a tárolásukhoz szükséges vektor méretét, általában érdemes először egy *vector* (§16.3.5) objektumba olvasni az elemeket és onnan másolni azokat egy *valarray* változóba.

22.4.4. Nem tagfüggvényként megvalósított műveletek

A szokásos bináris (kétoperandusú) operátorok és matematikai függvények így szerepelnek a könyvtárban:

```
template<class T> valarray<T> operator*(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator*(const valarray<T>&, const T&);
template<class T> valarray<T> operator*(const T&, const valarray<T>&);

// hasonlóan: /, %, +, -, ^, &, |, <<, >>, &&, ||, ==, !=, <, >, <=, >=, atan2, és pow

template<class T> valarray<T> abs(const valarray<T>&);

// hasonlóan: acos, asin, atan, cos, cosh, exp, log, log10, sin, sinh, sqrt, tan, és tanh
```

A bináris műveleteket két *valarray* objektumra, vagy egy *valarray* objektum és a megfelelő típusú skalár érték együttesére alkalmazhatjuk:

```
void f(valarray<double>& v, valarray<double>& v2, double d)
{
   valarray<double> v3 = v*v2;       // v3[i] = v[i]*v2[i] minden i-re
   valarray<double> v4 = v*d;        // v4[i] = v[i]*d minden i-re
   valarray<double> v5 = d*v2;       // v5[i] = d*v2[i] minden i-re

   valarray<double> v6 = cos(v);     // v6[i] = cos(v[i]) minden i-re
}
```

Ezek a vektorműveletek *valarray* operandusuk (operandusaik) minden elemére végrehajtják a megfelelő műveletet, úgy, ahogy a * és a *cos()* példákon keresztül bemutattuk. Természetesen minden művelet csak akkor alkalmazható, ha a megfelelő függvény definiált a sablonparaméterként megadott típusra. Ellenkező esetben a fordító hibaüzenetet ad, amikor megpróbálja példányosítani a sablont (§13.5).

Ha az eredmény egy *valarray*, akkor annak hossza megegyezik a paraméter(ek)ben használt *valarray* méretével. Ha két *valarray* paraméter mérete nem egyezik meg (bináris műveletnél), az eredmény nem meghatározott lesz.

Érdekes módon I/O műveletek nem állnak rendelkezésünkre a *valarray* típushoz (§22.4.3); a << és a >> operátor csak léptetésre szolgál. Ha mégis szükségünk van a két operátor ki- és bemenetet kezelő változatára, minden gond nélkül definiálhatjuk azokat (§22.9[5]).

Jegyezzük meg, hogy ezek a *valarray* műveletek új *valarray* objektumokat adnak vissza, és nem operandusaikat módosítják. Ez egy kicsit „költségesebbé” teheti a függvényeket, de ha kellően hatékony optimalizációs módszereket alkalmazunk, ez a veszteség nem jelentkezik (lásd például §22.4.7).

A *valarray* objektumokra alkalmazható operátorok és matematikai függvények ugyanúgy használhatók *slice_array* (§22.4.6), *gslice_array* (§22.4.8), *mask_array* (§22.4.9) és *indirect_array* (§22.4.10) objektumokra is, egyes nyelvi változatok azonban lehet, hogy a nem *valarray* típusú operandusokat először *valarray* típusra alakítják, majd ezen végzik el a kijelölt műveletet.

22.4.5. Szeletek

A *slice* (szelet) olyan elvont típus, amely lehetővé teszi, hogy vektorokat akárhány dimenziós mátrixként hatékonyan kezeljünk. Ez a típus a Fortran vektorok alapeleme és kulcsszerepet játszik a BLAS (Basic Linear Algebra Subprograms) könyvtárban, amely viszont a legtöbb számművelet kiindulópontja. A szelet alapjában véve nem más, mint egy *valarray* egy részletének minden *n*-ik eleme:

```
class std::slice {
  // kezdőindex, hossz, és lépésköz
public:
  slice();
  slice(size_t start, size_t size, size_t stride);

  size_t start() const;          // az első elem indexértéke
  size_t size() const;           // elemek száma
  size_t stride() const;         // az n-ik elem helye: start()+n*stride()
};
```

A *stride* a lépésköz, vagyis a távolság (az elemek számában kifejezve) a *slice* két, egymást követő eleme között. Tehát a *slice* egészek egy sorozatát írja le:

```
size_t slice_index(const slice& s, size_t i)        // i leképezése a megfelelő indexértékre
{
  return s.start()+i*s.stride();
}

void print_seq(const slice& s)                      // s elemeinek kiírása
{
  for (size_t i = 0; i<s.size(); i++) cout << slice_index(s,i) << " ";
}
```

```
void f()
{
  print_seq(slice(0,3,4));        // 0. sor
  cout << ", ";
  print_seq(slice(1,3,4));        // 1. sor
  cout << ", ";
  print_seq(slice(0,4,1));        // 0. oszlop
  cout << ", ";
  print_seq(slice(4,4,1));        // 1. oszlop
}
```

A megjelenő szöveg a következő lesz: *0 4 8 , 1 5 9 , 0 1 2 3 , 4 5 6 7*

Tehát egy *slice* nem tesz mást, mint hogy nemnegatív egész értékeket sorszámokra képez le. Az elemek száma (*size()*) nincs hatással a leképezésre (a címzésre), de lehetőséget ad számunkra a sorozat végének megtalálására. Ez a leképezés egy egyszerű, hatékony, általános és viszonylag kényelmes lehetőséget ad egy kétdimenziós tömb utánzására egy egydimenziós tömbben (például egy *valarray* objektumban). Például lássunk egy 3-szor 4-es mátrixot abban a formában, ahogy megszoktuk (§C.7):

00	01	02
10	11	12
20	21	22
30	31	32

A Fortran szokásainak megfelelően ezt a következő formában helyezhetnénk el a memóriában:

0				4				8			
00	10	20	30	01	11	21	31	02	12	22	32
0	1	2	3								

A C++ *nem* ezt a megoldást használja (lásd §C.7), ugyanakkor biztosítanunk kell egy rendszert, amely tiszta és logikus kezelőfelületet ad, és ehhez olyan ábrázolást kell választanunk, amely megfelel a probléma kötöttségeinek. Itt most a Fortran elrendezését választottam, hogy könnyen megvalósítható legyen az együttműködés az olyan numerikus prog-

ramokkal, melyek ezt az elvet követik. Annyira azért nem voltam hajlandó változtatni, hogy az indexelést *0* helyett *1*-ről indítsam – ha ilyen céljaink vannak, oldjuk meg a §22.9[9] feladatot. A legtöbb számműveletet továbbra is számos nyelv és még több könyvtár együttes használatával kell megoldanunk. A különböző nyelvek és könyvtárak eltérő formátumú adatainak kezelése gyakran létszükséglet.

Egy *x* sort a *slice(x,3,4)* kifejezéssel írhatunk le. Tehát az *x* sor első eleme a vektor *x -ik* eleme, a sor következő eleme az *(x+4)-ik* és így tovább. A fenti példában minden sor három elemet tartalmaz. Az ábrák szerint a *slice(0,3,4)* a *00*, a *01* és a *02* elemet jelöli.

Egy *y* oszlopot a *slice(4*y,4,1)* határoz meg. Az *y* oszlop első eleme a vektornak *4*y-ik* eleme, a következő oszlopelem a *(4*y+1)-ik* vektorelem stb. Minden oszlopban 4 elem szerepel. A *slice(0,4,1)* kifejezés tehát az ábrák szerint a *00*, a *10*, a *20* és a *30* elemet jelöli. Azon kívül, hogy könnyen utánozhatunk kétdimenziós tömböket, a *slice* segítségével számtalan más sorozatot is leírhatunk; az egyszerű sorozatok megadására így kellően általános módszert biztosít. Ezzel a §22.4.8 pont foglalkozik részletesen.

Egy szeletet elképzelhetünk úgy is, mint egy különleges bejárót (iterator): a *slice* lehetővé teszi, hogy leírjunk egy indexsorozatot egy *valarray* objektumban. Ez alapján egy valódi bejárót is felépíthetünk:

```
template<class T> class Slice_iter {
   valarray<T>* v;
   slice s;
   size_t curr;                          // az aktuális elem indexe

   T& ref(size_t i) const { return (*v)[s.start()+i*s.stride()]; }
public:
   Slice_iter(valarray<T>* vv, slice ss) :v(vv), s(ss), curr(0) { }

   Slice_iter end()
   {
            Slice_iter t = *this;
            t.curr = s.size();           // az utolsó utáni elem indexértéke
            return t;
   }
   Slice_iter& operator++() { curr++; return *this; }
   Slice_iter operator++(int) { Slice_iter t = *this; curr++; return t; }

   T& operator[ ](size_t i) { return ref(curr=i); }          // C stílusú index
   T& operator()(size_t i) { return ref(curr=i); }           // Fortran stílusú index
   T& operator*() { return ref(curr); }                      // az aktuális elem

   // ...
};
```

Mivel a *slice* rögzített méretű, tartományellenőrzést is végezhetünk. A fenti példában a *slice::size()* kihasználásával valósítottuk meg az *end()* műveletet, hogy a *valarray* utolsó utáni elemére állíthassuk a bejárót.

A *slice* leírhat egy sort és egy oszlopot is, így a *Slice_iter* is lehetővé teszi, hogy akár egy sort, akár egy oszlopot járjunk be a *valarray* objektumban.

Ahhoz, hogy a *Slice_iter* igazán jól használható legyen, meg kell határoznunk az ==, a != és a < operátort is:

```
template<class T> bool operator==(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
  return p.curr==q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}

template<class T> bool operator!=(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
  return !(p==q);
}

template<class T> bool operator<(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
  return p.curr<q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}
```

22.4.6. A slice_array

A *valarray* és a *slice* felhasználásával olyan szerkezetet építhetünk, amely úgy néz ki és úgy is viselkedik, mint egy *valarray*, pedig valójában csak a szelet által kijelölt területre hivatkozik. Ezt a *slice_array* típust a következőképpen definiálhatjuk:

```
template <class T> class std::slice_array {
public:
  typedef T value_type;

  void operator=(const valarray<T>&);
  void operator=(const T& val);                       // minden elem val-t kapja értékül

  void operator*=(const T& val);                      // v[i]*=val minden elemre
  // hasonlóan: /=, %=, +=, -=, ^=, &=, |=, <<=, >>=
```

```
    ~slice_array();
private:
    slice_array();                                  // létrehozás megakadályozása
    slice_array(const slice_array&);                // másolás megakadályozása
    slice_array& operator=(const slice_array&);     // másolás megakadályozása

    valarray<T>* p;                                 // megvalósítástól függő ábrázolás
    slice s;
};
```

A felhasználók nem hozhatnak létre közvetlenül egy *slice_array* objektumot, csak egy *valarray* „indexelésével" az adott szeletre. Miután a *slice_array* objektumnak kezdőértéket adtunk, minden rá történő hivatkozás közvetve arra a *valarray* objektumra hat majd, amelyhez létrehoztuk. Az alábbi formában például egy olyan szerkezetet hozhatunk létre, amely egy tömb minden második elemét választja ki:

```
void f(valarray<double>& d)
{
    slice_array<double>& v_even = d[slice(0,d.size()/2+d.size()%2,2)];
    slice_array<double>& v_odd = d[slice(1,d.size()/2,2)];

    v_even *= v_odd;        // az elempárok szorzatát tároljuk a páros elemekben
    v_odd = 0;              // 0 értékül adása a páratlan elemeknek
}
```

A *slice_array* objektumok másolását nem engedhetjük meg, mert csak így tehetjük lehetővé olyan optimalizációs módszerek alkalmazását, melyek kihasználják, hogy egy objektumra csak egyféleképpen, álnevek nélkül hivatkozhatunk. Ez a korlátozás néha nagyon kellemetlen:

```
slice_array<double> row(valarray<double>& d, int i)
{
    slice_array<double> v = d[slice(0,2,d.size()/2)];      // hiba: másolás kísérlete

    return d[slice(i%2,i,d.size()/2)];                       // hiba: másolás kísérlete
}
```

Egy *slice_array* másolása gyakran helyettesíthető a megfelelő *slice* másolásával.

A szeletek alkalmasak a tömbök bizonyos típusú részhalmazainak kifejezésére. A következő példában például szeleteket használunk egy folytonos résztartomány kezeléséhez:

```
inline slice sub_array(size_t first, size_t count)  // [first:first+count[
{
  return slice(first,count,1);
}

void f(valarray<double>& v)
{
  size_t sz = v.size();
  if (sz<2) return;
  size_t n = sz/2;
  size_t n2 = sz-n;

  valarray<double> half1(n);
  valarray<double> half2(n2);

  half1 = v[sub_array(0,n)];                  // v első felének másolása
  half2 = v[sub_array(n,n2)];                 // v második felének másolása

  // ...
}
```

A standard könyvtárban nem szerepel mátrix osztály. Ehelyett a *valarray* és a *slice* együttesen igyekszik biztosítani azokat az eszközöket, melyekkel különböző igényekhez igazított mátrixokat építhetünk fel. Nézzük meg, hogyan készíthetünk egy egyszerű, kétdimenziós mátrixot a *valarray* és a *slice_array* felhasználásával:

```
class Matrix {
  valarray<double>* v;
  size_t d1, d2;
public:
  Matrix(size_t x, size_t y);          // megjegyzés: nincs alapértelmezett konstruktor
  Matrix(const Matrix&);
  Matrix& operator=(const Matrix&);
  ~Matrix();

  size_t size() const { return d1*d2; }
  size_t dim1() const { return d1; }
  size_t dim2() const { return d2; }

  Slice_iter<double> row(size_t i);
  Cslice_iter<double> row(size_t i) const;

  Slice_iter<double> column(size_t i);
  Cslice_iter<double> column(size_t i) const;

  double& operator()(size_t x, size_t y);        // Fortran stílusú index
  double operator()(size_t x, size_t y) const;
```

```
    Slice_iter<double> operator()(size_t i) { return row(i); }
    Cslice_iter<double> operator()(size_t i) const { return row(i); }

    Slice_iter<double> operator[ ](size_t i) { return row(i); }                // C stílusú index
    Cslice_iter<double> operator[ ](size_t i) const { return row(i); }

    Matrix& operator*=(double);

    valarray<double>& array() { return *v; }
};
```

A *Matrix*-ot egy *valarray* ábrázolja. Erre a tömbre a kétdimenziós jelleget a szeletelés segítségével „húzhatjuk rá". Ezt az ábrázolást tekinthetjük egy-, két-, három- stb. dimenziós tömbnek is, ugyanúgy, ahogy az alapértelmezett kétdimenziós nézetet biztosítjuk a *row()* és a *column()* függvény megvalósításával. A *Slice_iter* objektumok segítségével megkerülhetjük a *slice_array* objektumok másolásának tilalmát. Egy *slice_array* típusú objektumot nem adhatunk vissza:

```
    slice_array<double> row(size_t i) { return (*v)(slice(i,d1,d2)); }  // hiba
```

Ezért a *slice_array* helyett egy bejárót (iterator) adunk meg, amely egy mutatót tartalmaz a *valarray* objektumra, illetve magát a *slice* objektumot.

Meg kell határoznunk egy további osztályt is, a „bejárót a konstans szeletekhez". Ez a *Cslice_iter*, amely egy *const Matrix* és egy nem konstans *Matrix* szeletei közötti különbséget fejezi ki.

```
inline Slice_iter<double> Matrix::row(size_t i)
{
    return Slice_iter<double>(v,slice(i,d1,d2));
}

inline Cslice_iter<double> Matrix::row(size_t i) const
{
    return Cslice_iter<double>(v,slice(i,d1,d2));
}

inline Slice_iter<double> Matrix::column(size_t i)
{
    return Slice_iter<double>(v,slice(i*d2,d2,1));
}

inline Cslice_iter<double> Matrix::column(size_t i) const
{
    return Cslice_iter<double>(v,slice(i*d2,d2,1));
}
```

A *Cslice_iter* definíciója megegyezik a *Slice_iter*-ével, a különbség mindössze annyi, hogy itt a szelet elemeire *const* referenciák fognak mutatni.

A többi tagfüggvény meglehetősen egyszerű:

```
Matrix::Matrix(size_t x, size_t y)
{
  // ellenőrzés: x és y értelmes-e
  d1 = x;
  d2 = y;
  v = new valarray<double>(x*y);
}

double& Matrix::operator()(size_t x, size_t y)
{
  return row(x)[y];
}

double mul(Cslice_iter<double>& v1, const valarray<double>& v2)
{
  double res = 0;
  for (size_t i = 0; i<v1.size(); i++) res+= v1[i]*v2[i];
  return res;
}

valarray<double> operator*(const Matrix& m, const valarray<double>& v)
{
  valarray<double> res(m.dim1());
  for (size_t i = 0; i<m.dim1(); i++) res[i] = mul(m.row(i),v);
  return res;
}

Matrix& Matrix::operator*=(double d)
{
  (*v) *= d;
  return *this;
}
```

A *Matrix* indexeléséhez az *(i,j)* jelölést használtam, mert a *( )* egy elég egyszerű operátor, és ez a jelölés a „matematikus társadalomban” eléggé elfogadott. A sor fogalma ennek ellenére lehetővé teszi a C és C++ világban megszokottabb *[i][j]* jelölést is:

```
void f(Matrix& m)
{
  m(1,2) = 5;                    // Fortran stílusú index
  m.row(1)(2) = 6;
```

```
  m.row(1)[2] = 7;
  m[1](2) = 8;                    // zavaros, kevert stílus (de működik)
  m[1][2] = 9;                    // C++ stílusú index
}
```

Ha a *slice_array* osztályt használjuk az indexeléshez, erős optimalizálásra van szükségünk.

Ennek a szerkezetnek az általánosítása *n*-dimenziós mátrixra és tetszőleges elemtípusra a megfelelő művelethalmaz biztosításával a §22.9[7] feladat témája.

Elképzelhető, hogy első ötletünk a kétdimenziós vektor megvalósítására a következő volt:

```
class Matrix {
  valarray< valarray<double> > v;
public:
  // ...
};
```

Ez a megoldás is működőképes (§22.9[10]), de így igen nehéz anélkül összeegyeztetni a hatékonyságot a nagyteljesítményű számítások által megkívánt követelményekkel, hogy a *valarray* és a *slice* osztály által ábrázolt alacsonyabb és hagyományosabb szintre térnénk át.

22.4.7. Ideiglenes változók, másolás, ciklusok

Ha megpróbálunk készíteni egy vektor vagy egy mátrix osztályt, rövid időn belül rájöhetünk, hogy három, egymással összefüggő problémát kell figyelembe vennünk a teljesítményt hangsúlyozó felhasználók igényeinek kielégítéséhez:

1. Az ideiglenes változók számát a lehető legkisebbre kell csökkentenünk.
2. A mátrixok másolásának számát a lehető legkisebbre kell csökkentenünk.
3. Az összetettebb műveletekben az ugyanazon adatokon végighaladó ciklusok számát a lehető legkisebbre kell csökkentenünk.

A standard könyvtár nem közvetlenül ezekkel a célokkal foglalkozik. Ennek ellenére létezik olyan eljárás, melynek segítségével optimális megoldást biztosíthatunk.

Vegyük például az $U=M*V+W$ műveletet, ahol U, V és W vektor, M pedig mátrix. A legegyszerűbb megoldásban külön-külön ideiglenes változóra van szükségünk az $M*V$ és az $M*V+W$ számára, és másolnunk kell az $M*V$ és az $M*V+W$ eredményét is. Egy „okosabb”

változatban elkészíthetjük a *mul_add_and_assign(&U, &M, &V, &W)* függvényt, amelynek nincs szüksége ideiglenes változókra, sem a vektorok másolására, és ráadásul minden mátrix minden elemét a lehető legkevesebbszer érinti.

Ilyen mérvű optimalizálásra egy-két kifejezésnél többször ritkán van szükségünk. Így a hatékonysági problémák megoldására a *mul_add_and_assign()* jellegű függvények írása, az egyik legegyszerűbb módszer melyeket a felhasználó szükség esetén meghívhat. Ugyanakkor lehetőség van olyan *Matrix* kifejlesztésére is, amely automatikusan alkalmaz ilyen optimalizációt, ha a kifejezéseket megfelelő formában fogalmazzuk meg. A lényeg az, hogy az *U=M*V+W* kifejezést tekinthetjük egyetlen, négy operandussal rendelkező operátor alkalmazásának. Az eljárást már bemutattuk az *ostream* módosítóknál (§21.4.6.3). Az ott bemutatott módszer általánosan használható arra, hogy *n* darab kétoperandusú operátor együttesét egy *(n+1)* paraméterű operátorként kezeljük. Az *U=M*V+W* kifejezés kezeléséhez be kell vezetnünk két segédosztályt. Ennek ellenére bizonyos rendszerekben ez a eljárás igen jelentős (akár 30-szoros) sebességnövekedést is elérhet azzal, hogy további optimalizálást tesz lehetővé.

Először is meg kell határoznunk egy *Matrix* és egy *Vector* szorzásakor képződő eredmény típusát:

```
struct MVmul {
   const Matrix& m;
   const Vector& v;

   MVmul(const Matrix& mm, const Vector &vv) :m(mm), v(vv) { }

   operator Vector();              // az eredmény kiszámítása és visszaadása
};

inline MVmul operator*(const Matrix& mm, const Vector& vv)
{
   return MVmul(mm,vv);
}
```

A „szorzás” semmi mást nem tesz, mint hogy referenciákat tárol az operandusairól; az *M*V* tényleges kiszámítását későbbre halasztjuk. Az objektum, amelyet a * művelet eredményez, igen közel áll ahhoz, amit gyakran *lezárás* (closure; inkább „befoglaló objektum”) néven emlegetnek. Ugyanígy kezelhetjük egy *Vector* hozzáadását az eddigi eredményhez:

```
struct MVmulVadd {
   const Matrix& m;
   const Vector& v;
   const Vector& v2;
```

```
    MVmulVadd(const MVmul& mv, const Vector& vv) :m(mv.m), v(mv.v), v2(vv) { }

    operator Vector();              // az eredmény kiszámítása és visszaadása
};

inline MVmulVadd operator+(const MVmul& mv, const Vector& vv)
{
    return MVmulVadd(mv,vv);
}
```

Ezzel az *M*V+W* kifejezés kiszámítását is elhalasztjuk. Most már csak azt kell biztosítanunk, hogy hatékony algoritmussal számoljuk ki a tényleges eredményt, amikor a kifejezés eredményét értékül adjuk egy *Matrix* objektumnak:

```
class Vector {
    // ...
public:
    Vector(const MVmulVadd& m)                  // kezdeti értékadás m eredményével
    {
            // elemek lefoglalása, stb.
            mul_add_and_assign(this,&m.m,&m.v,&m.v2);
    }

    Vector& operator=(const MVmulVadd& m)       // m eredményének értékül adása *this-nek
    {
            mul_add_and_assign(this,&m.m,&m.v,&m.v2);
            return *this;
    }
    // ...
};
```

Így tehát az *U=M*V+W* kifejezés automatikusan a következő kifejezésre változik:

```
U.operator=(MVmulVadd(MVmul(M,V),W))
```

Ezt pedig a helyben kifejtés alkalmazása az eredeti, megkívánt függvényhívásra alakítja:

```
mul_add_and_assign(&U,&M,&V,&W)
```

Ezzel a megoldással kiküszöböltük a másolásokat és az ideiglenes változókat, ráadásul a *mul_add_and-assign()* függvényen további optimalizálást is végrehajthatunk. Sőt, ha a *mul_add_and_assign()* függvényt a lehető legegyszerűbb módon készítjük el, akkor is olyan formát alakítottunk ki, amely az automatikus optimalizáló eszközök számára sokkal kedvezőbb.

A fenti példában egy új *Vector* típust használtunk (nem az egyszerű *valarray* osztályt), mert új értékadó operátorokat kellett meghatároznunk és az értékadásoknak mindenképpen tagfüggvényeknek kell lenniük (§11.2.2). Nagy valószínűséggel azonban a *valarray* osztályt használjuk a *Vector* megvalósításához. Ennek a módszernek abban áll a jelentősége, hogy a leginkább időigényes vektor- és mátrixműveletek elvégzését néhány egyszerű nyelvi elemmel fogalmazzuk meg. Annak igazán soha sincs értelme, hogy így optimalizáljunk egy féltucat operátort használó kifejezést, ezekre a hagyományosabb eljárások (§11.6) is megfelelőek.

Az ötlet lényege, hogy fordítási idejű vizsgálatokat végzünk és befoglaló objektumokat (closure) használunk, azzal a céllal, hogy a részkifejezések kiszámítását a teljes műveletet ábrázoló objektumra bízzuk. Az elv számos területen használható; lényege, hogy több önálló információelemet egyetlen függvénybe gyűjtünk össze, mielőtt a tényleges műveleteket elkezdenénk. A késleltetett kiszámítás érdekében létrehozott objektumokat ***kompozíciós lezárt (befoglaló) objektumoknak*** (composition closure object) vagy egyszerűen ***kompozitoroknak*** (compositor) nevezzük.

22.4.8. Általánosított szeletek

A §22.4.6 pontban szereplő *Matrix* példa megmutatta, hogyan használhatunk két *slice* objektumot egy kétdimenziós tömb sorainak és oszlopainak leírására. Általánosan az igaz, hogy egy *slice* alkalmas egy *n*-dimenziós tömb bármely sorának vagy oszlopának kijelölésére (§22.9[7]), de néha szükségünk lehet olyan résztömb kijelölésére is, amely nem egyetlen sora, vagy egyetlen oszlopa az *n*-dimenziós tömbnek. Tegyük fel, hogy egy 3-szor 4-es mátrix bal felső sarkában elhelyezkedő 2-szer 3-as részmátrixot akarunk kijelölni:

00	01	02
10	11	12
20	21	22
30	31	32

Sajnos ezek az elemek nem úgy helyezkednek el a memóriában, hogy egy *slice* segítségével leírhatnánk azokat:

0	1	2									
00	10	20	30	01	11	21	31	02	12	22	32
				4	5	6					

A *gslice* egy olyan általánosított *slice*, amely (majdnem) *n* darab szelet információit tartalmazza:

```
class std::gslice {
  // ahelyett, hogy a slice-hoz hasonlóan 1 lépésköz és 1 méret lenne,
  // a gslice-ban n lépésköz és n méret van
public:
  gslice();
  gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

  size_t start() const;                // az első elem indexértéke
  valarray<size_t> size() const;       // elemek száma a dimenzióban
  valarray<size_t> stride() const;     // lépésköz index[0], index[1], ... között
};
```

Az új értékek lehetővé teszik, hogy a *gslice* leképezést határozzon meg *n* darab egész és egy index között, amit a tömb elemeinek eléréséhez használhatunk fel. Egy 2-szer 3-as mátrix elhelyezkedését például két (hosszúság, lépésköz) elempárral írhatjuk le. A §22.4.5 pontban bemutattuk, hogy a *2* hosszúság és a *4* lépésköz a 3-szor 4-es mátrix egy sorának két elemét írja le, ha a Fortran elrendezését használjuk. Ugyanígy, a *3* hosszúság és az *1* lépésköz egy oszlop 3 elemét jelöli ki. A kettő együtt képes leírni egy 2-szer 3-as részmátrix összes elemét. Az elemek felsorolásához az alábbi eljárásra lesz szükségünk:

```
size_t gslice_index(const gslice& s, size_t i, size_t j)
{
  return s.start()+i*s.stride()[0]+j*s.stride()[1];
}

size_t len[ ] = { 2, 3 };              // (len[0],str[0]) egy sort ír le
size_t str[ ] = { 4, 1 };              // (len[1],str[1]) egy oszlopot ír le

valarray<size_t> lengths(len,2);
valarray<size_t> strides(str,2);

void f()
{
  gslice s(0,lengths,strides);

  for (int i = 0 ; i<s.size()[0]; i++) cout << gslice_index(s,i,0) << " ";      // sor
  cout << ", ";
  for (int j = 0 ; j<s.size()[1]; j++) cout << gslice_index(s,0,j) << " ";      // oszlop
}
```

Az eredmény *0 4 , 0 1 2* lesz.

Ezzel a módszerrel egy *gslide* két (hosszúság, lépésköz) pár segítségével képes megadni egy kétdimenziós tömb tetszőleges részmátrixát. Három (hosszúság, lépésköz) párral leírhatjuk egy háromdimenziós tömb résztömbjeit és így tovább. Ha egy *gslice* objektumot használunk egy *valarray* indexeléséhez, akkor egy *gslice_array* típusú objektumot kapunk, ami a *gslice* által kijelölt elemeket tartalmazza. Például:

```
void f(valarray<float>& v)
{
  gslice m(0,lengths,strides);
  v[m] = 0;           // értékül adja 0-t v[0],v[1],v[2],v[4],v[5],v[6] elemeknek
}
```

A *gslice_array* ugyanazokat a tagfüggvényeket biztosítja, mint a *slice_array*, így egy *gslice_array* objektumot sem hozhat létre közvetlenül a felhasználó, és nem is másolhatja (§22.4.6), viszont *gslice_array* objektumot kapunk eredményként, ha egy *valarray* (§22.4.2) objektumot egy *gslice* objektummal indexelünk.

22.4.9. Maszkok

A *mask_array* típus a *valarray* tömb valamely része kijelölésének másik módja. Ráadásul az eredményt is egy *valarray*-szerű formában kapjuk meg. A *valarray* osztály szempontjából a maszk egyszerűen egy *valarray<bool>* objektum. Amikor maszkot használunk egy *valarray* indexeléséhez, a *true* bitek jelzik, hogy a *valarray* megfelelő elemét az eredményben is meg szeretnénk kapni. Ez a megoldás lehetővé teszi, hogy egy *valarray* objektumnak olyan részhalmazát dolgozzuk fel, amely nem valamilyen egyszerű elrendezésben (például egy szeletben) helyezkedik el:

```
void f(valarray<double>& v)
{
  bool b[ ] = { true , false, false, true, false, true };
  valarray<bool> mask(b,6);                // a 0, 3, és 5 elem

  valarray<double> vv = cos(v[mask]);      // vv[0]==cos(v[0]), vv[1]==cos(v[3]),
                                           // vv[2]==cos(v[5])
}
```

A *mask_array* osztály ugyanazokat a műveleteket biztosítja, mint a *slice_array*. Egy *mask_array* objektumot sem hozhatunk létre közvetlenül, és nem is másolhatjuk (§22.4.6). Egy *mask_array* meghatározásához egy *valarray* (§22.4.2) objektumot kell indexelnünk *valarray<bool>* objektummal. A maszkoláshoz használt *valarray* mérete nem lehet nagyobb, mint azon *valarray* elemeinek száma, amelyben ezt indexelésre akarjuk használni.

22.4.10. Indirekt tömbök

Az *indirect_array* (indirekt vagyis közvetett tömb) lehetőséget ad arra, hogy egy *valarray* objektumot tetszőlegesen indexeljünk és átrendezzünk:

```
void f(valarray<double>& v)
{
    size_t i[ ] = { 3, 2, 1, 0 };              // az első 4 elem fordított sorrendben
    valarray<size_t> index(i,4);              // a 3, 2, 1, 0 elemek (ebben a sorrendben)

    valarray<double> vv = log(v[index]);      // vv[0]==log(v[3]), vv[1]==log(v[2]),
                                              // vv[2]==log(v[1]), vv[3]==log(v[0])
}
```

Ha egy sorszám kétszer szerepel, akkor a *valarray* objektum valamely elemére kétszer hivatkozunk egy műveleten belül. Ez pontosan az a típusú többszörös hivatkozás (álnév), amit a *valarray* nem enged meg. Így az *indirect_array* működése nem meghatározott, ha többször használjuk ugyanazt az indexértéket.

Az *indirect_array* osztály ugyanazokat a műveleteket biztosítja, mint a *slice_array*, tehát nem hozhatunk létre közvetlenül *indirect_array* objektumokat, és nem is másolhatjuk azokat (§22.4.6); erre a célra egy *valarray* (§22.4.2) objektum *valarray<size_t>* objektummal való indexelése szolgál.

22.5. Komplex aritmetika

A standard könyvtár tartalmaz egy *complex* sablont, körülbelül azokkal a tulajdonságokkal, amelyekkel a §11.3 pontban szereplő *complex* osztályt meghatároztuk. A könyvtárban szereplő *complex* osztálynak azért kell sablonként (template) szerepelnie, hogy különböző skalár típusokra épülő komplex számokat is kezelni tudjunk. A könyvtárban külön-külön változat szerepel a *float*, a *double* és a *long double* skalártípushoz.

A *complex* osztály az *std* névtérhez tartozik és a *<complex>* fejállomány segítségével érhető el:

```
template<class T> class std::complex {
    T re, im;
public:
    typedef T value_type;
```

```
    complex(const T& r = T(), const T& i = T()) : re(r), im(i) { }
    template<class X> complex(const complex<X>& a) : re(a.real()), im(a.imag()) { }

    T real() const { return re; }
    T imag() const { return im; }

    complex<T>& operator=(const T& z);                    // complex(z,0) értékül adása
    template<class X> complex<T>& operator=(const complex<X>&);
    // hasonlóan: +=, -=, *=, /=
};
```

Az itt bemutatott megvalósítás és a helyben kifejtett (inline) függvények csak illusztrációként szerepelnek. Ha nagyon akarunk, el tudunk képzelni más megvalósítást is a standard könyvtár *complex* osztályához. Figyeljük meg a sablon tagfüggvényeket, amelyek lehetővé teszik, hogy bármilyen típusú *complex* objektumnak egy másikkal adjunk kezdőértéket, vagy azt egyszerű értékadással rendeljük hozzá (§13.6.2).

A könyv folyamán a *complex* osztályt nem sablonként használtuk, csak „egyszerű” osztályként. Erre azért volt lehetőség, mert a névterekkel ügyeskedve a *double* értékek *complex* osztályát tettük „alapértelmezetté”:

```
typedef std::complex<double> complex;
```

Rendelkezésünkre állnak a szokásos egyoperandusú (unáris) és kétoperandusú (bináris) operátorok is:

```
template<class T> complex<T> operator+(const complex<T>&, const complex<T>&);
template<class T> complex<T> operator+(const complex<T>&, const T&);
template<class T> complex<T> operator+(const T&, const complex<T>&);

// hasonlóan: -, *, /, ==, és !=

template<class T> complex<T> operator+(const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&);
```

A koordináta-függvények a következők:

```
template<class T> T real(const complex<T>&);
template<class T> T imag(const complex<T>&);

template<class T> complex<T> conj(const complex<T>&);

// polár koordinátarendszer szerinti létrehozás (abs(),arg()):
```

```
template<class T> complex<T> polar(const T& rho, const T& theta);

template<class T> T abs(const complex<T>&);          // néha rho a neve
template<class T> T arg(const complex<T>&);          // néha theta a neve

template<class T> T norm(const complex<T>&);         // az abs() négyzetgyöke
```

A szokásos matematikai függvényeket is használhatjuk:

```
template<class T> complex<T> sin(const complex<T>&);
// hasonlóan: sinh, sqrt, tan, tanh, cos, cosh, exp, log, és log10

template<class T> complex<T> pow(const complex<T>&,int);
template<class T> complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow(const complex<T>&, const complex<T>&);
template<class T> complex<T> pow(const T&, const complex<T>&);
```

A ki- és bemeneti adatfolyamok kezelésére pedig a következők szolgálnak:

```
template<class T, class Ch, class Tr>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, complex<T>&);
template<class T, class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const complex<T>&);
```

A komplex számok kiírási formája *(x,y)*, míg beolvasásra használhatjuk az *x*, az *(x)* és az *(x,y)* formátumot is (§21.2.3, §21.3.5). A *complex<float>*, a *complex<double>* és a *complex<long double>* specializációk azért szerepelnek, hogy korlátozzuk a konverziókat (§13.6.2) és lehetőséget adjunk az optimalizálásra is:

```
template<> class complex<double> {
  double re, im;
public:
  typedef double value_type;

  complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
  complex(const complex<float>& a) : re(a.real()), im(a.imag()) { }
  explicit complex(const complex<long double>& a) : re(a.real()), im(a.imag()) { }

  // ...
};
```

Ezek után egy *complex<float>* érték gond nélkül *complex<double>* számra konvertálható, de egy *complex<long double>* már nem. Ugyanilyen specializációk biztosítják, hogy egy *complex<float>* vagy egy *complex<double>* automatikusan konvertálható *complex<long double>* értékre, de egy *complex<long double>* nem alakítható „titokban" *complex<float>* vagy *complex<double>* számmá. Érdekes módon az értékadások nem ugyanazt a védelmet biztosítják, mint a konstruktorok:

```
void f(complex<float> cf, complex<double> cd, complex<long double> cld, complex<int> ci)
{
  complex<double> c1 = cf;          // jó
  complex<double> c2 = cd;          // jó
  complex<double> c3 = cld;         // hiba: esetleg csonkol
  complex<double> c4(cld);          // rendben: explicit konverzió
  complex<double> c5 = ci;          // hiba: nincs konverzió

  c1 = cld;                         // rendben, de vigyázat: esetleg csonkol
  c1 = cf;                          // rendben
  c1 = ci;                          // rendben
}
```

22.6. Általánosított numerikus algoritmusok

A *<numeric>* fejállományban a standard könyvtár biztosít néhány általánosított numerikus algoritmust is, az *<algorithm>* fejállomány (18. fejezet) nem numerikus algoritmusainak stílusában:

Általánosított numerikus algoritmusok *<numeric>*	
accumulate()	Egy sorozat elemein végez el egy műveletet.
inner_product()	Két sorozat elemein végez el egy műveletet.
partial_sum()	Egy sorozatot állít elő egy másik sorozatra alkalmazott művelettel.
adjacent_difference()	Egy sorozatot állít elő egy másik sorozatra alkalmazott művelettel.

Ezek az algoritmusok egy-egy jellegzetes, gyakran használt műveletet általánosítanak. Az egyik például kiszámítja egy sorozat elemeinek összegét, de úgy, hogy bármilyen típusú sorozatra használhassuk az eljárást, a műveletet pedig, amit a sorozat elemeire el kell végezni, paraméterként adjuk meg. Mindegyik algoritmus esetében az általános műveletet kiegészíti egy olyan változat, amely közvetlenül a leggyakoribb operátort használja az adott algoritmushoz.

22.6.1. Az accumulate()

Az *accumulate()* algoritmust felfoghatjuk úgy, mint egy vektor elemeit összegző eljárás általánosítását. Az *accumulate()* algoritmus az *std* névtérhez tartozik és a *<numeric>* fejállományból érhetjük el:

```
template <class In, class T> T accumulate(In first, In last, T init)
{
  while (first != last) init = init + *first++;          // összeadás
  return init;
}

template <class In, class T, class BinOp> T accumulate(In first, In last, T init, BinOp op)
{
  while (first != last) init = op(init,*first++);        // általános művelet
  return init;
}
```

Az *accumulate()* legegyszerűbb változata egy sorozat elemeit adja össze, az elemekre értelmezett + operátor segítségével:

```
void f(vector<int>& price, list<float>& incr)
{
  int i = accumulate(price.begin(),price.end(),0);          // int-be összegzés
  double d = 0;
  d = accumulate(incr.begin(),incr.end(),d);                // double-ba összegzés
  // ...
}
```

A visszatérési érték típusát az átadott kezdőérték típusa határozza meg.

Gyakran azok az értékek, melyeket összegezni szeretnénk, nem egy sorozat elemeiként állnak rendelkezésünkre. Ha ilyen helyzetbe kerülünk, akkor az *accumulate()* számára megadható műveletet felhasználhatjuk arra is, hogy az értékeket előállítsuk. A legegyszerűbb ilyen eljárás csak annyit tesz, hogy a sorozatban tárolt adatszerkezetből kiválasztja a kívánt értéket:

```
struct Record {
  // ...
  int unit_price;
  int number_of_units;
};

long price(long val, const Record& r)
{
   return val + r.unit_price * r.number_of_units;
}

void f(const vector<Record>& v)
{
  cout << "Összeg: " << accumulate(v.begin(),v.end(),0,price) << '\n';
}
```

Az *accumulate()* szerepének megfelelő műveletet egyes fejlesztők *reduce* vagy *reduction* néven valósítják meg.

22.6.2. Az inner_product

Egyetlen sorozat elemeinek összegzése, illetve az ilyen jellegű műveletek nagyon gyakoriak, ugyanakkor a sorozatpárokon ilyen műveletet végző eljárások viszonylag ritkák. Az *inner_product()* algoritmus meghatározása az *std* névtérben szerepel, deklarációját pedig a *<numeric>* fejállományban találhatjuk meg:

```
template <class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
{
  while (first != last) init = init + *first++ * *first2++;
  return init;
}

template <class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
  while (first != last) init = op(init,op2(*first++,*first2++));
  return init;
}
```

Szokás szerint, a második sorozatnak csak az elejét kell megadnunk paraméterként. A második bemeneti sorozatról azt feltételezzük, hogy legalább olyan hosszú, mint az első.

Amikor egy *Matrix* objektumot egy *valarray* objektummal akarunk összeszorozni, akkor a legfontosabb művelet az *inner_product*:

```
valarray<double> operator*(const Matrix& m, valarray<double>& v)
{
  valarray<double> res(m.dim2());

  for (size_t i=0; i<m.dim1(); i++) {
          const Cslice_iter<double>& ri = m.row(i);
          res[i] = inner_product(ri,ri.end(),&v[0], double(0));
  }
  return res;
}

valarray<double> operator*(valarray<double>& v, const Matrix& m)
{
  valarray<double> res(m.dim1());

  for (size_t i=0; i<m.dim2(); i++) {
          const Cslice_iter<double>& ci = m.column(i);
          res[i] = inner_product(ci, ci.end(), &v[0], double(0));
  }
  return res;
}
```

Az *inner_product* (belső szorzat) bizonyos formáit gyakran emlegetjük „pont-szorzás" (dot product) néven is.

22.6.3. Növekményes változás

A *partial_sum()* és az *adjacent_difference()* algoritmus egymás fordítottjának (inverzének) tekinthető, de mindkettő az inkrementális vagy növekményes változás (incremental change) fogalmával foglalkozik. Leírásuk az *std* névtérben és a *<numeric>* fejállományban szerepel:

```
template <class In, class Out> Out adjacent_difference(In first, In last, Out res);

template <class In, class Out, class BinOp>
  Out adjacent_difference(In first, In last, Out res, BinOp op);
```

Az *a, b, c, d, …* sorozatból például az *adjacent_difference()* a következő sorozatot állítja elő: *a, b-a, c-b, d-c, …*

Képzeljünk el egy sorozatot, amely hőmérséklet-értékeket tartalmaz. Ebből könnyen készíthetünk egy olyan vektort, amely a hőmérséklet-változásokat tartalmazza:

```
vector<double> temps;

void f()
{
  adjacent_difference(temps.begin(),temps.end(),temps.begin());
}
```

A *17, 19, 20, 20, 17* sorozatból például a *17, 2, 1, 0, -3* eredményt kapjuk.

Megfordítva, a *partial_sum()* azt teszi lehetővé, hogy több növekményes változás végeredményét kiszámítsuk:

```
template <class In, class Out, class BinOp>
Out partial_sum(In first, In last, Out res, BinOp op)
{
  if (first==last) return res;
  *res = *first;
  T val = *first;
  while (++first != last) {
            val = op(val,*first);
            *++res = val;
  }
  return ++res;
}

template <class In, class Out> Out partial_sum(In first, In last, Out res)
{
  return partial_sum(first,last,res,plus);          // §18.4.3
}
```

Az *a, b, c, d, …* sorozatból a *partial_sum() a* következő eredményt állítja elő: *a, a+b, a+b+c, a+b+c+d, …* Például:

```
void f()
{
  partial_sum(temps.begin(),temps.end(),temps.begin());
}
```

Figyeljük meg, hogy a *partial_sum()* növeli a *res* értékét, mielőtt új értéket adna a hivatkozott területnek. Ez lehetővé teszi, hogy a *res* ugyanaz a sorozat legyen, mint a bemeneti. Ugyanígy működik az *adjacent_difference()* is. Tehát a következő utasítás az *a, b, c, d* sorozatot az *a, a+b, a+b+c, a+b+c+d* sorozatra képezi le:

```
partial_sum(v.begin(),v.end(),v.begin());
```

Az alábbi utasítással pedig visszaállíthatjuk az eredeti sorozatot:

```
adjacent_difference(v.begin(),v.end(),v.begin());
```

Egy másik példa: a *partial_sum()* a *17, 2, 1, 0, -3* sorozatból a *17, 19, 20, 20, 17* sorozatot állítja vissza.

Akik szerint a hőmérséklet-változások figyelése csupán rendkívül unalmas részletkérdése a meteorológiának vagy a tudományos, laboratóriumi kísérleteknek, azok számára megjegyezzük, hogy a készlettartalékok változásainak vizsgálata pontosan ugyanezen a két műveleten alapul.

22.7. Véletlenszámok

A véletlenszámok fontos szerepet töltenek be igen sok szimulációs és játékprogramban. A *<cstdlib>* és az *<stdlib.h>* fejállományban a standard könyvtár egyszerű alapot biztosít a véletlenszámok előállításához:

```
#define RAND_MAX megvalósítás_függő /* nagy pozitív egész */

int rand();          // ál-véletlenszám 0 és RAND_MAX között
int srand(int i);    // a véletlenszám-előállító beállítása i-vel
```

Jó véletlenszám-előállító (generátor) készítése nagyon nehéz feladat, ezért sajnos nem minden rendszerben áll rendelkezésünkre jó *rand()* függvény. Különösen igaz ez a véletlenszámok alsó bitjeire, aminek következtében a *rand()%n* kifejezés egyáltalán nem tekinthető általános (más rendszerre átvihető) módszernek arra, hogy *0* és *n-1* közé eső véletlenszámokat állítsunk elő. A *(double(rand())/RAND_MAX)*n* általában sokkal elfogadhatóbb eredményt ad.

Az *srand()* függvény meghívásával egy új véletlenszám-sorozatot kezdhetünk a paraméterként megadott alapértéktől. A programok tesztelésekor gyakran nagyon hasznos, hogy egy adott alapértékből származó véletlenszám-sorozatot meg tudjunk ismételni. Ennek ellenére általában a program minden futtatásakor más alapértékről akarunk indulni. Ahhoz hogy játékainkat tényleg megjósolhatatlanná tegyük, gyakran szükség van arra, hogy az alapértéket a program környezetéből szerezzük be. Az ilyen programokban a számítógép órája által jelzett idő néhány bitje általában jó alapérték.

Ha saját véletlenszám-előállítót kell készítenünk, elengedhetetlen az alapos tesztelés (§22.9[14]).

Egy véletlenszám-előállító gyakran használhatóbb, ha osztály formájában áll rendelkezésünkre. Így könnyen készíthetünk véletlenszám-előállítókat a különböző értéktartományokhoz:

```
class Randint {                          // egyenletes eloszlás 32 bites long-ot feltételezve
   unsigned long randx;
public:
   Randint(long s = 0) { randx=s; }
   void seed(long s) { randx=s; }

   // bűvös számok: 31 bitet használunk egy 32 bites long-ból:

   long abs(long x) { return x&0x7fffffff; }
   static double max() { return 2147483648.0; }           // figyelem: double
   long draw() { return randx = randx*1103515245 + 12345; }

   double fdraw(){ return abs(draw())/max(); }            //a [0,1] tartományban

   long operator()() { return abs(draw()); }              //a [0,pow(2,31)] tartományban
};

class Urand : public Randint {                 // egyenletes eloszlás [0:n[ intervallumban
   long n;
public:
   Urand(long nn) { n = nn; }

   long operator()() { long r = n*fdraw(); return (r==n) ? n-1 : r; }
};

class Erand : public Randint {           // exponenciális eloszlású véletlenszám-előállító
   long mean;
public:
   Erand(long m) { mean=m; }
   long operator()() { return -mean * log( (max()-draw())/max() + .5); }
};
```

Íme egy rövid próbaprogram:

```
int main()
{
  Urand draw(10);
  map<int,int> bucket;
  for (int i = 0; i< 1000000; i++) bucket[draw()]++;
  for(int j = 0; j<10; j++) cout << bucket[j] << '\n';
}
```

Ha nem minden *bucket* értéke van a 100 000 közelében, valahol valamilyen hibát vétettünk.

Ezek a véletlenszám-előállítók annak a megoldásnak kissé átalakított változatai, amit a C++ könyvtár legelső megvalósításában (pontosabban az első „osztályokkal bővített C" könyvtárban, §1.4) alkalmaztam.

22.8. Tanácsok

[1] A numerikus problémák gyakran nagyon árnyaltak. Ha nem vagyunk 100 százalékig biztosak valamely probléma matematikai vonatkozásaiban, mindenképpen kérjük szakember segítségét vagy kísérletezzünk sokat. §22.1.

[2] A beépített adattípusok tulajdonságainak megállapításához használjuk a *numeric_limits* osztályt. §22.2.

[3] A felhasználói skalártípusokhoz adjunk meg külön *numeric_limits* osztályt. §22.2.

[4] Ha a futási idejű hatékonyság fontosabb, mint a rugalmasság az elemtípusokra és a műveletekre nézve, a számműveletekhez használjuk a *valarray* osztályt. §22.4.

[5] Ha egy műveletet egy tömb részére kell alkalmaznunk, ciklusok helyett használjunk szeleteket. §22.4.6.

[6] Használjunk kompozitorokat, ha egyedi algoritmusokkal és ideiglenes változók kiküszöbölésével akarjuk növelni rendszerünk hatékonyságát. §22.4.7.

[7] Ha komplex számokkal kell számolnunk, használjuk az *std::complex* osztályt. §22.5.

[8] Ha egy értéket egy lista elemeinek végignézésével kell kiszámítanunk, akkor mielőtt saját ciklus megvalósításába kezdenénk, vizsgáljuk meg, nem felel-e meg céljainknak az *accumulate()*, az *inner_product()* vagy az *adjacent_difference()* algoritmus. §22.6.

[9] Az adott eloszlásokhoz készített véletlenszám-előállító osztályok hasznosabbak, mint a *rand()* közvetlen használata. §22.7.

[10] Figyeljünk rá, hogy véletlenszámaink valóban véletlenek legyenek. §22.7.

22.9. Gyakorlatok

1. (*1.5) Készítsünk egy függvényt, amely úgy viselkedik, mint az *apply()*, azzal a különbséggel, hogy nem tagfüggvény és függvényobjektumokat is elfogad.
2. (*1.5) Készítsünk függvényt, amely csak abban tér el az *apply()* függvénytől, hogy nem tagfüggvényként szerepel, elfogad függvényobjektumokat, és a saját *valarray* paraméterét módosítja.
3. (*2) Fejezzük be a *Slice_iter*-t (§22.4.5). Különösen figyeljünk a destruktor létrehozásakor.
4. (*1.5) Írjuk meg újból a §17.4.1.3 pontban szereplő programot az *accumulate()* felhasználásával.
5. (*2) Készítsük el a << és a >> ki-, illetve bemeneti operátort a *valarray* osztályhoz. Készítsünk egy *get_array()* függvényt, amely úgy hoz létre egy *valarray* objektumot, hogy annak méretét is maga a bemenet adja meg.
6. (*2.5) Határozzunk meg és készítsünk el egy háromdimenziós tömböt a megfelelő műveletek létrehozásával.
7. (*2.5) Határozzunk meg és készítsünk el egy n-dimenziós mátrixot a megfelelő műveletek létrehozásával.
8. (*2.5) Készítsünk egy *valarray*-szerű osztályt, és adjuk meg hozzá a + és a * műveletet. Hasonlítsuk össze ennek hatékonyságát a saját C++-változatunk *valarray* osztályának hatékonyságával. Ötlet: próbáljuk ki többek között az *x=0.5*(x+y)-z* kifejezés kiértékelését különböző méretű *x*, *y* és *z* vektor felhasználásával.
9. (*3) Készítsünk egy Fortran stílusú tömböt (például *Fort_array* néven), ahol az indexek nem *0*-tól, hanem *1*-től indulnak.
10. (*3) Készítsük el a *Matrix* osztályt úgy, hogy az egy saját *valarray* objektumot használjon az elemek ábrázolásához (nem pedig egy mutatót vagy referenciát a *valarray* objektumra).
11. (*2.5) Kompozitorok (§22.4.7) segítségével készítsünk egy hatékony többdimenziós indexelési rendszert a *[]* jelöléssel. A következő kifejezések mindegyike a megfelelő elemeket, illetve résztömböket jelölje ki, méghozzá egyszerű indexműveletek segítségével: *v1[x]*, *v2[x][y]*, *v2[x]*, *v3[x][y][z]*, *v3[x][y]*, *v3[x]* stb.

12. (*2) A §22.7 pontban szereplő program ötletének általánosításával írjunk olyan függvényt, amely paraméterként egy véletlenszám-előállítót fogad és egyszerű grafikus formában szemlélteti annak eloszlását, úgy, hogy segítségével szemléletesen is ellenőrizhessük a véletlenszám-előállító helyességét.
13. (*1) Ha *n* egy *int* érték, akkor milyen a *(double(rand())/RAND_MAX)*n* kifejezés eloszlása?
14. (*2.5) Rajzoljunk pontokat egy négyzet alakú területen. Az egyes pontok koordináta-párjait az *Urand(N)* osztály segítségével állítsuk elő, ahol *N* a megjelenítési terület oldalának képpontban mért hossza. Milyen következtetést vonhatunk le az eredmény alapján az *Urand* által létrehozott véletlenszámok eloszlásáról?
15. (*2) Készítsünk egy Normal eloszlású véletlenszám-előállítót *Nrand* néven.

Negyedik rész
Tervezés a C++ segítségével

Ez a rész a programfejlesztés átfogóbb szemszögéből mutatja be a C++-t és az általa támogatott eljárásokat. A hangsúlyt a tervezésre és a nyelv lehetőségein alapuló hatékony megvalósításra fektetjük.

Fejezetek

23. Fejlesztés és tervezés
24. Tervezés és programozás
25. Az osztályok szerepe

„…Csak most kezdem felfedezni, milyen nehéz a gondolatainkat papírra vetni. Amíg csupán leírásból áll, elég könnyen megy, de amint érvelni kell, a gondolatok között megfelelő kapcsolatokat kell teremteni, világosan és gördülékenyen kell fogalmazni, s ez, mint mondottam, számomra oly nehézséget jelent, amire nem is gondoltam…”

(Charles Darwin)

23

Fejlesztés és tervezés

„Nincs ezüstgolyó.”
(F. Brooks)

Programépítés • Célok és eszközök • A fejlesztési folyamat • fejlesztés ciklus • Tervezési célok • Tervezési lépések • Osztályok keresése • Műveletek meghatározása • Függések meghatározása • Felületek meghatározása • Osztályhierarchiák újraszervezése • Modellek • Kísérletezés és elemzés • Tesztelés • A programok karbantartása • Hatékonyság • Vezetés • Újrahasznosítás • Méret és egyensúly • Az egyének fontossága • Hibrid tervezés • Bibliográfia – Tanácsok

23.1. Áttekintés

Ez az első a három fejezetből, melyek részletesebben bemutatják a szoftvertermék készítését, a viszonylag magas szintű tervezési szemléletmódtól azokig a programozási fogalmakig és eljárásokig, amelyekkel a C++ a tervezést közvetlenül támogatja. Ez a fejezet a bevezetőn és a szoftverfejlesztés céljait és eszközeit röviden tárgyaló §23.3 ponton kívül két fő részből áll:

§23.4 A szoftverfejlesztés folyamatának áttekintése.
§23.5 Gyakorlati tanácsok a szoftverfejlesztői munka megszervezéséhez.

A 24. fejezet a tervezés és a programozási nyelv közti kapcsolatot tárgyalja, a 25. pedig az osztályoknak a tervezésben játszott szerepével foglalkozik. Egészében véve a 4. rész célja, hogy áthidalja a szakadékot a nyelvfüggetlenségre törekvő tervezés és a rövidlátóan a részletekre összpontosító programozás között. Egy nagyobb program elkészítési folyamatában mindkettőnek megvan a helye, de a tervezési szempontok és a használt eljárások között megfelelő összhangot kell teremteni, hogy elkerüljük a katasztrófákat és a felesleges kiadásokat.

23.2. Bevezetés

A legegyszerűbbek kivételével minden program elkészítése összetett és gyakran csüggesztő feladat. A programutasítások megírása még az önállóan dolgozó programozó számára is csupán egy része a folyamatnak. A probléma elemzése, az átfogó programtervezés, a dokumentálás, a tesztelés és a program fenntartásának, módosításának kérdései, valamint mindennek az összefogása mellett eltörpül az egyes kódrészek megírásának és kijavításának feladata. Természetesen nevezhetnénk e tevékenységek összességét „programozásnak", majd logikus módon kijelenthetnénk: „Én nem tervezek, csak programozok". Akárminek nevezzük is azonban a tevékenységet, néha az a fontos, hogy a részletekre összpontosítsunk, néha pedig az, hogy az egész folyamatot tekintsük. Sem a részleteket, sem a végcélt nem szabad szem elől tévesztenünk – bár néha pontosan ez történik –, csak így készíthetünk teljes értékű programokat.

Ez a fejezet a programfejlesztés azon részeivel foglalkozik, melyek nem vonják magukkal egyes kódrészek írását és javítását. A tárgyalás kevésbé pontos és részletes, mint a korábban bemutatott nyelvi tulajdonságok és programozási eljárások tárgyalása, de nem is lehet olyan tökéletes „szakácskönyvet" írni, amely leírná, hogyan kell jó programot készíteni. Létezhetnek részletes „hogyan kell" leírások egyes programfajtákhoz, de az általánosabb alkalmazási területekhez nem. Semmi sem pótolja az intelligenciát, a tapasztalatot, és a programozási érzéket. Következésképpen ez a fejezet csak általános tanácsokat ad, illetve megközelítési módokat mutat be és tanulságos megfigyeléseket kínál.

A problémák megközelítését bonyolítja a programok eleve elvont természete és az a tény, hogy azok a módszerek, melyek kisebb projekteknél (például amikor egy vagy két ember ír 10 000 sornyi kódot) működnek, nem szükségszerűen alkalmazhatók közepes vagy nagy

projektekben. Ezért számos kérdést inkább a kevésbé elvont mérnöki tudományokból átvett hasonlatokon keresztül közelítünk meg, kódpéldák használata helyett. A programtervezés tárgyalása a C++ fogalmaival, illetve a kapcsolódó példák a 24. és 25. fejezetben találhatók, de az itt bemutatott elvek tükröződnek mind magában a C++ nyelvben, mind a könyv egyes példáiban.

Arra is emlékeztetném az olvasót, hogy az alkalmazási területek, emberek és programfejlesztő környezetek rendkívül sokfélék, ezért nem várható el, hogy minden itteni javaslatnak közvetlenül hasznát vehessük egy adott probléma megoldásában. A megfigyelések valós helyzetekben születtek és számos területen felhasználhatók, de nem tekinthetők általános érvényűnek, ezért nem árt némi egészséges szkepticizmus.

A C++ úgy is tekinthető, mint egy „jobb C". De ha így teszünk, kihasználatlanul maradnak a C++ igazi erősségei, így a nyelv előnyeinek csak töredéke érvényesül. Ez a fejezet kimondottan azokra a tervezési megközelítésekre összpontosít, melyek lehetővé teszik a C++ elvont adatábrázolási és objektumközpontú programozási lehetőségeinek hatékony használatát. Az ilyen módszereket gyakran nevezzük *objektumorientált* (object-oriented) *tervezésnek*.

A fejezet fő témái a következők:

- A szoftverfejlesztés legfontosabb szempontja, hogy tisztában legyünk vele, mit is készítünk.
- A sikeres szoftverfejlesztés sokáig tart.
- Az általunk épített rendszerek összetettségének határait saját tudásunk és eszközeink képességei szabják meg.
- Semmiféle tankönyvi módszer nem helyettesítheti az intelligenciát, a tapasztalatot, a jó tervezési és programozási érzéket.
- A kísérletezés minden bonyolultabb program elkészítésénél lényeges.
- A programkészítés különböző szakaszai – a tervezés, a programozás és a tesztelés – nem választhatók el szigorúan egymástól.
- A programozás és a tervezés nem választhatók el e tevékenységek megszervezésének kérdésétől.

Könnyen abba a hibába eshetünk – és persze megfizetjük az árát –, hogy ezeket a szempontokat alábecsüljük, pedig az elvont ötleteket nehéz a gyakorlatba áttenni. Tudomásul kell vennünk a tapasztalat szükségességét: éppúgy, mint a csónaképítés, a kerékpározás, vagy éppen a programozás, a tervezés sem olyan képesség, amely pusztán elméleti tanulmányokkal elsajátítható.

Gyakran elfeledkezünk a rendszerépítés emberi tényezőiről, és a programfejlesztés folyamatát egyszerűen meghatározott lépések sorozatának tekintjük, mely „a bemenetből adott műveletek végrehajtásával előállítja a kívánt kimenetet, a lefektetett szabályoknak megfelelően." A programozási nyelv, amelyet használunk, elfedi az emberi tényező jelenlétét, márpedig a tervezés és a programozás emberi tevékenységek – ha erről megfeledkezünk, nem járhatunk sikerrel.

Ez a fejezet olyan rendszerek tervezésével foglalkozik, melyek a rendszert építő emberek tapasztalatához és erőforrásaihoz képest igényesek. Úgy tűnik, a fejlesztők szeretik feszegetni lehetőségeik határait. Az olyan munkáknál, melyek nem jelentenek ilyen kihívást, nincs szükség a tervezés megvitatására, hiszen ezeknek kidolgozott kereteik vannak, amelyeket nem kell széttörni. Csak akkor van szükség új, jobb eszközök és eljárások elsajátítására, amikor valami igényes dologra vállalkozunk. Arra is hajlamosak vagyunk, hogy hozzánk képest újoncokra olyan feladatokat bízzunk, melyekről „mi tudjuk, hogyan kell elvégezni", de ők nem.

Nem létezik „egyetlen helyes módszer" minden rendszer tervezésére és építésére. A hitet az „egyetlen helyes módszerben" gyermekbetegségnek tekinteném, ha nem fordulna elő oly gyakran, hogy tapasztalt programozók és tervezők engednek neki. Emlékeztetek arra, hogy azért, mert egy eljárás a múlt évben egy adott munkánál jól működött, még nem biztos, hogy ugyanaz módosítás nélkül működni fog valaki másnál vagy egy másik munka során is. A legfontosabb, hogy mentesek legyünk az ilyesfajta feltételezésektől.

Az itt leírtak természetesen a nagyobb méretű programok fejlesztésére vonatkoznak. Azok, akik nem működnek közre ilyen fejlesztésben, visszaülhetnek és örülhetnek, látván, milyen rémségektől menekültek meg, illetve nézegethetik az egyéni munkával foglalkozó részeket. A programok méretének nincs alsó határa, ahol a kódolás előtti tervezés még ésszerű, de létezik ilyen határ, amennyiben a tervezés és dokumentálás megközelítési módjáról van szó. A célok és a méret közötti egyensúly fenntartásának kérdéseivel a §23.5.2 pont foglalkozik.

A programfejlesztés központi problémája a bonyolultság. A bonyolultsággal pedig egyetlen módon lehet elbánni, az „oszd meg és uralkodj!" elvet követve. Ha egy problémát két önmagában kezelhető részproblémára választhatunk szét, félig máris megoldottuk azt. Ez az egyszerű elv bámulatosan változatos módokon alkalmazható. Nevezetesen, egy modul vagy egy osztály használata a rendszer megtervezésénél két részre osztja a programot – a tényleges megvalósításra és a felhasználói kódra – amelyeket ideális esetben csak egy jól definiált felület (interface) kapcsol össze: ez a programmal járó bonyolultság kezelésének alapvető megközelítése. Ugyanígy a programtervezési folyamat is külön tevékeny-

ségekre bontható, s így a közreműködő és egymással kapcsolatban álló fejlesztők között feladatokra bontva szétosztható: ez pedig a fejlesztési folyamat és a résztvevő programozók közötti bonyolult kapcsolatok kezelésének alapvető megközelítése.

Mindkét esetben a felosztás és a kapcsolatot megteremtő felületek meghatározása az, ahol a legnagyobb tapasztalatra és érzékre van szükség. Ez nem egyszerű, mechanikusan megoldható feladat; jellemzően éleslátást igényel, melyre csak egy rendszer alapos tanulmányozása és az elvonatkoztatási szintek megfelelő megértése által tehetünk szert (lásd: §23.4.2, §24.3.1 és §25.3). Ha egy programot vagy fejlesztési folyamatot csak bizonyos szemszögből vizsgálunk, súlyos hibákat véthetünk. Azt is vegyük észre, hogy *különválasztani* mind az emberek feladatait, mind a programokat könnyű. A feladat nehéz része a hatékony *kapcsolattartás* biztosítása a válaszfal két oldalán levő felek között anélkül, hogy lerombolnánk a válaszfalat vagy elnyomnánk az együttműködéshez szükséges kommunikációt.

Ez a fejezet egy tervezési megközelítést mutat be, nem egy teljes tervezési módszert; annak bemutatása túlmutatna e könyv keretein. Az itt bemutatott megközelítés az általánosítás – vagyis a formális megfogalmazás – különböző fokozataival és a mögöttük megbúvó alapvető elvekkel ismertet meg. Nem szakirodalmi áttekintés és nem szándékszik érinteni minden, a szoftverfejlesztésre vonatkozó témát, vagy bemutatni minden szempontot. Ez ismét csak meghaladná e könyv lehetőségeit. Ilyen áttekintést találhatunk [Booch,1994]-ben. Feltűnhet, hogy a kifejezéseket itt elég általános és hagyományos módon használom. A „legérdekesebbeknek" – *tervezés*, *prototípus*, *programozó* – a szakirodalomban számos különböző és gyakran egymással ellentétes definíciója található. Legyünk óvatosak, nehogy olyan nem szándékolt jelentést olvassunk ki az itt elmondottakból, melyek az egyes kifejezések önmagában vett vagy csupán „helyileg" pontos meghatározásain alapulnak.

23.3. Célok és eszközök

A professzionális programozás célja olyan terméket létrehozni, mellyel felhasználói elégedettek lesznek. Ennek elsődleges módja olyan programot alkotni, melynek belső felépítése tiszta, és csapatot kovácsolni olyan tervezőkből és programozókból, akik elég ügyesek és lelkesek ahhoz, hogy gyorsan és hatékonyan reagáljanak a változásokra és élni tudjanak lehetőségeikkel.

Miért? A program belső szerkezete és keletkezésének folyamata ideális esetben nem érinti a végfelhasználót. Nyíltabban fogalmazva: ha a végfelhasználónak aggálya van, hogyan írták meg a programot, akkor azzal a programmal valami baj van. Ezt figyelembe véve feltehetjük a kérdést: miben áll a program szerkezetének és a programot megalkotó embereknek a fontossága?

Először is, egy programnak tiszta belső felépítéssel kell rendelkeznie, hogy megkönnyítse

- a tesztelést,
- a más rendszerekre való átültetést,
- a program karbantartását és módosítását,
- a bővítést,
- az újraszervezést és
- a kód megértését.

A lényeg, hogy egyetlen sikeres nagy program története sem ér véget a piacra kerüléssel; újabb és újabb programozók és tervezők dolgoznak rajta, új hardverre viszik át, előre nem látott célokra használják fel és többször is átalakítják szerkezetét. A program élete folyamán új változatokat kell készíteni, elfogadható mennyiségű hibával, elfogadható idő alatt. Ha ezzel nem számolunk, kudarcra vagyunk ítélve.

Vegyük észre, hogy bár a végfelhasználók ideális esetben nem kell, hogy ismerjék egy rendszer belső felépítését, előfordulhat, hogy kíváncsiak rá, például azért, hogy fel tudják becsülni megbízhatóságát, lehetséges felülvizsgálatát és bővítését. Ha a kérdéses program nem egy teljes rendszer, csak más programok építését segítő könyvtárak készlete, a felhasználók még több részletet akarnak tudni, hogy képesek legyenek jobban kihasználni a könyvtárakat és ötleteket meríthessenek belőlük.

Egyensúlyt kell teremteni a program átfogó tervezésének mellőzése és a szerkezetre fektetett túlzott hangsúly között. Az előbbi vég nélküli javítgatásokhoz vezet („ezt most leszállítjuk, a problémát meg majd kijavítjuk a következő kiadásban"), az utóbbi túlbonyolítja a tervezést és a lényeg elvész a formai tökéletességre való törekvés közben, ami azt eredményezi, hogy a tényleges megvalósítás késedelmet szenved a program szerkezetének folytonos alakítgatása miatt („de ez az új felépítés *sokkal* jobb, mint a régi; az emberek hajlandóak várni rá"). A forma tartalom fölé rendelése gyakran eredményez olyan erőforrás-igényű rendszereket is, melyeket a leendő felhasználók többsége nem engedhet meg magának. Az egyensúly megtartása a tervezés legnehezebb része és ez az a terület, ahol a tehetség és a tapasztalat megmutatkozik. A választás az önálló programozó számára is nehéz, de még nehezebb a nagyobb programoknál, melyek több, különböző képességű ember munkáját igénylik.

A programot egy olyan szervezett közösségnek kell megalkotnia és fenntartania, mely a személyi és vezetési változások ellenére képes a feladatot megoldani. Népszerű megközelítés a programfejlesztést néhány viszonylag alacsony szintű feladatra szűkíteni, melyek egy merev vázba illeszkednek. Az elgondolás egy könnyen betanítható (olcsó) és „cserélhető", alacsony szintű programozókból („kódolókból") álló osztály és egy valamivel kevésbé olcsó, de ugyanúgy cserélhető (és ugyanúgy mellőzhető) tervezőkből álló osztály létrehozása. A kódolókról nem tételezzük fel, hogy tervezési döntéseket hoznak, míg a tervezőkről nem tételezzük fel, hogy a kódolás „piszkos" részleteivel törődnek. Ez a megközelítés gyakran csődöt mond; ahol pedig működik, nagy méretű, de gyenge teljesítményű programokat eredményez.

E megoldás buktatói a következők:

- Elégtelen kapcsolattartás a megvalósítást végzők és a tervezők között, ami alkalmak elmulasztását, késést, rossz hatékonyságot, és a tapasztalatból való tanulás képtelensége miatt ismétlődő problémákat eredményez.
- A programozók kezdeményezési hatáskörének elégtelensége, ami a szakmai fejlődés hiányához, felületességhez és káoszhoz vezet.

E rendszer alapvető gondja a visszajelzés hiánya, mely lehetővé tenné, hogy a közreműködők egymás tapasztalataiból tanuljanak. Ez a szűkös emberi tehetség elvesztegetése. Az olyan keretek biztosítása, melyen belül mindenki megcsillanthatja tehetségét, új képességeket fejleszthet ki, ötletekkel járulhat hozzá a sikerhez és jól érezheti magát, nemcsak az egyedüli tisztességes megoldás, hanem gyakorlati és gazdasági értelme is van.

Másrészt egy rendszert nem lehet felépíteni, dokumentálni és a végtelenségig fenntartani valamilyen áttekinthető szerkezet nélkül. Egy újításokat is igénylő munka elején gyakran az a legjobb, ha egyszerűen megkeressük a legjobb szakembereket és hagyjuk őket, hogy úgy közelítsék meg a problémát, ahogy a legjobbnak tartják. A munka előrehaladtával azonban egyre inkább ügyelni kell a helyes ütemezésre, a részfeladatok kiosztására, a bevont személyek közötti kapcsolattartásra és bizonyos – a jelölésre, elnevezésekre, dokumentációra, tesztelésre stb. vonatkozó – irányelvek betartására. Ismét csak egyensúlyra és a helyénvaló iránti érzékre van szükség. Egy túl merev rendszer akadályozhatja a fejlődést és elfojthatja az innovációt. Ebben az esetben a vezető tehetsége és tapasztaltsága mérettetik meg, de az önálló programozó számára is ugyanilyen dilemma kiválasztani, hol próbáljon ügyeskedni és hol csinálja „a könyv szerint".

Az adott munkánál ajánlatos nem csak a következő kiadásra, hanem hosszú távra tervezni. Az előrelátás hiánya mindig megbosszulja magát. A szervezeti felépítést és a programfejlesztési stratégiát úgy kell megválasztani, hogy több program több kiadásának megalkotását célozzuk meg – vagyis sikerek sorozatát kell megterveznünk.

A „tervezés” célja a program számára tiszta és viszonylag egyszerű belső szerkezet – *architektúra* – létrehozása. Más szóval, olyan vázat akarunk alkotni, melybe beilleszthetők az egyes kódrészek és ezáltal irányelvként szolgál e kódrészek megírásához.

A terv a tervezési folyamat végterméke (amennyiben egy körkörös folyamatnak *vég*terméke lehet). Ez áll a tervező és a programozó, illetve az egyes programozók közti kapcsolattartás középpontjában. Fontos, hogy itt kellő arányérzékünk legyen. Ha én – mint önálló programozó – egy kis programot tervezek, melyet holnap fogok elkészíteni, megfelelő pontosságú és részletességű lehet egy boríték hátára firkált pár sor. A másik véglet: egy tervezők és programozók százait foglalkoztató rendszer fejlesztése könyveket kitevő, gondosan megírt „szabványokat” kívánhat, valamilyen szabályhoz részben vagy teljes egészében igazodó jelölések használatával. Egy terv megfelelő részletességi, pontossági és formalitási szintjének meghatározása önmagában véve is kihívó szakmai és vezetési feladat.

Ebben és a következő fejezetekben feltételezem, hogy egy program terve osztálydeklarációk egy halmazával (a privát deklarációikat, mint zavaró részleteket, általában elhagyom) és a köztük levő kapcsolatokkal van kifejezve. Ez persze egyszerűsítés: egy konkrét tervben sok más kérdés is felmerül; például a párhuzamos folyamatok, a névterek kezelése, a nem tag függvények és adatok használata, az osztályok és függvények paraméterezése, az újrafordítás szükségességét a lehető legalacsonyabb szintre visszaszorító kódszervezés, a perzisztencia és a több-számítógépes használat. A tárgyalásnak ezen a szintjén mindenképpen szükség van egyszerűsítésre és a C++-ban a tervezéshez az osztályok megfelelő kiindulópontot jelenthetnek. A fent említett témakörök közül néhányat futólag ez a fejezet is érint, de a C++-ban való tervezésre közvetlenül hatást gyakorlókat a 24. és 25. fejezet tárgyalja. Ha egy bizonyos objektumorientált tervezési modellre részleteiben vagyunk kíváncsiak, [Booch, 1994] lehet segítségünkre.

Az elemzés (analízis, analysis) és a tervezés (design) közötti különbségre nem térek ki különösebben, egyrészt mert nem témája e könyvnek, másrészt az egyes tervezési módszerek esetében e különbséget más-más módon határozhatnánk meg. Röviden annyit mondhatunk, hogy az elemzési módszert mindig az adott tervezési megközelítéshez kell igazítani, azt pedig a használt programozási nyelv és stílus alapján kell megválasztani.

23.4. A fejlesztési folyamat

A szoftverfejlesztés ismétlést és gyarapodást jelent: a folyamat minden szakasza a fejlesztés alatt újra és újra felülvizsgálatra kerül és minden egyes felülvizsgálat finomítja az adott szakasz végtermékét. A fejlesztési folyamatnak általában nincs kezdete és nincs vége. Amikor egy programrendszert megtervezünk és elkészítünk, más emberek terveit, könyvtárait és programjait vesszük alapul. Amikor befejezzük, a terv és a kód törzsét másokra hagyjuk, hogy finomítsák, felülvizsgálják, bővítsék és más gépekre áttegyék azt. Természetesen egy adott munkának lehet meghatározott kezdete és vége, és fontos is (bár gyakran meglepően nehéz), hogy azt időben és térben tisztán és pontosan behatároljuk. Ha azonban úgy teszünk, mintha tiszta lappal indulnánk, komoly problémákat okozhatunk. Úgy tenni, mintha a világ a program „végső átadásánál" végződne, egyaránt fejfájást okozhat utódaink és gyakran saját magunk számára.

Ebből következik, hogy a következő fejezetek bármilyen sorrendben olvashatók, mivel egy valós munka során a tervezési és megvalósítási szempontok majdnem tetszés szerint átfedhetik egymást. Ez azt jelenti, hogy a „tervezés" majdnem mindig újratervezés, amely egy előző tervezésen és némi fejlesztési tapasztalaton alapul. Továbbá, a tervezést korlátozzák az ütemtervek, a résztvevő emberek képességei, a kompatibilitási kérdések és így tovább. A tervező, vezető vagy programozó számára nagy kihívást jelent rendet teremteni e folyamatban anélkül, hogy elfojtanánk az újítási törekvéseket és tönkretennénk a visszajelzés rendszerét, melyek a sikeres fejlesztéshez szükségesek.

A fejlesztési folyamat három szakaszból áll:

- Elemzés: a megoldandó probléma kiterjedésének meghatározása
- Tervezés: a rendszer átfogó felépítésének megalkotása
- Megvalósítás: a kód megírása és ellenőrzése

Még egyszer emlékeztetnék e folyamat ismétlődő természetére – nem véletlen, hogy a szakaszok között nem állítottam fel sorrendet. Vegyük észre azt is, hogy a programfejlesztés néhány fő szempontja nem külön szakaszként jelentkezik, mivel ezeknek a folyamat során mindvégig érvényesülniük kell:

- Kísérletezés
- Ellenőrzés
- A tervek és a megvalósítás elemzése
- Dokumentálás
- Vezetés

A program fenntartása vagy „karbantartása" egyszerűen e fejlesztési folyamat többszöri ismétlését jelenti (§23.4.6).

A legfontosabb, hogy az elemzés, a tervezés és a megvalósítás ne váljon el túlzottan egymástól, és hogy a bevont emberek közös szellemiség, kultúra és nyelv révén hatékonyan tudjanak egymással kommunikálni. Nagyobb programok fejlesztésénél ez sajnos ritkán van így. Ideális esetben az egyének a munka során több szakaszban is részt vesznek; ez az ismeretek és tapasztalatok átadásának legjobb módja. Sajnos a programfejlesztő cégek gyakran akadályozzák az ilyen mozgást, például azáltal, hogy a tervezőknek magasabb beosztást és/vagy magasabb fizetést adnak, mint a „csak programozóknak". Ha gyakorlati szempontból nem célszerű, hogy a munkatársak vándorolva tanuljanak és tanítsanak, legalább arra biztassuk őket, hogy rendszeresen beszélgessenek a fejlesztés „más szakaszaiba" bevontakkal.

Kisebb és közepes projekteknél gyakran nem különböztetik meg az elemzést és a tervezést; e két szakasz egyesül. A kis programok készítésénél általában ugyanígy nem válik szét a tervezés és a programozás, ami persze megoldja a kommunikációs problémákat. Fontos, hogy a formalitások és a szakaszok elkülönítése mindig az adott munkához igazodjanak (§23.5.2); nem létezik egyetlen, örök érvényű út.

Az itt leírt programfejlesztési modell gyökeresen különbözik a hagyományos „vízesés" modelltől, amelyben a fejlesztés szabályosan és egyenes irányban halad a fejlesztési szakaszokon át, az elemzéstől a tesztelésig. A vízesés modell alapvető baja, hogy az információ folyása egyirányú. Ha a „folyás irányában haladva" problémákba ütközünk, a szervezeti felépítés és a munkamódszerek arra kényszerítenek, hogy helyben oldjuk meg azokat, az előző szakaszokra való hatás nélkül. A visszacsatolás ezen hiánya gyenge tervekhez vezet, a helyi javítások pedig torz megvalósítást eredményeznek. Vannak elkerülhetetlen esetek, amikor az információ mindenképpen visszafelé áramlik és megváltoztatja az eredeti terveket. A gerjesztett „hullám" csak lassan és nehézkesen jut el céljához, hiszen a rendszert úgy alkották meg, hogy ne legyen szükség változtatásokra és a rendszer emiatt lassan és kelletlenül reagál. A „semmi változtatás" vagy a „helyi javítás" elve tehát olyan elvvé változik, mely szerint egy részleg nem adhat többletmunkát más részlegeknek „a saját kényelme érdekében". Lehetséges, hogy mire egy nagyobb hibát észlelnek, már olyan sok papírmunkát végeztek, hogy a kód javításához szükséges erőfeszítés eltörpül ahhoz képest, ami a dokumentáció módosításához kell. Ilyenkor a papírmunka válik a programfejlesztés fő kerékkötőjévé. Természetesen ilyen problémák előfordulhatnak és elő is fordulnak, jóllehet a nagy rendszerek fejlesztését alaposan megszervezik. Mindenképpen elengedhetetlen *némi* papírmunka. Az egyenes vonalú (vízesés) fejlesztési modell alkalmazása azonban nagyban növeli a valószínűségét, hogy a probléma kezelhetetlenné válik.

A vízeséses modellel az a probléma, hogy nincs benne megfelelő visszacsatolás és képtelen a változásokra reagálni. Az itt vázolt ciklikus megközelítés veszélye viszont a kísértés, hogy a valódi gondolkodás és haladás helyett nem azonos cél érdekében végrehajtott módosítások sorát végezzük el. Mindkét problémát könnyebb felismerni, mint megoldani, és bármilyen jól is szervezzük meg a feladat végrehajtását, könnyen összetéveszthetjük a puszta munkát a haladással. Természetesen a munka előrehaladtával a fejlesztési folyamat különböző szakaszainak jelentősége változik. Kezdetben a hangsúly az elemzésen és a tervezésen van, és kevesebb figyelmet fordítunk a programozási kérdésekre. Ahogy múlik az idő, az erőforrások a tervezés és a programozás felé tolódnak, majd inkább a programozás és a tesztelés felé. Kulcsfontosságú azonban, hogy soha ne összpontosítsunk csupán az elemzés/tervezés/megvalósítás egyikére, kizárva látókörünkből a többit.

Ne felejtsük el, hogy nem segíthet rajtunk semmilyen, „a részletekre is kiterjedő" figyelem, vezetési módszer vagy fejlett technológia, ha nincs tiszta elképzelésünk arról, mit is akarunk elérni. Több terv hiúsul meg amiatt, hogy nincsenek jól meghatározott és valószerű céljai, mint bármilyen más okból. Bármit teszünk is és bárhogyan fogunk is hozzá, tűzzünk ki kézzelfogható célokat, jelöljük ki a hozzá vezető út állomásait, és használjunk fel minden *megfelelő* technológiát, ami csak elérhető – még akkor is, ha az beruházást igényel –, mert az emberek jobban dolgoznak megfelelő eszközökkel és megfelelő környezetben. Ne higgyük persze, hogy könnyű ezt a tanácsot megfogadni.

23.4.1. A fejlesztés körforgása

Egy program fejlesztése során állandó felülvizsgálatra van szükség. A fő ciklus a következő lépések ismételt végrehajtásából áll:

1. Vizsgáljuk meg a problémát.
2. Alkossunk átfogó tervet.
3. Keressünk szabványos összetevőket.
   - Igazítsuk ezeket a tervhez.
4. Készítsünk új szabványos összetevőket.
   - Igazítsuk ezeket is a tervhez.
5. Állítsuk össze a konkrét tervet.

Hasonlatként vegyünk egy autógyárat. A projekt beindításához kell, hogy legyen egy átfogó terv az új autótípusról. Az előzetes tervnek valamilyen elemzésen kell alapulnia és általánosságban kell leírnia az autót, inkább annak szándékolt használatát, mintsem a kívánt tulajdonságok kialakításának részleteit szem előtt tartva. Eldönteni, mik is a kívánt tulajdonságok – vagy még inkább, a döntéshez viszonylag egyszerű irányelveket adni – gyakran

a munka legnehezebb része. Ezt sikeresen általában egyetlen, nagy áttekintéssel rendelkező egyén tudja elvégezni, és ez az, amit gyakran *látomásnak* (vision) nevezünk. Igen gyakori, hogy hiányoznak az ilyen tiszta célok – márpedig a terv ezek nélkül meginoghat vagy meg is hiúsulhat.

Tegyük fel, hogy egy közepes nagyságú autót akarunk építeni, négy ajtóval és erős motorral. A tervezés első szakasza a leghatározottabban nem az autó (és az alkatrészek) megtervezése a „semmiből". Egy meggondolatlan programtervező vagy programozó hasonló körülmények között (ostobán) pontosan ezt próbálná tenni.

Az első lépés annak számbavétele, milyen alkatrészek szerezhetők be a gyár saját készleteiből és megbízható szállítóktól. Az így talált alkatrészek nem kell, hogy pontosan illeszkedjenek az új autóba. Lesz mód az alkatrészek hozzáigazítására. Azt is megtehetjük, hogy az ilyen alkatrészek „következő kiadásához" irányelveket adunk, hogy saját elképzeléseinkhez jobban illeszkedjenek. Például létezhet egy motor, melynek megfelelőek a tulajdonságai, azzal a kivétellel, hogy kicsit gyengébb a leadott teljesítménye. A cég vagy a motor szállítója rászerelhet egy turbófeltöltőt, hogy ezt helyrehozza, anélkül, hogy befolyásolná az alapvető terveket. Vegyük észre, hogy az ilyen, „az alaptervet nem befolyásoló" változtatás valószínűsége kicsi, hacsak az eredeti terv nem előlegezi meg valamilyen formában az igazíthatóságot. Az ilyen igazíthatóság általában együttműködést kíván köztünk és a motor szállítója között. A programozó hasonló lehetőségekkel rendelkezik: a többalakú (polimorf) osztályok és sablonok (template) például hatékonyan felhasználhatók egyedi változatok készítésére. Nem szabad azonban elvárnunk, hogy az osztály készítője a mi szájunk íze szerint bővítse alkotását; ehhez a készítő előrelátására vagy a velünk való együttműködésre van szükség.

Ha kimerült a megfelelő szabványos alkatrészek választéka, az autótervező nem rohan, hogy megtervezze az új autóhoz az optimális új alkatrészeket. Ez egyszerűen túl költséges lenne. Tegyük fel, hogy nem kapható megfelelő légkondicionáló egység és hogy egy megfelelő L alakú tér áll rendelkezésre a motortéren belül. Az egyik megoldás egy L alakú légkondicionáló egység megtervezése lenne, de annak valószínűsége, hogy ez a különleges forma más autótípusoknál is használható, még alapos igazítás esetén is csekély. Ebből következik, hogy autótervezőnk nem lesz képes az ilyen egységek termelési költségeit más autótípusok tervezőivel megosztani, így az egység kihasználható élettartama rövid lesz. Érdemes tehát olyan egységet tervezni, mely szélesebb körben számíthat érdeklődésre; vagyis egy tisztább vonalú egységet kell tervezni, mely jobban az igényekhez igazítható, mint a mi elméletben elkészített L alakú furcsaságunk. Ez valószínűleg több munkával jár, mint az L alakú egység és magával vonhatja az autó átfogó terveinek módosítását is, hogy illeszkedjen az általánosabb célú egységhez. Mivel az új egység szélesebb körben használható, mint a mi L alakú csodánk, feltételezhető, hogy szükség lesz egy kis igazításra, hogy tökéletesen

illeszkedjen a mi felülvizsgált igényeinkhez. A programozó ismét hasonló lehetőségek közül választhat: ahelyett, hogy az adott programra nézve egyedi kódot írna, új, általánosabb összetevőt tervezhet, amely remélhetőleg valamilyen szabvánnyá válik.

Amikor végképp kifogytunk a lehetséges szabványos összetevőkből, összeállítjuk a „végleges" tervet. A lehető legkevesebb egyedi tervezésű alkatrészt használjuk, mert jövőre – kissé más formában – ismét végig kell csinálnunk ugyanezt a munkát a következő új modellhez, és az egyedi alkatrészek lesznek azok, amelyeket a legvalószínűbb, hogy újra kell építenünk vagy el kell dobnunk. Sajnos a hagyományosan tervezett programokkal kapcsolatban az a tapasztalat, hogy kevés bennük az olyan rész, amelyet egyáltalán önálló összetevőnek lehetne tekinteni és az adott programon kívül máshol is fel lehetne használni.

Nem állítom, hogy minden autótervező olyan ésszerűen gondolkodik, mint ahogy a hasonlatban felvázoltam vagy hogy minden programtervező elköveti az említett hibákat. Éppen ellenkezőleg, a bemutatott modell a programtervezésben is használható. Ez a fejezet és a következők olyan eljárásokat ismertetnek, melyek a C++-ban működnek. A programok nem kézzelfogható természete miatt viszont a hibákat kétségtelenül nehezebb elkerülni (§24.3.1, 24.3.4), és rámutatok majd arra is (§23.5.3.1), hogy a cégek működési elve gyakran elveszi az emberek bátorságát, hogy az itt vázolt modellt használják. Ez a fejlesztési modell valójában csak akkor működik jól, ha hosszabb távra tervezünk. Ha látókörünk csak a következő kiadásig terjed, a szabványos összetevők megalkotása és fenntartása értelmetlen, egyszerűen felesleges túlmunkának fogjuk látni. E modell követése olyan fejlesztőközösségek számára javasolt, melyek élete számos projektet fog át és melyek mérete miatt érdemes pénzt fektetni a szükséges eszközökbe (a tervezéshez, programozáshoz és projektvezetéshez) és oktatásba (a tervezők, programozók és vezetők részére). Modellünk valójában egyfajta „szoftvergyár" vázlata, amely furcsa módon csak méreteiben különbözik a legjobb önálló programozók gyakorlatától, akik az évek során módszerek, tervek, eszközök és könyvtárak készletét építették fel, hogy fokozzák személyes hatékonyságukat. Sajnos úgy tűnik, a legtöbb cég elmulasztotta kihasználni a legjobbak gyakorlati tudását, mert hiányzott belőle az előrelátás és a képesség, hogy az általuk használt megközelítést nagyobb programokra is alkalmazza.

Vegyük észre, hogy nincs értelme minden „szabványos összetevőtől" elvárni, hogy általános érvényű szabvány legyen. Nemzetközileg szabványos könyvtárból csak néhány maradhat fenn. A legtöbb összetevő vagy alkatrész (csak) egy országon, egy iparágon, egy gyártósoron, egy osztályon, egy alkalmazási területen stb. belül lesz szabvány. A világ egyszerűen túl nagy ahhoz, hogy valóságos vagy igazán kívánatos cél legyen minden eszközre egyetemes szabványt alkotni.

Ha már az elején az egyetemességet tűzzük ki célul, a tervet arra ítéljük, hogy soha ne legyen befejezve. Ennek egyik oka, hogy a fejlesztés körkörös folyamat, és elengedhetetlen, hogy legyen egy működő rendszere, amelyből tapasztalatokat meríthetünk (§23.4.3.6).

23.4.2. Tervezési célok

Melyek a tervezés átfogó céljai? Az egyik természetesen az egyszerűség, de milyen szempontok szerint? Feltételezzük, hogy a tervnek fejlődnie kell, vagyis a programot majd bővíteni, más rendszerre átvinni, finomhangolni és többféle, előre nem látható módon módosítani lesz szükséges. Következésképpen olyan tervet és rendszert kell megcéloznunk, amely egyszerű, de könnyen és sokféle módon módosítható. A valóságban a rendszerrel szemben támasztott követelmények már a kezdeti terv és a program első kibocsátása között többször megváltoznak.

Ez magával vonja, hogy a programot úgy kell megtervezni, hogy a módosítások sorozata alatt a lehető legegyszerűbb *maradjon*. A módosíthatóságot figyelembe véve a következőket kell célul kitűznünk:

- Rugalmasság
- Bővíthetőség
- Hordozhatóság (más rendszerekre való átültethetőség)

A legjobb, ha megpróbáljuk elszigetelni a program azon területeit, melyek valószínűleg változni fognak, a felhasználók számára pedig lehetővé tesszük, hogy módosításaikat e részek érintése nélkül végezhessék el.

Ezt a program kulcsfogalmainak azonosításával és azzal érhetjük el, hogy minden egyes osztályra kizárólagos felelősséggel rábízzuk egy fogalommal kapcsolatos minden információ kezelését. Ez esetben mindennemű változtatás csupán az adott osztály módosításával érhető el. Ideális esetben egy fogalom módosításához elég egy új osztály származtatása (§23.4.3.5) vagy egy sablon eltérő paraméterezése. Természetesen az elvet a gyakorlatban sokkal nehezebb követni.

Vegyünk egy példát: egy meteorológiai jelenségeket utánzó programban egy esőfelhőt akarunk megjeleníteni. Hogyan csináljuk? Nem lehet általános eljárásunk a felhő megjelenítésére, mivel a felhő kinézete függ a felhő belső állapotától és ez az állapot a felhő kizárólagos „felelőssége” alá kell, hogy tartozzon.

A probléma első megoldása, hogy hagyjuk a felhőre a saját megjelenítését. Ez bizonyos körülmények között elfogadható, de nem általános megoldás, mert egy felhőt sokféle módon lehet megjeleníteni: részletes képpel, elnagyolt vázlatként, vagy mint egy jelet egy térképen. Más szóval az, hogy egy felhő mire hasonlít, függ mind a felhőtől, mind a környezetétől.

A másik megoldás, hogy a felhőt „tájékoztatjuk" környezetéről, majd hagyjuk, hogy megjelenítse magát. Ez még több esetben elfogadható, de még mindig nem általános módszer. Ha tudatjuk a felhővel környezetének részleteit, megsértjük azt a szabályt, hogy egy osztály csak egy dologért lehet felelős és hogy minden „dolog" valamelyik osztály felelőssége. Nem biztos, hogy eljuthatunk „a felhő környezetének" egy következetes jelöléséhez, mert általában az, hogy egy felhő mihez hasonlít, függ mind a felhőtől, mind a nézőtől. Az, hogy számomra mihez hasonlít egy felhő, még a valós életben is meglehetősen függ attól, hogyan nézem: puszta szemmel, polárszűrőn keresztül vagy meteorológiai radarral. A nézőn és a felhőn kívül lehet, hogy valamilyen „általános hátteret" is figyelembe kell venni, például a nap viszonylagos helyzetét. Tovább bonyolítja a dolgot, ha egyéb objektumokat is – más felhőket, repülőgépeket – számításba veszünk. Hogy a tervező életét igazán megnehezítsük, adjuk hozzá azt a lehetőséget is, hogy egyszerre több nézőnk van.

A harmadik megoldás, hogy hagyjuk a felhőt és a többi objektumot (a repülőgépeket és a napot), hogy leírják magukat a nézőknek. E megoldás elég általános ahhoz, hogy a legtöbb célnak megfeleljen, de jelentős árat fizethetünk érte, mert a program túlságosan bonyolult és lassú lehet. Hogyan értetjük meg például a nézővel a felhők és más objektumok által készített leírásokat?

A programokban nem sűrűn fordulnak elő esőfelhők (bár a példa kedvéért lásd §15.2), de a különböző I/O műveletekbe szükségszerűen bevont objektumok is hasonlóan viselkednek. Ez teszi a felhő példát találóvá a programok és különösen a könyvtárak tervezésére nézve. Logikailag hasonló példa C++ kódját mutattam be az adatfolyamok be- és kimeneti rendszerének tárgyalásakor, a formázott kimenetre használt módosítóknál (manipulator, §21.4.6, §21.4.6.3). Le kell azonban szögeznünk, hogy a harmadik megoldás nem „a helyes", csupán a legáltalánosabb megoldás. A tervezőnek mérlegelnie kell a rendszer különböző szükségleteit, hogy megválassza az adott problémához az adott rendszerben megfelelő mértékű általánosságot és elvont ábrázolásmódot. Alapszabály, hogy egy hosszú életűnek tervezett programban az elvonatkoztatási szintnek a még megérthető és megengedhető legáltalánosabbnak, *nem* az abszolút legáltalánosabbnak kell lennie. Ha az általánosítás túlhaladja az adott program kereteit és a készítők tapasztalatát, káros lehet, mert késedelmet, elfogadhatatlanul rossz hatékonyságot, kezelhetetlen terveket és nyilvánvaló kudarcot von maga után.

Az ilyen módszerek kezelhetővé és gazdaságossá tételéhez az újrahasznosítás mikéntjét is meg kell terveznünk (§23.5.1) és nem szabad teljesen elfeledkezni a hatékonyságról (§23.4.7).

23.4.3. Tervezési lépések

Vegyük azt az esetet, amikor egyetlen osztályt tervezünk. Ez általában *nem* jó ötlet. Fogalmak *nem* léteznek elszigetelten; más fogalmakkal való összefüggéseik határozzák meg azokat. Ugyanez érvényes az osztályokra is; meghatározásuk logikailag kapcsolódó osztályok meghatározásával együtt történik. Jellemzően egymással kapcsolatban lévő osztályok egy halmazán dolgozunk. Az ilyen halmazt gyakran *osztálykönyvtárnak* (class library) vagy *összetevőnek* (komponens, component) nevezzük. Néha az adott összetevőben lévő összes osztály egyetlen osztályhierarchiát alkot, néha egyetlen névtér tagjai, néha viszont csupán deklarációk ad hoc gyűjteményét képezik (§24.4).

Az egy összetevőben lévő osztályok halmazát logikai kötelékek egyesítik; közös stílusuk van vagy azonos szolgáltatásokra támaszkodnak. Az összetevő tehát a tervezés, a dokumentáció, a tulajdonlás és az újrafelhasználás egysége. Ez nem jelenti azt, hogy ha valakinek egy összetevőből csak egy osztályra van szüksége, értenie és használnia kell az összetevő minden osztályát vagy be kell töltenie a programjába az összes osztály kódját. Éppen ellenkezőleg, általában arra törekszünk, hogy az osztályoknak a lehető legkevesebb gépi és emberi erőforrásra legyen szükségük. Ahhoz azonban, hogy egy összetevő bármely részét használhassuk, értenünk kell az összetevőt összetartó és meghatározó logikai kapcsolatokat, (a dokumentációban ezek remélhetőleg eléggé világosak), a felépítésében és dokumentációjában megtestesült elveket és stílust, illetve a közös szolgáltatásokat (ha vannak).

Lássuk tehát, hogyan tervezhetünk meg egy összetevőt. Mivel ez általában komoly kihívást jelent, megéri lépésekre bontani, hogy könnyebb legyen a különböző részfeladatokra összpontosítani. Szokott módon, az összetevők megtervezésének sincs egyetlen helyes útja. Itt csupán egy olyan lépéssorozatot mutatok be, amely egyesek számára már bevált:

1. Keressük meg a fogalmakat/osztályokat és legalapvetőbb kapcsolataikat.
2. Finomítsuk az osztályokat a rajtuk végezhető műveletek meghatározásával.
   - Osztályozzuk a műveleteket. Különösen figyeljünk a létrehozás, a másolás és a megsemmisítés szükségességére.
   - Törekedjünk a minél kisebb méretre, a teljességre és a kényelemre.
3. Finomítsuk az osztályokat az összefüggések meghatározásával.
   - Ügyeljünk a paraméterezésre és az öröklésre, és használjuk ki a függőségeket.

4. Határozzuk meg a felületeket.
   - Válasszuk szét a függvényeket privát és védett műveletekre.
   - A műveleteket határozzuk meg pontosan az adott osztályokra.

Észrevehetjük, hogy ezek is egy ismétlődő folyamat lépései. Általában többször kell rajtuk végigmenni, hogy kényelmesen használható tervet kapjunk, melyet felhasználhatunk egy első vagy egy újabb megvalósításnál. Ha az itt leírtak szerinti elemezzük a tennivalókat és készítjük el az elvont adatábrázolást, az azzal az előnnyel jár, hogy viszonylag könnyű lesz az osztálykapcsolatok átrendezése a kód megírása után is. (Bár ez sohasem túl egyszerű.)

Ha végeztünk, elkészítjük az osztályokat és visszatérünk felülvizsgálni a tervet annak alapján, amit a megvalósítás során megtudtunk. A következőkben ezeket a lépéseket egyenként vesszük sorra.

23.4.3.1. Első lépés: keressünk osztályokat

Keressük meg a fogalmakat/osztályokat és legalapvetőbb kapcsolataikat. A jó tervezés kulcsa a „valóság" valamely részének közvetlen modellezése – vagyis a program fogalmainak osztályokra való leképezése, az osztályok közti kapcsolatok ábrázolása meghatározott módon (például örökléssel), és mindezek ismételt végrehajtása különböző elvonatkoztatási szinteken. De hogyan fogjunk hozzá a fogalmak megkereséséhez? Milyen megközelítést célszerű követni, hogy eldönthessük, mely osztályokra van szükség?

A legjobb, ha először magában a programban nézünk szét, ahelyett, hogy a számítógéptudós „fogalomzsákjában" keresgélnénk. Hallgassunk valakire, aki a rendszer elkészülte után annak szakértő felhasználója lesz, és valakire, aki a kiváltandó rendszer kissé elégedetlen felhasználója. Figyeljük meg, milyen szavakat használnak.

Gyakran mondják, hogy a főnevek fognak megfelelni a programhoz szükséges osztályoknak és objektumoknak; és ez többnyire így is van. Ezzel azonban semmiképpen sincs vége a történetnek. Az igék jelenthetik az objektumokon végzett műveleteket, a hagyományos (globális) függvényeket, melyek paramétereik értékei vagy osztályok alapján új értékeket állítanak elő. Az utóbbira példák a függvényobjektumok (function object, §18.4) és a módosítók (manipulator, §21.4.6). Az olyan igék, mint a „bejár" vagy a „véglegesít" ábrázolhatók egy bejáró (iterátor) objektummal, illetve egy adatbázis *commit* műveletét végrehajtó objektummal. Még a melléknevek is gyakran használható módon ábrázolhatók osztályokkal. Vegyük a „tárolható", „párhuzamosan futó", „bejegyzett" és „lekötött" mellékneveket. Ezek is lehetnek osztályok, melyek lehetővé teszik a tervezőnek vagy a programozónak, hogy virtuális bázisosztályok meghatározása által kiválassza a később megtervezendő osztályok kívánt jellemzőit (§15.2.4).

Nem minden osztály felel meg programszintű fogalmaknak. Vannak például rendszer-erőforrásokat vagy a megvalósítás fogalmait ábrázoló osztályok (§24.3.1) is. Az is fontos, hogy elkerüljük a régi rendszer túl közeli modellezését. Például biztosan nem akarunk olyan rendszert készíteni, mely egy adatbázis köré épülve egy „kézi rendszer" lehetőségeit utánozza és az egyénnek csak papírok fizikai tologatásának irányítását engedi meg.

Az öröklés (inheritance) fogalmak közös tulajdonságainak ábrázolására szolgál. Legfontosabb felhasználása a hierarchikus felépítés ábrázolása az egyes fogalmakat képviselő osztályok viselkedése alapján (§1.7, §12.2.6, §24.3.2). Erre néha úgy hivatkoznak, mint *osztályozás* (classification), sőt *rendszertan* (taxonomy). A közös tulajdonságokat aktívan keresni kell. Az általánosítás és az osztályozás magas szintű tevékenységek, melyek éleslátást igényelnek, hogy hasznos és tartós eredményt adjanak. Egy közös alapnak egy általánosabb fogalmat kell ábrázolnia, nem pedig egy hasonló, az ábrázoláshoz esetleg kevesebb adatot igénylőt.

Vegyük észre, hogy az osztályozást a rendszerben modellezett fogalmak alapján kell végezni, nem más területeken érvényes szemszögből. A matematikában például a kör az ellipszis egy fajtája, de a legtöbb programban a kört nem az ellipszisből és az ellipszist sem a körből származtatják. Azok a gyakran hallható érvek, hogy „azért, mert a matematikában ez a módja" és „azért, mert a kör az ellipszis részhalmaza", nem meggyőzőek és általában rosszak. Ennek az az oka, hogy a legtöbb programban a kör fő tulajdonsága az, hogy középpontja van és távolsága a kerületéig rögzített. A kör minden viselkedése (minden művelete) meg kell, hogy tartsa ezt a tulajdonságot (invariáns; §24.3.7.1). Másrészt az ellipszist két fókuszpont jellemzi, melyeket sok programban egymástól függetlenül lehet módosítani. Ha ezek a fókuszpontok egybeesnek, az ellipszis olyan lesz, mint egy kör, de nem kör, mivel a műveletei nem tartják körnek. A legtöbb rendszerben ez a különbség úgy tükröződik, hogy van egy kör és egy ellipszis, amelyeknek vannak művelethalmazaik, de azok nem egymás részhalmazai.

Nem az történik, hogy kiagyalunk egy osztályhalmazt az osztályok közötti kapcsolatokkal és felhasználjuk azokat a végső rendszerben. Inkább osztályok és kapcsolatok kezdeti halmazát alkotjuk meg, majd ezeket ismételten finomítjuk (§23.4.3.5), hogy eljussunk egy olyan osztálykapcsolat-halmazig, mely eléggé általános, rugalmas és stabil ahhoz, hogy valódi segítséget nyújtson a rendszer későbbi megalkotásához.

A kezdeti fő fogalmak/osztályok felvázolására a legjobb eszköz egy tábla, finomításukra pedig az alkalmazási terület szakértőivel és néhány barátunkkal folytatott vita, melynek során kialakíthatunk egy használható kezdeti szótárat és fogalmi vázlatot. Kevés ember tudja ezt egyedül elvégezni. A kiinduló osztályhalmazból valóban használható osztályhalmaz kialakításának egyik módja a rendszer utánzása, ahol a tervezők veszik át az osztályok szerepét.

A nyilvános vita feltárja a kezdeti ötletek hibáit, más megoldások keresésére ösztönöz és elősegíti a kialakuló terv közös megértését. A tevékenység lapokra írott feljegyzésekkel támogatható, amelyek később visszakereshetők. Az ilyen kártyákat rendszerint CRC kártyáknak nevezik („Class, Responsibility, Collaborators", vagyis „Osztályok, Felelősség, Együttműködők", [Wirfs-Brock, 1990]) a rájuk feljegyzett információk miatt.

A *használati eset* (use case) a rendszer egy konkrét felhasználásának leírása. Íme a használati eset egyszerű példája egy telefonrendszerre: felvesszük a kagylót, tárcsázunk egy számot, a telefon a másik oldalon csenget, a másik oldalon felveszik a kagylót. Ilyen használati esetek halmazának meghatározása hatalmas értékű lehet a fejlesztés minden szakaszában, a tervezés elején pedig a használati esetek megkeresése segít, hogy megértsük, mit is próbálunk építeni. A tervezés alatt nyomon követhetjük velük a rendszer működési útját (például CRC kártyák használatával), hogy ellenőrizzük, van-e értelme a felhasználó szempontjából a rendszer osztályokkal és objektumokkal való viszonylag statikus leírásának; a programozás és tesztelés alatt pedig lehetséges helyzeteket modellezhetünk velük. Ily módon a használati eseteket a rendszer valószerűségének ellenőrzésére használhatjuk.

A használati esetek a rendszert (dinamikusan) működő egységként tekintik. Ezért a tervezőt abba a csapdába ejthetik, hogy a rendszer rendeltetését tartsa szem előtt, ami elvonja a figyelmét az osztályokkal ábrázolható, használható fogalmak keresésének elengedhetetlen feladatától. Különösen azok, akiknek a rendszerezett elemzés erősségük, de az objektumorientált programozásban és tervezésben kevés tapasztalattal rendelkeznek, a használati esetek hangsúlyozásával hajlamosak az eszközöket a célnak alárendelni. A használati esetek halmaza még nem tekinthető tervnek. Nem elég a rendszer használatát szem előtt tartani, gondolni kell annak felépítésére is.

A tervezők csapata eredendően hiábavaló – és költséges – kísérlet csapdájába eshet, ha az *összes* használati esetet meg akarja keresni és le akarja írni. Amikor a rendszerhez osztályokat keresünk, ugyanígy eljön az idő, amikor ki kell mondani: „Ami sok, az sok. Itt az idő, hogy kipróbáljuk, amink van és lássuk, mi történik." A további fejlesztés során csak egy elfogadható osztályhalmaz és egy elfogadható használatieset-halmaz használatával kaphatjuk meg a jó rendszer eléréséhez elengedhetetlen visszajelzést. Persze nehéz felismerni, mikor álljunk meg egy hasznos tevékenységben, különösen akkor, ha tudjuk, hogy később vissza kell térnünk a feladatot befejezni.

Mennyi eset elegendő? Erre a kérdésre általában nem lehet választ adni. Az adott munka során azonban eljön az idő, amikor a rendszer fő szolgáltatásainak legtöbbje működik és a szokatlanabb problémák, illetve a hibakezelési kérdések jó részét is érintettük. Ekkor ideje elvégezni a tervezés és programozás következő lépését.

Amikor a program felhasználási területeit használati esetek segítségével próbáljuk felbecsülni, hasznos lehet azokat elsődleges és másodlagos használati esetekre szétválasztani. Az elsődlegesek a rendszer legáltalánosabb, „normális” tevékenységeit, a másodlagosak a szokatlanabb, illetve a hibakezeléssel kapcsolatos tevékenységeket írják le. A másodlagos használati esetre példa a „telefonhívás” egy változata, ahol a hívott állomás kagylóját felemelték, mert éppen a hívó számát tárcsázzák. Gyakran mondják, hogy ha az elsődleges használati esetek 80%-át és néhány másodlagosat már kipróbáltunk, ideje továbbmenni, de mivel előre nem tudhatjuk, mi alkotja az „összes esetet”, ez csupán irányelv. Itt a tapasztalat és a jó érzék számít.

A fogalmak, műveletek és kapcsolatok a programok alkalmazási területeiből természetszerűen adódnak vagy az osztályszerkezeten végzett további munkából születnek. Ezek jelzik, hogy alapvetően értjük a program működését és képesek vagyunk az alapfogalmak osztályozására. Például a tűzoltókocsi egy tűzoltó gép, ami egy teherautó, ami egy jármű. A §23.4.3.2 és §23.4.5 pontok bemutatnak néhány módszert, amelyekkel az osztályokra és osztályhierarchiákra a továbbfejlesztés szándékával nézhetünk.

Óvakodjunk a rajzos tervezéstől! Természetesen a fejlesztés bizonyos szakaszában megkérnek majd, hogy mutassuk be a tervet valakinek, és akkor bemutatunk egy sereg diagramot, melyek az épülő rendszer felépítését magyarázzák. Ez nagyon hasznos lehet, mert segít, hogy figyelmünket arra összpontosítsuk, ami a rendszerben fontos, és kénytelenek vagyunk ötleteinket mások által is érthető szavakkal kifejezni. A bemutató értékes tervezési eszköz. Elkészítése azzal a céllal, hogy valóban megértsék azok, akiket érdekel és akik képesek építő bírálatot alkotni, jó gyakorlat a gondolatok megfogalmazására és tiszta kifejezésére.

A terv hivatalos bemutatása azonban veszélyes is lehet, mivel erős a kísértés, hogy az ideális rendszert mutassuk be – olyat, amit bárcsak meg tudnánk építeni, egy rendszert, melyre a vezetőség vágyik – ahelyett, amivel rendelkezünk, és amit elfogadható idő alatt meg tudunk csinálni. Amikor különböző megközelítések vetélkednek és a döntéshozók nem igazán értik a részleteket (vagy nem is törődnek azokkal), a bemutatók hazugságversennyé válnak, amelyben a leggrandiózusabb rendszert bemutató csapat nyeri el a munkát. Ilyen esetekben a gondolatok világos kifejezését gyakran szakzsargon és rövidítések helyettesítik. Ha ilyen bemutatót hallgatunk – és különösen, ha döntéshozók vagyunk és fejlesztési erőforrásokról rendelkezünk – létfontosságú, hogy képesek legyünk különbséget tenni vágyálom és reális terv között. A jó bemutatóanyag nem garantálja a leírt rendszer minőségét. Tapasztalatom szerint a valós problémákra összpontosító cégek, amikor eredményeik bemutatására kerül sor, rövidre fogják a szót azokhoz képest, akik kevésbé érdekeltek valós rendszerek elkészítésében.

Amikor osztályokkal ábrázolandó fogalmakat keresünk, vegyük észre, hogy a rendszernek vannak olyan fontos tulajdonságai is, melyeket nem ábrázolhatunk osztályokkal. A megbízhatóság, a teljesítmény, a tesztelhetőség fontos mérhető rendszertulajdonságok. Mégsem lehet még a legtisztábban objektumokra alapozott rendszer megbízhatóságát sem egy „megbízhatósági objektummal" ábrázolni. A rendszerben mindenütt jelenlevő tulajdonságok meghatározhatók, megtervezhetők és méréssel igazolhatók; gondot kell fordítani rájuk minden osztálynál, és tükröződniük kell az egyes osztályok és összetevők tervezési és megvalósítási szabályaiban (§23.4.3).

23.4.3.2. Második lépés: határozzuk meg a műveleteket

Finomítsuk az osztályokat a rajtuk végezhető műveletek meghatározásával. Természetes, hogy nem lehet az osztályok keresését elválasztani attól, hogy kigondoljuk, milyen műveleteket kell rajtuk végezni. Gyakorlati különbség van azonban abban, hogy az osztályok keresésekor a fő fogalmakra összpontosítunk és tudatosan háttérbe szorítjuk a számítástechnikai szempontokat, míg a műveletek meghatározásakor azt tartjuk szem előtt, hogy teljes és használható művelethalmazt találjunk. Általában túl nehéz egyszerre mindkettőre gondolni, különösen azért, mert az egymással kapcsolatban lévő osztályokat együtt kell megtervezni, de ebben is segíthetnek a CRC kártyák (§23.4.3.1).

Amikor az a kérdés, milyen szolgáltatásokról gondoskodjunk, többféle megközelítés lehetséges. Én a következő stratégiát javaslom:

1. Gondoljuk végig, hogyan jön létre, hogyan másolódik (ha egyáltalán másolódik) és hogyan semmisül meg az osztály egy objektuma.
2. Határozzuk meg az osztály által igényelt műveletek minimális halmazát. Általában ezek lesznek a tagfüggvények.
3. Gondoljuk végig, milyen műveleteket lehetne még megadni a kényelmesebb jelölés érdekében. Csak néhány valóban fontos műveletet adjunk meg. Ezekből lesznek a nem tag segédfüggvények (§10.3.2).
4. Gondoljuk végig, mely műveleteknek kell virtuálisaknak lenniük, vagyis olyan műveleteknek, melyekkel az osztály felületként szolgálhat egy származtatott osztály által adott megvalósítás számára.
5. Gondoljuk végig, milyen közös elnevezési módszert és szolgáltatásokat biztosíthatunk egy összetevő minden osztályára vonatkozóan.

Világos, hogy ezek a minimalizmus elvei. Sokkal könnyebb minden hasznosnak gondolt függvényt megadni és minden műveletet virtuálissá tenni. Minél több függvényünk van azonban, annál valószínűbb, hogy nem fogják használni azokat, és hogy a fölös függvé-

nyek korlátozni fogják a rendszer megvalósítását, további fejlődését. Nevezetesen azok a függvények, melyek egy osztály egy objektuma állapotának valamely részét közvetlenül olvassák vagy írják, gyakran egyedi megvalósítást követelnek az osztálytól és komoly mértékben korlátozzák az újratervezés lehetőségét. Az ilyen függvények az elvont ábrázolást a fogalom szintjéről a megvalósítás szintjére szállítják le. A függvények hozzáadása a programozónak is több munkát jelent – és a tervezőnek is, a következő újratervezéskor. *Sokkal* könnyebb egy függvényt megadni, amikor világossá vált, hogy szükség van rá, mint eltávolítani azt, ha koloncá válik.

Annak oka, hogy egy függvényről világosan el kell dönteni, hogy virtuális legyen vagy sem, és ez nem alapértelmezett viselkedés vagy megvalósítási részlet, az, hogy egy függvény virtuálissá tétele alapjaiban befolyásolja osztályának használatát és az osztály kapcsolatait más osztályokkal. Egy olyan osztály objektumainak elrendezése, melyben akár csak egyetlen virtuális függvény is van, jelentős mértékben különbözik a C vagy Fortran nyelvek objektumaitól. A virtuális függvényt is tartalmazó osztályok felületet nyújthatnak a később definiálandó osztályok számára, de egy virtuális függvény egyben függést is jelent azoktól (§24.3.2.1).

Vegyük észre, hogy a minimalizmus inkább több, mint kevesebb munkát követel a tervezőtől.

Amikor kiválasztjuk a műveleteket, fontos, hogy inkább arra összpontosítsunk, mit kell tenni, nem pedig arra, hogyan tegyük. Vagyis inkább a kívánt viselkedésre, mint a megvalósítás kérdéseire figyeljünk.

Néha hasznos az osztályok műveleteit aszerint osztályozni, hogyan használják az objektumok belső állapotát:

- Alapvető műveletek: konstruktorok, destruktorok és másoló operátorok.
- Lekérdezések: műveletek, melyek nem módosítják egy objektum állapotát.
- Módosítók: műveletek, melyek módosítják egy objektum állapotát.
- Konverziók: műveletek, melyek más típusú objektumot hoznak létre annak az objektumnak az értéke (állapota) alapján, amelyre alkalmazzuk őket.
- Bejárók: operátorok, melyek a tartalmazott objektumsorozatokhoz való hozzáférést, illetve azok használatát teszik lehetővé.

A fentiek nem egymást kizáró kategóriák. Egy bejáró például lehet egyben lekérdező vagy módosító is. Ezek a kategóriák egyszerűen egy osztályozást jelentenek, mely segíthet az osztályfelületek megtervezésében. Természetesen lehetségesek más osztályozás is. Az ilyen osztályozások különösen hasznosak az összetevőkön belüli osztályok egységességének fenntartásában.

A C++ a lekérdezők és módosítók megkülönböztetését a konstans és nem konstans tagfüggvényekkel segíti. Ugyanígy a konstruktorok, destruktorok, másoló operátorok és konverziós függvények is közvetlenül támogatottak.

23.4.3.3. Harmadik lépés: határozzuk meg az összefüggéseket

Finomítsuk az osztályokat az összefüggések meghatározásával. A különféle összefüggéseket a §24.3 tárgyalja. A tervezéssel kapcsolatosan vizsgálandó fő összefüggések a *paraméterezési*, *öröklési* (inheritance) és *használati* (use) kapcsolatok. Mindegyiknél meg kell vizsgálni, mit jelent, hogy az osztály a rendszer egyetlen tulajdonságáért felelős. Felelősnek lenni biztos, hogy nem azt jelenti, hogy az osztály az összes adatot maga kell, hogy tartalmazza, vagy hogy az összes szükséges műveletet közvetlenül tagfüggvényei kell, hogy végrehajtsák. Ellenkezőleg: az, hogy minden osztálynak egyetlen felelősségi területe van, azt biztosítja, hogy egy osztály munkája jobbára abból áll, hogy a kéréseket egy másik osztályhoz irányítja, mely az adott részfeladatért felelős. Legyünk azonban óvatosak, mert e módszer túlzott mértékű használata kis hatékonyságú és áttekinthetetlen tervekhez vezet, azáltal, hogy olyan mértékben elburjánzanak az osztályok és objektumok, hogy más munka nem történik, mint a kérések továbbítása. Amit az adott helyen és időben meg *lehet* tenni, azt meg *kell* tenni.

Annak szükségessége, hogy az öröklési és használati kapcsolatokat a tervezési szakaszban vizsgáljuk (és nem csak a megvalósítás során), közvetlenül következik abból, hogy az osztályokat fogalmak ábrázolására használjuk. Ez pedig azt is magában foglalja, hogy nem az egyedi osztály, hanem a komponens (§23.4.3, §24.4) a tervezés valódi egysége.

A paraméterezés – mely gyakran sablonok (template) használatához vezet – egy mód arra, hogy a mögöttes függéseket nyilvánvalóvá tegyük, úgy, hogy új fogalmak hozzáadása nélkül több lehetséges megoldás legyen. Gyakran választhatunk, meghagyunk-e valamit környezettől függő – egy öröklési fa ágaként ábrázolt – elemként, vagy inkább paramétert használunk (§24.4.1).

23.4.3.4. Negyedik lépés: határozzuk meg a felületeket

Határozzuk meg a felületeket. A privát függvényekkel a tervezési szakaszban nem kell foglalkozni. Azt, hogy ekkor a megvalósítás mely kérdéseit vesszük figyelembe, legjobb a harmadik lépésben, a függési viszonyok meghatározásakor eldönteni. Világosabban kifejezve: vegyük alapszabálynak, hogy ha egy osztálynak nem lehetséges legalább két jelentősen eltérő megvalósítása, akkor valószínű, hogy ezzel az osztállyal valami nincs rendben, és valójában álcázott megvalósítással (implementation), nem pedig igazi fogalommal állunk

szemben. Sok esetben az „Eléggé független ennek az osztálynak a felülete a megvalósítás módjától?" kérdés jó megközelítése, ha megvizsgáljuk, alkalmazható-e az osztályra valamilyen takarékos kiértékelési módszer („lusta kiértékelés", lazy evaluation).

Vegyük észre, hogy a nyilvános (public) alaposztályok és barát függvények (friend) az osztály nyilvános felületének részei (lásd még §11.5 és §24.4.2). Kifizetődő lehet, ha külön felületekről gondoskodunk az öröklés, illetve az általános felhasználók számára, azáltal, hogy külön *protected* és *public* felületeket adunk meg.

Ez az a lépés, amelyben a paraméterek pontos típusát meghatározzuk. Az az ideális, ha annyi statikus, programszintű típusokat tartalmazó felületünk van, amennyi csak lehetséges (§24.2.3 és §24.4.2).

Amikor a felületeket (interface) meghatározzuk, vigyázni kell azoknál az osztályoknál, ahol a műveletek látszólag több fogalmi szintet támogatnak. A *File* osztály egyes tagfüggvényei például *File_descriptor* (fájlleíró) típusú paramétereket kaphatnak, mások pedig karakterlánc paramétereket, melyek fájlneveket jelölnek. A *File_descriptor* műveletek más fogalmi szinten dolgoznak, mint a fájlnév-műveletek, így elgondolkozhatunk azon, vajon ugyanabba az osztályba tartoznak-e. Lehet, hogy jobb lenne, ha két fájlosztályunk lenne, az egyik a fájlleíró fogalmának támogatására, a másik a fájlnév fogalomhoz. Általában egy osztály minden művelete ugyanazt a fogalmi szintet kell, hogy támogassa. Ha nem így van, át kell szervezni az osztályt és a vele kapcsolatban lévő osztályokat.

23.4.3.5. Osztályhierarchiák újraszervezése

Az első és a harmadik lépésben megvizsgáltuk az osztályokat és osztályhierarchiákat, hogy lássuk, megfelelően kiszolgálják-e igényeinket. A válasz általában nem és ilyenkor át kell szerveznünk, osztályainkat, hogy tökéletesítsük a szerkezetet, a tervet vagy éppen a megvalósítás módját. Az osztályhierarchia legközönségesebb átszervezési módszere két osztály közös részének kiemelése egy új osztályba, illetve az osztály két új osztályra bontása. Mindkét esetben három osztály lesz az eredmény: egy bázisosztály (base class) és két származtatott osztály (derived class). Mikor kell így újraszervezni? Mi jelzi azt, hogy egy ilyen újraszervezés hasznos lehet?

Sajnos ezekre a kérdésekre nincsenek egyszerű, általános válaszok. Ez nem igazán meglepő, mert amiről beszélünk, nem apró részletek, hanem egy rendszer alapfogalmainak módosítása. Az alapvető – bár nem egyszerű – feladat az osztályok közös tulajdonságainak megkeresése és a közös rész kiemelése. Azt, hogy mi számít közösnek, nem lehet pontosan meghatározni, de a közös résznek a fogalmak közti, és nem csak a megvalósítás kényelmét szolgáló közösséget kell tükröznie. Arról, hogy két vagy több osztály olyan közös tu-

lajdonságokkal rendelkezik, melyeket egy bázisosztályban foglalhatunk össze, a közös használati minta, a művelethalmazok hasonlósága, illetve a hasonló megvalósítás árulkodik, valamint az, hogy ezek az osztályok gyakran együtt merülnek fel a tervezési viták során. Ezzel szemben egy osztály valószínűleg két másikra bontható, ha műveleteinek részhalmazai eltérő használati mintájúak, ha ezen részhalmazok az ábrázolás külön részhalmazaihoz férnek hozzá, vagy ha az osztály egymással nyilvánvalóan nem kapcsolatos tervezési viták során merül fel. A rokon osztályok halmazának sablonba (template) foglalása rendszerezett módot ad a szükséges alternetívák biztosítására (§24.4.1).

Az osztályok és fogalmak közti szoros kapcsolatok miatt az osztályhierarchiák szervezési problémái gyakran mint az osztályok elnevezési problémái merülnek fel. Ha az osztálynevek nehézkesen hangzanak vagy az osztályhierarchiákból következő osztályozások túlságosan bonyolultak, valószínűleg itt az alkalom, hogy tökéletesítsük a hierarchiákat. Véleményem szerint két ember sokkal jobban tud egy osztályhierarchiát elemezni, mint egy. Ha történetesen nincs valaki, akivel egy tervet meg tudnánk vitatni, hasznos lehet, ha írunk egy bevezető tervismertetőt, amelyben az osztályok neveit használjuk.

A terv egyik legfontosabb célkitűzése olyan felületek (interface) biztosítása, melyek a változások során állandóak maradnak (§23.4.2). Ezt gyakran úgy érhetjük el legjobban, ha egy osztályt, amelytől számos osztály és függvény függ, olyan absztrakt osztállyá teszünk, mely csak nagyon általános műveleteket biztosít. A részletesebb műveletek jobb, ha egyedi célú származtatott osztályokhoz kapcsolódnak, melyektől kevesebb osztály és függvény függ közvetlenül. Világosabban: minél több osztály függ egy osztálytól, annál általánosabbnak kell lennie.

Erős a kísértés, hogy egy sokak által használt osztályhoz műveleteket (és adatokat) tegyünk hozzá. Ezt gyakran úgy tekintik, mint ami egy osztályt használhatóbbá és (további) változtatást kevésbé igénylővé tesz. Pedig az eredmény csak annyi, hogy a felület meghízik („kövér felület", §24.4.3) és olyan adattagjai lesznek, melyek számos gyengén kapcsolódó szolgáltatást támogatnak. Ez ismét azzal jár, hogy az osztályt módosítani kell, amikor az általa támogatott osztályok közül akár csak egy is jelentősen megváltozik. Ez viszont magával vonja olyan felhasználói osztályok és származtatott osztályok módosítását is, melyek nyilvánvalóan nincsenek vele kapcsolatban. Ahelyett, hogy bonyolítanánk egy, a tervben központi szerepet játszó osztályt, rendszerint általánosnak és elvontnak kellene hagynunk. Ha szükséges, az egyedi szolgáltatásokat származtatott osztályokként kell biztosítani. (Példákat lásd [Martin,1995]).

Ez a gondolatfüzér absztrakt (abstract) osztályok hierarchiájához vezet, ahol a gyökérhez közeli osztályok a legáltalánosabbak, és ezektől függ a legtöbb más osztály és függvény. A levél osztályok a legegyedibbek és csak nagyon kevés kódrész függ közvetlenül tőlük. Példaként említhetnénk az *Ival_box* hierarchia végső változatát (§12.4.3, §12.4.4).

23.4.3.6. Modellek használata

Amikor cikket írok, egy megfelelő modellt próbálok követni. Vagyis ahelyett, hogy azonnal elkezdem a gépelést, hasonló témájú cikkeket keresek, hogy lássam, találok-e olyat, amely a cikkem kezdeti mintájaként szolgálhat. Ha az általam választott modell egy saját, rokon témáról szóló cikk, még arra is képes vagyok, hogy helyükön hagyok szövegrészeket, másokat szükség szerint módosítok és új információt csak ott teszek hozzá, ahol mondandóm logikája megkívánja. Ez a könyv például ilyen módon íródott, az első és második kiadás alapján. Ennek a módszernek szélsőséges esete egy levélsablon használata. Ekkor egyszerűen egy nevet írok be és esetleg hozzáírok még néhány sort, hogy a levél „személyre szóló” legyen. Az ilyen leveleket lényegében úgy írom meg, hogy a sablont csak az alapmodelltől eltérő részekkel egészítem ki.

A létező rendszerek ilyen, új rendszerek modelljeiként való felhasználása általános eljárás az alkotó törekvések minden formájánál. Amikor csak lehetséges, a tervezés és programozás előző munkára kell, hogy alapozódjon. Ez korlátozza a tervező szabadságát, de lehetővé teszi, hogy figyelmét egyszerre csak néhány kérdésre összpontosítsa. Egy nagy projekt tiszta lappal indítása frissítő hatással lehet, de könnyen „mérgezővé” is válhat, mert a tervezési módszerek közötti kóválygást eredményezheti. Ennek ellenére, ha van modellünk, az nem jelent korlátozást és nem követeli meg, hogy szolgai módon kövessük; egyszerűen megszabadítja a tervezőt attól, hogy egy tervet egyszerre csak egy szempontból közelíthessen meg.

Vegyük észre, hogy a modellek használata elkerülhetetlen, mivel minden terv tervezőinek tapasztalataiból tevődik össze. Egy adott modell birtokában a modellválasztás tudatos elhatározássá válik, feltételezéseink megalapozottak, a használt kifejezések köre pedig meghatározott lesz, a tervnek lesz egy kezdeti váza és nő a valószínűsége, hogy a tervezők meg tudnak egyezni egy közös megközelítésben.

Természetesen a kiinduló modell kiválasztása önmagában is fontos tervezési döntés, amely gyakran csak a lehetséges modellek keresése és az alternatívák gondos kiértékelése után hozható meg. Továbbá sok esetben egy modell csak akkor felel meg, ha megértjük, hogy az elképzeléseknek egy új konkrét alkalmazáshoz való igazítása számos módosítást igényel. A programtervezés nem könnyű, így minden megszerezhető segítségre szükségünk van. Nem szabad visszautasítanunk a modellek használatát, csak a „rossz utánzás” erőltetését. Az utánzás a hízelgés legőszintébb módja, és a modellek és előző munkák ösztönzésként való felhasználása – a tulajdon és a másolás jogának törvényes határain belül – az innovatív munka minden területen elfogadható módszer: ami jó volt Shakespeare-nek, az jó nekünk is. Vannak, akik a modellek ilyen használatát a tervezésben „terv-újrahasznosításnak” nevezik.

Az általánosan használt, több programban előforduló elemek feljegyzése – az általuk megoldott probléma és használatuk feltételeinek leírásával együtt – kézenfekvő ötlet; feltéve, hogy eszünkbe jut. Az ilyen általános és hasznos tervezési elemek leírására általában a *minta* (pattern) szót használjuk, a minták dokumentálásának és használatának pedig komoly irodalma van (pl. [Gamma, 1994] és [Coplien, 1995]).

Célszerű, hogy a tervező ismerje az adott alkalmazási terület népszerű mintáit. Mint programozó, olyan mintákat részesítek előnyben, melyekhez valamilyen kód is tartozik, ami példaként tekinthető. Mint a legtöbb ember, egy általános ötletet (ebben az esetben mintát) akkor tartok a legjobbnak, ha segítségemre van egy konkrét példa is (ebben az esetben egy, a minta használatát illusztráló kódrészlet). Akik gyakran használnak mintákat, szakzsargont használnak egymás között, ami sajnos privát nyelvvé válhat, ami kívülállók számára lehetetlenné teszi a megértést. Mint mindig, elengedhetetlen a megfelelő kommunikáció biztosítása a projekt különböző részeiben résztvevők (§23.3) és általában a tervező és programozó közösségek között.

Minden sikeres nagy rendszer egy valamivel kisebb működő rendszer áttervezése. Nem ismerek kivételt e szabály alól. Még leginkább olyan tervek jutnak eszembe, melyek évekig szenvedtek zűrzavaros állapotban és nagy költséggel, évekkel a tervezett befejezési dátum után esetleg sikeresek lettek. Az ilyen projektek eredménye – szándékolatlanul és persze elismerés nélkül – először működésképtelen rendszer lett, amit később működőképessé tettek, végül újraterveztek olyan rendszerré, mely megközelíti az eredeti célkitűzéseket. Ebből következik, hogy ostobaság egy nagy rendszert újonnan megtervezni és megépíteni, pontosan követve a legfrissebb elveket. Minél nagyobb és nagyratörőbb a rendszer, melyet megcéloztunk, annál fontosabb, hogy legyen egy modellünk, melyből dolgozunk. Egy nagy rendszerhez az egyetlen valóban elfogadható modell egy valamivel kisebb, *működő* rokon rendszer.

23.4.4. Kísérletezés és elemzés

Egy igényes fejlesztés kezdetén nem tudjuk a módját, hogyan alkossuk meg legjobban a rendszert. Gyakran még azt sem tudjuk pontosan, mit kellene a rendszernek tennie, mivel a konkrétumok csak akkor tisztulnak ki, amikor a rendszert építjük, teszteljük és használjuk. Hogyan kapjuk meg a teljes rendszer felépítése előtt a szükséges információkat ahhoz, hogy megértsük, melyek a jelentős tervezési döntések és felbecsüljük azok buktatóit?

Kísérleteket folytatunk. Mihelyt van mit, elemezzük a terveket és azok megvalósítását is. A leggyakoribb és legfontosabb a lehetőségek megvitatása. A tervezés általában közösségi tevékenység, melyben a tervek bemutatók és viták által fejlődnek. A legfontosabb tervezőeszköz a tábla; enélkül az embrionális állapotú tervezési fogalmak nem tudnak kifejlődni és elterjedni a tervezők és programozók között.

A kísérlet legnépszerűbb formája egy prototípus építése, vagyis a rendszer vagy egy részének lekicsinyített változatban való létrehozása. A prototípus teljesítményére nézve nincsenek szigorú előírások, általában elegendő a gépi és programozási környezetbeli erőforrás, a tervezők és programozók pedig igyekeznek különösen képzettnek, tapasztaltnak és motiváltnak látszani. Arról van szó, hogy egy változatot a lehető leggyorsabban futtatni lehessen, hogy lehetővé tegyük a tervezési és megvalósítási módszerek választékának felderítését.

Ez a megközelítés nagyon sikeres lehet, ha jól csináljuk, de lehet ürügy a pongyolaságra is. Az a probléma, hogy a prototípus súlypontja a lehetőségek felderítésétől eltolódhat „a lehető leghamarabb valamilyen működőképes rendszert kapni" felé. Ez könnyen a prototípus belső felépítése iránti érdektelenséghez („végül is csak prototípusról van szó") és a tervezési erőfeszítések elhanyagolásához vezet, mert a prototípus gyors elkészítése ezekkel szemben előnyt élvez. Az a bökkenő, hogy az ily módon elkészített változatban sokszor nyoma sincs az erőforrások helyes felhasználásának vagy a módosíthatóság lehetőségének, miközben egy „majdnem teljes" rendszer illúzióját adja. Szinte törvényszerű, hogy a prototípus nem rendelkezik azzal a belső felépítéssel, hatékonysággal és áttekinthetőséggel, mely megengedné, hogy rajta keresztül a valós használatot felmérhessük. Következésképpen egy „prototípus", melyből „majdnem termék" lesz, elszívja azt az időt és energiát, amit a valódi termékre lehetett volna fordítani. Mind a fejlesztők, mind a vezetők számára kísértést jelent, hogy a prototípusból terméket csináljanak és a „teljesítmény megtervezését" elhalasszák a következő kiadásig. A prototípuskészítésnek ez a helytelen használata tagadása mindennek, amiért a tervezés létezik.

Rokon probléma, hogy a prototípus fejlesztői beleszerethetnek az eszközeikbe. Elfelejthetik, hogy az általuk élvezett kényelem költségeit egy valódi rendszer nem mindig engedheti meg magának, és az a korlátok nélküli szabadság, melyet kis kutatócsoportjuk nyújtott, nemigen tartható fenn egy nagyobb közösségben, mely szoros – és egymástól függő – határidők szerint dolgozik.

Másrészről, a prototípusok felbecsülhetetlen értékűek. Vegyük például egy felhasználói felület tervezését. Ebben az esetben a rendszer a felhasználóval közvetlen kölcsönhatásban nem levő részének belső felépítése tényleg lényegtelen, és a prototípus használatán kívül nincs más lehetséges mód, hogy tapasztalatokat szerezzünk arról, mi a véleménye a felhasználóknak a rendszer külsejével és használatával kapcsolatban. De példaként vehetjük

az ellenkezőjét is, egy olyan prototípust, melyet szigorúan egy program belső működésének tanulmányozására terveztek. Itt a felhasználói felület lehet kezdetleges – lehetőség szerint a valódi felhasználók helyett szimuláltakkal.

A prototípuskészítés a kísérletezés egy módja. Eredményei pedig azok a meglátások kellenek, hogy legyenek, melyeket nem maga a prototípus, hanem annak építése hoz magával. Tulajdonképpen a prototípus legfőbb ismérveként a tökéletlenséget kellene meghatározni, hogy világos legyen, ez csupán kísérleti eszköz és nem válhat termékké nagymértékű újratervezés nélkül. Ha „tökéletlen" prototípusunk van, az segít, hogy a kísérletre összpontosítsunk és a lehető legkisebbre csökkenti annak veszélyét, hogy a prototípusból termék legyen, valamint megszünteti azt a kísértést is, hogy a termék tervezését túl szorosan a prototípusra alapozzuk, elfelejtve vagy elhanyagolva annak eredendő korlátait. A prototípus használat után eldobandó.

Vannak olyan kísérleti módszerek is, melyeket a prototípuskészítés helyett alkalmazhatunk. Ahol ezek használhatók, gyakran előnyben is részesítjük őket, mert szigorúbbak és kevesebb tervezői időt és rendszer-erőforrást igényelnek. Ilyenek például a matematikai modellek és a különféle szimulátorok. Tulajdonképpen fel is vázolhatunk egy folyamatot, a matematikai modellektől az egyre részletesebb szimulációkon, a prototípusokon, és a részleges megvalósításon át a teljes rendszerig.

Ez ahhoz az ötlethez vezet, hogy egy rendszert a kezdeti tervből és megvalósításból többszöri újratervezéssel és újbóli elkészítéssel finomítsunk. Ez az ideális módszer, de nagyon nagy igényeket támaszthat a tervezői és fejlesztői eszközökkel szemben. A megközelítés azzal a kockázattal is jár, hogy túl sok kód készül a kezdeti döntések alapján, így egy jobb tervet nem lehet megvalósítani. Így ez a stratégia egyelőre csak kis és közepes méretű programok készítésénél működik jól, melyekben nem valószínű, hogy az átfogó tervek különösebben módosulnának, illetve a programok első kiadása utáni újratervezésekre és újbóli megvalósításokra, ahol elkerülhetetlen az ilyen megközelítés.

A tervezési lehetőségek áttekintésére szolgáló kísérleteken kívül egy terv vagy megvalósítás elemzése önmagában is ötleteket adhat. A legnagyobb segítség például az osztályok közti különböző összefüggések (§24.3) tanulmányozása lehet, de nem elhanyagolhatók a hagyományos eszközök sem, mint a hívási fák (call graphs) megrajzolása, a teljesítménymérések stb.

Ne feledjük, hogy a fogalmak meghatározása (az elemzési szakasz „kimenete") és a tervezés során éppúgy elkövethetünk hibákat, mint a megvalósítás folyamán. Valójában még többet is, mivel ezen tevékenységek eredménye kevésbé konkrét, kevésbé pontosan meghatározott, nem „végrehajtható" és általában nem támogatják olyan kifinomult eszközök,

mint amilyenek a megvalósítás elemzéséhez és ellenőrzéséhez használhatók. Ha a tervezés kifejezésére használt nyelvet vagy jelölésmódot formálisabbá tesszük, valamilyen mértékig ilyen eszközt adhatunk a tervező kezébe, ezt azonban nem tehetjük annak az árán, hogy szegényebbé tesszük a megvalósításra használt programozási nyelvet (§24.3.1). Ugyanakkor egy formális jelölésmód maga is problémák forrása lehet, például ha a használt jelölésrendszer nem illik jól arra a gyakorlati problémára, melyre alkalmazzuk; amikor a formai követelmények szigora túllépi a résztvevő tervezők és programozók matematikai felkészültségét és érettségét; vagy amikor a formális leírás elszakad a leírandó rendszertől.

A tervezés eredendően ki van téve a hibáknak és nehéz támogatni hatékony eszközökkel. Ezért nélkülözhetetlen a tapasztalat és a visszajelzés. Következésképpen alapvetően hibás a programfejlesztést egyenes vonalú folyamatként tekinteni, mely az elemzéssel kezdődik és a teszteléssel végződik. Figyelmet kell fordítani az ismételt tervezésre és megvalósításra, hogy a fejlesztés különböző szakaszaiban elegendő visszajelzésünk legyen a tapasztaltakról.

23.4.5. Tesztelés

Az a program, melyet nem teszteltek, nem tekinthető működőnek. Az eszménykép, hogy egy programot úgy készítsünk el, hogy már az első alkalommal működjön, a legegyszerűbb programokat kivéve elérhetetlen. Törekednünk kell erre, de nem szabad ostobán azt hinnünk, hogy tesztelni könnyű.

A „Hogyan teszteljünk?" olyan kérdés, melyre nem lehet csak úgy általában válaszolni. A „Mikor teszteljünk?" kérdésre azonban van általános válasz: olyan hamar és olyan gyakran, amennyire csak lehetséges. A tesztelési módszert célszerű már a tervezés és megvalósítás részeként, de legalább azokkal párhuzamosan kidolgozni. Mihelyt van egy futó rendszer, el kell kezdeni a tesztelést. A komoly tesztelés elhalasztása a „befejezés" utánig a határidők csúszását és /vagy hibás kiadást eredményezhet.

Ha csak lehetséges, a rendszert úgy kell megtervezni, hogy viszonylag könnyen tesztelhető legyen. Az ellenőrzést gyakran bele lehet tervezni a rendszerbe. Néha ezt nem tesszük, attól való félelmünkben, hogy a futási idejű tesztelés rontja a hatékonyságot, vagy hogy az ellenőrzéshez szükséges redundáns elemek túlságosan nagy adatszerkezeteket eredményeznek. Az ilyen félelem rendszerint alaptalan, mert a legtöbb tesztkód szükség esetén leválasztható a tényleges kódról, mielőtt a rendszert átadjuk. Az *assertion*-ök (§24.3.7.2) használata néha hasznos lehet.

Maguknál a teszteknél is fontosabb, hogy a rendszer olyan felépítésű legyen, ami elfogadható esélyt ad arra, hogy saját magunkat és felhasználóinkat/fogyasztóinkat meggyőzzük

arról, hogy a hibákat statikus ellenőrzés, statikus elemzés és tesztelés együttes használatával ki tudjuk küszöbölni. Ahol a hibatűrés biztosítására kidolgoztak valamilyen módszert (§14.9), ott általában a tesztelés is beépíthető a teljes tervbe.

Ha a tervezési szakaszban a tesztelési kérdésekkel egyáltalán nem számoltunk, akkor az ellenőrzéssel, a határidők betartásával és a program módosításával problémáink lesznek. Az osztályfelületek és az osztályfüggések rendszerint jó kiindulópontot jelentenek a tesztelési módszer kidolgozásához(§24.3, §24.4.2).

Rendszerint nehéz meghatározni, mennyi tesztelés elegendő. A túl kevés tesztelés azonban általánosabb probléma, mint a túl sok. Az, hogy a tervezéshez és megvalósításhoz viszonyítva pontosan mennyi erőforrást kell a tesztelésre kijelölni, természetesen függ a rendszer természetétől és a módszerektől, melyeket építéséhez használunk. Irányelvként javasolhatom azonban, hogy időben, erőfeszítésben és tehetségben több erőforrást fordítsunk a rendszer tesztelésére, mint első változatának elkészítésére. A tesztelésnek olyan problémákra kell összpontosítania, melyeknek végzetes következményei lennének és olyanokra, melyek gyakran fordulnának elő.

23.4.6. A programok „karbantartása"

A „program-karbantartás" (software maintenance) helytelen elnevezés. A „karbantartás" szó félrevezető hasonlatot sugall a hardverrel. A programoknak nincs szükségük olajozásra, nincsenek mozgó alkatrészeik, amelyek kopnak, és nincsenek repedéseik, ahol a víz összegyűlik és rozsdásodást okoz. A szoftver *pontosan* lemásolható és percek alatt nagy távolságra átvihető. A szoftver nem hardver.

Azok a tevékenységek, melyeket program-karbantartás néven emlegetünk, valójában az újratervezés és az újbóli megvalósítás, tehát a szokásos programfejlesztési folyamathoz tartoznak. Amikor a tervezésben hangsúlyosan szerepel a rugalmasság, bővíthetőség és hordozhatóság, közvetlenül a program fenntartásával kapcsolatos hibák hagyományos forrásaira összpontosítunk.

A teszteléshez hasonlóan a karbantartás sem lehet utólagos megfontolás vagy a fejlesztés fő sodrától elválasztott tevékenység. Különösen fontos, hogy a projektben végig ugyanazok a személyek vegyenek részt, vagy legalábbis biztosítsuk a folytonosságot . Nem könnyű ugyanis „karbantartást" végeztetni egy olyan új (és általában kisebb tapasztalatú) emberekből álló csoporttal, akik az eredeti tervezőkkel és programozókkal nem állnak kapcsolatban. Ha mégis másoknak kell átadni a munkát, hangsúlyt kell fektetni arra, hogy az új emberek tisztában legyenek a rendszer felépítésével és célkitűzéseivel. Ha a „karbantartó

személyzet" magára marad abban, hogy kitalálja a rendszer szerkezetét, vagy hogy a rendszer összetevőinek célját megvalósításukból következtesse ki, a rendszer a helyi „foltozgatás" hatására gyorsan degenerálódik. A dokumentáció nem segít, mert az inkább a részletek leírására való, nem pedig arra, hogy az új munkatársakat segítse a fő ötletek és elvek megértésében.

23.4.7. Hatékonyság

Donald Knuth megfigyelte, hogy „az idő előtti optimalizálás a gyökere minden rossznak". Némelyek túlságosan megtanulták ezt a leckét és minden hatékonysági törekvést rossznak tekintenek, pedig a hatékonyságra végig ügyelni kell a tervezés és megvalósítás során. Ez azonban nem azt jelenti, hogy a tervezőnek minden „apróságot" hatékonnyá kell tennie, hanem csak azt, hogy az elsőrendű hatékonysági kérdésekkel foglalkoznia kell.

A hatékonyság eléréséhez a legjobb módszer egy világos és egyszerű terv készítése. Csak egy ilyen terv maradhat viszonylag állandó a projekt élettartama alatt és szolgálhat a teljesítmény behangolásának alapjául. Lényeges, hogy elkerüljük a nagy projekteket sújtó megalomániát. Túl gyakori, hogy a programozók „ha szükség lenne rá" tulajdonságokat (§23.4.3.2, §23.5.3) építenek be, majd a „sallangok" támogatásával megkétszerezik, megnégyszerezik a rendszerek méretét és futási idejét. Ennél is rosszabb, hogy az ilyen túlfinomított rendszereket gyakran szükségtelenül nehéz elemezni, így nehéz lesz megkülönböztetni az elkerülhető túlterhelést az elkerülhetetlentől. Ez pedig még az alapvető elemzést és optimalizálást is akadályozza. Az optimum beállítása elemzés és teljesítménymérés eredménye kell, hogy legyen, nem találomra piszmogás a kóddal. A nagy rendszerek esetében a tervezői vagy programozói „intuíció" különösen megbízhatatlan útmutató a hatékonysági kérdésekben.

Fontos, hogy elkerüljük az eredendően rossz hatékonyságú szerkezeteket, illetve az olyanokat, melyeknek elfogadható teljesítményszintre való optimalizálása sok időt és ügyességet kíván. Hasonlóképpen fontos, hogy csökkentsük az eredendően „hordozhatatlan" szerkezetek és eszközök használatát, mert ez arra kárhoztatja a projektet, hogy régebbi (kisebb teljesítményű) vagy éppen drágább gépeken fusson.

23.5. Vezetés

Feltéve, hogy legalább valamilyen értelme van, a legtöbb ember azt teszi, amire ráveszik. Nevezetesen, ha egy projekttel összefüggésben bizonyos „működési módokat" díjazunk és másokat büntetünk, csak kivételes programozók és tervezők fogják kockáztatni a karrierjüket, hogy azt tegyék, amit helyesnek tartanak, a vezetés ellenvéleményével, közönyével és bürokráciájával szemben?.[3] Ebből következik, hogy a cégnek kell, hogy legyen egy jutalmazási rendszere, mely megfelel az általa megállapított tervezési és programozási célkitűzéseknek. Mindazonáltal gyakran nem ez a helyzet: a programozási stílusban nagy változás csak a tervezési stílus megfelelő módosításával érhető el és ahhoz, hogy hatásos legyen, általában mindkettő a vezetési stílus megváltoztatását igényli. A szellemi és szervezeti tehetetlenség könnyen eredményezhet olyan helyi változtatásokat, melyeket nem támogatnak a sikerét biztosító globális változtatások. Meglehetősen jellemző példa egy objektumorientált programozást támogató nyelvre (mondjuk a C++-ra) való átállás, a nyelv lehetőségeit kihasználó megfelelő tervezési stratégiabeli változtatás nélkül, vagy éppen fordítva, „objektumorientált tervezésre" váltás azt támogató programozási nyelv bevezetése nélkül.

23.5.1. Újrahasznosítás

A kódok és tervek újrahasznosítását gyakran emlegetik, mint fő okot egy új programozási nyelv vagy tervezési módszer bevezetésére. A legtöbb szervezet azonban azokat az egyéneket támogatja, akik a melegvíz újrafeltalálását választják. Például ha egy programozó teljesítményét kódsorokban mérik; vajon kis programokat fog írni a standard könyvtárakra támaszkodva, bevételének és esetleg státuszának rovására? És ha egy vezetőt a csoportjában lévő emberek számával arányosan fizetnek, vajon fel fogja-e használni a mások által készített programot, ha helyette a saját csoportjába újabb embereket vehet fel? Vagy ha egy cégnek odaítélnek egy kormányszerződést, ahol a profit a fejlesztési költségeknek egy rögzített százaléka, vajon ez a cég közvetetten csökkenteni fogja-e a profitját a leghatékonyabb fejlesztőeszközök használatával? Az újrahasznosítást nehéz jutalmazni, de ha a vezetés nem talál módot az ösztönzésére és jutalmazására, nem lesz újrahasznosítás. Az újrahasznosításnak más vonatkozásai is vannak. Akkor használhatjuk fel valaki más programját, ha:

1. Működik: ahhoz, hogy újrahasznosítható legyen, a programnak előbb használhatónak kell bizonyulnia.

[3] Az olyan szervezetnek, mely a programozóival úgy bánik, mint az idiótákkal, hamarosan olyan programozói lesznek, akik csak idiótaként akarnak és képesek cselekedni.

2. Áttekinthető: fontos a program szerkezete, a megjegyzések, a dokumentáció és az oktatóanyag.
3. Együtt létezhet egy olyan programmal, melyet nem kimondottan a vele való együttműködésre írtak.
4. Támogatott (vagy magunk akarjuk támogatni; én általában nem akarom).
5. Gazdaságos (megoszthatom-e más felhasználókkal a fejlesztési és karbantartási költségeket?).
6. Meg lehet találni.

Ehhez hozzátehetjük, hogy egy összetevő nem újrahasznosítható, amíg nincs valaki, aki „újra felhasználta". Egy összetevő valamely környezetbe beillesztése jellemzően működésének finomításával, viselkedésének általánosításával és más programokkal való együttműködési képességének javításával jár együtt. Amíg legalább egyszer el nem végeztük ezt a feladatot, még a leggondosabban megtervezett és elkészített összetevőknek is lehetnek szándékolatlan és váratlan egyenetlenségei.

Az a tapasztalatom, hogy az újrahasznosításnak csak akkor vannak meg a szükséges feltételei, ha valaki közös ügynek tekinti, hogy ilyen munkamegosztás működjön. Egy kis csoportban ez általában azt jelenti, hogy valaki – tervezetten vagy véletlenül – a közös könyvtárak és dokumentáció gazdája lesz, egy nagyobb cégnél pedig azt, hogy megbíznak egy csoportot vagy osztályt, hogy gyűjtse össze, készítse el, dokumentálja, népszerűsítse és „tartsa karban" a több csoport által felhasznált programösszetevőket. Nem lehet túlbecsülni a „szabványos összetevőkkel" foglalkozó csoport fontosságát. Vegyük észre, hogy egy program azt a közösséget tükrözi, amely létrehozta. Ha a közösségnek nincs együttműködést és munkamegosztást elősegítő és jutalmazó rendszere, ritkán lesz együttműködés és munkamegosztás. A szabványos összetevőkkel foglalkozó csoport aktívan kell, hogy népszerűsítse ezeket az összetevőket. Ebből következik, hogy a jól megírt dokumentáció nélkülözhetetlen, de nem elegendő. A csoportnak ezen kívül gondoskodnia kell oktatóanyagokról és minden más információról is, ami lehetővé teszi, hogy a reménybeli felhasználó megtaláljon egy összetevőt és megértse, miért segíthet az neki. Ennek következménye, hogy a piackutatással és az oktatással kapcsolatos tevékenységeket ennek a csoportnak kell vállalnia.

Ha lehetséges, e csoport tagjainak szorosan együtt kell működniük az alkalmazásfejlesztőkkel. Csak így tudnak megfelelően értesülni a felhasználók igényeiről és felhívni a figyelmet az alkalmakra, amikor egyes összetevőket különböző programok közösen használhatnak. Ezzel amellett érvelek, hogy az ilyen csoportoknak legyen vélemény-nyilvánítási és tanácsadói szerepe, és hogy működjenek a belső kapcsolatok a csoportba bejövő és onnan kimenő információ átvitelére.

Az „komponens-csoport" sikerét az ügyfelek sikerével kell mérni. Ha a sikert egyszerűen azzal mérjük, hogy mennyi eszközt és szolgáltatást tudnak elfogadtatni a fejlesztő cégekkel, az ilyen csoport kereskedelmi ügynökké alacsonyodik és állandóan változó hóbortok előmozdítója lesz.

Nem minden kódnak kell újrahasznosíthatónak lennie, ez nem „mindenható" tulajdonság. Ha azt mondjuk, hogy egy összetevő „újrahasznosítható", ez azt jelenti, hogy felhasználása bizonyos kereteken belül kevés vagy semmi munkát nem igényel. A legtöbb esetben egy más vázba történő átköltöztetés jelentős munkával jár. Ebből a szempontból az újrahasznosítás erősen emlékeztet a hordozhatóságra (vagyis más rendszerre való átültetésre). Fontos megjegyezni, hogy az újrahasznosítás az ezt célzó tervezésnek, az összetevők tapasztalat alapján való finomításának és annak a szándékos erőfeszítésnek az eredménye, hogy már létező összetevőket keressünk (újbóli) felhasználásra. Az újrahasznosítás nem az egyes nyelvi tulajdonságok vagy kódolási módszerek véletlenszerű használatából keletkezik, mintegy csodaként. A C++ olyan szolgáltatásai, mint az osztályok, a virtuális függvények és a sablonok lehetővé teszik, hogy a tervezés kifejezésmódja az újrahasznosítást könnyebbé (és ezáltal valószínűbbé) tegye, de önmagukban nem biztosítják azt.

23.5.2. Méret és egyensúly

Az önálló vagy közösségben dolgozó programozót sokszor arra kényszerítik, hogy dolgát „a megfelelő módon" végezze. Intézményes felállásban ez gyakran így hangzik: „a megfelelő eljárások kifejlesztése és szigorú követése szükséges". Mindkét esetben elsőként a józan ész eshet áldozatul annak a kívánságnak, hogy mindenáron javítsunk „a dolgok végzésének" módján. Sajnos, ha egyszer hiányzik a józan ész, az akaratlanul okozható károknak nincs határa.

Vegyük a fejlesztési folyamat §23.4-ben felsorolt szakaszait és a §23.4.3-ban felsorolt tervezési lépéseket. E lépésekből viszonylag könnyen kidolgozható egy megfelelő tervezési módszer, ahol minden szakasz pontosan körülhatárolt, meghatározott „bemenete és kimenete" van, valamint rendelkezik az ezek kifejezéséhez szükséges, részben formális jelölésmóddal. Ellenőrző listákkal biztosíthatjuk, hogy a tervezési módszer következetesen támogassa a nagy számú eljárási és jelölésbeli szabály érvényre juttatását, és hogy ehhez eszközöket lehessen kifejleszteni. Továbbá, az összefüggések §24.3-ban bemutatott osztályozását nézve valaki kinyilatkoztathatná, hogy bizonyos kapcsolatok jók, mások pedig rosszak, és elemző eszközöket adna annak biztosítására, hogy ezeket az értékítéleteket az adott projekten belül egységesen alkalmazzák. A programfejlesztési folyamat e „megerősítését" tökélyre fejlesztendő, valaki leírhatná a dokumentálás szabályait (beleértve a helyesírási, nyelvtani és gépelési szabályokat) és a kód általános küllemére vonatkozó előírásokat

(beleértve azt is, mely nyelvi tulajdonságokat és könyvtárakat szabad és melyeket nem szabad használni, hol használható behúzás, hogyan lehet elnevezni a függvényeket, változókat és típusokat stb.).

Ezek között akad olyan, amely elősegítheti a sikert. Természetesen ostobaság lenne egy esetleg tízmillió kódsorból álló rendszer tervezését elkezdeni – melyet több száz ember fejlesztene és több ezer ember „tartana karban" és támogatna tíz vagy még több évig – egy becsületesen kidolgozott és meglehetősen szilárd váz nélkül.

Szerencsére a legtöbb program nem ebbe a kategóriába esik. Ha azonban egyszer elfogadtuk, hogy egy ilyen tervezési módszer vagy egy ilyen kódolási és dokumentálási szabványgyűjteményhez való ragaszkodás „a helyes út", kötelező lesz annak általános és minden részletre kiterjedő használata. Ez kis programoknál nevetséges korlátozásokhoz és többletmunkához vezethet: a haladás és siker mértékét jelentő hatékony munka helyett aktatologatás és űrlaptöltögetés folyik majd. Ha ez történik, az igazi tervezők és programozók távozni fognak és bürokraták lépnek a helyükbe.

Ha egy közösségen belül már előfordult, hogy egy tervezési módszert ilyen nevetségesen rosszul alkalmaztak, annak sikertelensége ürügyet szolgáltat arra, hogy elkerüljenek szinte minden szabályozást a fejlesztési folyamatban. Ez viszont éppen olyan zűrzavarhoz és balsikerekhez vezet, mint amilyeneket az új tervezési módszer kialakításával meg akartunk előzni.

Az igazi probléma megtalálni az egyensúlyt a megfelelő fokú szabályozás és az adott program fejlesztéséhez szükséges szabadság között. Ne higgyük, hogy erre a problémára könnyű lesz megoldást találni. Lényegében minden megközelítés csak kis projekteknél működik, pontosabban –ami még rosszabb, hiszen a rendszer rosszul tervezett és kegyetlen a belevont egyénekkel szemben – nagy projekteknél is, feltéve, hogy szemérmetlenül sok időt és pénzt akarunk elpocsékolni a problémára.

Minden programfejlesztői munka kulcsproblémája, hogyan tartsuk magunkat a terv eredeti szempontjaihoz. Ez a probléma a mérettel hatványozottan növekszik. Csak egy önálló programozó vagy egy kis csoport tudja szilárdan kézbentartani és betartani egy nagyobb munka átfogó célkitűzéseit. A legtöbb embernek olyan sok időt kell töltenie részprojektekkel, technikai részletkérdésekkel, napi adminisztrációval stb., hogy az átfogó célok könnyen feledésbe merülnek vagy alárendelődnek helyi és közvetlenebb céloknak. Az is sikertelenséghez vezet, ha nincs egy egyén vagy egy csoport, melynek kizárólag az a feladata, hogy fenntartsa a terv sértetlenségét. Recept a siker ellen, ha egy ilyen egyénnek vagy csoportnak nem engedjük, hogy a munka egészére hasson.

Egy ésszerű hosszú távú célkitűzés hiánya károsabb, mint bármilyen egyedi tulajdonság hiánya. Egy ilyen átfogó célkitűzés megfogalmazása, észben tartása, az átfogó tervdokumentáció megírása, a fő fogalmak ismertetése, és általában másokat segíteni, hogy észben tartsák az átfogó célt, kis számú egyén feladata kell, hogy legyen.

23.5.3. Egyének

Az itt leírtak szerinti tervezés jutalom a tehetséges tervezők és programozók számára, viszont a sikerhez elengedhetetlen, hogy ezeket a tervezőket és programozókat megtaláljuk.

A vezetők gyakran elfelejtik, hogy a szervezetek egyénekből állnak. Népszerű vélemény, hogy a programozók egyformák és csereszabatosak. Ez téveszme, mely tönkreteheti a közösséget, azáltal, hogy elzavarja a leghatékonyabb egyéneket és még a megmaradtakat is arra ítéli, hogy jóval a képességeik alatti szinten dolgozzanak. Az egyének csak akkor felcserélhetők, ha nem engedjük, hogy hasznosítsák azokat a képességeiket, melyek a kérdéses feladat által megkövetelt minimum fölé emelik őket. A csereszabatosság elve tehát nem humánus és eredendően pazarló.

A legtöbb programozási teljesítménymérés ösztönzi a pazarlást és kihagyja a számításból az egyéni hozzájárulást. A legkézenfekvőbb példa az a viszonylag elterjedt gyakorlat, hogy a haladást a megírt kódsorok, a dokumentáció lapjainak, vagy az elvégzett tesztek számával mérik. Az ilyen számok jól mutatnak diagramokon, de a valósághoz vajmi kevés közük van. Például, ha a termelékenységet a kódsorok számával mérjük, akkor egy összetevő sikeres újrahasznosítása negatív programozói teljesítménynek minősülhet. Egy nagy programrész újratervezésénél a legjobb elvek sikeres alkalmazása ugyanilyen hatású.

A teljesített munka minőségét sokkal nehezebb mérni, mint a „kimenet" mennyiségét, az egyéneket és csoportokat viszont a munka minősége alapján kell jutalmazni, nem durva mennyiségmérés alapján. Sajnos a minőség gyakorlati mérése – legjobb tudásom szerint – gyerekcipőben jár. Ezenkívül azon teljesítmény-előírások, melyek a munkának csak egy szakaszára vonatkoznak, hajlamosak befolyásolni a fejlesztést. Az emberek alkalmazkodnak a helyi határidőkhöz és az egyéni és csoportteljesítményt a kvóták által előírthoz igazítják. Ezt közvetlenül a rendszer átfogó épsége és teljesítménye szenvedi meg. Ha egy határidőt például az eltávolított vagy a megmaradt programhibákkal határoznak meg, beláthatjuk, hogy a határidő csak a futási idejű teljesítmény figyelmen kívül hagyásával vagy a rendszer futtatásához szükséges hardver-erőforrások optimalizálásának mellőzésével lesz tartható. Ellenben ha csak a futási idejű teljesítményt mérjük, a hibaarány biztosan növekedni fog, amikor a fejlesztők a rendszert a futási teljesítményt mérő programokra igyekeznek optimalizálni. Az értelmes minőségi előírások hiánya nagy követelményeket támaszt a ve-

zetők szakértelmével szemben, de az alternatíva az, hogy a haladás helyett egyes találomra kiválasztott tevékenységeket fognak jutalmazni. Ne feledjük, hogy a vezetők is emberek. Nekik is legalább annyi oktatásra van szükségük, mint az általuk vezetetteknek. Mint a programfejlesztés más területein, itt is hosszabb távra kell terveznünk. Lényegében lehetetlen megítélni egy egyén teljesítményét egyetlen év munkája alapján. A következetesen és hosszú távon vezetett feljegyzések azonban megbízható előrejelzést biztosítanak az egyes munkatársak teljesítményével kapcsolatban és hasznos segítséget adnak a közvetlen múlt értékeléséhez. Az ilyen feljegyzések figyelmen kívül hagyása – ahogy akkor történik, amikor az egyéneket pusztán cserélhető fogaskeréknek tekintik a szervezet gépezetében – a vezetőket kiszolgáltatja a félrevezető mennyiségi méréseknek.

A hosszú távra tervezés egyik következménye, hogy az egyének (mind a fejlesztők, mind a vezetők) az igényesebb és érdekesebb feladatokhoz hosszabb időt igényelnek. Ez elveszi a kedvét az állásváltoztatóknak éppúgy, mint a „karriercsinálás" miatt munkakört változtatóknak. Törekedni kell arra, hogy kicsi legyen a fluktuáció mind a műszakiak, mind a kulcsfontosságú vezetők körében. Egyetlen vezető sem lehet sikeres a kulcsemberekkel való egyetértés és friss, a tárgyra vonatkozó technikai tudás nélkül, de ugyanígy egyetlen tervezőkből és fejlesztőkből álló csoport sem lehet hosszú távon sikeres alkalmas vezetők támogatása nélkül és anélkül, hogy alapjaiban értsék azt a környezetet, melyben dolgoznak.

Ahol szükséges az innováció, az idősebb szakemberek, elemzők, tervezők, programozók stb. létfontosságú szerepet játszanak az új eljárások bevezetésében. Ők azok, akiknek új módszereket kell megtanulniuk és sok esetben el kell felejteniük a régi szokásokat, ami nem könnyű, hiszen általában komoly erőfeszítéseket tettek a régi munkamódszerek elsajátítására, ráadásul az e módszerekkel elért sikereikre, szakmai tekintélyükre támaszkodnak. Sok szakmai vezetővel ugyanez a helyzet.

Természetesen az ilyen egyének gyakran félnek a változástól. Ezért a változással járó problémákat túlbecsülik és nehezen ismerik el a régi módszerekkel kapcsolatos problémákat. Ugyanilyen természetes, hogy a változás mellett érvelők hajlamosak túlbecsülni az új módszerek előnyeit és alábecsülni a változással járó problémákat. A két csoportnak kommunikálnia ***kell***: meg ***kell*** tanulniuk ugyanazon a nyelven beszélni és segíteniük ***kell*** egymásnak, hogy az átmenetre megfelelő módszert dolgozhassanak ki. Ha ez nem történik meg, a szervezet megbénul és a legjobb képességű egyének távoznak mindkét csoportból. Mindkét csoportnak emlékeznie kell arra, hogy a legsikeresebb „öregek" gyakran azok, akik tavaly az „ifjú titánok" voltak. Ha adott az esély arra, hogy megalázkodás nélkül tanuljanak, a tapasztaltabb programozók és tervezők lehetnek a legsikeresebb és legnagyobb betekintéssel rendelkező hívei a változtatásnak. Egészséges szkepticizmusuk, a felhasználók ismerete és a szervezet működésével kapcsolatban szerzett tapasztalataik felbecsülhetetlenül értékesek lehetnek. Az azonnali és gyökeres változások javaslói észre kell vegyék, hogy az

átmenet, amely az új eljárások fokozatos elsajátításával jár, többnyire elengedhetetlen. Azoknak viszont, akik nem kívánnak változtatni, olyan területeket kell keresniük, ahol nem szükséges változtatni, nem pedig dühös hátvédharcot vívni olyan területeken, ahol az új követelmények már jelentősen megváltoztatták a siker feltételeit.

23.5.4. Hibrid tervezés

Új munkamódszerek bevezetése egy cégnél fáradságos lehet. Jelentős törést okozhat mind a szervezetben, mind az egyénekben. Egy váratlan változás, mely a „régi iskola" hatékony és tapasztalt tagjait egyik napról a másikra az „új iskola" zöldfülű újoncaivá változtatja, általában elfogadhatatlan. Változások nélkül azonban ritkán érhetünk el nagy nyereséget, a jelentős változások pedig többnyire kockázattal járnak.

A C++-t úgy tervezték, hogy a kockázatot a lehető legkisebbre csökkentse, azáltal, hogy lehetővé teszi a módszerek fokozatos elsajátítását. Bár világos, hogy a C++ használatának legnagyobb előnyei az elvont adatábrázolásból és az objektumorientált szemléletből adódnak, nem biztos, hogy e nyereségeket a leggyorsabban a múlttal való gyökeres szakítással lehet elérni. Egyszer-egyszer keresztülvihető az ilyen egyértelmű szakítás, de gyakoribb, hogy a javítás vágyát egy időre félre kell tennünk, hogy meggondoljuk, hogyan kezeljük az átmenetet. A következőket kell figyelembe vennünk:

- A tervezőknek és programozóknak idő kell az új szakismeretek megszerzéséhez.
- Az új kódnak együtt kell működnie a régi kóddal.
- A régi kódot karban kell tartani (gyakran a végtelenségig).
- A létező terveket és programokat be kell fejezni (időre).
- Az új eljárásokat támogató eszközöket be kell vezetni az adott környezetbe.

Ezek a tényezők természetesen hibrid (kevert) tervezési stílushoz vezetnek – még ott is, ahol némelyik tervezőnek nem ez a szándéka. Az első két pontot könnyű alulértékelni.

Azáltal, hogy számos programozási irányelvet támogat, a C++ változatos módon támogatja a nyelv használatának fokozatos bevezetését:

- A programozók produktívak maradhatnak a C++ tanulása közben.
- A C++ jelentős előnyöket nyújt egy eszközszegény környezetben.
- A C++ programrészek jól együttműködnek a C-ben vagy más hagyományos nyelven írt kóddal.
- A C++-nak jelentős „C-kompatibilis" részhalmaza van.

Az alapötlet az, hogy a programozók egy hagyományos nyelvről úgy térhetnek át a C++-ra, hogy a nyelvre áttérve először még megtartják a hagyományos (eljárásközpontú) programozási stílust, azután használni kezdik az elvont adatábrázolás módszereit, végül – amikor már elsajátították a nyelvet és a hozzá tartozó eszközök használatát – áttérnek az objektumorientált (object-oriented) és az általánosított programozásra (generic programming). Egy jól tervezett könyvtárat sokkal könnyebb használni, mint megtervezni és elkészíteni, így egy kezdő már az előrehaladás korai szakaszaiban is részesülhet az elvont ábrázolás használatának előnyeiből.

Az objektumorientált tervezést és programozást, valamint a C++ fokozatosan történő megtanulását támogatják azok a szolgáltatások, melyekkel a C++ kódot keverhetjük olyan nyelveken írt kóddal, melyek nem támogatják a C++ elvont adatábrázolási és objektumorientált programozási fogalmait (§24.2.1). Sok felület eljárásközpontú maradhat, mivel nincs közvetlen haszna, ha valamit bonyolultabbá teszünk. Sok kulcsfontosságú könyvtárnál az átültetést már elvégezte a könyvtár létrehozója, így a C++ programozónak nem kell tudnia, mi a tényleges megvalósítás nyelve. A C-ben vagy hasonló nyelven írt könyvtárak használata az újrahasznosítás elsődleges és kezdetben legfontosabb formája a C++-ban.

A következő lépés – melyet csak akkor kell elvégezni, amikor ténylegesen szükség van a kifinomultabb eljárásokra – a C, Fortran vagy hasonló nyelven írt szolgáltatások osztályok formájában való „tálalása", az adatszerkezetek és függvények C++ nyelvű felületosztályokba zárása által. Egy egyszerű példa a jelentés bővítésére az „eljárás és adatszerkezet" szintjéről az elvont adatábrázolás szintjére a §11.12 *string* osztálya. Itt a C karakterlánc-ábrázolását és szabványos karakterlánc-függvényeit használjuk fel egy sokkal egyszerűbben használható karakterlánc-típus létrehozására.

Hasonló módszer használható egy beépített vagy egyedi típusnak egy osztályhierarchiába illesztésére (§23.5.1). Ez lehetővé teszi, hogy a C++-ra készült terveket az elvont adatábrázolás és az osztályhierarchiák használatához továbbfejlesszük még olyan nyelveken írt kód jelenlétében is, ahonnan hiányoznak ezek a fogalmak, sőt azzal a megszorítással is, hogy az eredményül kapott kód eljárásközpontú nyelvekből is meghívható legyen.

23.6. Jegyzetek

Ez a fejezet csak érintette a programozás tervezési és vezetési kérdéseit A további tanulmányokat segítendő összegyűjtöttünk egy rövid irodalomjegyzéket. Részletesebbet [Booch, 1994] alatt találunk.

[Anderson, 1990] Bruce Anderson és Sanjiv Gossain: *An Iterative Design Model for Reusable Object-Oriented Software*. Proc. OOPSLA'90. Ottawa, Canada. Egy tervező és újratervező modell leírása, példákkal és a tapasztalatok tárgyalásával.

[Booch, 1994] Grady Booch: *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings. 1994. ISBN 0-8053-5340-2. Részletes leírást tartalmaz a tervezésről, és egy grafikai jelölésmóddal támogatott tervezési módról. Számos nagyobb tervezési példa áll rendelkezésre C++ nyelven. Kiváló könyv, melyből e fejezet is sokat merített. Az e fejezetben érintett kérdések jó részét nagyobb mélységben tárgyalja.

[Booch, 1996] Grady Booch: *Object Solutions*. Benjamin/Cummings. 1996. ISBN 0-8053-0594-7. Az objektumközpontú rendszerek fejlesztését a vezetés szemszögéből vizsgálja. Számos C++ példát tartalmaz.

[Brooks, 1982] Fred Brooks: *The Mythical man Month*. Addison-Wesley. 1982. Ezt a könyvet mindenkinek pár évente újra el kellene olvasnia! Intés a nagyképűség ellen. Technikailag kissé eljárt felette az idő, de az emberi, szervezeti és méretezési vonatkozásai időtállóak. 1997-ben kibővítve újra kiadták. ISBN 1-201-83595-9.

[Brooks, 1987] Fred Brooks: *No Silver Bullet*. IEEE Computer. Vol 20. No. 4. 1987 április. A nagybani szoftverfejlesztés megközelítéseinek összegzése, a régóta aktuális figyelmeztetéssel: nincsenek csodaszerek („nincs ezüstgolyó").

[Coplien, 1995] James O. Coplien és Douglas C. Schmidt (szerk.): *Pattern Languages of Program Design*. Addison-Wesley. 1995. ISBN 1-201-60734-4.

[De Marco, 1987] T. DeMarco és T. Lister: *Peopleware*. Dorset House Publishing Co. 1987. Azon ritka könyvek egyike, melyek az egyénnek a programfejlesztésben betöltött szerepével foglalkoznak. Vezetőknek kötelező, kellemes esti olvasmány, számos ostoba hiba ellenszerét tartalmazza.

[Gamma, 1994] Eric Gamma és mások: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2. Gyakorlati katalógus, mely rugalmas és újrahasznosítható programok készítési módszereit tartalmazza, bonyolultabb, jól kifejtett példákkal. Számos C++ példát tartalmaz.

[Jacobson, 1992] Ivar Jacobson és mások: *Object-Oriented Software Engineering*. Addison-Wesley. 1992. ISBN 0-201-54435-0. Alapos és gyakorlati leírás, amely használati esetek (use case, §23.4.3.1) alkalmazásával írja le a szoftverfejlesztést ipari környezetben. Félreérti a C++ nyelvet, 10 évvel ezelőtti állapotával leírva.

[Kerr, 1987] Ron Kerr: A *Materialistic View of the Software "Engineering" Analogy.* SIGPLAN Notices, 1987 március. Az ebben és a következő fejezetekben használt hasonlatok nagyban építenek e cikkre és a Ron által tartott bemutatókra, illetve vele folytatott beszélgetésekre.

[Liskov, 1987] Barbara Liskov: *Data Abstraction and Hierarchy.* Proc. OOPSLA'87 (Függelék). Orlando, Florida. Az öröklődés szerepe az elvont adatábrázolásban. Megjegyzendő, hogy a C++ számos eszközt biztosít a felvetett problémák többségének megoldására (§24.3.4).

[Martin, 1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method.* Prentice-Hall. 1995. ISBN 0-13-203837-4. Rendszerezetten mutatja be, hogyan jutunk el a problémától a C++ kódig. Lehetséges tervezési módokat és a köztük való döntés elveit ismerteti. A többi tervezéssel foglalkozó könyvnél gyakorlatiasabb és konkrétabb. Számos C++ példát tartalmaz.

[Meyer, 1988] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall. 1988. Az 1-64. és 323-334. oldalakon jó bevezetést ad az objektumközpontú programozás és tervezés egy nézetéről, számos használható gyakorlati tanáccsal. A könyv maradék része az Eiffel nyelvet írja le. Hajlamos az Eiffelt és az általános elveket összekeverni.

[Parkinson, 1957] C. N. Parkinson: *Parkinson's Law and other Studies in Administration.* Houghton Mifflin. Boston. 1957. Az egyik leghumorosabb és legpontosabb leírás, melyet a bürokráciáról írtak.

[Shlaer, 1988] S. Shlaer és S. J. Mellor: *Object-Oriented Systems Analysis és Object Lifecycles.* Yourdon Press. ISBN 0-13-629023-X és 0-13-629940-7. Az elemzés, tervezés és programozás egy olyan szemléletét mutatja be, mely jelentősen eltér az itt bemutatottól és a C++ által támogatottól.

[Snyder, 1986] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages.* Proc. OOPSLA'86. Portland, Oregon. Valószínűleg a betokozás (enkapszuláció) és öröklődés kapcsolatának első jó leírása. A többszörös öröklődést ugyancsak tárgyalja.

[Wirfs-Brock, 1990] Rebecca Wirfs-Brock, Brian Wilkerson és Lauren Wiener: *Designing Object-Oriented Software.* Prentice Hall. 1990. Szerepmintákon alapuló emberközpontú tervezési módszertant ír le CRC kártyák használatával. A szöveg (talán a módszertan is) részrehajló a Smalltalk irányában.

23.7. Tanácsok

[1] Legyünk tisztában vele, mit akarunk elérni. §23.3.
[2] Tartsuk észben, hogy a programfejlesztés emberi tevékenység. §23.2, §23.5.3.
[3] Hasonlat által bizonyítani ámítás. §23.2.
[4] Legyenek meghatározott, kézzelfogható céljaink. §23.4.
[5] Ne próbáljunk emberi problémákat technikai megoldásokkal orvosolni. §23.4.
[6] Tekintsünk hosszabb távra a tervezésben és az emberekkel való bánásmódban. §23.4.1, §23.5.3.
[7] Nincs méretbeli alsó határa azon programoknak, melyeknél értelme van a kódolás előtti tervezésnek. §23.2.
[8] Ösztönözzük a visszajelzést. §23.4.
[9] Ne cseréljük össze a tevékenységet a haladással. §23.3, §23.4.
[10] Ne általánosítsunk jobban, mint szükséges, mint amivel kapcsolatban közvetlen tapasztalataink vannak, és ami tesztelhető. §23.4.1, §23.4.2.
[11] Ábrázoljuk a fogalmakat osztályokként. §23.4.2, §23.4.3.1.
[12] A program bizonyos tulajdonságait nem szabad osztályként ábrázolni. §23.4.3.1.
[13] A fogalmak közötti hierarchikus kapcsolatokat ábrázoljuk osztályhierarchiákként. §23.4.3.1.
[14] Aktívan keressük a közös vonásokat az alkalmazás és a megvalósítás fogalmaiban és az eredményül kapott általánosabb fogalmakat ábrázoljuk bázisosztályokként. §23.4.3.1, §23.4.3.5.
[15] Máshol alkalmazott osztályozások nem szükségszerűen használhatóak egy program öröklési modelljében. §23.4.3.1.
[16] Az osztályhierarchiákat a viselkedés és a nem változó (invariáns) tulajdonságok alapján építsük fel. §23.4.3.1, §23.4.3.5, §24.3.7.1.
[17] Vizsgáljuk meg a használati eseteket. §23.4.3.1.
[18] Használjunk CRC kártyákat, ha szükséges. §23.4.3.1.
[19] Modellként, ösztönzésként és kiindulópontként használjunk létező rendszereket. §23.4.3.6.
[20] Óvakodjunk a rajzos tervezéstől. §23.4.3.1.
[21] Dobjuk el a prototípust, mielőtt teherré válik. §23.4.4.
[22] Számoljunk a változtatás lehetőségével, összpontosítsunk a rugalmasságra, a bővíthetőségre, a hordozhatóságra és az újrahasznosíthatóságra. §23.34.2.
[23] A középpontba az összetevők tervezését helyezzük. §23.4.3.
[24] A felületek az egyes fogalmakat egyetlen elvonatkoztatási szinten ábrázolják. §23.4.3.1.
[25] A változtatás küszöbén tartsuk szem előtt a stabilitást. §23.4.2.

[26] A gyakran használt felületek legyenek kicsik, általánosak és elvontak, hogy az eredeti terv lényegét ne kelljen módosítani. §23.4.3.2, §23.4.3.5..
[27] Törekedjünk a minimalizmusra. Ne használjunk „ha szükség lenne rá” tulajdonságokat. §23.4.3.2.
[28] Mindig vizsgáljuk meg egy osztály más lehetséges ábrázolásait. Ha nincs kézenfekvő alternatíva, az osztály valószínűleg nem képvisel tiszta fogalmat. §23.4.3.4.
[29] Többször is vizsgáljuk felül és finomítsuk mind a tervezést, mind a megvalósítás módját. §23.4, §23.4.3.
[30] Használjuk az elérhető legjobb eszközöket a teszteléshez és a probléma, a terv, illetve a megvalósítás elemzéséhez. §23.3, §23.4.1, §23.4.4.
[31] Kísérletezzünk, elemezzünk és teszteljünk a lehető leghamarabb és leggyakrabban. §23.4.4, §23.4.5.
[32] Ne feledkezzünk meg a hatékonyságról. §23.4.7.
[33] Teremtsünk egyensúlyt a formalitások szintje és a projekt mérete között. §23.5.2.
[34] Mindenképpen bízzunk meg valakit, aki az átfogó tervezésért felel. §23.5.2.
[35] Dokumentáljuk, népszerűsítsük és támogassuk az újrahasznosítható összetevőket. §23.5.1.
[36] Dokumentáljuk a célokat és elveket éppúgy, mint a részleteket. §23.4.6.
[37] Gondoskodjunk oktatóanyagról az új fejlesztők részére a dokumentáció részeként. $§23.4.6.
[38] Jutalmazzuk és ösztönözzük a tervek, könyvtárak és osztályok újrahasznosítását. §23.5.1.

24

Tervezés és programozás

„Legyen egyszerű: annyira egyszerű,
amennyire csak lehet – de ne egyszerűbb."
(A. Einstein)

A tervezés és a programozási nyelv • Osztályok • Öröklés • Típusellenőrzés • Programozás • Mit ábrázolnak az osztályok? • Osztályhierarchiák • Függőségek • Tartalmazás • Tartalmazás és öröklés • Tervezési kompromisszumok • Használati kapcsolatok • „Beprogramozott" kapcsolatok • Invariánsok • Hibaellenőrzés feltételezésekkel • Betokozás • Komponensek • Sablonok • Felület és megvalósítás • Tanácsok

24.1. Áttekintés

Ebben a fejezetben azt vizsgáljuk meg, hogy a programozási nyelvek – és maga a C++ – hogyan támogatják a tervezést.

§24.2 Az osztályok, osztályhierarchiák, a típusellenőrzés és maga a programozás alapvető szerepe

§24.3 Az osztályok és osztályhierarchiák használata, különös tekintettel a programrészek közötti függőségekre

§24.4 A *komponens* – mint a tervezés alapegysége – fogalma, és a felületek felépítésére vonatkozó gyakorlati megfigyelések

Az általánosabb tervezési kérdésekkel a 23. fejezetben foglalkoztunk, míg az osztályok különböző használati módjait részletesebben a 25. fejezet tárgyalja.

24.2. A tervezés és a programozási nyelv

Ha hidat szeretnénk építeni, figyelembe kellene vennünk az anyagot, amiből építjük. A híd tervezését erősen befolyásolná az anyag megválasztása és viszont. A kőhidakat másképp kell megtervezni, mint az acélhidakat vagy a fahidakat és így tovább. Nem lennénk képesek kiválasztani a hídhoz a megfelelő anyagot, ha nem tudnánk semmit a különböző anyagokról és használatukról. Természetesen nem kell ácsmesternek lenni ahhoz, hogy valaki fahidat tervezzen, de ismernie kell a faszerkezetek alapelveit, hogy választani tudjon, fából vagy vasból építsen-e hidat. Továbbá – bár nem kell valakinek személyesen ácsmesternek lennie egy fahíd tervezéséhez – kell, hogy részletesen ismerje a fa tulajdonságait és az ácsok szokásait.

Ehhez hasonlóan, ahhoz, hogy valamilyen programhoz nyelvet válasszunk, több nyelv ismerete szükséges, a program egyes részeinek sikeres megtervezéséhez pedig meglehetősen részletesen kell ismernünk a megvalósításhoz választott nyelvet – még akkor is, ha személyesen egyetlen kódsort sem írunk. A jó hídtervező figyelembe veszi az anyagok tulajdonságait és azok megfelelő felhasználásával növeli a terv értékét. A jó szoftvertervező ugyanígy a választott programozási nyelv erősségeire épít és – amennyire csak lehet – elkerüli annak olyan használatát, ami problémákat okozhat a kód íróinak.

Azt gondolhatnánk, hogy ez a nyelvi kérdések iránti érzékenység természetes, ha csak egyetlen tervezőt vagy programozót érint. Sajnos azonban még az önálló programozó is „kísértésbe" eshet, hogy a nyelvet – hiányos tapasztalata vagy gyökeresen eltérő nyelvekben kialakult programozási stílusa – miatt helytelenül használja. Amikor a tervező és a programozó nem azonos – és különösen ha szakmai és kulturális hátterük különböző –, szinte bizonyos, hogy a program hibás, körülményes vagy nem hatékony lesz.

Mit nyújthat tehát a programozási nyelv a tervezőnek? Olyan tulajdonságokat, melyek lehetővé teszik a terv alapfogalmainak közvetlen ábrázolását a programban. Ez megkönnyíti

a kód megírását, könnyebbé teszi a tervezés és megvalósítás közötti kölcsönös megfelelés fenntartását, javítja a tervezők és programozók közötti kapcsolattartást és jobb eszközök készítését teszi lehetővé mindkét csoport támogatására.

A legtöbb tervezési módszer a program különböző részei közti függőségekkel foglalkozik (rendszerint azért, hogy a lehető legkisebbre csökkentse számukat és biztosítsa, hogy a függőségek pontosan meghatározottak és átláthatóak legyenek). Egy nyelv, mely támogatja a programrészek kapcsolatát biztosító felületeket, képes támogatni az ezekre épülő tervezést is, illetve garantálni tudja, hogy ténylegesen csak az előre látott függőségek létezzenek. Mivel az ilyen nyelvekben sok függés közvetlenül a kódban is megjelenik, beszerezhetők olyan eszközök, melyek a programot olvasva függőségi diagramokat készítenek. Ez megkönnyíti a tervezők és azok dolgát, akiknek szükségük van a program szerkezetének megértésére. Egy olyan programozási nyelv, mint a C++ felhasználható a terv és a program közti szakadék kisebbítésére és a zavarok és félreértések körének következetes szűkítésére.

A C++ legfontosabb fogalma az osztály. A C++ osztályai típusok. A névterekkel együtt az osztályok is az adatrejtés elsődleges eszközei. A programok felhasználói típusok hierarchiáiként építhetők fel. Mind a beépített, mind a felhasználói típusok a statikusan ellenőrzött típusokra vonatkozó szabályoknak engedelmeskednek. A virtuális függvények ezen szabályok megsértése nélkül a futási idejű kötésről gondoskodnak. A sablonok a paraméterezett típusok tervezését támogatják, a kivételek pedig szabályozottabb hibakezelésre adnak módot. A C++ ezen szolgáltatásai anélkül használhatók, hogy többletterhet jelentenének a C programokhoz képest. Ezek a C++ azon elsőrendű tulajdonságai, melyeket a tervezőnek meg kell értenie és tekintetbe kell vennie. Ezenkívül a széles körben elérhető nagy programkönyvtárak – a mátrixkönyvtárak, adatbázis-felületek, a grafikus felhasználói felületek könyvtárai és a párhuzamosságot támogató könyvtárak – is erősen befolyásolhatják a tervezési döntéseket.

Az újdonságtól való félelem néha a C++ optimálisnál rosszabb felhasználásához vezet. A más nyelveknél, más rendszereken és alkalmazási területeken tanultak helytelen alkalmazása ugyanezt eredményezi. A gyenge tervezőeszközök szintén elronthatják a terveket. Íme öt a leggyakrabban elkövetett – a nyelvi tulajdonságok rossz kihasználását és korlátozások felrúgását eredményező – tervezői hibák közül:

1. Az osztályok figyelmen kívül hagyása és olyan tervezés, amely a programozókat arra kényszeríti, hogy csak a C részhalmazt használják.
2. A származtatott osztályok és virtuális függvények figyelmen kívül hagyása, csak az absztrakt adatábrázolási módszerek részhalmazának használata.
3. A statikus típusellenőrzés figyelmen kívül hagyása és olyan tervezés, amely a programozókat arra kényszeríti, hogy utánozzák a dinamikus típusellenőrzést.

4. A programozás figyelmen kívül hagyása és a rendszer olyan megtervezése, amely a programozók kiküszöbölését célozza.
5. Az osztályhierarchiák kivételével mindennek a figyelmen kívül hagyása.

Ezeket a hibákat általában a következő háttérrel rendelkező tervezők követik el:

1. akik korábban a C-vel, a hagyományos CASE eszközzel, vagy struktúrált tervezéssel foglalkoztak,
2. akik korábban Ada83, Visual Basic vagy más absztrakt ábrázolást támogató nyelven dolgoztak,
3. akik Smalltalk vagy Lisp múlttal rendelkeznek,
4. akik nem műszaki vagy nagyon speciális területen dolgoztak, és
5. akik olyan területről érkeztek, ahol erős hangsúlyt kapott a „tiszta" objektumorientált programozás.

Mindegyik esetben kételkednünk kell, vajon jól választották-e meg a megvalósítás nyelvét, a tervezési módszert, illetve hogy a tervező elsajátította-e a kezében lévő eszközök használatát.

Nincs semmi szokatlan vagy szégyellni való az ilyen problémákban. Ezek egyszerűen olyan hiányosságok, amelyek nem optimális terveket eredményeznek és a programozókra felesleges terheket hárítanak. A tervezők ugyanezekkel a problémákkal találják magukat szemben, ha a tervezési módszer fogalmi felépítése észrevehetően szegényesebb, mint a C++-é. Ezért ahol lehetséges, kerüljük az ilyen hibákat.

A következő fejtegetés ellenvetésekre adott válaszokból áll, mivel ez a valóságban is így szokott lenni.

24.2.1. Az osztályok figyelmen kívül hagyása

Vegyük azt a tervezést, amely figyelmen kívül hagyja az osztályokat. Az eredményül kapott C++ program nagyjából egyenértékű az ugyanezen tervezési folyamat eredményeként kapható C programmal – és ez a program ugyancsak nagyjából egyenértékű azzal a COBOL programmal, melyet ugyanezen tervezési folyamat eredményeként kapnánk. A tervezés lényegében „programozási nyelvtől függetlenül" folyt, a programozót arra kényszerítve, hogy a C és a COBOL közös részhalmazában kódoljon. Ennek a megközelítésnek vannak előnyei. Például az adat és a kód szigorú elkülönítése, ami könnyűvé teszi az ilyen programokhoz tervezett hagyományos adatbázisok használatát. Mivel egy „minimális" programozási nyelvet használunk, kevesebb tudást – vagy legalább is kevesebb féle tudást – követelünk

meg a programozóktól. Sok programnál – mondjuk egy hagyományos, szekvenciális adatbázist frissítőnél – ez a gondolkodásmód egészen ésszerű, az évtizedek alatt kifejlesztett hagyományos eljárások pedig megfelelőek a feladathoz.

Tegyük fel azonban, hogy a program a rekordokat (vagy karaktereket) a hagyományos szekvenciális feldolgozástól eltérően kezeli, vagy bonyolultabb – mondjuk, egy interaktív CASE rendszerről van szó. Az absztrakt adatábrázolás nyelvi támogatásának hiánya, amit az osztályok elhanyagolása melletti döntés okoz, fájó lesz. Az eredendő bonyolultság az alkalmazásban valahol meg fog mutatkozni, és ha a rendszert egy szegényes nyelven készítették, a kód nem fogja a tervet közvetlenül tükrözni. A program kódja túl hosszú lesz, hiányzik belőle a típusellenőrzés és általában nem megközelíthető segédeszközök számára. A program fenntartása és későbbi módosíthatósága szempontjából ez igazi rémálom.

A problémára általános megoldás, ha eszközöket készítünk a tervezési módszer fogalmainak támogatására. Ezek az eszközök magasabb szintű építkezést és ellenőrzést tesznek lehetővé, ami ellensúlyozza a (szándékosan legyengített) programozási nyelv gyengeségét. A tervezési módszer tehát egy egyedi célú (és általában testületi tulajdont képező) programozási nyelvvé válik. Az ilyen programozási nyelvek legtöbb esetben csak gyenge pótlékai a széles körben elérhető általános célú programozási nyelveknek, melyeket hozzájuk való tervezőeszközök támogatnak.

Az osztályok tervezésből való kihagyásának legáltalánosabb oka egyszerűen a tehetetlenség. A hagyományos programozási nyelvek nem támogatják az osztály fogalmát, a hagyományos tervezési módszerek pedig tükrözik ezt a gyengeséget. A tervezés legtöbbször a problémák eljárásokra bontására összpontosul, melyek a kívánt műveleteket hajtják végre. Ezt a 2. fejezetben eljárásközpontú (procedurális) programozásnak nevezett fogalmat a tervezéssel összefüggésben általában *funkcionális* (függvényekre vagy műveletekre való) *lebontásnak* (functional decomposition) nevezzük. Gyakori kérdés, hogy „tudjuk-e használni a C++-t egy funkcionális lebontáson alapuló tervezési módszerrel együtt?" A válasz igen, de a legvalószínűbb, hogy a C++-t végül egyszerűen csak mint egy jobb C-t fogjuk használni és a fentebb említett problémákkal fogunk kínlódni. Átmeneti időszakban, már befejezett tervezésnél, vagy olyan alrendszereknél, ahol (a bevont személyek tapasztalatát figyelembe véve) nem várható, hogy az osztályok jelentős előnnyel járnak, ez elfogadható. Hosszabb távon és általában azonban az osztályok használatának – a funkcionális lebontásból következő – ellenzése nem összeegyeztethető a C++ vagy bármely más, az absztrakt ábrázolást támogató nyelv hatékony használatával.

A programozás eljárásközpontú és objektumorientált szemléletei alapvetően különböznek és ugyanarra a problémára jellemzően gyökeresen különböző megoldásokat adnak. Ez

a megfigyelés éppúgy igaz a tervezési, mint a megvalósítási szakaszra: lehet összpontosítani az elvégzendő tevékenységekre és az ábrázolandó fogalmakra, de nem lehet egyszerre mindkettőre.

Miért részesítjük előnyben az „objektumorientált tervezést" a funkcionális lebontáson alapuló hagyományos tervezési módszerekkel szemben? Elsősorban azért, mert az utóbbi nem biztosít elegendő lehetőséget az absztrakt adatábrázolásra. Ebből pedig az következik, hogy az eredményül kapott terv

- kevésbé módosítható,
- eszközökkel kevésbé támogatható,
- kevésbé alkalmas párhuzamos fejlesztésre,
- kevésbé alkalmas párhuzamos végrehajtásra.

A probléma az, hogy a funkcionális lebontás következtében a lényeges adatok globálisak lesznek, mivel amikor egy rendszer függvényekből álló fa szerkezetű, bármely adat, melyre két függvénynek van szüksége, mindkettő számára elérhető kell, hogy legyen. Ez azt eredményezi, hogy az „érdekes" adatok egyre feljebb vándorolnak a fán a gyökér felé (ne feledjük, a számítástechnikában a fák mindig a gyökértől lefelé növekednek), ahogy egyre több függvény akar hozzájuk férni. Pontosan ugyanez a folyamat figyelhető meg az egygyökerű osztályhierarchiákban, melyekben az „érdekes" adatok és függvények hajlamosak felfelé vándorolni egy gyökérosztály felé (§24.4). A problémát úgy oldhatjuk meg, ha az osztályok meghatározására és az adatok „betokozására" (encapsulation) összpontosítunk, így ugyanis a programrészek közötti függéseket áttekinthetővé tehetjük, és – ami még fontosabb – csökkentjük a programban levő függőségek számát, azáltal, hogy az adatokra való hivatkozások lokálisak lesznek.

Egyes problémákat azonban a legjobban a megfelelő eljárások megírásával oldhatunk meg. Az objektumorientált megközelítésnél a tervezés lényege nem az, hogy egyetlen nem tag függvény se legyen a programban vagy hogy a rendszer egyetlen része se legyen eljárásközpontú. Lényegesebb, hogy a program különböző részeit úgy válasszuk el, hogy jobban tükrözzék a fogalmakat. Ez általában úgy érhető el a legjobban, ha elsősorban az osztályok és nem a függvények állnak a tervezés középpontjában. Az eljárásközpontú stílus használata tudatos döntés kell, hogy legyen, nem pedig az „alapértelmezés". Az osztályokat és eljárásokat az alkalmazásnak megfelelően kell használni, nem egy rugalmatlan tervezési módszer „melléktermékeiként".

24.2.2. Az öröklés elkerülése

Tegyük fel, hogy a tervezésnél nem építünk az öröklésre. Az eredményül kapott program nem fog élni a C++ egyik legelőnyösebb tulajdonságával, miközben persze kihasználja a C++ sok más előnyét a C, Pascal, Fortran, COBOL stb. nyelvekkel szemben. A leggyakrabban hangoztatott érvek – a „tehetetlenségtől" eltekintve – „az öröklés használata csak részletkérdés a megvalósítás során", „az öröklés megsérti az adatrejtés elvét" és „az öröklés megnehezíti az együttműködést más programokkal".

Az öröklést pusztán részletkérdésnek tekintve nem vesszük figyelembe azt, hogy az osztályhierarchiák közvetlenül ábrázolják az alkalmazási terület fogalmai közötti kapcsolatokat. Márpedig az ilyen kapcsolatokat nyilvánvalóvá kell tenni a tervezésben, hogy a tervezők vitatkozhassanak róluk.

Előfordulhat az is, hogy az öröklést olyan C++ programrészekből zárjuk ki, amelyek közvetlenül érintkeznek más nyelveken írott kóddal. Ez azonban *nem* elégséges ok arra, hogy a program egészében elkerüljük az öröklést, csupán a program „külvilág" felé mutatott felületét kell gondosan leírnunk és „betokoznunk". Hasonlóképpen, az adatrejtésnek az öröklés általi veszélyeztetése miatti aggályok (§24.3.2.1) csak arra adnak okot, hogy elővigyázatosak legyünk a virtuális függvények és védett tagok használatával (§15.3), az öröklődés általános elkerülésére nem.

Sok esetben nem származik valódi előny az öröklésből. A nagyobb programoknál azonban a „nincs öröklés" megközelítés kevésbé áttekinthető és rugalmatlanabb rendszert eredményez. Az öröklést ekkor csak „tettetjük", hagyományos nyelvi szerkezetek és tervezési módok használatával. Az is valószínű, hogy az ilyen hozzáállás ellenére az öröklést mégis használni fogjuk, mert a C++ programozók a program több részében is meggyőző érveket fognak találni az öröklés alapú tervezés mellett. Ezért a „nincs öröklés" csak azt fogja eredményezni, hogy a program felépítése nem lesz következetes és az osztályhierarchiák használata csak egyes alrendszerekre fog korlátozódni.

Más szóval, ne legyünk elfogultak. Az osztályhierarchiák nem minden jó program nélkülözhetetlen részei, de sok esetben segíteni tudnak mind az alkalmazás megértésében, mind egy megoldás kifejezésében. Az a tény, hogy az öröklést lehet helytelen vagy túlzott módon használni, ok az óvatosságra, de a tiltásra nem.

24.2.3. A statikus típusellenorzés figyelmen kívül hagyása

Vegyünk azt az esetet, amikor a tervezésnél elhanyagoljuk a statikus típusellenőrzést. Ezt általában a következőkkel indokolják: „a típusok a programozási nyelv termékei", „természetesebb objektumokban és nem típusokban gondolkodni" és „a statikus típusellenőrzés arra kényszerít, hogy túl korán gondoljunk a megvalósítás kérdéseire". Ez a hozzáállás addig jó, amíg nem okoz kárt. Tervezéskor ésszerűnek tűnhet nem foglalkozni a típusellenőrzés részleteivel, és az elemzési és a korai tervezési szakaszban általában nyugodtan figyelmen kívül is hagyhatjuk ezeket a kérdéseket. Az osztályok és osztályhierarchiák azonban nagyon hasznosak a tervezésben. Nevezetesen megengedik, hogy a fogalmakat ábrázolhassuk, kapcsolataikat meghatározhassuk, és segítenek, hogy a fogalmakról vitázzunk. A tervezés előrehaladtával ez a pontos ábrázolás az osztályokról és felületeikről tett egyre precízebb megállapítások formájában jelentkezik.

Fontos, hogy észrevegyük, hogy a pontosan meghatározott és erősen típusos (lényegében típusokra építő) felületek a tervezés alapvető eszközei. A C++ felépítése is ennek figyelembe vételével történt. Egy erősen típusos felület biztosítja (egy határig), hogy csak kompatibilis programrészeket lehessen együtt fordítani és összeszerkeszteni, ami lehetővé teszi, hogy ezek a programrészek egymásról viszonylag erős feltételezésekkel élhessenek. Ezeket a feltételezéseket a típusrendszer biztosítja; hatására csökkenteni lehet a futási idejű ellenőrzést, ezáltal nő a hatékonyság és jelentősen rövidül a többszemélyes projektek integrálási szakasza. Valójában az erősen típusos felületekről gondoskodó rendszerek integrálásában szerzett nagyon pozitív tapasztalatok okozzák, hogy az integrálás nem kap nagy teret e fejezetben.

Nézzünk egy hasonlatot. Gyakran kapcsolunk össze különböző szerkentyűket, a csatlakozó-szabványok száma pedig látszólag végtelen. A dugaszoknál kézenfekvő, hogy egyedi célra tervezettek, ami lehetetlenné teszi két szerkezet egymással való összekapcsolását, hacsak nem pont erre tervezték őket, ez esetben viszont csak a helyes módon kapcsolhatók össze. Nem lehet egy villanyborotvát egy nagyfeszültségű aljzatba bedugni. Ha lehetne, az eredmény vagy egy „sült villanyborotva" vagy égési sérülés lenne. A tervezők igen találékonynak bizonyultak, hogy biztosítsák, hogy az össze nem illő hardvereszközöket ne lehessen egymással összedugni. A nem megfelelő dugaszok ellen lehet olyan készülékeket készíteni, melyek az aljzataikba dugott készülékek nemkívánatos viselkedésével szemben megvédik magukat. Jó példa erre egy elektromos zavarvédő. Miután a dugaszok szintjén nem garantálható a teljes összeegyeztethetőség, alkalmanként szükségünk van drágább védőáramkörökre, melyek dinamikusan alkalmazkodnak a bemenethez vagy védelmet nyújtanak azzal szemben.

A hasonlat majdnem pontos. A statikus típusellenőrzés a dugasz megfelelőségének biztosításával egyenértékű, a dinamikus ellenőrzés pedig az alkalmazkodó/védő áramkörnek felel meg. Ha mindkét ellenőrzés hiányzik, az komoly kárt okozhat. Nagy rendszerekben mindkét ellenőrzési formát használják. A tervezés korai szakaszában ésszerű lehet egyszerűen kijelenteni, hogy „ezt a két készüléket össze kell dugni", hamarosan fontos lesz azonban, hogy pontosan megmondjuk, *hogyan* kell összedugni őket. Milyen garanciákat ad a dugasz a viselkedéssel kapcsolatban? Milyen körülmények között fordulhatnak elő hibák? Milyen költségekkel jár a megfelelő viselkedés biztosítása?

A statikus típusellenőrzés használata nem korlátozódik a „fizikai világra". A mértékegységek (pl. méter, kilogramm, másodperc) használata a fizikában és a mérnöki tudományokban az össze nem egyeztethető elemek összekeverését akadályozza meg.

Amikor a §23.4.3-ban a tervezés lépéseit ismertettük, a típusinformációk a 2. lépésben kerültek elő (az 1. lépésben rendszerint csak felületesen foglalkozunk velük), és a 4. lépésben váltak központi kérdéssé.

A statikusan ellenőrzött felületek a különböző programozói csoportok által fejlesztett C++ programok együttműködésének fő biztosítékai. Ezek dokumentációja (beleértve a használt típusokét is) az elsődleges kapcsolattartási eszköz az egyes programozói csoportok között. Ezen felületek jelentik a tervezési folyamat legfontosabb eredményét és ezek állnak a tervezők és programozók közötti kapcsolat középpontjában.

A típusok elhanyagolása a felületek kialakításánál olyan felépítéshez vezet, amely homályba burkolja a program szerkezetét és a futás idejéig elhalasztja a hibák észlelését. Tegyük fel például, hogy egy felületet önazonosító objektumokkal írunk le:

```
// a példa dinamikus típusellenőrzést feltételez statikus ellenőrzés helyett

Stack s; // a verem bármilyen típusú objektumra hivatkozó mutatókat tárolhat

void f()
{
  s.push(new Saab900);       // ez egy autótípus
  s.push(new Saab37B);       // ez egy repülőtípus

  s.pop()->takeoff();        // jó: a Saab 37B egy repülőgép
  s.pop()->takeoff();        // futási idejű hiba: egy autó nem tud felszállni
}
```

Ez a felület (a *Stack::push()* felületének) komoly túlegyszerűsítése, ami statikus ellenőrzés helyett dinamikus ellenőrzésre épít. Az *s* verem „repülőgépek” (*Plane*) tárolására való, de ez a kódban rejtett maradt, így a felhasználó kötelessége lesz e követelmény betartását biztosítani.

Egy precízebb meghatározás – egy sablon és egy virtuális függvény a megszorítás nélküli dinamikus típusellenőrzés helyett – a hibák észlelését a futási időből átteszi a fordítási időbe:

```
Stack<Plane*> s;            // a verem Plane-ekre hivatkozó mutatókat tárolhat

void f()
{
   s.push(new Saab900);     // hiba: a Saab900 nem Plane típusú
   s.push(new Saab37B);

   s.pop()->takeoff();      // rendben: a Saab 37B egy repülőgép
   s.pop()->takeoff();
}
```

Hasonló kérdést tárgyal a §16.2.2 pont. A különbség a futási idejű dinamikus ellenőrzés és a statikus ellenőrzés között jelentős lehet. A dinamikus ellenőrzés rendszerint 3-10-szer több feladatot ró a rendszerre. Nem szabad viszont a másik végletbe sem esni. Nem lehet statikus ellenőrzéssel minden hibát elcsípni. Még a legalaposabb statikusan ellenőrzött program is ki van téve a hardverhibák okozta sérülésnek. A §25.4.1 pontban további példát találhatunk arra, hogy nem lehet tökéletes statikus ellenőrzést megvalósítani, az ideális azonban az, ha a felületek nagy többsége alkalmazásszintű statikus típusokat használ (lásd §24.4.2).

Egy másik probléma, hogy a terv elvont szinten lehet tökéletesen ésszerű, de komoly problémákat okozhat, ha nem számol a használt eszköz (esetünkben a C++) korlátaival. Például, egy *f()* függvény, mely egy paraméterén a *turn_right()* műveletet hajtja végre, csak akkor hozható létre, ha minden paramétere ugyanolyan típusú:

```
class Plane {
   // ...
   void turn_right();
};

class Car {
   // ...
   void turn_right();
};
```

```
void f(X* p)          // milyen típus kell legyen X?
{
  p->turn_right();
  // ...
}
```

Egyes nyelvek (mint a Smalltalk és a CLOS) megengedik két ugyanazon műveletekkel rendelkező típus felcserélt használatát, azáltal, hogy minden típust egy közös bázisosztály által kapcsolnak össze és a futási időre halasztják a név feloldását. A C++ azonban ezt (szándékosan) csak sablonokkal (template) és fordítási idejű feloldással támogatja. Egy nem sablon függvény két különböző típusú paramétert csak akkor fogad el, ha a két típus automatikusan közös típusra konvertálható. Az előbbi példában tehát az *X*-nek a *Plane* és *Car* (Autó) közös bázisosztályának kell lennie (pl. a *Vehicle* (Jármű) osztálynak).

A C++-tól idegen fogalmakra épülő rendszerek természetesen ábrázolhatók a C++-ban is, ha a kapcsolatokra vonatkozó feltételezéseket kifejezetten megadjuk. A *Plane* és a *Car* például (közös bázisosztály nélkül is) osztályhierarchiába helyezhető, ami lehetővé teszi, hogy átadjunk egy *Car*-t vagy *Plane*-t tartalmazó objektumot *f(X*)*-nek (§25.4.1). Ha azonban ezt tesszük, az gyakran nemkívánatos mennyiségű műveletet és ügyességet követel, de a sablonok hasznos eszköznek bizonyulhatnak az ilyen leképezések egyszerűsítésére. A tervezési fogalmak és a C++ közti rossz megfeleltetés általában „természetellenes kinézetű” és kis hatékonyságú kódhoz vezet. A „karbantartó” programozók nem szeretik a nyelvben szokatlan kódot, amely ilyen rossz megfeleltetésekből származik.

A tervezési mód és a megvalósításhoz használt nyelv közti rossz megfeleltetés hasonlít a (természetes nyelvek esetében végzett) „szóról szóra” fordításhoz. Például az angol nyelv magyar nyelvtannal ugyanolyan nehézkes, mint a magyar nyelv angol nyelvtannal, annak pedig, aki csak a két nyelv egyikét beszéli folyékonyan, mindkét változat érthetetlen lehet.

A programban lévő osztályok a tervezés fogalmainak konkrét ábrázolásai. Következésképpen, ha az osztályok közötti kapcsolatok nem világosak, a terv alapfogalmai sem lesznek azok.

24.2.4. A programozás elkerülése

A programozás sok más tevékenységhez képest költséges és előre nehezen felmérhető munka, az eredményül kapott kód pedig gyakran nem 100%-ig megbízható. A programozás munkaigényes és – számos okból – a munkát általában az hátráltatja a legkomolyabban, ha egy kódrész nem átadásra kész. Nos, miért ne küszöböljük ki a programozást, mint tevékenységet, egészében véve?

Sok vezető számára jelentős előnnyel járna megszabadulni az arrogáns, túlfizetett, szakmailag megszállott, nem megfelelő öltözékű stb. programozóktól?[4]. Egy programozónak persze ez a javaslat abszurdnak hangzik. Vannak azonban olyan fontos területek, melyeknél a programozásnak vannak valós alternatívái. Egyes esetekben lehet közvetlenül egy magas szintű ábrázolásból létrehozni a kódot; másutt a képernyőn lévő alakzatok kezelésével. Közvetlen kezeléssel használható felhasználói felületeket lehet építeni annak az időnek tört része alatt, ami ugyanezen felületnek hagyományos kóddal való leírásához kellene. Ugyanígy kódot készíthetünk adatbázis-kapcsolatokhoz és az adatok ilyen kapcsolatok szerinti hozzáféréséhez pusztán azokból a specifikációkból, melyek sokkal egyszerűbbek, mint a műveletek közvetlen kifejezéséhez szükséges – C++-ban vagy más, általános célú programozási nyelven – írt kód. Ilyen leírásokból/meghatározásokból vagy egy közvetlen kezelőfelület segítségével állapotautomaták (state machines) készíthetők, melyek kisebbek, gyorsabbak és jobban működnek, mint amit a legtöbb programozó képes volna alkotni.

Ezek a módszerek olyan területeken használhatók jól, ahol erősek az elméleti alapok (pl. matematika, állapotautomaták, relációs adatbázisok) vagy van egy általános váz, amelybe be lehet ágyazni kis programtöredékeket (pl. grafikus felhasználói felületek, hálózatszimulációk, adatbázis-sémák). Az a tény, hogy ezen módszerek egyes lényeges területeken (bár ezek köre korlátozott) igen hasznosak lehetnek, elhitethetik velünk, hogy a hagyományos programozás kiváltása e módszerek segítségével „már a küszöbön áll". Nem így van: ha az ábrázolási módszerek az erős elméleti vázon túllépnek, a leíró nyelv szükségszerűen éppoly bonyolult lesz, mint egy általános célú programozási nyelv.

Néha elfelejtjük, hogy a váz, mely valamely területen lehetőséget ad a hagyományos programozás kiküszöbölésére, valójában egy hagyományos módon tervezett, programozott és tesztelt rendszer vagy könyvtár. A C++ és az e könyvben leírt eljárások egyik népszerű felhasználása is pontosan ilyen rendszerek tervezése és építése.

A legrosszabb eset, ha egy általános célú nyelv kifejező képességének csak a töredékét biztosító kompromisszumos megoldást az eredeti (korlátozott) alkalmazási területen kívül kell felhasználnunk. A tervezők, akik egy magas szintű modellezési szemponthoz ragaszkodnak, bosszankodnak a bonyolultság miatt és olyan rendszerleírást készítenek, melyből szörnyűséges kód jön létre, a közönséges programozási eljárásokat használó programozók pedig csalódottak lesznek a nyelvi támogatás hiánya miatt, és jobb kódot csak túlzott erőfeszítéssel és a magasszintű modellek elhagyásával lesznek képesek készíteni.

Nem látom jelét annak, hogy a programozás, mint tevékenység sikeresen kiküszöbölhető lenne az olyan területeken kívül, amelyeknek jól megalapozott elmélete van vagy amelyekben az alapvető programozási módszer egy vázhoz igazodik. A hatékonyság mindkét esetben drámai módon lecsökken, amint elhagyjuk az eredeti vázat és általánosabb célú mun-

[4] Igen. Én programozó vagyok.

kát kísérelünk meg elvégezni. Mást színlelni csábító, de veszélyes dolog. Ugyanakkor őrültség lenne figyelmen kívül hagyni a magas szintű leírásokat és a közvetlen kezelésre szolgáló eljárásokat olyan területeken, ahol azok jól megalapozottak és meglehetősen kiforrottak.

Az eszközök, könyvtárak és vázak tervezése a tervezés és programozás egyik legmagasabb rendű fajtája. Jól használható matematikai alapú modellt építeni egy alkalmazási területre az egyik legmagasabb rendű elemzésfajta. Adni egy eszközt, nyelvet, vázat stb., amely az ilyen munka eredményét ezrek számára teszi elérhetővé, módot ad a programozóknak és a tervezőknek elkerülni a csapdát, hogy tucattermékek készítőivé váljanak.

A legfontosabb, hogy az adott leíró rendszer vagy alapkönyvtár képes legyen felületként hatásosan együttműködni egy általános célú programozási nyelvvel. Egyébként az adott váz magában hordja korlátait. Ebből következik, hogy azon leíró vagy közvetlen kezelést biztosító rendszereknek, melyek megfelelően magas szintű kódot készítenek valamilyen elfogadott általános célú programozási nyelven, nagy előnyük van. Az egyedi nyelvek hosszú távon csak készítőiknek jelentenek könnyebbséget. Ha a létrehozott kód olyan alacsony szintű, hogy a hozzátett általános kódot az absztrakt ábrázolás előnyei nélkül kell megírni, elveszítjük a megbízhatóságot, a módosíthatóságot és a gazdaságosságot. Egy kódkészítő rendszert lényegében úgy kell megírni, hogy egyesítjük a magasabb szintű leírások és a magasabb szintű nyelvek erősségeit. Kihagyni az egyiket vagy a másikat annyi, mint feláldozni a rendszerépítők érdekeit az eszközkészítők érdekeiért. A sikeres nagy rendszerek többszintűek, modulárisak és folytonosan fejlődnek. Következésképpen az ilyen rendszerek megalkotását célzó sikeres erőfeszítésekbe sokféle nyelvet, könyvtárat, eszközt és módszert kell bevonni.

24.2.5. Az osztályhierarchiák kizárólagos használata

Amikor úgy találjuk, hogy egy újdonság tényleg működik, gyakran esünk túlzásba, és nyakra-főre azt alkalmazzuk. Más szóval, az egyes problémák esetében jó megoldásról gyakran hisszük, hogy gyógyírt jelenthet majdnem minden problémára. Az osztályhierarchiák és az objektumokon végzett többalakú (polimorf) műveletek sok problémára adnak jó megoldást, de nem minden fogalom ábrázolható a legjobban egy hierarchia részeként, és nem minden programkomponens legjobb ábrázolása egy osztályhierarchia.

Miért nem? Az osztályhierarchia kapcsolatokat fejez ki osztályai között, az osztály pedig egy fogalmat képvisel. Nos, akkor mi a közös kapcsolat egy mosoly, a CD-meghajtóm, Richard Strauss Don Juanjának egy felvétele, egy sor szöveg, egy műhold, az orvosi leleteim és egy valósidejű óra között? Ha az egészet egyetlen hierarchiába helyezzük, miközben egyetlen közös tulajdonságuk, hogy mindnyájan programozási elemek („objektumok”), csak kevés

értékkel bíró, zavaros rendszert hozunk létre (§15.4.5). Ha mindent egyetlen hierarchiába erőltetünk, mesterséges hasonlóságok jöhetnek létre és elhomályosíthatják a valódi egyezéseket. Hierarchiát csak akkor szabad használnunk, ha az elemzés fogalmi közösséget mutat ki, vagy ha a tervezés és programozás fed fel egyezéseket a fogalmak ábrázolására használt szerkezetekben. Az utóbbi esetben nagyon figyelnünk kell arra, hogy megkülönböztessük a valódi (altípusok által örökölt nyilvános tulajdonságban tükröződő) közösséget és a hasznos egyszerűsítéseket (ami privát öröklésben tükröződik, §24.3.2.1).

Ez a gondolatmenet olyan programhoz vezet, melyben számos egymással kapcsolatban nem lévő, vagy gyengén kapcsolódó osztályhierarchia van, és ezek mindegyike szorosan összekapcsolt fogalmak halmazát képviseli. Elvezet a konkrét osztály (§25.2) fogalmához is, mely nem hierarchia tagja, mert egy ilyen osztályt hierarchiába helyezve csorbítanánk az osztály teljesítőképességét és függetlenségét a rendszer többi részétől. A hatékonyság szemszögéből egy osztályhierarchia részét képező osztály leglényegesebb műveleteinek virtuális függvényeit kell tekintenünk, továbbá az osztály számos adatának privát (private) helyett védettnek (protected) kell lennie. Ez persze sebezhetővé teszi a további származtatott osztályok módosításaival szemben és komoly bonyodalmakat okozhat a tesztelésnél. Ott, ahol tervezési szempontból szigorúbb betokozást (enkapszulációt) érdemes használni, nem virtuális függvényeket és privát adatokat kell alkalmazni (§24.3.2.1).

Ha egy műveletnek egyetlen paramétere van (az, amelyik „az objektumot" jelöli), a terv általában torzul. Ha több paraméterrel egyformán lehet bánni, a művelet egy nem tag függvénnyel ábrázolható a legjobban. Ebből nem következik, hogy az ilyen függvényeket globálissá kell tennünk, csupán az, hogy majdnem minden ilyen önálló függvénynek egy névtér tagjának kell lennie (§24.4).

24.3. Osztályok

Az objektumorientált tervezés és programozás alapvető elve, hogy a program a valóság valamely részének modellje. A programban az osztályok a modellezett „valóság" alapfogalmait, míg ezen osztályok objektumai a valós világbeli objektumokat és a megvalósítás során létrehozott elemeket ábrázolják.

Az osztályok közti és egy osztály részein belüli kapcsolatok elemzése a rendszer tervezésében központi szerepet játszik:

§24.3.2 Öröklési kapcsolatok
§24.3.3 Tartalmazási kapcsolatok
§24.3.5 Használati kapcsolatok
§24.2.4 Beprogramozott kapcsolatok
§24.3.7 Osztályon belüli kapcsolatok

Mivel a C++ osztályai típusok, az osztályok és az osztályok közötti kapcsolatok jelentős támogatást kapnak a fordítóprogram részéről és általában a statikus elemzés alá tartoznak.

Ahhoz, hogy a rendszerben fontos legyen, egy osztálynak nemcsak használható fogalmat kell ábrázolnia; megfelelő felületet is kell nyújtania. Az ideális osztály alapvetően csekély mértékben, de pontosan meghatározottan függ a többi programelemtől, és olyan felületet nyújt, amely azok számára csak a feltétlenül szükséges információkat biztosítja (§24.4.2).

24.3.1. Mit ábrázolnak az osztályok?

Egy rendszerben lényegében kétfajta osztály van:

1. Osztályok, melyek közvetlenül az alkalmazási terület fogalmait ábrázolják; azaz olyan fogalmakat, melyeket a végfelhasználók használnak a problémák és megoldások ábrázolására.
2. Osztályok, melyek a megvalósításból következnek, vagyis olyan fogalmak, melyeket a tervezők és programozók a megvalósítási módszerek leírására használnak.

Néhány osztály, mely a megvalósítás „terméke", ábrázolhat valóságos dolgot is. A rendszer hardver- és szoftver-erőforrásai például alkalmasak arra, hogy egy programban osztályok legyenek. (Ez azt a tényt tükrözi, hogy egy rendszer több nézőpontból is tekinthető.) Ebből következik, hogy ami valaki számára csak a megvalósítás részletkérdése, az más számára a teljes alkalmazás lehet. Egy jól tervezett rendszer olyan osztályokat tartalmaz, melyek a rendszer logikailag önálló nézeteit ábrázolják:

1. Felhasználói szintű fogalmakat ábrázoló osztályok (pl. autók és teherautók)
2. Felhasználói fogalmak általánosításait ábrázoló osztályok (pl. járművek)
3. Hardver-erőforrásokat ábrázoló osztályok (pl. egy memóriakezelő osztály)
4. Rendszer-erőforrásokat ábrázoló osztályok (pl. kimeneti adatfolyamok)
5. Más osztályok megvalósítása használt osztályok (pl. listák, várakozási sorok, zárak)
6. Beépített adattípusok és vezérlési szerkezetek

Nagyobb rendszerekben kihívást jelent a logikailag önálló osztálytípusok elválasztása és a különböző elvonatkoztatási (fogalmi) szintek közötti elkülönítés fenntartása. Vegyünk egy egyszerű példát, ahol három fogalmi szintünk van:

1+2	A rendszer alkalmazásszintű nézete
3+4	Annak a gépnek az ábrázolása, amelyen a „modell" fut
5+6	A megvalósítás alacsonyszintű (programozási nyelvi) nézete

Minél nagyobb a rendszer, annál több fogalmi szintre van szükség annak leírásához és annál nehezebb a szinteket meghatározni és fenntartani. Vegyük észre, hogy az ilyen fogalmi szinteknek közvetlen megfelelői vannak mind a természetben, mind a más típusú, ember által létrehozott rendszerekben. Egy ház például úgy is tekinthető, mint ami az alábbiakból áll:

1. atomok,
2. molekulák,
3. faanyag és tégla,
4. padló, falak és mennyezet,
5. szobák.

Mindaddig, amíg ezek a szintek külön maradnak, fenntartható a ház fogalmának következetes megközelítése, ha azonban keverjük őket, abszurditások keletkeznek. Például az a kijelentés, hogy „Az én házam többezer kiló szénből, komplex polimerből, kb. 5000 téglából, két fürdőszobából és 13 mennyezetből áll", ostobaság. A programok absztrakt természetéből adódik, hogy ha egy bonyolult programrendszerről hasonló kijelentést teszünk, azt nem mindig lehet ilyen egyszerűen minősíteni.

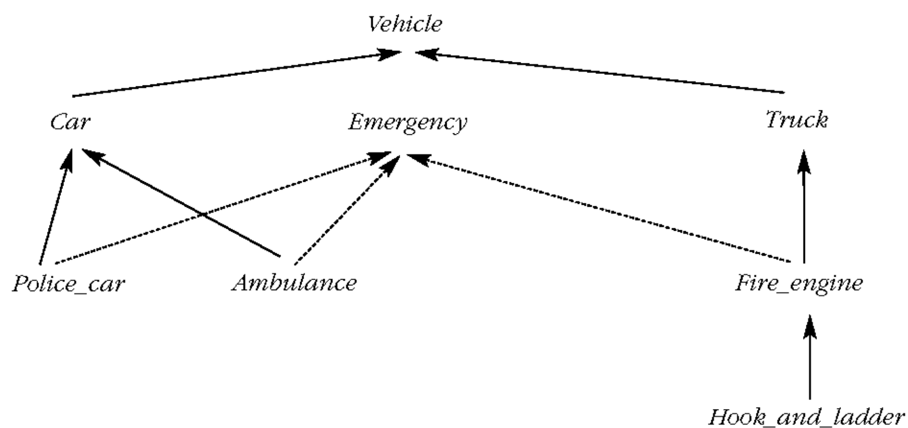
Az alkalmazási terület valamely fogalmának „lefordítása" egy tervezési osztályra nem egyszerű, mechanikus művelet, gyakran jelentős áttekintést követel. Vegyük észre, hogy magukat az adott alkalmazási terület fogalmait is elvonatkoztatás segítségével írjuk le. Például „adófizetők", „szerzetesek" és „alkalmazottak" a természetben nem léteznek; az ilyen fogalmak csupán címkék, melyeket egyénekre aggatunk, hogy valamely rendszer szerint osztályozzuk őket. A valós, sőt a képzelt világból (az irodalomból, különösen a tudományos fantasztikumból) merített fogalmak gyökeresen megváltoznak, amikor osztályokkal ábrázoljuk azokat. A PC képernyője például nem igazán emlékeztet az íróasztalra, sokszor mégis ezzel a hasonlattal írják le?[5], a képernyőn lévő ablakok pedig csak halványan emlékeztetnek azokra a szerkezetekre, melyek huzatot engednek be az irodába. A valóság modellezésénél nem az a lényeg, hogy szolgai módon kövessük azt, amit látunk, hanem hogy a tervezés kiinduló pontjaként, ösztönző forrásként, és horgonyként használjuk azt, melybe kapaszkodhatunk, amikor a program megfoghatatlan természete azzal fenyeget, hogy legyőzi azt a képességünket, hogy megértsük saját programjainkat.

5 Én semmiképpen sem tűrnék akkora rendetlenséget a képernyőmön.

Egy figyelmeztetés: a kezdők gyakran nehezen „találják meg az osztályokat”, de ezt a problémát rendszerint hamarosan leküzdik, „maradandó” káros hatások nélkül. Ezt azonban gyakran követi egy szakasz, amelyben az osztályok – és azok öröklési kapcsolatai – látszólag ellenőrizhetetlenül megsokszorozódnak, ami hosszú távon viszont bonyolultabbá és áttekinthetetlenebbé teheti az eredményül kapott programot és ronthatja annak hatékonyságát. Nem kell minden kis részletet külön osztállyal és minden osztályok közötti kapcsolatot öröklési kapcsolattal ábrázolni. Próbáljuk észben tartani, hogy a tervezés célja a rendszer megfelelő részletességgel és megfelelő elvonatkoztatási szinten való modellezése. Az egyszerűség és az általánosság között nem könnyű az egyensúlyt megtalálni.

24.3.2. Osztályhierarchiák

Vegyük egy város forgalmának szimulációját: meg kell határoznunk, várhatóan mennyi időre van szükség, hogy a mentőjárművek rendeltetési helyükre érjenek. Világos, hogy ábrázolnunk kell autókat, teherautókat, mentőautókat, különféle tűzoltójárműveket, rendőrautókat, buszokat és így tovább. Az öröklés mindenképpen szerephez jut, mivel a valós világbeli fogalmak nem léteznek elszigetelten, csak más fogalmakkal kapcsolatban. A kapcsolatok megértése nélkül nem érthetjük meg a fogalmakat sem. Következésképpen az a modell, amely nem ábrázol ilyen kapcsolatokat, nem ábrázolja megfelelően a fogalmakat sem. Programjainkban tehát szükségünk van osztályokra a fogalmak ábrázolásához, de ez nem elég; szükségünk van az osztályok kapcsolatainak ábrázolására is. Az öröklés kitűnő módszer hierarchikus kapcsolatok közvetlen ábrázolására. Példánkban a mentőjárműveket valószínűleg különlegesnek tekintenénk és megkülönböztetnénk „autószerű” és „teherautószerű” járműveket is. Az osztályhierarchia ezek alapján így nézne ki:

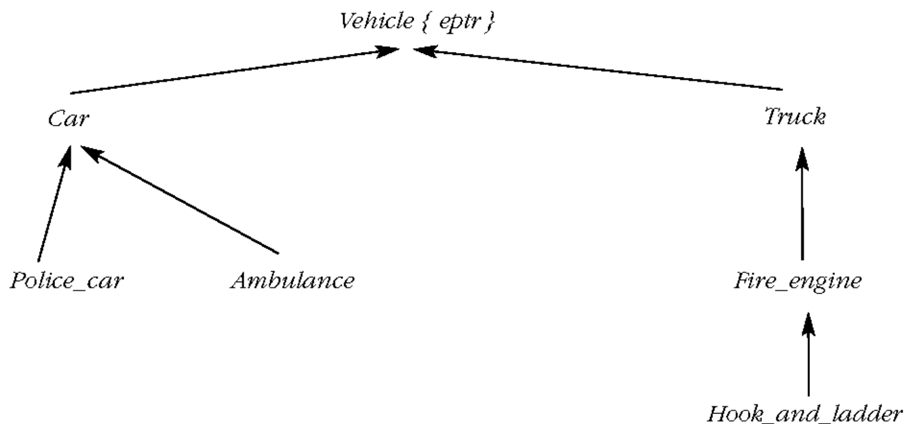


Itt az *Emergency* a megkülönböztetett járművek azon jellemzőit képviseli, melyek a szimuláció szempontjából lényegesek: megsérthet bizonyos forgalmi szabályokat, elsőbbsége van az útkereszteződésekben, diszpécser irányítja stb.

Íme a C++ változat:

```
class Vehicle { /* ... */ };                                              // Jármű
class Emergency { /* ... */ };                                            // Megkülönböztetett
class Car : public Vehicle { /* ... */ };                                 // Autó
class Truck : public Vehicle { /* ... */ };                               // Teherautó
class Police_car : public Car , protected Emergency { /* ... */ };        // Rendőrautó
class Ambulance : public Car , protected Emergency { /* ... */ };         // Mentőautó
class Fire_engine : public Truck , protected Emergency { /* ... */ };     // Tűzoltóautó
class Hook_and_ladder : public Fire_engine { /* ... */ };                 // Létrásautó
```

Az öröklés a C++-ban közvetlenül ábrázolható legmagasabb szintű kapcsolat, és a tervezés korai szakaszában a legnagyobb a szerepe. Gyakran választhatunk, hogy öröklést vagy tagságot használunk-e egy kapcsolat ábrázolására. Nézzünk egy másik megközelítést, mit is jelent megkülönböztetett járműnek lenni: egy jármű megkülönböztetett, ha villogó fényjelzője van. Ez lehetővé tenné, hogy úgy egyszerűsítsük az osztályhierarchiát, hogy az *Emergency* osztályt a *Vehicle* osztály egyik tagjával helyettesítjük:



Az *Emergency* osztályt most egyszerűen azon osztályok tagjaként használjuk, melyeknek szükségük lehet arra, hogy megkülönböztetett járműként szerepeljenek:

```
class Emergency { /* ... */ };
class Vehicle { protected: Emergency* eptr; /* ... */ }; // jobb: helyes hozzáférést biztosít eptr-
hez
class Car : public Vehicle { /* ... */ };
class Truck : public Vehicle { /* ... */ };
class Police_car : public Car { /* ... */ };
class Ambulance : public Car { /* ... */ };
class Fire_engine : public Truck { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };
```

Itt egy jármű akkor megkülönböztetett, ha a *Vehicle::eptr* nem nulla. A „sima" autóknál és teherautóknál a *Vehicle::eptr* kezdőértéke nulla; a többinél nem nulla:

```
Car::Car()                    // Car konstruktor
{
  eptr = 0;
}

Police_car::Police_car()      // Police_car konstruktor
{
  eptr = new Emergency;
}
```

Az elemek ilyen meghatározása lehetővé teszi, hogy egy megkülönböztetett járművet közönséges járművé alakítsunk át és megfordítva:

```
void f(Vehicle* p)
{
  delete p->eptr;
  p->eptr = 0;                    // Többé nem megkülönböztetett jármű

  // ...

  p->eptr = new Emergency;        // Ismét megkülönböztetett jármű
}
```

Nos, melyik jobb osztályhierarchia? Az általános válasz: „Az a program, amely a valós világ bennünket legjobban érdeklő részét a legközvetlenebb módon modellezi." Vagyis a modellek közti választáskor a cél a valóság minél jobb megközelítése, persze a hatékonyságra és egyszerűségre való törekvés mellett. Esetünkben a könnyű „átváltozás" a közönséges és megkülönböztetett járművek között számomra nem tűnik valószerűnek. A tűzoltóautók és a mentőautók különleges célra készült járművek, kiképzett személyzettel; eligazításuk pedig egyedi kommunikációs berendezéseket igényel. Ez a szemlélet jelzi, hogy megkülön-

böztetett járműnek lenni alapvető fogalom és a programban közvetlenül kell ábrázolni, hogy segítse a típusellenőrzést és más eszközök használatát. Ha olyan helyet modelleznénk, ahol a járművek szerepe kevésbé szigorúan meghatározott – mondjuk egy olyan területet, ahol magánjárműveket szokás a mentőszemélyzet helyszínre szállítására használni és ahol a kommunikáció elsősorban hordozható rádiókon keresztül folyik, más modellezési módszer megfelelőbb lenne. Azok számára, akik a forgalomszimulációt elvontnak tekintik, érdemes lehet rámutatni, hogy az öröklődés és a tagság közti választás szinte minden esetben elkerülhetetlen (lásd a §24.3.3 gördítősáv példáját).

24.3.2.1. Osztályhierarchián belüli függések

A származtatott osztályok természetesen függnek bázisosztályaiktól. Ritkábban esik szó róla, de az ellenkezője is igaz lehet?[6]. Ha egy osztálynak van virtuális függvénye, az osztály függ a belőle származtatott osztályoktól, melyek e függvény felülbírálásával az osztály egyes szolgáltatásait biztosítják. Ha magának a bázisosztálynak egy tagja hívja meg az osztály egyik virtuális függvényét, megint csak a bázisosztály függ a származtatott osztályaitól, saját megvalósítása miatt. Vegyük az alábbi példát:

```
class B {
  // ...
protected:
  int a;
public:
  virtual int f();
  int g() { int x = f(); return x-a; }
};
```

Mit csinál *g()* ? A választ jelentősen befolyásolja, definiálja *f()*-et valamelyik származtatott osztály. Íme egy változat, mely biztosítja, hogy *g()* visszatérési értéke *1* lesz:

```
class D1 : public B {
  int f() { return a+1; }
};
```

Íme egy másik, amely kiírja a *„Helló, világ!"* szöveget és nullával tér vissza:

```
class D2 : public B {
  int f() { cout<<"Helló, világ!\n"; return a; }
};
```

[6] Ez a megfigyelés így összegezhető: „Az esztelenség örökölhető. A gyerekeidtől kapod meg."

A fentiek a virtuális függvényekkel kapcsolatos egyik legfontosabb dolgot szemléltetik. Miért rossz a példa? Miért nem írna programozó soha ilyet? Azért, mert a virtuális függvények a bázisosztály felületének részei, és az osztály feltehetően a belőle származó osztályok ismerete nélkül is használható. Következésképpen úgy kell tudnunk meghatározni a bázisosztály egy objektumának elvárt viselkedését, hogy a származtatott osztályok ismerete nélkül is írhassunk programokat. Minden osztály, mely felülírja (override) a virtuális függvényt, e viselkedésnek egy változatát kell, hogy leírja. Például a *Shape* (Alak) osztály *rotate()* (forgatás) virtuális függvénye egy alakzatot forgat el. A származtatott *Circle* (Kör) és *Triangle* (Háromszög) osztályok *rotate()* függvényeinek a nekik megfelelő típusú objektumokat kell forgatniuk, különben megsértenének egy alapvető feltételezést a *Shape* osztályról. A fenti *B* osztályról és a belőle származó *D1* és *D2* osztályokról ilyesmit nem tételeztünk fel, ezért a példa értelmetlen. Még a *B, D1, D2, f* és *g* neveket is úgy választottuk meg, hogy bármilyen lehetséges jelentést homályban hagyjanak. A virtuális függvények elvárt viselkedésének meghatározása az osztályok tervezésének egyik *fő* szempontja. Jó neveket választani az osztályok és függvények számára szintén fontos – de nem mindig könnyű.

Jó vagy rossz-e egy függés az ismeretlen (esetleg még meg sem írt) származtatott osztályoktól? Természetesen ez függ a programozó szándékától. Ha egy osztályt úgy akar elszigetelni minden külső befolyástól, hogy az meghatározott módon viselkedjen, akkor legjobb elkerülni a védett (protected) tagokat és a virtuális függvényeket. Ha azonban egy vázat akar adni, amelyhez egy későbbi programozó (vagy saját maga néhány héttel később) kódot adhat hozzá, a virtuális függvények használata ehhez elegáns módszert jelenthet, a *protected* tagfüggvények pedig jól támogatják az ilyen megoldásokat. Ezt a megoldást választottuk az adatfolyam I/O könyvtárban (§21.6), az *Ival_box* hierarchia végső változatánál (§12.4.2) pedig példát is mutattunk rá.

Ha egy *virtuális* függvényt csak a származtatott osztályok általi közvetett használatra szánunk, *private* maradhat. Példaként vegyük egy átmeneti tár (puffer) egyszerű sablonját:

```
template<class T> class Buffer {
public:
   void put(T);        // overflow(T) meghívása, ha az átmeneti tár megtelt
   T get();            // underflow() meghívása, ha a tár üres
   // ...
private:
   virtual int overflow(T);
   virtual int underflow();
   // ...
};
```

A *put()* és *get()* függvények a virtuális *overflow()*, illetve *underflow()* függvényeket hívják meg. A felhasználó ezen függvények felülírásával a különböző igények szerint egy sor tártípust határozhat meg:

```
template<class T> class Circular_buffer : public Buffer<T> {
  int overflow(T);     // körbelép, ha teli
  int underflow();
  // ...
};

template<class T> class Expanding_buffer : public Buffer<T> {
  int overflow(T);     // megnöveli az átmeneti tárat, ha megtelt
  int underflow();
  // ...
};
```

Az *overflow()* és *underflow()* függvényeknek csak akkor kellene *private* helyett *protected*-nek lenniük, ha a származtatott osztálynak szüksége lenne e függvények közvetlen meghívására.

24.3.3. Tartalmazási kapcsolatok

Ott, ahol tartalmazást használunk, egy *X* osztály egy objektumát két fő módszerrel ábrázolhatjuk:

1. Bevezetünk egy *X* típusú tagot.
2. Bevezetünk egy *X** típusú vagy egy *X&* típusú tagot.

Ha a mutató értéke sohasem változik, ezek a megoldások egyenértékűek, kivéve a hatékonyság kérdéseit és azt a módot, ahogyan konstruktorokat és destruktorokat írunk:

```
class X {
public:
  X(int);
  // ...
};

class C {
  X a;
  X* p;
  X& r;
public:
```

```
    C(int i, int j, int k) : a(i), p(new X(j)), r(*new X(k)) { }
    ~C() { delete p; delete &r; }
};
```

Ilyen esetekben rendszerint előnyben kell részesítenünk magának az objektumnak a tagságát (*C::a*), mert ez biztosítja a leggyorsabb működést, ez igényli a legkevesebb helyet, és persze ezt lehet a leggyorsabban leírni. Kevesebb hiba forrása is, mivel a tartalmazó és a tartalmazott objektum kapcsolata a létrehozás és megsemmisítés szabálya alá tartozik (§10.4.1, §12.2.2, §14.4.1, de lásd még: §24.4.2 és §25.7).

A mutatót használó megoldást akkor használjuk, ha a „tartalmazó” objektum élettartama alatt a „tartalmazott” objektumra hivatkozó mutatót meg kell változtatnunk:

```
class C2 {
    X* p;
public:
    C2(int i) : p(new X(i)) { }
    ~C2() { delete p; }

    X* change(X* q)
    {
            X* t = p;
            p = q;
            return t;
    }
};
```

Egy másik ok a mutató tag használatára, hogy meg akarjuk engedni a „tartalmazott” objektum paraméterként való szereplését:

```
class C3 {
    X* p;
public:
    C3(X* q) : p(q) { }
    // ...
};
```

Azáltal, hogy olyan objektumaink vannak, melyek más objektumokra hivatkozó mutatókat tartalmaznak, tulajdonképpen egy objektumhierarchiát hoztunk létre. Ez az osztályhierarchiák használatát egyszerre helyettesítheti és kiegészítheti. Mint a §24.3.2 „jármű” példája mutatta, gyakran nehéz tervezéskor választani, hogy egy osztálytulajdonságot bázisosztályként vagy tagként ábrázoljunk. Ha felülírt függvényeket kell használnunk, az azt jelzi, hogy az első a jobb választás, ha pedig arra van szükség, hogy egy tulajdonságot több típussal ábrázolhassunk, valószínűleg célszerűbb a második megoldás mellett dönteni:

```
class XX : public X { /* ... */ };

class XXX : public X { /* ... */ };

void f()
{
   C3* p1 = new C3(new X);          // C3 "tartalmaz" egy X objektumot
   C3* p2 = new C3(new XX);         // C3 "tartalmaz" egy XX objektumot
   C3* p3 = new C3(new XXX);        // C3 "tartalmaz" egy XXX objektumot
   // ...
}
```

Itt nem lenne megfelelő ábrázolás, ha *C3*-at *X*-ből származtatnánk vagy ha *C3*-nak egy *X* típusú tagja lenne, mivel a tag pontos típusát kell használni. Ez a virtuális függvényekkel rendelkező osztályoknál fontos, például egy alakzatosztálynál (§2.6.2) vagy egy absztrakt halmazosztálynál (§25.3).

A mutató tagokra épülő osztályok egyszerűsítésére azokban az esetekben, amikor a tartalmazó objektum élettartama alatt csak egyetlen objektumra hivatkozunk, referenciákat használhatunk:

```
class C4 {
   X& r;
public:
   C4(X& q) : r(q) { }
   // ...
};
```

Akkor is szükség van mutató és referencia típusú tagokra, amikor egy objektumon osztozni kell:

```
X* p = new XX;
C4 obj1(*p);
C4 obj2(*p);          // obj1 és obj2 most osztoznak az új XX objektumon
```

Természetesen a közösen használt objektumok kezelése külön óvatosságot kíván, főleg a párhuzamos feldolgozást támogató rendszerekben.

24.3.4. Tartalmazás és öröklés

Az öröklési kapcsolatok fontosságának ismeretében nem meglepő, hogy ezeket a kapcsolatokat gyakran túlzottan használják vagy félreértik. Ha egy *D* osztály nyilvános és a *B* osztályból származtatott, gyakran azt mondjuk, hogy egy *D* valójában egy *B* (*D is a B*):

```
class B { /* ... */ };
class D : public B { /* ... */ }; // D is a kind of B: D egyfajta B
```

Úgy is kifejezhetjük, hogy az öröklés egy *is-a* kapcsolat, vagy – valamivel precízebben – hogy egy *D* valójában egyfajta *B* (D is a kind of B). Ezzel szemben ha a *D* osztály valamelyik tagja egy *B* osztály, azt mondjuk, hogy van (have) egy B-je, illetve tartalmaz (contain) egy *B*-t:

```
class D {    // D tartalmaz egy B-t
public:
   B b;
   // ...
};
```

Másképpen ezt úgy fejezzük ki, hogy a tagság egy *has-a* kapcsolat.

Adott *B* és *D* osztályok esetében hogyan válasszunk öröklés és tagság között? Vegyünk egy *Repülőgép*-et és egy *Motor*-t. A kezdők gyakran kíváncsiak, jó ötlet-e egy *Repülőgép* osztályt a *Motor*-ból származtatni. Rossz ötlet, mert bár a repülőgépnek *van* motorja, a repülőgép *nem* motor. Vegyük figyelembe, hogy egy repülőgépnek kettő vagy több motorja is lehet. Mivel ésszerűnek látszik – még akkor is, ha az adott programban minden *Repülőgép* egymotoros –, használjunk öröklés helyett tagságot. A „lehet-e neki kettő?" kérdés sok esetben hasznos lehet, ha kétség merül fel. Mint rendszerint, a programok megfoghatatlan természete az, ami fontossá teszi ezt a vizsgálatot. Ha minden osztályt olyan könnyen elképzelhetnénk, mint a *Repülőgép*-et és a *Motor*-t, könnyen elkerülhetnénk az olyan nyilvánvaló tévedéseket, mint egy *Repülőgép* származtatása egy *Motor*-ból. Az ilyen tévedések azonban igen gyakoriak – különösen azoknál, akik a származtatást egyszerűen egy olyan eljárásnak tekintik, mellyel programozási nyelvi szintű szerkezeteket lehet együtt használni. Annak ellenére, hogy az öröklés használata kényelmes és a kódot is rövidíti, majdnem kivétel nélkül olyan kapcsolatok kifejezésére használjuk, melyeket a tervezésnél pontosan meghatároztunk. Vegyük a következőt:

```
class B {
public:
   virtual void f();
   void g();
};

class D1 {             // D1 tartalmaz egy B-t
public:
   B b;
   void f();           // nem írja felül a b.f() virtuális függvényt
};
```

```
void h1(D1* pd)
{
  B* pb = pd;          // hiba: nincs konverzió D1* -ról B* -ra
  pb = &pd->b;
  pb->g();             // B::g() meghívása
  pd->g();             // hiba: D1-nek nincs g() tagja
  pd->b.g();
  pb->f();             // B::f() meghívása (D1::f() nem írta felül)
  pd->f();             // D1::f() meghívása
}
```

Vegyük észre, hogy egy osztály nem konvertálható automatikusan (implicit módon) saját tagjává, és egy osztály, mely egy másik osztály egy tagját tartalmazza, nem írja felül a tag virtuális függvényeit. Ez ellentétes a nyilvános származtatással:

```
class D2 : public B {          // D2 egy B
public:
  void f();                    // felülírja a B::f() virtuális függvényt
};

void h2(D2* pd)
{
  B* pb = pd;          // rendben: automatikus konverzió D2* -ról B* -ra
  pb->g();             // B::g() meghívása
  pd->g();             // B::g() meghívása
  pb->f();             // virtuális hívás: D2::f() meghívása
  pd->f();             // D2::f() meghívása
}
```

A *D2* példa a *D1* példához képest kényelmesebb jelölést biztosít, és ez olyan tényező, ami a megoldás túlzott használatához vezethet. Emlékeztetni kell arra, hogy ezért a kényelemért a *B* és a *D2* közti nagyobb függéssel kell fizetni (lásd §24.3.2.1). A *D2*-ről *B*-re történő automatikus konverzióról különösen könnyű megfeledkezni. Hacsak az ilyen konverziók nem tartoznak szorosan a használt osztályok fogalomábrázolásához, kerüljük a *public* származtatást. Ha egy osztályt egy fogalom ábrázolására használunk, az öröklés pedig *is-a* kapcsolatot fejez ki, szinte biztos, hogy éppen ilyen konverziókra lesz szükségünk.

Vannak esetek, melyekben öröklést szeretnénk, de nem engedhetjük meg, hogy konverzió történjen. Vegyük egy *Cfield* osztály írását (controlled field, ellenőrzött mező), amely – egyebeken kívül – futási idejű hozzáférés-ellenőrzést biztosít egy másik *Field* osztály részére. Első ránézésre a *Cfield* meghatározása a *Field*-ből való származtatással éppen jónak látszik:

```
class Cfield : public Field { /* ... */ };
```

Ez kifejezi, hogy egy *Cfield* valójában egyfajta *Field*, kényelmesebb jelölést biztosít, amikor olyan *Cfield* függvényt írunk, mely a *Cfield Field* részének egy tagját használja és – ez a legfontosabb – megengedi, hogy egy *Cfield* felülírja a *Field* virtuális függvényeit. Az a baj, hogy a *Cfield**-ról *Field**-ra történő konverzió, amit a *Cfield* deklarációja sugall, meghiúsít minden, a *Field*-hez való hozzáférés ellenőrzésére irányuló kísérletet:

```
void g(Cfield* p)
{
  *p = "asdf";        // a Field elérése a Cfield értékadó operátorával vezérelt:
                      // p->Cfield::operator=("asdf")

  Field* q = p;       // automatikus átalakítás Cfield*-ról Field*-ra
  *q = "asdf";        // hoppá! nincs Cfield-en keresztüli ellenőrzés
}
```

Egy megoldás az lehetne, hogy úgy határozzuk meg a *Cfield*-et, mint amelynek *Field* egy tagja, de ha ezt tesszük, eleve kizárjuk, hogy *Cfield* felülírhassa *Field* virtuális függvényeit. Jobb megoldás, ha *private* öröklődést használunk:

```
class Cfield : private Field { /* ... */ };
```

Tervezési szempontból a privát származtatás egyenértékű a tartalmazással, kivéve a felülírás (alkalmanként lényeges) kérdését. E módszer fontos felhasználási területe egy osztály nyilvános származtatása egy felületet leíró absztrakt bázisosztályból, valamint a privát vagy védett származtatás egy konkrét osztályból, implementáció céljából (§2.5.4, §12.3, §25.3). Mivel a *private* és *protected* származtatásból következő öröklődés nem tükröződik a származtatott osztály típusában, néha ***implementációs öröklődésnek*** nevezzük a nyilvános származtatással szemben, melynél a bázisosztály felülete öröklődik és az alaptípus automatikus konverziója megengedett. Az utóbbira néha mint ***altípuskészítésre*** (subtyping) vagy ***felület-öröklésre*** (interface inheritance) hivatkozunk.

Úgy is megfogalmazhatjuk ezt, hogy rámutatunk, egy származtatott osztály objektuma használható kell legyen mindenütt, ahol bázisosztályának objektumai használhatók. Ezt néha „Liskov-féle helyettesítési elvnek" (Liskov Substitution Principle) nevezik (§23.6 [Liskov, 1987]). A nyilvános/védett/privát megkülönböztetés közvetlenül támogatja ezt, amikor többalakú típusokat mutatókon és hivatkozásokon keresztül kezelünk.

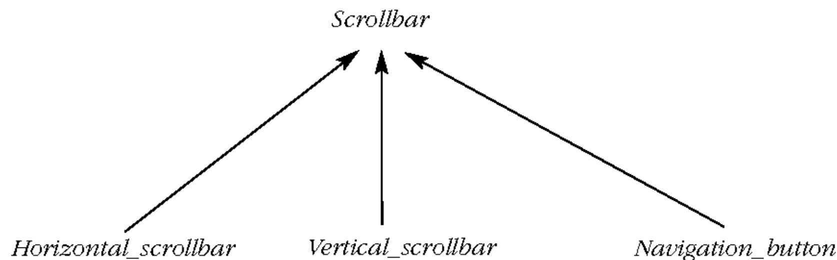
24.3.4.1. Tag vagy hierarchia?

Hogy tovább vizsgáljuk a tartalmazást és öröklést magával vonó tervezési választásokat, vegyük egy gördítősáv (scrollbar) ábrázolását egy interaktív grafikus rendszerben és azt, hogyan kapcsolhatunk egy gördítősávot egy ablakhoz. Kétféle gördítősávra van szükségünk: vízszintesre és függőlegesre. Ezt két típussal ábrázolhatjuk – *Horizontal_scrollbar* és *Vertical_scrollbar* – vagy egyetlen olyan *Scrollbar* típussal, mely paraméterként megkapja, vízszintes vagy függőleges-e. Az előbbi választás következménye, hogy egy harmadik típusra, a sima *Scrollbar*-ra is szükség van, mint a két gördítősáv-típus alaptípusára. Az utóbbi választás azt eredményezi, hogy egy külön paramétert kell használnunk és értékeket kell választanunk a két típus ábrázolására:

```
enum Orientation { horizontal, vertical };
```

Választásunk meghatározza, milyen módosítások szükségesek a rendszer bővítéséhez. Lehet, hogy be kell vezetnünk egy harmadik típusú gördítősávot. Eredetileg úgy gondolhattuk, elég kétféle gördítősáv („egy ablaknak végül is csak két dimenziója van"), de szinte minden esetben lehetségesek bővítések, melyek az újratervezés szükségességét vonják maguk után. Például lehet, hogy valaki a két gördítősáv helyett egy „navigáló gombot" szeretne használni. Egy ilyen gomb különböző irányú görgetést okozna, aszerint, hol nyomja meg a felhasználó. Felül középen nyomva „felfelé" görget, bal oldalon középen nyomva „balra", míg a bal felső sarkot nyomva „balra felfelé". Az ilyen gombok nem szokatlanok. A gördítősávok továbbfejlesztett változatainak tekinthetők, és különösen illenek olyan programokhoz, melyekben a görgetett adatok nem csupán szövegek, hanem képek.

Ha egy programba, melyben egy három gördítősávból álló osztályhierarchia van, egy navigáló gombot teszünk, létre kell hoznunk hozzá egy új osztályt, de a gördítősáv régi kódját nem kell módosítanunk:



Ez a „hierarchikus” megoldás szép oldala.

Ha paraméterben adjuk meg a görgetés irányát, a gördítősáv-objektumokban típusmezőknek, a gördítősáv-tagfüggvények kódjában pedig *switch* utasításoknak kell szerepelniük. Ez azt jelenti, hogy választanunk kell, deklarációk vagy kód által fejezzük ki a rendszer szerkezetének ezt a vonását. Az előbbi növeli a statikus ellenőrzés fokát és azt az adatmennyiséget, melyet az eszközöknek fel kell dolgozniuk. Az utóbbi a futási időre halasztja el a döntéseket és az egyes függvények módosításával anélkül tesz lehetővé változtatásokat, hogy befolyásolná a rendszer átfogó szerkezetét a típusellenőrző és más eszközök szempontjából. A legtöbb helyzetben azt javaslom, használjunk osztályhierarchiát a fogalmak kapcsolatainak közvetlen modellezésére.

Az egyetlen típust használó megoldás megkönnyíti a gördítősáv fajtáját leíró adatok tárolását és továbbítását:

```
void helper(Orientation oo)
{
  // ...
  p = new Scrollbar(oo);
  // ...
}

void me()
{
  helper(horizontal);
  // ...
}
```

Ez az ábrázolásmód megkönnyíti a görgetés irányának futási időben történő megváltoztatását. Nem valószínű, hogy ennek itt nagy jelentősége van, de más hasonló példák esetében fontos lehet. A lényeg, hogy mindig választani kell, és a választás gyakran nem könnyű.

24.3.4.2. Tartalmazás vagy hierarchia?

Most vegyük azt a kérdést, hogyan kapcsoljunk egy gördítősávot egy ablakhoz. Ha egy *Window_with_scrollbar*-t úgy tekintünk, mint ami egyszerre *Window* és *Scrollbar* is, valami ilyesmit kapunk:

```
class Window_with_scrollbar : public Window, public Scrollbar {
  // ...
};
```

Ez megengedi, hogy egy *Window_with_scrollbar* úgy is viselkedhessen, mint egy *Scrollbar* és úgy is, mint egy *Window*, de arra kényszerít, hogy az egyetlen típust használó megoldást használjuk.

Másrészt, ha a *Window_with_scrollbar*-t egy *Scrollbar*-ral rendelkező *Window*-nak tekintjük, valami ilyesmit kapunk:

```
class Window_with_scrollbar : public Window {
  // ...
  Scrollbar* sb;
public:
  Window_with_scrollbar(Scrollbar* p, /* ... */) : Window(/* ...*/), sb(p) { /* ... */ }
  // ...
};
```

Ez megengedi, hogy a gördítősáv-hierarchia megoldást használjuk. Ha a gördítősávot paraméterben adjuk át, az ablak figyelmen kívül hagyhatja annak pontos típusát. Sőt, egy *Scrollbar*-t ahhoz hasonlóan is átadhatunk, ahogy az *Orientation*-t a §24.3.4.1 pontban. Ha arra van szükség, hogy a *Window_with_scrollbar* gördítősávként működjön, hozzátehetünk egy konverziós műveletet:

```
Window_with_scrollbar::operator Scrollbar&()
{
  return *sb;
}
```

Én azt részesítem előnyben, ha az ablak a gördítősávot *tartalmazza*. Könnyebb egy olyan ablakot elképzelni, melynek gördítősávja van, mint egy olyat, amely amellett, hogy ablak, még gördítősáv is. Kedvenc módszerem az, hogy olyan gördítősávot határozok meg, amely egy különleges ablak, és ezt tartalmazza egy, a gördítősáv-szolgáltatásokat igénylő ablak. Ez a *tartalmazás* használatát igényli, de emellett szól egy másik érv is, mely a „lehet neki kettő is" szabályból következik (§24.3.4). Mivel nincs logikus indok, miért ne lehetne egy ablaknak két gördítősávja (valójában sok ablaknak van vízszintes és függőleges is), nem kötelező a *Window_with_scrollbar*-t a *Scrollbar*-ból származtatni.

Vegyük észre, hogy ismeretlen osztályból nem lehet származtatni, fordításkor ismerni kell a bázisosztály pontos típusát (§12.2). Másrészt, ha egy osztály egy tulajdonságát paraméterben adjuk át konstruktorának, akkor valahol az osztályban kell, hogy legyen egy tag, mely azt ábrázolja. Ha azonban ez a tag egy mutató vagy referencia, akkor a taghoz megadott osztályból származtatott osztály egy objektumát is átadhatjuk. Az előző példában például a *Scrollbar* sb* tagja mutathat egy olyan *Scrollbar* típusra, mint a *Navigation_button*, amely a *Scrollbar** felhasználója számára ismeretlen.

24.3.5. Használati kapcsolatok

Annak ismerete, hogy egy osztály milyen más osztályokat használ és milyen módon, gyakran létfontosságú a programszerkezet kifejezése és megértése szempontjából. Az ilyen függéseket a C++ csak rejtetten (implicit módon) támogatja. Egy osztály csak olyan elemeket használhat, melyeket (valahol) deklaráltak, de azok felsorolása nem szerepel a C++ forráskódban. Eszközök szükségesek (vagy megfelelő eszközök hiányában gondos elolvasás) az ilyen adatok kinyeréséhez. A módok, ahogy egy *X* osztály egy *Y* osztályt felhasználhat, többféleképpen osztályozhatók. Íme egy lehetséges változat:

- *X* használja az *Y* nevet.
- *X* használja *Y*-t.
  - *X* meghívja *Y* egy tagfüggvényét.
  - *X* olvassa *Y* egy tagját.
  - *X* írja *Y* egy tagját.
- *X* létrehoz egy *Y*-t.
  - *X* lefoglal egy *auto* vagy *static Y* változót.
  - *X* a *new* segítségével létrehoz egy *Y*-t.
- veszi egy *Y* méretét.

Az utolsó azért került külön kategóriába, mivel ehhez ismerni kell az osztály deklarációját, de független a konstruktoroktól. Az Y nevének használata szintén külön módszert jelent, mert ehhez önmagában – például egy Y* deklarálásánál vagy Y-nak egy külső függvény deklarációjában való említésénél – egyáltalán nem kell hozzáférni Y deklarációjához (§5.7):

```
class Y;      // Y az osztály neve
Y* p;
extern Y f(const Y&);
```

Gyakran fontos, hogy különbséget tegyünk egy osztály felületének (az osztály deklarációjának) és az osztály megvalósításának függései között. Egy jól tervezett rendszerben az utóbbinak általában jóval több függősége van, és azok egy felhasználó számára sokkal kevésbé érdekesek, mint az osztály deklarációjának függései (§24.4.2). A tervezéskor arra kell törekednünk, hogy a felületek függéseinek számát csökkentsük, mert ezekből az osztály felhasználóinak függései lesznek (§8.2.4.1, §9.3.2, §12.4.1.1 és §24.4).

A C++ nem kívánja meg egy osztály készítőjétől, hogy részletesen leírja, milyen más osztályokat és hogyan használ fel. Ennek egyik oka, hogy a legjelentősebb osztályok oly sok más osztálytól függnek, hogy az olvashatóság miatt ezen osztályok rövidített felsorolására – például egy *#include* utasításra – lenne szükség. A másik ok, hogy az ilyen függések osztályo-

zása nem a programozási nyelv kérdése. Az, hogy a „használja" (*use*) típusú függéseket pontosan hogyan szemléljük, függ a tervező, a programozó vagy az eszköz céljától, az pedig, hogy mely függések az érdekesek, függhet az adott fejlesztőkörnyezettől is.

24.3.6. Beprogramozott kapcsolatok

Egy programozási nyelv nem képes közvetlenül támogatni – és nem is szabad, hogy támogassa – minden tervezési módszer minden fogalmát. Hasonlóképpen egy tervezési nyelvnek sem célszerű támogatnia minden programozási nyelv minden tulajdonságát. Egy tervezési nyelv legyen gazdagabb és kevésbé törődjön a részletekkel, mint amennyire ez egy rendszerprogramozási nyelvnél szükséges. Megfordítva, egy programozási nyelvnek képesnek kell lennie többféle tervezési filozófia támogatására, különben csorbul az alkalmazhatósága.

Ha egy programozási nyelvben nincs lehetőség egy tervezési fogalom közvetlen ábrázolására, nem szabad hagyományos leképezést alkalmazni a tervezési szerkezetek és a programozási nyelv szerkezetei között. A tervezési módszer például használhatja a delegálás (delegation) fogalmát, vagyis a terv meghatározhatja, hogy minden olyan műveletet, amely A osztályhoz nincs definiálva, egy *p* mutató által mutatott *B* osztálybeli objektum szolgáltasson. A C++ ezt nem tudja közvetlenül kifejezni, de rendelkezésre állnak olyan szerkezetek, melyek segítségével könnyű elképzelni egy programot, mely a fogalomból kódot hoz létre. Vegyük az alábbiakat:

```
class B {
  // ...
  void f();
  void g();
  void h();
};

class A {
  B* p;
  // ...
  void f();
  void ff();
};
```

Annak meghatározása, hogy A osztály az *A::p* mutatón keresztül átruházza („delegálja") a feladatot *B*-re, ilyen kódot eredményezne:

```
class A {
  B* p;                      // delegálás p-n keresztül
  // ...
  void f();
  void ff();
  void g() { p->g(); }       // g() delegálás
  void h() { p->h();         // h() delegálás
};
```

Egy programozó számára nyilvánvaló, hogy mi történik itt, de világos, hogy egy tervezési fogalom kóddal való utánzása kevesebb, mint valamely „egy az egyhez" megfeleltetés. Az ilyen „beprogramozott" kapcsolatokat a programozási nyelv nem „érti" olyan jól, ezért nehezebb azokat eszközökkel támogatni. A szabványos eszközök például nem ismernék fel, hogy az *A::p*-n keresztül A feladatát ruháztuk át *B*-re, nem pedig más célra használtuk a *B**-ot.

Ahol csak lehetséges, használjunk „egy az egyhez" leképezést a tervezés és a programozási nyelv fogalmai között. Az ilyen leképezés biztosítja az egyszerűséget, és azt, hogy a program valóban tükrözi a tervezést, hogy a programozók és az eszközök hasznosíthassák annak fogalmait. A beprogramozott kapcsolatokkal rendelkező osztályok kifejezését a nyelv konverziós műveletekkel segíti. Ez azt jelenti, hogy az *X::operator Y()* konverziós operátor meghatározza, hogy ahol elfogadható egy *Y*, ott használhatunk egy *X*-et is (§11.4.1). A *Y::Y(X)* konstruktor ugyanezt a kapcsolatot fejezi ki. Vegyük észre, hogy a konverziós operátorok (és a konstruktorok) új objektumokat hoznak létre, nem pedig létező objektumok típusát változtatják meg. Egy konverziós függvény deklarálása *Y*-ra nem más, mint egy *Y*-t visszaadó függvény rejtett alkalmazásának egy módja. Mivel a konstruktorok és konverziós operátorok által definiált átalakítások automatikus alkalmazása csalóka lehet, néha hasznos, ha a terven belül külön elemezzük azokat.

Fontos, hogy biztosítsuk, hogy egy program konverziós diagramjai ne tartalmazzanak ciklusokat. Ha tartalmaznak, az ebből adódó többértelműségek megakadályozzák, hogy a ciklusokban szereplő típusokat együtt használhassuk:

```
class Rational;

class Big_int {
public:
  friend Big_int operator+(Big_int,Big_int);
  operator Rational();
  // ...
};
```

```
class Rational {
public:
  friend Rational operator+(Rational,Rational);
  operator Big_int();
  // ...
};
```

A *Rational* és a *Big_int* típusok nem fognak olyan gördülékenyen együttműködni, mint remélnénk:

```
void f(Rational r, Big_int i)
{
  g(r+i);   // hiba, többértelmű: operator+(r,Rational(i)) vagy operator+(Big_int(r),i) ?
  g(r+Rational(i));            // explicit feloldás
  g(Big_int(r)+i);             // másik explicit feloldás
}
```

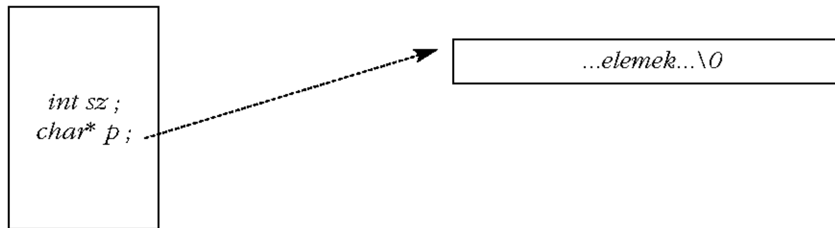
Az ilyen „kölcsönös" konverziókat elkerülhetjük, ha legalább néhányat közülük explicitté teszünk. A *Big_int* átalakítása *Rational*-lé például konverziós operátor helyett definiálható lett volna *make_Rational()*-ként, az összeadást pedig *g(Big_int®,i)*-re lehetett volna feloldani. Ahol nem kerülhetők el az ilyen „kölcsönös" konverziók, a keletkezett ellentmondásokat vagy explicit átalakításokkal (lásd fent), vagy kétoperandusú operátorok (pl. +) több külön változatának deklarálásával kell feloldanunk.

24.3.7. Osztályon belüli kapcsolatok

Az osztályok a megvalósítás módját (és a rossz megoldásokat) csaknem teljes egészében elrejthetik – és néha el is kell rejteniük. A legtöbb osztály objektumainak azonban szabályos szerkezete van és oly módon kezelhető, amit meglehetősen könnyű leírni. Egy osztály valamely objektuma alobjektumok (tagok) gyűjteménye, melyek közül sok más objektumokra mutat vagy hivatkozik. Egy objektum tehát úgy tekinthető, mint egy objektumfa gyökere, a benne szereplő objektumok pedig mint egy „objektum-hierarchia" alkotóelemei, ami az osztályhierarchia kiegészítője (lásd 25.3.2.1). Vegyünk például egy nagyon egyszerű *String*-et:

```
class String {
  int sz;
  char* p;
public:
  String(const char* q);
  ~String();
  // ...
};
```

Egy *String* objektum grafikusan így ábrázolható:



24.3.7.1. Invariánsok

A tagok és a tagok által hivatkozott objektumok értékeit gyűjtőnéven az objektum *állapotának* (vagy egyszerűen *értékének*) nevezzük. Egy osztály tervezésénél fontos az objektumok pontosan definiált állapotba hozása (kezdeti értékadás/létrehozás), ezen állapot fenntartása a műveletek végrehajtása során, és végül az objektumok megfelelő megsemmisítése. Az a tulajdonság, amely pontosan definiáltá teszi egy objektum állapotát, az objektum *invariánsa* (állapotbiztosítója, invariant).

A kezdeti értékadás célja tehát az objektumot abba az állapotba helyezni, amelyet az invariáns leír. Ezt általában egy konstruktorral végezzük. Egy osztályon végzett minden művelet feltételezheti, hogy belépéskor érvényesnek találja az állapotot (vagyis hogy az invariáns logikai értéke *igaz*) és kilépéskor ugyanez a helyzet áll fenn. Az invariánst végül a destruktor érvényteleníti, az objektum megsemmisítésével. A *String::String(const char*)* konstruktor például biztosítja, hogy *p* egy legalább *sz+1* elemű tömbre mutat, ahol *sz*-nek értelmes értéke van és *p[sz]==0*. Ezt az állítást minden karakterlánc-műveletnek „igaznak" kell hagynia.

Az osztálytervezésben a legtöbb szakértelem ahhoz kell, hogy az adott osztályt elég egyszerűvé tegyük, hogy használható, egyszerűen kifejezhető invariánssal valósíthassuk meg. Elég könnyű kijelenteni, hogy minden osztálynak szüksége van invariánsra. Ami nehéz, egy használható invariánssal „előhozakodni", amely könnyen érthető és nem jár elfogadhatatlan megszorításokkal a készítőre vagy a műveletek hatékonyságára vonatkozóan. Vegyük észre, hogy az „invariánst" itt egy olyan kódrészlet megjelölésére használjuk, melyet futtathatunk egy objektum állapotának ellenőrzésére. Világos, hogy be lehetne vezetni egy szigorúbb, „matematikaibb" és – bizonyos környezetben – megfelelőbb fogalmat is. Az invariáns, ahogyan itt tárgyaljuk, az objektum állapotának egyfajta gyakorlati – és emiatt általában gazdaságos, de logikailag nem tökéletes – ellenőrzése.

Az invariáns fogalma Floyd, Naur és Hoare előfeltételekkel és utófeltételekkel foglalkozó munkáiból ered és lényegében minden absztrakt adattípusokról és programhelyesség-ellenőrzésről szóló munka említi, amit az utóbbi 30 évben írtak, így a C hibakeresésnek is fontos elemét képezi. Az invariáns ellenőrzésére a tagfüggvények végrehajtása közben általában nem kerül sor. Azok a függvények, melyek meghívhatók, miközben az invariáns érvénytelen, nem lehetnek a nyilvános felület részei; e célra a privát és védett függvények szolgálhatnak.

Hogyan fejezhetjük ki egy C++ programban az invariáns fogalmát? Erre egy egyszerű mód egy invariáns-ellenőrző függvényt megadni és a nyilvános műveletekbe e függvény meghívását beiktatni:

```
class String {
  int sz;
  char* p;
public:
  class Range {};                    // kivételosztály
  class Invariant {};                // kivételosztály

  enum { TOO_LARGE = 16000 };        // mérethatár

  void check();                      // állapot-ellenőrzés

  String(const char* q);
  String(const String&);
  ~String();

  char& operator[](int i);
  int size() { return sz; }

  // ...
};

void String::check()
{
  if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant();
}

char& String::operator[](int i)
{
  check();                                    // ellenőrzés belépéskor
  if (i<0 || sz<=i) throw Range();
   // itt végezzük a tényleges munkát
  check();                                    // ellenőrzés kilépéskor
  return p[i];
}
```

Ez szépen fog működni és alig jelent munkát a programozó számára. Viszont egy egyszerű osztálynál, mint a *String*, az állapotellenőrzés döntően befolyásolja a futási időt és lehet, hogy még a kód méretét is. Ezért a programozók gyakran csak hibakeresés alatt hajtják végre az invariáns ellenőrzését:

```
inline void String::check()
{
#ifndef NDEBUG
  if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant();
#endif
}
```

Itt az *NDEBUG* makrót használjuk, hasonló módon, mint ahogy azt a szabványos C *assert()* makró használja. Az *NDEBUG* rendszerint azt jelzi, hogy *nem* történik hibakeresés.

Az invariánsok megadása és hibakeresés alatti használata felbecsülhetetlen segítséget jelent a kód „belövésénél" és – ami még fontosabb – akkor, amikor az osztályok által ábrázolt fogalmakat pontosan meghatározottá és szabályossá tesszük. A lényeg az, hogy az invariánsok beépítésével az osztályokat más szempontból vizsgálhatjuk és a kódban ellenőrzést helyezhetünk el. Mind a kettő növeli a valószínűségét, hogy megtaláljuk a hiányosságokat, következetlenségeket és tévedéseket.

24.3.7.2. Feltételezések

Az invariáns kód egyfajta feltételezés (assertion), ami viszont nem más, mint annak a kijelentése, hogy egy adott logikai feltételnek teljesülnie kell. Az a kérdés, mit kell tenni, amikor nem teljesül.

A C standard könyvtára – és következésképpen a C++ standard könyvtára – tartalmazza az *assert()* makrót a *<cassert>*-ben vagy *<assert.h>*-ban. Az *assert()* kiértékeli a paraméterét és meghívja az *abort()*-ot, ha az eredmény nulla (vagyis „hamis"):

```
void f(int* p)
{
  assert(p!=0);      // feltételezi, hogy p!=0; abort() meghívása, ha p nulla
  // ...
}
```

A program megszakítása előtt az *assert()* kiírja saját forrásfájljának nevét és annak a sornak a számát, amelyben előfordult. Ezáltal az *assert()* hasznos hibakereső eszközt jelent. Az *NDEBUG* beállítása rendszerint fordítói utasítások segítségével történik, fordítási egysé-

genként. Ebből következik, hogy az *assert()* használata tilos olyan helyben kifejtett (inline) függvényekben és sablon függvényekben, melyek több fordítási egységben is előfordulnak, hacsak nem fordítunk igen nagy gondot az *NDEBUG* következetes beállítására (§9.2.3). Mint minden „makró-varázslat", az *NDEBUG* használata is túl alacsony szintű, rendetlen és hibaforrást jelenthet. Általában jó ötlet, ha még a legjobban ellenőrzött programban is hagyunk néhány ellenőrző kódot; az *NDEBUG* viszont erre nem nagyon alkalmas, és az *abort()* meghívása is ritkán fogadható el a végleges kódban.

A másik megoldás egy *Assert()* sablon használata, mely a programból való kilépés helyett egy kivételt vált ki, így kívánságra a végleges kódban bennmaradhatnak az ellenőrzések. Sajnos a standard könyvtár nem gondoskodik ilyen *Assert()*-ről, de mi könnyen elkészíthetjük:

```
template<class X, class A> inline void Assert(A assertion)
{
  if (!assertion) throw X();
}
```

Az *Assert()* egy *X()* kivételt vált ki, ha az *assertion* hamis:

```
class Bad_arg { };

void f(int* p)
{
  Assert<Bad_arg>(p!=0);      // feltételezi, hogy p!=0; Bad_arg kivételt vált ki, hacsak p!=0
  // ...
}
```

Az ellenőrzésben a feltételt pontosan meg kell határoznunk, tehát ha csak hibakeresés alatt akarunk ellenőrizni, jeleznünk kell e szándékunkat:

```
void f2(int* p)
{
  Assert<Bad_arg>(NDEBUG || p!=0);      // vagy nem hibakeresés folyik vagy p!=0
  // ...
}
```

A || használata az ellenőrzésben && helyett meglepőnek tűnhet, de az *Assert<E> (a||b)* a *!(a||b)*-t ellenőrzi, ami *!a&&!b*.

Az *NDEBUG* ilyen módon való használata megköveteli, hogy az *NDEBUG*-nak megfelelő értéket adjunk, aszerint, hogy akarunk-e hibakeresést végezni vagy sem. A C++ egyes változatai alapértelmezés szerint nem teszik ezt meg nekünk, ezért jobb saját értéket használni:

```
#ifdef NDEBUG
const bool ARG_CHECK = false;          // nem hibakeresés: ellenőrzés kikapcsolása
#else
const bool ARG_CHECK = true;           // hibakeresés
#endif

void f3(int* p)
{
  Assert<Bad_arg>(!ARG_CHECK || p!=0);   // vagy nem hibakeresés folyik vagy p!=0
  // ...
}
```

Ha egy ellenőrzéssel kapcsolatos kivételt nem kapunk el, az *Assert()* leállítja a programot (*terminate()*), hasonlóan ahhoz, ahogyan a megfelelő *assert() abort()*-tal járna. Egy kivételkezelő azonban képes lehet valamilyen kevésbé durva beavatkozásra.

A valóságos méretű programokban azon veszem észre magamat, hogy egyes csoportokban be- és kikapcsolom a feltételezéseket, aszerint, hogy kell-e tesztelni. Ennek az eljárásnak az *NDEBUG* használata a legnyersebb formája. A fejlesztés elején a legtöbb ellenőrzés be van kapcsolva, míg az átadott kódban csak a legfontosabbak megengedettek. Ez akkor érhető el a legkönnyebben, ha a tényleges ellenőrzések két részből állnak, ahol az első egy megengedő feltétel (mint az *ARG_CHECK*), és csak a második maga a feltételezés.

Amennyiben a megengedő feltétel konstans kifejezés, az egész ellenőrzés kimarad a fordításból, ha nincs bekapcsolva. Lehet azonban változó is, mely a hibakeresés szükségletei szerint futási időben be- és kikapcsolható:

```
bool string_check = true;

inline void String::check()
{
  Assert<Invariant>(!string_check || (p && 0<=sz && sz<TOO_LARGE && p[sz]==0));
}

void f()
{
  String s = "csoda";
  // a karakterláncokat itt ellenőrizzük
  string_check = false;
  // itt a karakterláncokat nem ellenőrizzük
}
```

Természetesen ilyen esetekben létrejön a megfelelő programkód, tehát ügyelnünk kell, nehogy a program „meghízzon” az ilyen ellenőrzések kiterjedt használatától.

Ha azt mondjuk:

```
Assert<E>(a);
```

az egyszerűen egy másik módja ennek:

```
if (!a) throw E();
```

Így viszont felmerül a kérdés: minek bajlódjunk az *Assert()*-tel az utasítás közvetlen kiírása helyett? Azért, mert az *Assert()* használata nyilvánvalóvá teszi a tervező szándékát. Azt fejezi ki, hogy valamiről feltételezzük, hogy mindig igaz. Ez nem a program logikájának lényegtelen része, hanem értékes információ a program olvasója számára. Gyakorlati előnye, hogy egy *assert()*-et vagy *Assert()*-et könnyű megtalálni, míg a kivételeket kiváltó feltételes utasításokat nehéz.

Az *Assert()* általánosítható olyan kivételek kiváltására is, melyek paramétereket vagy kivétel-változókat vehetnek át:

```
template<class A, class E> inline void Assert(A assertion, E except)
{
  if (!assertion) throw except;
}

struct Bad_g_arg {
  int* p;
  Bad_g_arg(int* pp) : p(pp) { }
};

bool g_check = true;
int g_max = 100;

void g(int* p, exception e)
{
  Assert(!g_check || p!=0, e);                          // a mutató érvényes
  Assert(!g_check || (0<*p&&*p<=g_max),Bad_g_arg(p));   // az érték ésszerű
  // ...
}
```

Sok programban döntő fontosságú, hogy ne jöjjön létre kód olyan *Assert()* esetén, ahol az ellenőrzés a fordítás során kiértékelhető. Sajnos egyes fordítók az általánosított *Assert()*-nél

ezt képtelenek elérni. Következésképpen a kétparaméterű *Assert()*-et csak akkor használjuk, ha a kivétel nem *E()* alakú vagy ha valamilyen, az ellenőrzött értéktől független kódot kell készíteni.

A §23.4.3.5 pontban említettük, hogy az osztályhierarchia átszervezésének két legközönségesebb formája az osztály kettéhasítása, illetve két osztály közös részének kiemelése egy bázisosztályba. Az átszervezés lehetőségét mindkét esetben megfelelő invariánsokkal biztosíthatjuk. Viszont ha összehasonlítjuk az invariánst a műveletek kódjával, egy „hasításra érett” osztályban az állapotellenőrzést fölöslegesnek találhatjuk. Ilyen esetekben a műveletek részhalmazai csak az objektum állapotának részhalmazaihoz fognak hozzáférni. Megfordítva, azok az osztályok, melyek összefésülhetők, hasonló invariánsokkal rendelkeznek akkor is, ha megvalósításuk különbözik.

24.3.7.3. Előfeltételek és utófeltételek

A feltételezések (assert) egyik népszerű használata egy függvény előfeltételeinek és utófeltételeinek kifejezése. Ez azt jelenti, hogy ellenőrizzük a bemenetre vonatkozó alapfeltételezések érvényességét és azt, hogy kilépéskor a függvény a programot a várt állapotban hagyja-e. Sajnos amit a feltételezéssel ki szeretnénk fejezni, gyakran magasabb szinten van, mint aminek a hatékony kifejezésére a programozási nyelv lehetőséget ad:

```
template<class Ran> void sort(Ran first, Ran last)
{
  Assert<Bad_sequence>("A [first,last) érvényes sorozat"); // álkód

  // ... rendező algoritmus ...

  Assert<Failed_sort>("A [first,last) növekvő sorrendű");   // álkód
}
```

Ez alapvető probléma. Amit egy programról *feltételezünk*, az matematikai alapú, magasabb szintű nyelven fejezhető ki a legjobban, nem azon az algoritmusokra épülő programozási nyelven, amelyben a programot *írjuk*.

Az invariánsok meghatározásához hasonlóan a feltételezések megfogalmazása is bizonyos fokú ügyességet igényel, mert az ellenőrizni kívánt állítást úgy kell megadnunk, hogy az egy algoritmussal valóban ellenőrizhető legyen:

```
template<class Ran> void sort(Ran first, Ran last)
{
   // A [first,last) érvényes sorozat; a hihetőség ellenőrzése:
   Assert<Bad_sequence>(NDEBUG || first<=last);

   // ... rendező algoritmus ...

   // A [first,last) növekvő sorrendű; próba-ellenőrzés:
   Assert<Failed_sort>(NDEBUG ||
            (last-first<2 || (*first<=last[-1]
                     && *first<=first[(last-first)/2] && first[(last-first)/2]<=last[-1])));
}
```

Én gyakran egyszerűbbnek találom a közönséges kódellenőrző paraméterek és eredmények használatát, mint a feltételezések összeállítását. Fontos azonban, hogy megpróbáljuk a valós (ideális) elő- és utófeltételeket kifejezni – és legalább megjegyzések formájában dokumentálni azokat – mielőtt olyan, kevésbé absztrakt módon ábrázolnánk, ami a programozási nyelven hatékonyan kifejezhető.

Az előfeltételek ellenőrzése könnyen egyszerű paraméterérték-ellenőrzéssé fajulhat. Mivel a paraméterek gyakran több függvényen keresztül adódnak át, az ellenőrzés ismétlődővé és költségessé válhat. Annak az egyszerű kijelentése azonban, hogy minden függvényben minden mutató-paraméter nem nulla, nem sokat segít, és hamis biztonságérzetet adhat – különösen akkor, ha a többletterhelés elkerülése végett az ellenőrzéseket csak hibakeresés alatt végezzük. Ez az egyik fő oka, hogy javaslom, fordítsunk figyelmet az invariánsokra.

24.3.7.4. Betokozás

Vegyük észre, hogy a C++-ban az osztályok – nem az egyes objektumok – a betokozás (enkapszuláció, encapsulation) alapegységei:

```
class List {
   List* next;
public:
   bool on(List*);
   // ...
};

bool List::on(List* p)
{
   if (p == 0) return false;
   for(List* q = this; q; q=q->next) if (p == q) return true;
   return false;
}
```

A privát *List::next* mutató betokozása elfogadható, mivel a *List::on()* hozzáfér a *List* osztály minden objektumához, amelyre hivatkozni tud. Ahol ez kényelmetlen, egyszerűsíthetünk, ha nem használjuk ki azt a képességet, hogy egy tagfüggvényből más objektumok ábrázolásához hozzá lehet férni:

```
bool List::on(List* p)
{
  if (p == 0) return false;
  if (p == this) return true;
  if (next==0) return false;
  return next->on(p);
}
```

Ez azonban a bejárást (iterációt) ismétléssé (rekurzióvá) alakítja át, ami komoly teljesítményromlást okozhat, ha a fordító nem képes az optimális működéshez az eljárást visszaalakítani.

24.4. Komponensek

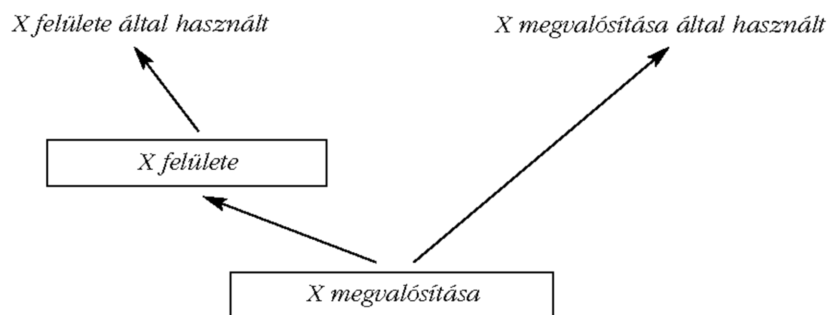
A tervezés egységei az osztályok, függvények stb. gyűjteményei, nem egy egyes osztályok. Ezeket a gyűjteményeket gyakran ***könyvtárnak*** (library) vagy ***keretrendszernek*** (framework) nevezzük (§25.8), és az újrahasznosítás (§23.5.1) és karbantartás is ezeket helyezi a középpontba. A logikai feltételek által összekapcsolt szolgáltatások halmazát jelentő fogalom kifejezésére a C++ három módot biztosít:

1. Osztály létrehozása, mely adat-, függvény-, sablon- és típustagok gyűjteményét tartalmazza.
2. Osztályhierarchia létrehozása, mely osztályok gyűjteményét foglalja magában.
3. Névtér meghatározása, mely adat-, függvény-, sablon- és típustagok gyűjteményéből áll.

Az osztályok számos lehetőséget biztosítanak arra, hogy kényelmesen hozhassunk létre általuk meghatározott típusú objektumokat. Sok jelentős komponens azonban nem írható le úgy, mint adott típusú objektumokat létrehozó rendszer. Az osztályhierarchia egymással rokon típusok halmazának fogalmát fejezi ki. A komponensek egyes tagjainak kifejezésére azonban nem mindig az osztályok jelentik a legjobb módszert és nem minden osztály illeszthető hasonlóság alapján osztályhierarchiába (§24.2.5). Ezért a komponens fogalmának

a C++-ban a névtér a legközvetlenebb és legáltalánosabb megtestesítése. A komponenseket néha „osztálykategóriáknak” nevezzük, de nem minden esetben állnak osztályokból (és ez nem is előírás).

Ideális esetben egy komponens a megvalósítására használt felületekkel, valamint azokkal a felületekkel írható le, melyeket felhasználói részére biztosít. Minden más „részletkérdés” és a rendszer többi része számára rejtett. A tervező szemszögéből ez a komponens meghatározása. A programozó ezt a fogalmat deklarációkra leképezve ábrázolhatja. Az osztályok és osztályhierarchiák adják a felületeket, melyek csoportosítására a névterek adnak lehetőséget, és ugyancsak a névterek biztosítják, hogy a programozó elválaszthassa a felhasznált felületeket a szolgáltatottaktól. Vegyük az alábbi példát:



A §8.2.4.1-ben leírt módszerek felhasználásával ebből az alábbi lesz:

```
namespace A {               // az X felülete által használt szolgáltatások
  // ...
}

namespace X {               // az X komponens felülete

  using namespace A;        // az A deklarációitól függ
  // ...
  void f();
}

namespace X_impl {          // az X megvalósításához szükséges szolgáltatások
  using namespace X;
  // ...
}
```

```
void X::f()
{
  using namespace X_impl;   // az X_impl deklarációitól függ
  // ...
}
```

Az *X* általános felület nem függhet az *X_impl* megvalósítási felülettől.

Egy komponensnek számos olyan osztálya lehet, melyek nem általános használatra szántak. Az ilyen osztályokat megvalósító osztályokba vagy névterekbe kell „elrejteni":

```
namespace X_impl {           // az X megvalósításának részletei

  class Widget {
        // ...
  };

  // ...
}
```

Ez biztosítja, hogy a *Widget*-et nem használhatjuk a program más részeiből. Az egységes fogalmakat ábrázoló osztályok általában újrahasznosíthatók, ezért érdemes befoglalni azokat a komponens felületébe:

```
class Car {
  class Wheel {
        // ...
  };

  Wheel flw, frw, rlw, rrw;
  // ...
public:
  // ...
};
```

A legtöbb környezetben el kell rejtenünk a kerekeket (*Wheel*), hogy megtartsuk az autó (*Car*) ábrázolásának pontosságát (egy valódi autó esetében sem tudjuk a kerekeket függetlenül működtetni). Maga a *Wheel* osztály viszont alkalmasnak tűnik szélesebb körű használatra, így lehet, hogy jobb a *Car* osztályon kívülre tenni:

```
class Wheel {
  // ...
};
```

```
class Car {
   Wheel flw, frw, rlw, rrw;
   // ...
public:
   // ...
};
```

A döntés, hogy beágyazzuk-e az egyik fogalmat a másikba, attól függ, mik a tervezés céljai és mennyire általánosak a fogalmak. Mind a beágyazás, mind a „nem beágyazás" széles körben alkalmazható módszer. Az alapszabály az legyen, hogy az osztályokat a lehető legönállóbbá („helyivé") tesszük, amíg általánosabb elérésüket szükségesnek nem látjuk.

Az „érdekes" függvényeknek és adatoknak van egy kellemetlen tulajdonsága, mégpedig az, hogy a globális névtér felé, a széles körben használt névterekbe vagy a hierarchiák gyökérosztályaiba „törekednek". Ez könnyen vezethet a megvalósítás részleteinek nem szándékos nyilvánossá tételéhez és a globális adatokkal és globális függvényekkel kapcsolatos problémákhoz. Ennek valószínűsége az egyetlen gyökerű hierarchiákban és az olyan programokban a legnagyobb, ahol csak nagyon kevés névteret használunk. E jelenség leküzdésére az osztályhierarchiákkal kapcsolatban a virtuális bázisosztályokat (§15.2.4) használhatjuk fel, a névtereknél pedig a kis „implementációs" névterek kialakítása lehet megoldás.

Megjegyzendő, hogy a fejállományok (header) komoly segítséget nyújthatnak abban, hogy a komponenseket felhasználóiknak különböző szempontból „mutathassuk", és kizárják azokat az osztályokat, melyek a felhasználó szempontjából a megvalósítás részeinek tekinthetők (§9.3.2).

24.4.1. Sablonok

A tervezés szempontjából a sablonok (template) két, laza kapcsolatban lévő szükségletet szolgálnak ki:

- Általánosított (generic) programozás
- Eljárásmódot (policy) meghatározó paraméterezés

A tervezés korai szakaszában a műveletek csupán műveletek. Később, amikor az operandusok típusát meg kell határozni, az olyan, statikus típusokat használó programozási nyelvekben, mint a C++, a sablonok lényeges szerepet kapnak. Sablonok nélkül a függvény-definíciókat újra meg újra meg kellene ismételni, vagy az ellenőrzéseket a futási időre kellene elhalasztani (§24.2.3). Azokat a műveleteket, amelyek egy algoritmust többféle operandustípusra kell, hogy leírjanak, célszerű sablonként elkészíteni. Ha az összes ope-

randus egyetlen osztályhierarchiába illeszthető (és különösen akkor, ha futási időben szükség van új operandustípusok hozzáadására is), az operandustípusok osztályként, mégpedig általában absztrakt osztályként ábrázolhatók a legjobban. Ha az operandustípusok nem illenek bele egyetlen hierarchiába (és különösen akkor, ha a futási idejű teljesítmény létfontosságú), a műveletet legjobb sablonként elkészíteni. A szabványos tárolók és az azokat támogató algoritmusok annak példái, amikor a többféle, nem rokon típusú operandus használatának és a futási idejű teljesítmény fokozásának egyidejű szükségessége vezet sablonok használatához (§16.2). Vegyük egy egyszerű bejárás általánosítását, hogy jobban szemügyre vehessük a „sablon vagy hierarchia?" dilemmát:

```
void print_all(Iter_for_T x)
{
  for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

Itt azzal a feltételezéssel éltünk, hogy az *Iter_for_T* olyan műveleteket biztosít, melyek *T**-okat hoznak létre.

Az *Iter_for_T*-t sablonparaméterré tehetjük:

```
template<class Iter_for_T> void print_all(Iter_for_T x)
{
  for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

Ez lehetőséget ad arra, hogy egy sereg egymással nem rokon bejárót (iterator) használjunk, ha ezek mindegyike biztosítja a helyes jelentésű *first()*-öt és *next()*-et és ha fordítási időben minden *print_all()* hívás bejárójának típusát ismerjük. A standard könyvtár tárolói és algoritmusai ezen az ötleten alapulnak.

Felhasználhatjuk azt a megfigyelést is, hogy a *first()* és a *next()* a bejárók számára felületet alkotnak és létrehozhatunk egy osztályt, amely ezt az felületet képviseli:

```
class Iter {
public:
  virtual T* first() const = 0;
  virtual T* next() = 0;
};

void print_all2(Iter& x)
{
  for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

Most már az *Iter*-ből származtatott valamennyi bejárót használhatjuk. A kódok azonosak, függetlenül attól, hogy sablonokat vagy osztályhierarchiát használunk a paraméterezés ábrázolására – csak a futási idő, újrafordítás stb. tekintetében különböznek. Az *Iter* osztály különösen alkalmas arra, hogy az alábbi sablon paramétere legyen:

```
void f(Iter& i)
{
  print_all(i);        // a sablon használata
  print_all2(i);
}
```

Következésképpen a két megközelítés egymást kiegészítőnek tekinthető.

Egy sablonnak gyakran van szüksége arra, hogy függvényeket és osztályokat úgy használjon, mint megvalósításának részeit. Soknak közülük maguknak is sablonoknak kell lenniük, hogy megtartsuk az általánosságot és a hatékonyságot. Így az algoritmusok több típusra általánosíthatók. A sablonok használatának ezt a módját nevezzük ***általánosított*** vagy ***generikus programozásnak*** (§2.7). Ha az *std::sort()*-ot egy *vector*-ra hívjuk meg, a *sort()* operandusai a vektor elemei lesznek, vagyis a *sort()* az egyes elemtípusokra általánosított. A szabványos rendező függvény emellett általános a tárolótípusokra nézve is, mert bármely, a szabványhoz igazodó tároló bejárójára meghívható (§16.3.1).

A *sort()* algoritmus az összehasonlítási feltételekre is paraméterezett (§18.7.1). Tervezési szempontból ez más, mint amikor veszünk egy műveletet és általánosítjuk az operandus típusára. Sokkal magasabb szintű tervezési döntés, hiszen egy objektumon (vagy műveleten) dolgozó algoritmust úgy paraméterezünk, hogy az az algoritmus működésmódját vezérli. Ez egy olyan döntés, ami részben a tervező/programozó kezébe adja a vezérlést az algoritmus működési elveinek kialakításában.

24.4.2. Felület és megvalósítás

Az ideális felület

- teljes és egységes fogalomkészletet ad a felhasználónak,
- a komponensek minden részére nézve következetes,
- nem fedi fel a megvalósítás részleteit a felhasználónak,
- többféleképpen elkészíthető,
- típusai fordítási időben ellenőrizhetők,
- alkalmazásszintű típusok használatával kifejezett,
- más felületektől korlátozott és pontosan meghatározott módokon függ.

Miután megjegyeztük, hogy az osztályok között, melyek a külvilág felé a komponens felületét képezik, következetességre van szükség (§24.4), a tárgyalást egyetlen osztályra egyszerűsíthetjük. Vegyük az alábbi példát:

```
class Y { /* ... */ };          // X által igényelt

class Z { /* ... */ };          // X által igényelt

class X {    // példa a szegényes felületre
  Y a;
  Z b;
public:
  void f(const char * ...);
  void g(int[],int);
  void set_a(Y&);
  Y& get_a();
};
```

Ennél a felületnél több probléma is felmerülhet:

- A felület *Y* és *Z* típust úgy használja, hogy a fordítónak ismernie kell *Y* és *Z* deklarációját.
- Az *X::f()* függvény tetszőleges számú ismeretlen típusú paramétert vehet át (valószínűleg egy, az első paraméterben megkapott „formázott karakterlánc” által vezérelt módon, §21.8).
- Az *X::g()* egy *int[]* paramétert vesz át. Ez elfogadható lehet, de annak a jele, hogy az elvonatkoztatás szintje túl alacsony. Egy egészekből álló tömb nem önleíró, tehát nem magától értetődő, hány eleme van.
- A *set_a()* és *get_a()* függvények valószínűleg felfedik *X* osztály objektumainak ábrázolását, azáltal, hogy *X::a*-hoz közvetlen hozzáférést tesznek lehetővé.

Ezek a tagfüggvények nagyon alacsony elvonatkoztatási (fogalmi) szinten nyújtanak felületet. Az ilyen szintű felületekkel rendelkező osztályok lényegében egy nagyobb komponens megvalósításának részletei közé tartoznak – ha egyáltalán tartoznak valahová. Ideális esetben egy felületfüggvény paramétere elég információt tartalmaz ahhoz, hogy önleíró legyen. Alapszabály, hogy a kérelmek „vékony dróton át” legyenek továbbíthatóak a távoli kiszolgáló felé.

A C++ a programozónak lehetőséget ad arra, hogy az osztályokat a felület részeként ábrázolja. Az ábrázolás lehet rejtett (*private* vagy *protected* használatával), de a fordító megengedi automatikus változók létrehozását, függvények helyben kifejtését stb. is. Ennek negatív hatása, hogy az osztálytípusok használata az osztály ábrázolásában nemkívánatos

függéseket idézhet elő. Az, hogy probléma-e *Y* és *Z* típusok tagjainak használata, attól függ, valójában milyen fajta típus *Y* és *Z*. Ha egyszerű típusok, mint a *list*, a *complex* vagy a *string*, használatuk általában teljesen megfelelő. Az ilyen típusok stabilnak minősülnek és osztálydeklarációik beépítése a fordító részére elfogadható terhelés. Ha azonban *Y* és *Z* maguk is jelentős komponensek (például egy grafikus rendszer vagy egy banki kezelőrendszer osztályai), bölcsebb lenne, ha nem függnénk tőlük túl közvetlenül. Ilyen esetekben gyakran jobb választás egy mutató vagy referencia használata:

```
class Y;
class Z;

class X {    // X csak mutatón és referencián át éri el Y-t és Z-t
  Y* a;
  Z& b;
  // ...
};
```

Ez leválasztja *X* definícióját *Y* és *Z* definícióiról, vagyis *X* definíciója csak *Y* és *Z* nevétől függ. *X* megvalósítása természetesen még mindig függ *Y* és *Z* definíciójától, de ez nem érinti *X* felhasználóit.

Ezzel egy fontos dolgot szemléltettünk: egy felületnek, mely jelentős adatmennyiséget rejt el – ahogyan ez egy használható felülettől elvárható – sokkal kevesebb lesz a függése, mint az általa elrejtett megvalósításnak. Az *X* osztály definíciója például anélkül fordítható le, hogy *Y* és *Z* definícióihoz hozzáférnénk. A függések elemzésekor külön kell kezelni a felület és külön a megvalósítás függéseit. Mindkét esetben ideális, ha a rendszer függőségi ábrái irányított, körmentes gráfok, melyek megkönnyítik a rendszer megértését és tesztelését. Ennek elérése a felületeknél fontosabb (és könnyebb is), mint a megvalósításnál.

Vegyük észre, hogy egy osztály három felületet írhat le:

```
class X {
private:
  // csak tagok és barát függvények számára elérhető
protected:
  // csak tagok és "barátok", valamint
  // a származtatott osztály tagjai számára elérhető
public:
  // általánosan elérhető
};
```

Ezenkívül még a nyilvános felület részét képezik a „barátok” is (*friend*, §11.5).

A tagokat a legkorlátozottabb elérésű felület részeként célszerű megadni. Ez azt jelenti, hogy a tagok mindig privátok legyenek, hacsak nincs okunk arra, hogy hozzáférhetőbbé tegyük azokat. Ha hozzáférhetőbbnek kell lenniük, legyenek védettek (*protected*), hacsak nincs okunk arra, hogy nyilvánosként (public) adjuk meg őket. Az adattagokat szinte soha nem szerencsés nyilvánossá vagy védetté tenni. A nyilvános felületet alkotó függvényeknek és osztályoknak az osztály olyan nézetét kell biztosítaniuk, amely illik ahhoz a szerephez, hogy az osztály egy fogalmat ábrázol.

Az ábrázolás elrejtésének egy további szintjét absztrakt osztályok használatával biztosíthatjuk (§2.5.4, §12.3, §25.3).

24.4.3. Kövér felületek

Ideális esetben egy felület csak olyan műveleteket kínálhat fel, melyeknek értelmük van és a felületet megvalósító bármely származtatott osztály által megfelelően leírhatók. Ez azonban nem mindig könnyű. Vegyük a listákat, tömböket, asszociatív tömböket, fákat stb. Mint ahogy a §16.2.2 pontban rámutattunk, csábító és néha hasznos ezeket a típusokat általánosítani – rendszerint egy tároló (konténer) használatával, melyet mindegyikük felületeként használhatunk. Ez (látszólag) felmenti a felhasználót a tároló részleteinek kezelése alól. Egy általános tárolóosztály felületének kialakítása azonban nem könnyű feladat. Tegyük fel, hogy a *Container*-t absztrakt típusként akarjuk meghatározni. Milyen műveleteket biztosítson a *Container*? Megadhatnánk csak azokat a műveleteket, melyeket minden tároló képes támogatni – a művelethalmazok metszetét –, de ez nevetségesen szűk felület. Sok esetben a metszet valójában üres. A másik mód, ha az összes művelethalmaz unióját adjuk meg, futáskor pedig hibajelzést adunk, ha ezen a felületen keresztül valamelyik objektumra egy „nem létező" művelet alkalmazása történik. Az olyan felületet, mely egy fogalomhalmaz felületeinek uniója, ***kövér felületnek*** (fat interface) nevezzük. Vegyünk egy *T* típusú objektumokat tartalmazó „általános tárolót":

```
class Container {
public:
   struct Bad_oper {                // kivételosztály
             const char* p;
             Bad_oper(const char* pp) : p(pp) { }
   };

   virtual void put(const T*) { throw Bad_oper("Container::put"); }
   virtual T* get() { throw Bad_oper("Container::get"); }
```

```
    virtual T*& operator[](int) { throw Bad_oper("Container::[](int)"); }
    virtual T*& operator[](const char*) { throw Bad_oper("Container::[](char*)"); }
    // ...
};
```

A *Container*-eket ezután így vezethetjük be:

```
class List_container : public Container, private list {
public:
    void put(const T*);
    T* get();
    // ... nincs operator[] ...
};

class Vector_container : public Container, private vector {
public:
    T*& operator[](int);
    T*& operator[](const char*);
    // ... nincs put() vagy get() ...
};
```

Amíg óvatosak vagyunk, minden rendben van:

```
void f()
{
    List_container sc;
    Vector_container vc;
    // ...
    user(sc,vc);
}

void user(Container& c1, Container& c2)
{
    T* p1 = c1.get();
    T* p2 = c2[3];
    // ne használjuk c2.get() vagy c1[3] műveletet
    // ...
}
```

Kevés azonban az olyan adatszerkezet, mely mind az indexeléses, mind a lista stílusú műveleteket jól támogatja. Következésképpen valószínűleg nem jó ötlet olyan felületet meghatározni, mely mindkettőt megköveteli. Ha ezt tesszük, a hibák elkerülése érdekében futási idejű típuslekérdezést (§15.4) vagy kivételkezelést (14. fejezet) kell alkalmaznunk:

```
void user2(Container& c1, Container& c2)  // észlelni könnyű, de a helyreállítás nehéz
lehet
{
  try {
            T* p1 = c1.get();
            T* p2 = c2[3];
            // ...
  }
  catch(Container::Bad_oper& bad) {
            // Hoppá!
            // Most mit tegyünk?
  }
}
```

vagy

```
void user3(Container& c1, Container& c2)
                      // a korai észlelés fárasztó, de a helyreállítás még mindig nehéz
{
  if (dynamic_cast<List_container*>(&c1) && dynamic_cast<Vector_container*>(&c2)) {
            T* p1 = c1.get();
            T* p2 = c2[3];
            // ...
  }
  else {
            // Hoppá!
            // Most mit tegyünk?
  }
}
```

A futási idejű teljesítmény mindkét esetben csökkenhet és a létrehozott kód meglepően nagy lehet. Ez azt eredményezi, hogy a programozók hajlamosak figyelmen kívül hagyni a lehetséges hibákat, remélve, hogy valójában nem is fordulnak elő, amikor a program a felhasználók kezébe kerül. E megközelítéssel az a probléma, hogy a kimerítő tesztelés szintén nehéz és drága munka.

Következésképpen a kövér felületeket a legjobb elkerülni ott, ahol a futási idejű teljesítmény elsőrendűen fontos, ahol megköveteljük a kód helyességére vonatkozó garanciát, és általában ott, ahol van más jó megoldás. A kövér felületek használata gyengíti a megfeleltetést a fogalmak és osztályok között és így szabad utat enged annak, hogy a származtatást pusztán a kényelmesebb megvalósítás kedvéért használjuk.

24.5. Tanácsok

[1] Törekedjünk az absztrakt adatábrázolásra és az objektumorientált programozásra. §24.2.
[2] A C++ tulajdonságait és technikáit (csak) szükség szerint használjuk. §24.2.
[3] Teremtsünk összhangot a tervezés és programozás stílusa között. §24.2.1.
[4] A függvények és a feldolgozás helyett a tervezésben elsődlegesen az osztályokra és fogalmakra összpontosítsunk. §24.2.1.
[5] A fogalmak ábrázolására használjunk osztályokat. §24.2.1, §24.3.
[6] A fogalmak közötti hierarchikus kapcsolatokat (és csak azokat) örökléssel ábrázoljuk. §24.2.2, §24.2.5, §24.3.2.
[7] A felületekre vonatkozó garanciákat fejezzük ki alkalmazásszintű statikus típusokkal. §24.2.3.
[8] A pontosan meghatározható feladatok elvégzésének megkönnyítésére használjunk programgenerátorokat és közvetlen kezelőeszközöket. §24.2.4.
[9] Kerüljük az olyan programgenerátorokat és közvetlen kezelőeszközöket, melyek nem illeszkednek pontosan az adott általános célú programozási nyelvhez. §24.2.4.
[10] Az elhatárolható fogalmi szinteket válasszuk el. §24.3.1.
[11] Összpontosítsunk a komponensek tervezésére. §24.4.
[12] Mindig győződjünk meg róla, hogy egy adott virtuális függvénynek pontosan definiált jelentése van-e és hogy az azt felülbíráló minden függvény a kívánt viselkedés egy változatát írja-e le. §24.3.5, §24.3.2.1.
[13] Használjunk nyilvános öröklést az *is-a* kapcsolatok ábrázolására. §24.3.4.
[14] Használjunk tagságot a *has-a* kapcsolatok ábrázolására. §24.3.4.
[15] Önállóan létrehozott objektumra hivatkozó mutatók helyett használjunk inkább közvetlen tagságot az egyszerű tartalmazási kapcsolatok kifejezésére. §24.3.3, §24.3.4.
[16] A *uses* függések legyenek világosak, (ahol lehetséges) kölcsönös függéstől mentesek, és minél kevesebb legyen belőlük. §24.3.5.
[17] Minden osztályhoz adjunk meg invariánsokat. §24.3.7.1.
[18] Az előfeltételeket, utófeltételeket és más állításokat feltételezésekkel (lehetőleg az *Assert()* használatával) fejezzük ki. §24.3.7.2.
[19] A felületek csak a feltétlenül szükséges információkat tegyék elérhetővé. §24.4.
[20] Csökkentsük a felületek függéseit más felületektől. §24.4.2.
[21] A felületek legyenek erősen típusosak. §24.4.2.
[22] A felületeket alkalmazásszintű típusokkal fejezzük ki. §24.4.2.
[23] A felületek tegyék lehetővé, hogy a kérések átvihetők legyenek egy távoli kiszolgálóra. §24.4.2.
[24] Kerüljük a kövér felületeket. §24.4.3.

[25] Használjunk *private* adatokat és tagfüggvényeket, ahol csak lehetséges. §24.4.2.
[26] A tervezők, illetve az általános felhasználók által igényelt származtatott osztályok elkülönítésére használjuk a *public/protected* megkülönböztetést. §24.4.2.
[27] Az általánosított programozás érdekében használjunk sablonokat. §24.4.1.
[28] Az algoritmusok eljárásmódot megadó paraméterezésére szintén sablonokat használjunk. §24.4.1.
[29] Ha fordítási idejű típusfeloldásra van szükség, használjunk sablonokat. §24.4.1.
[30] Ha futási idejű típusfeloldásra van szükség, használjunk osztályhierarchiákat. §24.4.1.

25

Az osztályok szerepe

Vannak dolgok, melyek jobb, ha változnak...
de az alapvető témáknak állandónak kell maradniuk.
(Stephen J. Gould)

Az osztályok fajtái • Konkrét típusok • Absztrakt típusok • Csomópontok • Változó felületek • Objektum I/O • Műveletek • Felületosztályok • Leírók • Használatszámlálás • Keretrendszerek • Tanácsok • Gyakorlatok

25.1. Az osztályok fajtái

A C++ osztály egy programozási nyelvi szerkezet, többféle tervezési igény kiszolgálására. A bonyolultabb tervezési problémák megoldása általában új osztályok bevezetésével jár (esetleg más osztályok elhagyásával). Az új osztályokkal olyan fogalmakat ábrázolunk, amelyeket az előző tervvázlatban még nem határoztunk meg pontosan. Az osztályok sokféle szerepet játszhatnak, ebből pedig az adódik, hogy a konkrét igényekhez egyedi osztályfajtákra lehet szükségünk. E fejezetben leírunk néhány alapvető osztálytípust, azok eredendő erősségeivel és gyengéivel együtt:

§25.2 Konkrét típusok
§25.2 Absztrakt típusok
§25.4 Csomópontok
§25.5 Műveletek
§25.6 Felületek
§25.7 Leírók
§25.8 Keretrendszerek

Ezek az „osztályfajták" tervezési fogalmak, nem nyelvi szerkezetek. A valószínűleg elérhetetlen ideál az lenne, ha rendelkezésre állna egyszerű és önálló osztályfajták egy olyan legkisebb halmaza, melyből az összes megfelelően viselkedő és használható osztályt megalkothatnánk. Fontos, hogy észrevegyük: a fenti osztályfajták mindegyikének helye van a tervezésben és egyik sem eredendően jobb minden használatra a többinél. A tervezési és programozási viták során számos félreértés adódhat abból, hogy vannak, akik kizárólag egy vagy két fajta osztályt próbálnak használni. Ez rendszerint az egyszerűség nevében történik, de a „kedvenc" osztályfajták torz és természetellenes használatához vezet.

Jelen leírás ezeknek az osztályfajtáknak a „tiszta" alakjaival foglalkozik, de természetesen kevert formák is használhatók. A keverékek azonban tervezési döntések eredményeként kell, hogy szülessenek, a lehetséges megoldások kiértékelésével, nem pedig a döntéshozatalt elkerülve, céltalan kísérletezéssel. A „döntések elhalasztása" sokszor valójában a „gondolkodás elkerülését" jelenti. A kezdő tervezők rendszerint jól teszik, ha óvakodnak a kevert megoldásoktól és egy létező összetevő stílusát követik, melynek tulajdonságai az új komponens kívánt tulajdonságaira emlékeztetnek. Csak tapasztalt programozók kíséreljék meg egy általános célú komponens vagy könyvtár megírását, és minden könyvtártervezőt arra kellene „ítélni", hogy néhány évig a saját alkotását használja, dokumentálja és támogassa. (Lásd még: §23.5.1.)

25.2. Konkrét típusok

Az olyan osztályok, mint a *vector* (§16.3), a *list* (§17.2.2), a *Date* (§10.3) és a *complex* (§11.3, §22.5) ***konkrétak*** abban az értelemben, hogy mindegyikük egy-egy viszonylag egyszerű fogalmat ábrázol, az összes, e fogalom támogatásához nélkülözhetetlen művelettel együtt. Mindegyiküknél fennáll az „egy az egyhez" megfelelés a felület és a megvalósítás között és egyiket sem szánták bázisosztálynak származtatáshoz. A konkrét típusok általában nem illenek bele egymással kapcsolatos osztályok hierarchiájába. Minden konkrét osztály önma-

gában megérthető, a lehető legkevesebb hivatkozással más osztályokra. Ha egy konkrét típust megfelelően írunk le, az azt használó programok méretben és sebességben összemérhetők lesznek azokkal a programokkal, melyekben a programozó egyénileg dolgozza ki a fogalom ábrázolását, majd annak egyedi célú (származtatott) változataival dolgozik. Ezenkívül, ha a megvalósítás jelentősen változik, rendszerint a felület is módosul, hogy tükrözze a változást. A konkrét típusok mindezekben a beépített típusokra emlékeztetnek. A beépített típusok természetesen mind konkrétak. A felhasználói konkrét típusok, mint a komplex számok, mátrixok, hibaüzenetek és szimbolikus hivatkozások, gyakran az egyes alkalmazási területek alaptípusaiként használtak.

Az adott osztály felületének természete határozza meg, mi minősül jelentős változtatásnak a megvalósításban; az elvontabb felületek ezen változtatásoknak több teret hagynak, de ronthatják a futási idejű hatékonyságot. Ezenkívül egy jó megvalósítás nem függ más osztályoktól a feltétlenül szükségesnél erősebben, így az osztály a programban lévő „hasonló" osztályokhoz való alkalmazkodás szükségességének elkerülésével fordítási vagy futási idejű túlterhelés nélkül használható.

Összegezve, egy konkrét típust leíró osztály céljai a következők:

1. Szoros illeszkedés egy konkrét fogalomhoz és a megvalósítás módjához.
2. Az „egyénileg kidolgozott" kódéhoz mérhető gyorsaság és kis méret, a helyben kifejtés, valamint a fogalom és megvalósítása összes előnyét kihasználó műveletek használatával.
3. A más osztályoktól való lehető legkisebb arányú függés.
4. Érthetőség és önálló használhatóság.

Az eredmény a felhasználói és a megvalósító kód közti szoros kötés. Ha a megvalósítás bármilyen módon megváltozik, a felhasználói kódot újra kell fordítani, mivel a felhasználói kód szinte mindig tartalmaz helyben kifejtett (inline) függvényhívásokat vagy a konkrét típushoz tartozó helyi (lokális) változókat.

A „konkrét típus" elnevezést az általánosan használt „absztrakt típus" elnevezés ellentéteként választottuk. A konkrét és absztrakt típusok közti viszonyt a §25.3 pont tárgyalja.

A konkrét típusok nem tudnak közvetlenül közösséget kifejezni. A *list* és a *vector* művelethalmaza például hasonló és néhány sablon függvényben egymást helyettesíthetik. A *list<int>* és a *vector<int>*, vagy a *list<Shape*>* és a *list<Circle*>* között azonban nincs rokonság (§13.6.3), jóllehet *mi* észrevesszük a hasonlóságokat.

Ez azt is jelenti, hogy az egyes konkrét típusokat hasonló módon használó kódok külsejükben különbözők lesznek. Egy *List* bejárása a *next()* művelettel például jelentősen különbözik egy *Vector* indexeléssel történő bejárásától:

```
void my(List& sl)
{
  for (T* p = sl.first(); p; p = sl.next()) {          // "természetes" lista-bejárás
            // egyik kód
  }
  // ...
}

void your(Vector& v)
{
  for (int i = 0; i<v.size(); i++) {          // "természetes" vektor-bejárás
            // másik kód
  }
  // ...
}
```

A bejárás stílusbeli különbsége természetes, abban az értelemben, hogy a „vedd a következő elemet" művelet nélkülözhetetlen a lista fogalmánál, de nem ilyen magától értetődő egy vektornál, illetve hogy az indexelés nélkülözhetetlen a vektor fogalmánál, de a listánál nem az. Az, hogy a választott megvalósítási módhoz a „természetes" műveletek rendelkezésre álljanak, általában létfontosságú, mind a hatékonyság szempontjából, mind azért, hogy a kód könnyen megírható legyen.

Magától értetődő nehézség, hogy az alapvetően hasonló műveletekhez (például az előbbi két ciklushoz) írott kód eltérően nézhet ki, a hasonló műveletekhez különböző konkrét típusokat használó kódok pedig nem cserélhetők fel. Valós programoknál fejtörést igényel, hogy megtaláljuk a hasonlóságokat, és jelentős újratervezést, hogy a megtalált hasonlóságok kihasználására módot adjunk. A szabványos tárolók és algoritmusok is így teszik lehetővé a konkrét típusok hasonlóságainak kihasználását, hatékonyságuk és „eleganciájuk" elvesztése nélkül (§16.2).

Egy függvénynek ahhoz, hogy paraméterként elfogadjon egy konkrét típust, pontosan az adott típust kell megadnia paramétertípusként. Nem állnak rendelkezésre öröklési kapcsolatok, melyeket arra használhatnánk, hogy a paraméterek deklarálását általánosabbá tegyük. Következésképpen a konkrét típusok közti hasonlóságok kihasználására tett kísérlet magával vonja a sablonok használatát és az általánosított (generikus) programozást, amint arról a §3.8 pontban már említést tettünk. Ha a standard könyvtárat használjuk, a bejárás így fog kinézni:

```
template<class C> void ours(const C& c)
{
  for (C::const_iterator p = c.begin(); p!=c.end(); ++p) {  // standard könyvtárbeli bejárás
            // ...
  }
}
```

Itt kihasználtuk a tárolók alapvető hasonlóságát, és ezzel megnyitottuk az utat a további hasonlóságok kihasználása felé, ahogyan azt a szabványos algoritmusok is teszik (18. fejezet).

A felhasználónak a konkrét típus megfelelő használatához meg kell értenie annak pontos részleteit. (Általában) nem léteznek olyan általános tulajdonságok, melyek egy könyvtárban az összes konkrét típusra érvényesek lennének, és amelyekre támaszkodva a felhasználónak nem kell az egyes osztályok ismeretével törődnie. Ez az ára a futási idejű tömörségnek és hatékonyságnak. Néha megéri, néha nem. Olyan eset is előfordul, hogy könnyebb megérteni és használni egy konkrét osztályt, mint egy általánosabb (absztrakt) osztályt. A jól ismert adattípusokat (pl. tömböket, listákat) ábrázoló osztályoknál gyakran ez a helyzet.

Vegyük észre azonban, hogy az ideális továbbra is az, ha a megvalósításból a lehető legnagyobb részt elrejtjük, anélkül, hogy a teljesítményt komolyabban rontanánk. A helyben kifejtett függvények ebben a környezetben nagy nyereséget jelenthetnek. Szinte soha nem jó ötlet, ha a tag változókat nyilvánossá tesszük vagy *set* és *get* függvényekről gondoskodunk, melyekkel a felhasználó közvetlenül kezelheti azokat (§24.4.2). A konkrét típusok maradjanak meg típusoknak, és ne „bitcsomagok” legyenek, melyeket a kényelem kedvéért néhány függvénnyel látunk el.

25.2.1. A konkrét típusok újrahasznosítása

A konkrét típusok ritkán használhatók alaptípusként további származtatáshoz. Minden konkrét típus célja egyetlen fogalom világos és hatékony ábrázolása. Az olyan osztály, mely ezt a követelményt jól teljesíti, ritkán alkalmas különböző, de egymással rokon osztályok nyilvános származtatás általi létrehozására. Az ilyen osztályok gyakrabban tehetők tagokká vagy privát bázisosztályokká, mert így anélkül használhatók hatékonyan, hogy felületüket és megvalósításukat keverni és rontani kellene új osztályokéval. Vegyük egy új osztály származtatását a *Date*-ből:

```
class My_date : public Date {
  // ...
};
```

Szabályos-e a *My_date*-et úgy használni, mint az egyszerű *Date*-et? Nos, ez attól függ, mi a *My_date*, de tapasztalatom szerint ritkán lehet olyan konkrét típust találni, mely módosítás nélkül megfelel bázisosztálynak.

A konkrét típusok ugyanúgy módosítás nélkül újrahasznosíthatók, mint az *int*-hez hasonló beépített típusok (§10.3.4):

```
class Date_and_time {
private:
  Date d;
  Time t;
public:
  // ...
};
```

A felhasználásnak (újrahasznosításnak) ez a módja rendszerint egyszerű és hatékony.

Lehet, hogy hiba volt a *Date*-et nem úgy tervezni, hogy származtatással könnyen lehessen módosítani? Néha azt mondják, *minden* osztálynak módosíthatónak kell lennie felülírás és származtatott osztályok tagfüggvényeiből való hozzáférés által. Ezt a szemléletet követve a *Date*-ből az alábbi változatot készíthetjük el:

```
class Date2 {
public:
  // nyilvános felület, elsősorban virtuális függvényekből áll
protected:
  // egyéb megvalósítási részletek (esetleg némi ábrázolást is tartalmaz)
private:
  // adatábrázolás és egyéb megvalósítási részletek
};
```

A felülíró függvények hatékony megírását megkönnyítendő, az ábrázolást *protected*-ként deklaráltuk. Ezzel elértük célunkat: a *Date2*-t származtatással tetszőlegesen „hajlíthatóvá" tettük, változatlanul hagyva felhasználói felületét. Ennek azonban ára van:

1. *Kevésbé hatékony alapműveletek.* A C++ virtuális függvényeinek meghívása kissé lassabb, mint a szokásos függvényhívások; a virtuális függvények nem fordíthatók helyben kifejtve olyan gyakran, mint a nem virtuális függvények; ráadásul a virtuális függvényekkel rendelkező osztályok egy gépi szóval több helyet igényelnek.
2. *A szabad tár használatának szükségessége.* A *Date2* célja, hogy a belőle származtatott osztályok objektumai felcserélhetőek legyenek. Mivel e származtatott

osztályok mérete különböző, kézenfekvő, hogy a szabad tárban foglaljunk számukra helyet és mutatókkal vagy referenciákkal férjünk hozzájuk. A valódi lokális változók használata tehát drámaian csökken.
3. *Kényelmetlenség a felhasználónak.* A virtuális függvények többalakúságát (polimorfizmus) kihasználandó, a *Date2*-höz mutatók vagy referenciák által kell hozzáférnünk.
4. *Gyengébb betokozás.* A virtuális műveletek felülírhatók és a védett adatok a származtatott osztályokból módosíthatók (§12.4.1.1).

Természetesen ezek a „költségek” nem mindig jelentősek és az így létrehozott osztály gyakran pontosan úgy viselkedik, mint ahogy szerettük volna (§25.3, §25.4). Egy egyszerű konkrét típusnál (mint a *Date2*) azonban ezek nagy és szükségtelen költségek.

A „hajlíthatóbb” típus ideális ábrázolására sokszor egy jól megtervezett konkrét típus a legalkalmasabb:

```
class Date3 {
public:
   // nyilvános felület, elsősorban virtuális függvényekből áll
private:
   Date d;
};
```

Így a konkrét típusok (a beépített típusokat is beleértve) osztályhierarchiába is illeszthetők, ha szükséges. (Lásd még §25.10[1].)

25.3. Absztrakt típusok

Az osztályokat, valamint az objektumokat létrehozó és az azokat felhasználó kód közötti kötelék lazításának legegyszerűbb módja egy absztrakt osztály bevezetése, mely a fogalom különböző megvalósításainak halmazához közös felületet biztosít. Vegyünk egy természetes *Set*-et (halmazt):

```
template<class T> class Set {
public:
   virtual void insert(T*) = 0;
   virtual void remove(T*) = 0;
```

```
  virtual int is_member(T*) = 0;

  virtual T* first() = 0;
  virtual T* next() = 0;

  virtual ~Set() { }
};
```

Ez meghatározza a halmaz felületét és tartalmazza az elemek bejárásának módját is. A konstruktor hiánya és a virtuális destruktor jelenléte szokványos (§12.4.2). Többféle megvalósítás lehetséges (§16.2.1):

```
template<class T> class List_set : public Set<T>, private list<T> {
  // ...
};

template<class T> class Vector_set : public Set<T>, private vector<T> {
  // ...
};
```

Az egyes megvalósításokhoz az absztrakt osztály adja a közös felületet, vagyis a *Set*-tel anélkül dolgozhatunk, hogy tudnánk, melyik megvalósítását használjuk:

```
void f(Set<Plane*>& s)
{
  for (Plane** p = s.first(); p; p = s.next()) {
          // saját kód
  }
  // ...
}

List_set<Plane*> sl;
Vector_set<Plane*> v(100);

void g()
{
  f(sl);
  f(v);
}
```

A konkrét típusoknál megköveteltük a megvalósító osztályok újratervezését, hogy kifejezzék a közösséget, annak kiaknázására pedig sablont használtunk. Itt a közös felületet kell megterveznünk (esetünkben a *Set*-et), de semmi más közöset nem kívánunk a megvalósításra használt osztályoktól, mint azt, hogy meg tudják valósítani a felületet.

Továbbá, a *Set*-et felhasználó programelemeknek nem kell ismerniük a *List_set* és a *Vector_set* deklarációit, tehát nem kell függniük ezektől. Ezenkívül nem kell sem újrafordítanunk, sem bármilyen más változtatást végeznünk, ha a *List_set* vagy a *Vector_set* megváltozik, vagy ha bevezetjük a *Set* egy újabb megvalósítását, mondjuk a *Tree_set*-et. Minden függést a függvények tartalmaznak, melyek a *Set*-ből származtatott osztályokat használják. Vagyis – feltételezve, hogy a szokásos módon használjuk a fejállományokat – az *f(Set&)* megírásakor csak a *Set.h*-t kell beépítenünk (*#include*), a *List_set.h*-t vagy a *Vector_set.h*-t nem. Csak ott van szükség egy „implementációs fejállományra", ahol egy *List_set*, illetve egy *Vector_set* létrehozása történik. Az egyes megvalósítások további elszigetelését jelentheti a tényleges osztályoktól, ha egy absztrakt osztályt vezetünk be, mely az objektum-létrehozási kéréseket kezeli („gyár", factory, §12.4.4).

A felületnek ez az elválasztása a megvalósítástól azzal a következménnyel jár, hogy „eltűnik" a hozzáférés azon műveletekhez, melyek „természetesek" egy adott megvalósítás számára, de nem elég általánosak ahhoz, hogy a felület részei legyenek. Például, mivel a *Set* nem rendelkezik a rendezés képességével, a *Set* felületében nem támogathatunk egy indexkezelő operátort még akkor sem, ha történetesen egy tömböt használó *Set*-et hozunk létre. A „kézi" optimalizálás hiánya miatt ez növeli a futási időt, ezenkívül általában nem élhetünk a helyben kifejtés előnyeivel sem (kivéve azokat az egyedi eseteket, amikor a fordító ismeri a valódi típust), és a felület minden lényeges művelete virtuális függvényhívás lesz. Mint a konkrét típusok, az absztrakt típusok használata is néha megéri; néha nem. Összefoglalva, az absztrakt típusok céljai a következők:

1. Egy egyszerű fogalom oly módon való leírása, mely lehetővé teszi, hogy a programban a fogalomnak több megvalósítása is létezhessen.
2. Elfogadható futási idő és takarékos helyfoglalás biztosítása virtuális függvények használatával.
3. Az egyes megvalósítások lehető legkisebb függése más osztályoktól.
4. Legyen érthető önmagában.

Az absztrakt típusok nem jobbak, mint a konkrét típusok, csak másmilyenek. A felhasználónak kell döntenie, melyiket használja. A könyvtár készítője elkerülheti a kérdést azáltal, hogy mind a kettőt biztosítja, a felhasználóra hagyva a választást. Az a fontos, hogy világos legyen, az adott osztály melyik típusba tartozik. Ha korlátozzuk egy absztrakt típus általánosságát, hogy sebességben állja a versenyt egy konkrét típussal, rendszerint kudarcot vallunk. Ez ugyanis veszélyezteti azt a képességét, hogy egyes megvalósításait egymással helyettesíthessük, a változtatás utáni újrafordítás szükségessége nélkül. Hasonlóképpen rendszerint kudarccal jár, ha konkrét típusokat „általánosabbá" akarunk tenni, hogy az absztrakt típusok tulajdonságaival összemérhetők legyenek, mert így veszélybe kerül az

egyszerű osztály hatékonysága és pontossága. A két elképzelés együtt létezhet – valójában együtt *kell*, hogy létezzen, mivel az absztrakt típusok megvalósítását konkrét típusok adják –, de nem szabad őket összekeverni egymással.

Az absztrakt típusok gyakran közvetlen megvalósításukon túl nem szolgálnak további származtatások alapjául. Új felület azonban felépíthető egy absztrakt osztályból, ha belőle egy még tágabb absztrakt osztályt származtatunk. Ezt az új absztrakt osztályt kell majd további származtatáson keresztül megvalósítani egy nem absztrakt osztállyal (§15.2.5).

Miért nem származtattunk a *Set*-ből egy lépésben *List* és *Vector* osztályokat, elhagyva a *List_set* és *Vector_set* osztályok bevezetését? Más szóval, miért legyenek konkrét típusaink, ha absztrakt típusaink is lehetnek?

1. *Hatékonyság.* Azt akarjuk, hogy konkrét típusaink legyenek (mint a *vector* és a *list*), azon túlterhelés nélkül, amit a megvalósításnak a felülettől való elválasztása okoz (az absztrakt típusoknál ez történik).
2. *Újrahasznosítás.* Szükségünk van egy módra, hogy a „máshol” tervezett típusokat (mint a *vector* és a *list*) beilleszthessük egy új könyvtárba vagy alkalmazásba, azáltal, hogy új felületet adunk nekik (nem pedig újraírjuk őket).
3. *Több felület.* Ha egyetlen közös alapot használunk többféle osztályhoz, az kövér felületekhez vezet (§24.4.3). Gyakran jobb, ha az új célra használni kívánt osztálynak új felületet adunk (mint egy *vector*-nak egy *Set* felületet), ahelyett, hogy a több célra való használhatóság kedvéért az eredeti felületet módosítanánk.

Természetesen ezek egymással összefüggő kérdések. Az *Ival_box* példában (§12.4.2, §15.2.5) és a tárolók tervezésével kapcsolatban (§16.2) részletesebben tárgyaltuk ezeket. Ha a *Set* bázisosztályt használtuk volna, az egy csomópont-osztályokon nyugvó alaptárolót eredményezett volna (§25.4).

A §25.7 pont egy rugalmasabb bejárót (iterátort) ír le, ahol a bejáró a kezdeti értékadásnál köthető az objektumokat biztosító megvalósításhoz és ez a kötés futási időben módosítható.

25.4. Csomópont-osztályok

Az osztályhierarchiák más származtatási szemlélet alapján épülnek fel, mint az absztrakt osztályoknál használt, felületből és megvalósításból álló rendszer. Itt az osztályokat alapnak tekintjük, melyre más osztályokat építünk. Még ha absztrakt osztály is, rendszerint van valamilyen ábrázolása és ad valamilyen szolgáltatásokat a belőle származtatott osztályoknak. Ilyen csomópont-osztályok például a *Polygon* (§12.3), a (kiinduló) *Ival_slider* (§12.4.1) és a *Satellite* (§15.2).

A hierarchián belüli osztályok jellemzően egy általános fogalmat ábrázolnak, míg a belőlük származtatott osztályok úgy tekinthetők, mint az adott fogalom konkretizált (egyedi célú, „szakosított") változatai. Az egy hierarchia szerves részeként tervezett osztályok – a *csomópont-osztályok* (node class) – a bázisosztály szolgáltatásaira támaszkodva biztosítják saját szolgáltatásaikat, vagyis a bázisosztály tagfüggvényeit hívják meg. A csomópont-osztályok általában nem csupán a bázisosztályuk által meghatározott felület egy megvalósítását adják (mint egy absztrakt típusnak egy megvalósító osztály), hanem maguk is hozzátesznek új függvényeket, ezáltal szélesebb felületet biztosítanak. Vegyük a §24.3.2 forgalomszimulációs példájában szereplő *Car*-t:

```
class Car : public Vehicle {
public:
   Car(int passengers, Size_category size, int weight, int fc)
            : Vehicle(passengers,size,weight), fuel_capacity(fc) { /* ... */ }

   // a Vehicle lényeges virtuális függvényeinek felülírása

   void turn(Direction);
   // ...

   // Car-ra vonatkozó függvények hozzáadása

   virtual void add_fuel(int amount);   // az autó üzemanyagot igényel
   // ...
};
```

A fontos függvények: a konstruktor, mely által a programozó a szimuláció szempontjából lényeges alaptulajdonságokat határozza meg és azok a (virtuális) függvények, melyek a szimulációs eljárásoknak lehetővé teszik, hogy egy *Car*-t anélkül kezeljenek, hogy tudnák annak pontos típusát. Egy *Car* az alábbi módon hozható létre és használható:

```
void user()
{
  // ...
  Car* p = new Car(3,economy,1500,60);
  drive(p,bs_home,MH);        // belépés a szimulált forgalmi helyzetbe
  // ...
}
```

A csomópont-osztályoknak rendszerint szükségük van konstruktorokra és ez a konstruktor gyakran bonyolult. Ebben a csomópont-osztályok eltérnek az absztrakt osztályoktól, melyeknek ritkán van konstruktoruk.

A *Car* műveleteinek megvalósításai általában a *Vehicle* bázisosztályból vett műveleteket használják, a *Car*-okat használó elemek pedig a bázisosztályok szolgáltatásaira támaszkodnak. A *Vehicle* például megadja a súllyal és mérettel foglalkozó alapfüggvényeket, így a *Car*-nak nem kell gondoskodnia azokról:

```
bool Bridge::can_cross(const Vehicle& r)
{
  if (max_weight<r.weight()) return false;
  // ...
}
```

Ez lehetővé teszi a programozó számára, hogy új osztályokat (*Car* és *Truck*) hozzon létre a *Vehicle* csomópont-osztályból, úgy, hogy csak az attól különböző tulajdonságokat és műveleteket kell megadnia, illetve elkészítenie. Ezt az eljárást gyakran úgy említik, mint „különbség általi programozást" vagy „bővítő programozást".

Sok csomópont-osztályhoz hasonlóan a *Car* maga is alkalmas a további származtatásra. Az *Ambulance*-nek például további adatokra és műveletekre van szüksége a vészhelyzetek kezeléséhez:

```
class Ambulance : public Car, public Emergency {
public:
  Ambulance();

   // a Car lényeges virtuális függvényeinek felülírása

  void turn(Direction);
  // ...

   // az Emergency lényeges virtuális függvényeinek felülírása
```

```
    virtual void dispatch_to(const Location&);
    // ...

    // Ambulance-ra vonatkozó függvények hozzáadása

    virtual int patient_capacity();          // hordágyak száma
    // ...
};
```

Összegezzük a csomópont-osztályok jellemzőit:

1. Mind megvalósítását, mind a felhasználóknak adott szolgáltatásokat tekintve bázisosztályaira támaszkodik.
2. Szélesebb felületet (vagyis több nyilvános tagfüggvénnyel rendelkező felületet) ad felhasználóinak, mint bázisosztályai.
3. Elsődlegesen (de nem feltétlenül kizárólagosan) a nyilvános felületében levő virtuális függvényekre támaszkodik.
4. Függ az összes (közvetlen és közvetett) bázisosztályától.
5. Csak bázisosztályaival összefüggésben érthető meg.
6. További származtatás bázisosztályaként felhasználható.
7. Objektumok létrehozására használható.

Nem minden csomópont-osztály fog eleget tenni az 1, 2, 6 és 7 mindegyikének, de a legtöbb igen. A 6-nak eleget nem tevő osztályok a konkrét típusokra emlékeztetnek, így *konkrét csomópont-osztályoknak* nevezhetők. A konkrét csomópont-osztályok például absztrakt osztályok megvalósításához használhatók (§12.4.2), az ilyen osztályok változói számára pedig statikusan és a veremben is foglalhatunk helyet. Az ilyen osztályokat néha *levél osztályoknak* nevezzük. Ne feledjük azonban, hogy minden kód, amely egy osztályra hivatkozó mutatón vagy referencián virtuális függvények által végez műveletet, számításba kell, hogy vegye egy további osztály származtatásának lehetőségét (vagy nyelvi támogatás nélkül el kell fogadnia, hogy nem történt további származtatás). Azok az osztályok, melyek nem tesznek eleget a 7-es pontnak, az absztrakt típusokra emlékeztetnek, így *absztrakt csomópont-osztályoknak* hívhatók. A hagyományok miatt sajnos sok csomópont-osztálynak van legalább néhány *protected* tagja, hogy a származtatott osztályok számára kevésbé korlátozott felületről gondoskodjon (§12.4.1.1).

A 4-es pontból az következik, hogy a csomópont-osztályok fordításához a programozónak be kell építenie (*#include*) azok összes közvetlen és közvetett bázisosztály-deklarációit és az összes olyan deklarációt, melyektől azok függnek. Ez megint csak egy különbség a csomópont-osztályok és az absztrakt típusok között, az utóbbiakat használó programelemek ugyanis nem függnek a megvalósításukra használt osztályoktól, így nem kell azokat a fordításhoz beépíteni.

25.4.1. Felületek módosítása

Minden csomópont-osztály egy osztályhierarchia tagja, egy hierarchián belül viszont nem minden osztálynak kell ugyanazt a felületet nyújtania. Nevezetesen, egy származtatott osztály több tagfüggvénnyel rendelkezhet és egy testvérosztály függvényhalmaza is teljesen más lehet. A tervezés szempontjából a *dynamic_cast* (§15.4) úgy tekinthető, mint az az eljárás, mellyel egy objektumot megkérdezhetünk, biztosít-e egy adott felületet.

Példaként vegyünk egy egyszerű objektum I/O (bemeneti/kimeneti) rendszert. A felhasználó egy adatfolyamból objektumokat akar olvasni, meg kívánja határozni, hogy a várt típusúak-e, majd fel akarja használni azokat:

```
void user()
{
   // ... feltételezés szerint alakzatokat (Shape) tartalmazó fájl megnyitása
   //     ss által azonosított bemeneti adatfolyamként ...

   Io_obj* p = get_obj(ss);         // objektum olvasása az adatfolyamról

   if (Shape* sp = dynamic_cast<Shape*>(p)) {
            sp->draw();             // alakzat használata
            // ...
   }
   else {
                                    // hoppá: ez nem alakzat
   }
}
```

A *user()* függvény az alakzatokat kizárólag az absztrakt *Shape* osztályon keresztül kezeli és ezáltal minden fajta alakzatot képes használni. A *dynamic_cast* használata lényeges, mert az objektum I/O rendszer számos objektumfajtát kezel, a felhasználó pedig véletlenül olyan fájlt is megnyithatott, amely olyan osztályok – egyébként tökéletesen jó – objektumait tartalmazza, melyekről a felhasználó nem is hallott.

A rendszer feltételezi, hogy minden olvasott vagy írt objektum olyan osztályhoz tartozik, amely az *Io_obj* leszármazottja. Az *Io_obj* osztálynak többalakúnak (polimorfnak) kell lennie, hogy lehetővé tegye számunkra a *dynamic_cast* használatát:

```
class Io_obj {
public:
   virtual Io_obj* clone() const =0;     // többalakú
   virtual ~Io_obj() {}
};
```

A rendszer létfontosságú eleme a *get_obj()* függvény, mely egy *istream* adatfolyamból adatokat olvas és osztályobjektumokat hoz létre azokra alapozva. Tegyük fel, hogy a bemeneti adatfolyamban az egyik objektumot képviselő adatok előtagként egy, az objektum osztályát azonosító karakterláncot tartalmaznak. A *get_obj()* függvény feladata ezt elolvasni, majd egy olyan függvényt meghívni, amely képes a megfelelő osztályú objektumot létrehozni és kezdőértéket adni neki:

```
typedef Io_obj* (*PF)(istream&);    // Io_obj* visszatérési típusú függvényre hivatkozó
mutató

map<string,PF> io_map;              // karakterláncok hozzárendelése a létrehozó függvényekhez

bool get_word(istream& is, string& s); // szó beolvasása is-ből s-be

Io_obj* get_obj(istream& s)
{
  string str;
  bool b = get_word(s,str);          // a kezdő szó beolvasása str-be
  if (b == false) throw No_class();  // io formátumhiba

  PF f = io_map[str];                // az "str" kikeresése, hogy kiválasszuk a függvényt
  if (f == 0) throw Unknown_class(); // nem találjuk "str"-t

  return f(s);                       // objektum létrehozása az adatfolyamból
}
```

Az *io_map* nevű *map* neveket tartalmazó karakterláncok és a nevekhez tartozó osztályok objektumait létrehozni képes függvények párjaiból áll.

A *Shape* osztályt a szokásos módon határozhatjuk meg, azzal a kivétellel, hogy a *user()* által megkövetelten az *Io_obj*-ból származtatjuk:

```
class Shape : public Io_obj {
  // ...
};
```

Még érdekesebb (és sok esetben valószerűbb) lenne azonban egy meghatározott *Shape*-et (§2.6.2) módosítás nélkül felhasználni:

```
class Io_circle : public Circle, public Io_obj {
public:
  Io_circle* clone() const { return new Io_circle(*this); } // másoló konstruktor használata
  Io_circle(istream&);          // kezdeti értékadás a bemeneti adatfolyamból
```

```
  static Io_obj* new_circle(istream& s) { return new Io_circle(s); }
  // ...
};
```

Ez annak példája, hogyan lehet egy osztályt egy absztrakt osztály használatával beilleszteni egy hierarchiába, ami kevesebb „előrelátást" igényel, mint amennyi ahhoz kellene, hogy először csomópont-osztályként hozzuk létre (§12.4.2, §25.3).

Az *Io_circle(istream&)* konstruktor az objektumokat az *iostream*-ből kapott adatokkal tölti fel. A *new_circle()* függvényt az *io_map*-ba tesszük, hogy az osztályt a bemeneti/kimeneti rendszer számára ismertté tegyük:

```
io_map["Io_circle"]=&Io_circle::new_circle;
```

Más alakzatokat hasonló módon hozhatunk létre:

```
class Io_triangle : public Triangle, public Io_obj {
  // ...
};
```

Ha az objektum I/O rendszer elkészítése túl fáradságos, segíthet egy sablon:

```
template<class T> class Io : public T, public Io_obj {
public:
  Io* clone() const { return new Io(*this); }     // felülbírálja Io_obj::clone()-t

  Io(istream&);     // kezdeti értékadás bemeneti adatfolyamból

  static Io* new_io(istream& s) { return new Io(s); }
  // ...
};
```

Ha a fenti adott, meghatározhatjuk az *Io_circle*-t:

```
typedef Io<Circle> Io_circle;
```

Természetesen továbbra is pontosan definiálnunk kell az *Io<Circle>::Io(istream&)*-et, mivel annak részleteiben ismernie kell a *Circle*-t. Az *Io* sablon példa arra, hogyan illeszthetünk be konkrét típusokat egy osztályhierarchiába egy leíró (handle) segítségével, amely az adott hierarchia egy csomópontja. A sablon saját paraméteréből származtat, hogy lehetővé tegye az átalakítást az *Io_obj*-ról, de ez sajnos kizárja az *Io*-nak beépített típusokra történő alkalmazását:

```
typedef Io<Date> Io_date;      // konkrét típus beburkolása
typedef Io<int> Io_int;        // hiba: nem származtathatunk beépített típusból
```

A probléma úgy kezelhető, ha a beépített típusok részére külön sablont biztosítunk vagy ha egy beépített típust képviselő osztályt használunk (§25.10[1]).

Ez az egyszerű objektum I/O rendszer nem teljesíthet minden kívánságot, de majdnem elfér egyetlen lapon és fő eljárásait számos célra felhasználhatjuk, például meghívhatjuk velük a felhasználó által karakterlánccal megadott függvényeket vagy ismeretlen típusú objektumokat kezelhetünk futási idejű típusazonosítással felderített felületen keresztül.

25.5. Műveletek

A C++-ban a műveletek meghatározásának legegyszerűbb és legkézenfekvőbb módja függvények írása. Ha azonban egy adott műveletet végrehajtás előtt késleltetni kell, át kell vinni „máshová", párosítani kell más műveletekkel vagy a művelet saját adatokat igényel (§25.10 [18, 19]), gyakran célszerűbb azt osztály alakjában megadni, mely végre tudja hajtani a kívánt eljárást és egyéb szolgáltatásokat is nyújthat. Jó példák erre a szabványos algoritmusok által használt függvény-objektumok (§18.4), valamint az *iostream*-mel használt módosítók (§21.4.6). Az előbbi esetben a tényleges műveletet a függvényhívó operátor hajtja végre, míg az utóbbinál a << vagy a >> operátor. A *Form* (§21.4.6.3) és a *Matrix* (§22.4.7) esetében a végrehajtás késleltetésére egyedi (compositor) osztályokat használunk, amíg a hatékony végrehajtáshoz elegendő információ össze nem gyűlik.

A műveletosztályok általános formája egy (jellemzően valamilyen „csináld" (do_it) nevű) virtuális függvényt tartalmazó egyszerű osztály:

```
class Action {
public:
   virtual int do_it(int) = 0;
   virtual ~Action() { }
};
```

Ha ez adott, olyan kódot írhatunk – mondjuk egy menüét – amely a műveleteket későbbi végrehajtásra „elraktározhatja", anélkül, hogy függvénymutatókat használna, bármit is tudna a meghívott objektumokról vagy egyáltalán ismerné a meghívott művelet nevét:

```
class Write_file : public Action {
  File& f;
public:
  int do_it(int) { return f.write().succeed(); }
};

class Error_response : public Action {
  string message;
public:
  int do_it(int);
};

int Error_response::do_it(int)
{
  Response_box db(message.c_str(), "Folytat","Mégse","Ismét");

  switch (db.get_response()) {
  case 0:
          return 0;
  case 1:
          abort();
  case 2:
          current_operation.redo();
          return 1;
  }
}

Action* actions[] = {
  new Write_file(f),
  new Error_response("Megint elrontotta"),
  // ...
};
```

Az *Action*-t használó kód teljesen elszigetelhető, nem kell, hogy bármit is tudjon az olyan származtatott osztályokról, mint a *Write_file* és az *Error_response*.

Ez igen erőteljes módszer, mellyel óvatosan kell bánniuk azoknak, akik leginkább a funkcionális elemekre bontásban szereztek tapasztalatot. Ha túl sok osztály kezd az *Action*-re hasonlítani, lehet, hogy a rendszer átfogó terve túlzottan műveletközpontúvá vált.

Megemlítendő, hogy az osztályok jövőbeni használatra is tárolhatnak műveleteket, illetve olyanokat is tartalmazhatnak, melyeket egy távoli gép hajt majd végre (§25.10 [18]).

25.6. Felületosztályok

Az egyik legfontosabb osztályfajta a szerény és legtöbbször elhanyagolt felületosztály. Egy felületosztály nem sok dolgot csinál – ha csinálna, nem lenne felületosztály –, csupán helyi szükségletekhez igazítja egy szolgáltatás megjelenését. Mivel elvileg lehetetlen mindig, minden szükségletet egyformán jól kielégíteni, a felületosztályok nélkülözhetetlenek, mert anélkül teszik lehetővé a közös használatot, hogy minden felhasználót egyetlen, közös „kényszerzubbonyba" erőszakolnának.

A felületek legtisztább formájukban még gépi kódot sem igényelnek. Vegyük a §13.5-ből a *Vector* specializált változatát:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
  typedef Vector<void*> Base;

  Vector() : Base() {}
  Vector(int i) : Base(i) {}

  T*& operator[](int i) { return static_cast<T*&>(Base::operator[](i)); }

  // ...
};
```

Ez a (részlegesen) specializált változat a nem biztonságos *Vector<void*>*-ot egy sokkal használhatóbb, típusbiztos vektorosztály-családdá teszi. A helyben kifejtett (inline) függvények gyakran nélkülözhetetlenek ahhoz, hogy a felületosztályokat megengedhessük magunknak. A fentihez hasonló esetekben, ahol a helyben kifejtett közvetítő függvény csak típusigazítást végez, sem a futási idő, sem a tárigény nem nő.

Természetesen az absztrakt típust ábrázoló absztrakt bázisosztályok, melyeket konkrét típusok (§25.2) valósítanak meg, szintén a felületosztályok egy formáját jelentik, mint ahogy a §25.7 leírói is azok. Itt azonban azon osztályokra összpontosítunk, melyeknek a felület igazításán kívül nincs más feladatuk.

Vegyük két, többszörös öröklést használó hierarchia összefésülésének problémáját. Mit lehet tenni, ha névütközés lép fel, vagyis két osztály ugyanazt a nevet használja teljesen eltérő műveleteket végző virtuális függvényekhez? Vegyünk például egy vadnyugati videojátékot, melyben a felhasználói beavatkozásokat egy általános ablakosztály kezeli:

```
class Window {
  // ...
  virtual void draw();          // kép megjelenítése
};

class Cowboy {
  // ...
  virtual void draw();          // pisztoly előrántása a pisztolytáskából
};

class Cowboy_window : public Cowboy, public Window {
  // ...
};
```

A játékban a cowboy mozgatását egy *Cowboy_window* képviseli és ez kezeli a felhasználó (játékos) és a cowboy figura közötti „kölcsönhatásokat" is. A *Window* és a *Cowboy* tagokként való bevezetése helyett többszörös öröklést szeretnénk használni, mivel számos kiszolgáló függvényt kell meghatároznunk mind a *Window*-k, mind a *Cowboy*-ok részére, a *Cowboy_window*-kat pedig olyan függvényeknek szeretnénk átadni, melyek nem kívánnak a programozótól különleges munkát. Ez azonban ahhoz a problémához vezet, hogy meg kell adni a *Cowboy::draw()* és *Window::draw()* függvények *Cowboy_window* változatát.

A *Cowboy_window*-ban csak egy *draw()* nevű függvény lehet. Ennek viszont – mivel a *Window*-kat és *Cowboy*-okat a *Cowboy_window*-k ismerete nélkül kezeljük – felül kell írnia mind a *Cowboy*, mind a *Window draw()* függvényét. Egyetlen változattal nem írhatjuk felül mind a kettőt, mert a közös név ellenére a két *draw()* függvény szerepe más. Végezetül azt is szeretnénk, hogy a *Cowboy_window* egyedi, egyértelmű neveket biztosítson az örökölt *Cowboy::draw()* és *Window:: draw()* függvények számára.

A probléma megoldásához szükségünk van egy-egy további osztályra a *Cowboy*-hoz és a *Window*-hoz. Ezek az osztályok vezetik be a két új nevet a *draw()* függvényekre és biztosítják, hogy a *Cowboy*-ban és a *Window*-ban a *draw()* függvényhívások az új néven hívják a függvényeket:

```
class CCowboy : public Cowboy {      // felület a Cowboy-hoz: a draw()-t átnevezi
public:
  virtual int cow_draw() = 0;
  void draw() { cow_draw(); }        // felülírja Cowboy::draw()-t
};

class WWindow : public Window {      // felület a Window-hoz: a draw()-t átnevezi
public:
  virtual int win_draw() = 0;
  void draw() { win_draw(); }        // felülírja Window::draw()-t
};
```

Most már összerakhatunk egy *Cowboy_window*-t a *CCowboy* és a *WWindow* felületosztályokból és a kívánt hatással felülbírálhatjuk a *cow_draw()*-t és *win_draw()*-t:

```
class Cowboy_window : public CCowboy, public WWindow {
  // ...
  void cow_draw();
  void win_draw();
};
```

Vegyük észre, hogy a probléma csak amiatt volt komoly, hogy a két *draw()* függvénynek ugyanaz a paramétertípusa. Ha a paramétertípusok különböznek, a szokásos túlterhelés-feloldási szabályoknak köszönhetően nem lesz probléma, annak ellenére, hogy az egymással kapcsolatban nem lévő függvényeknek ugyanaz a nevük.

Egy felületosztály minden használatára elképzelhetünk egy különleges célú nyelvi bővítést, mely a kívánt igazítást hatékonyabban vagy elegánsabban tudná elvégezni. A felületosztályok egyes használatai azonban ritkák, és ha mindet egyedi nyelvi szerkezettel támogatnánk, túlzottan bonyolulttá tennénk programjainkat. Az osztályhierarchiák összefésüléséből eredő névütközések nem általánosak (ahhoz képest, hogy milyen gyakran ír osztályt egy programozó), inkább abból erednek, hogy eltérő környezetben létrehozott hierarchiákat olvasztunk össze (pl. játékokat és ablakrendszereket). Az ilyen eltérő hierarchiákat nem könnyű összefésülni és a névütközések feloldása gyakran csak csepp a tengerben a programozó számára. Problémát jelenthet az eltérő hibakezelés, kezdeti értékadás és tárkezelési mód is. A névütközések feloldását itt azért tárgyaljuk, mert a közvetítő/továbbító szerepet betöltő felületosztályok bevezetése más területeken is alkalmazható; nem csak nevek megváltoztatására, hanem paraméterek és visszatérési értékek típusának módosítására, futási idejű ellenőrzés bevezetésére stb. is. Minthogy a *CCowboy::draw()* és *WWindow::draw()* függvények virtuálisak, nem optimalizálhatók egyszerű helyben kifejtéssel. Lehetséges viszont, hogy a fordító felismeri, hogy ezek egyszerű továbbító függvények, és a rajtuk átmenő hívási láncok alapján képes optimális kódot készíteni.

25.6.1. Felületek igazítása

A felületfüggvények egyik fő felhasználása egy felület igazítása úgy, hogy az jobban illeszkedjék a felhasználó elvárásaihoz; vagyis egyetlen felületbe helyezzük azt a kódot, amely különben a felhasználói kódon belül szét lenne szórva. A szabványos *vector* például nulla alapú, azaz az első elem indexe 0. Azoknak a felhasználóknak, akik a *0*-tól *size-1*-ig terjedő tartománytól eltérőt akarnak, igazítást kell végezniük:

```
void f()
{
  vector v<int>(10);          // [0:9] tartomány

  // úgy teszünk, mintha v a [1:10] tartományban lenne

  for (int i = 1; i<=10; i++) {
          v[i-1] = 7;         // ne felejtsük az indexet igazítani
          // ...
  }
}
```

Még jobb, ha a *vector*-nak tetszőleges határokat biztosítunk:

```
class Vector : public vector<int> {
  int lb;
public:
  Vector(int low, int high) : vector<int>(high-low+1) { lb=low; }

  int& operator[](int i) { return vector<int>::operator[](i-lb); }

  int low() { return lb; }
  int high() { return lb+size()-1; }
};
```

A *Vector* az alábbi módon használható:

```
void g()
{
  Vector v(1,10);             // [1:10] tartomány

  for (int i = 1; i<=10; i++) {
          v[i] = 7;
          // ...
  }
}
```

Ez az előző példához képest nem okoz többletterhelést. Világos, hogy a *Vector* változatot könnyebb olvasni és kisebb a hibázás lehetősége is.

A felületosztályok rendszerint meglehetősen kicsik és kevés feladatot végeznek, de mindenütt felbukkannak, ahol különböző hagyományok szerint megírt programoknak kell együttműködniük, mivel ilyenkor a különböző szabályok között közvetítésre van szükség.

A felületosztályokat gyakran használják például arra, hogy nem C++ kódnak C++ felületet adjanak, és az alkalmazás kódját elszigeteljék a könyvtárakétól (nyitva hagyva egy könyvtár másikkal való helyettesítésének a lehetőségét).

A felületosztályok másik fontos felhasználási területe az ellenőrzött vagy korlátozott felületek biztosítása. Nem szokatlan például, ha olyan egész típusú változóink vannak, melyek értéke csak egy adott tartományon belül mozoghat. Ez (futási időben) egy egyszerű sablonnal kényszeríthető ki:

```
template<int low, int high> class Range {
   int val;
public:
   class Error { };     // kivételosztály

   Range(int i) { Assert<Error>(low<=i&&i<high); val = i; }                    // lásd §24.3.7.2
   Range operator=(int i) { return *this=Range(i); }

   operator int() { return val; }
   // ...
};

void f(Range<2,17>);
void g(Range<-10,10>);


void h(int x)
{
   Range<0,2001> i = x;        // Range::Error kivételt válthat ki
   int i1 = i;

   f(3);
   f(17);                      // Range::Error kivételt vált ki
   g(-7);
   g(100);                     // Range::Error kivételt vált ki
}
```

A *Range* sablon könnyen bővíthető úgy, hogy tetszőleges skalár típusú tartományok kezelésére legyen képes (§25.10[7]).

Azokat a felületosztályokat, amelyek a más osztályokhoz való hozzáférést ellenőrzik vagy azok felületét igazítják, néha *beburkoló* (csomagoló, wrapper) osztályoknak nevezzük.

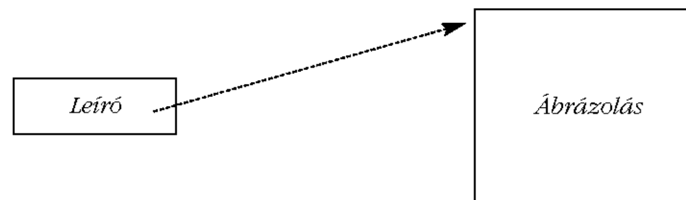
25.7. Leíró osztályok

Az absztrakt típusok hatékony elválasztást biztosítanak a felületek és megvalósításaik között, de az absztrakt típus által adott felület és annak egy konkrét típus által nyújtott megvalósítása közötti kapcsolat – ahogyan a §25.3-ban használtuk – tartós. Nem lehet például egy absztrakt bejárót az egyik forrásról (mondjuk egy halmazról) átkötni egy másikra (mondjuk egy adatfolyamra), ha az eredeti forrás kimerült.

Továbbá, hacsak nem mutatókon vagy referenciákon keresztül kezelünk egy absztrakt osztályt képviselő objektumot, elveszítjük a virtuális függvények előnyeit. A felhasználói kód függni fog a megvalósító osztályoktól, mivel az absztrakt típusok számára nem foglalható hely statikusan vagy a veremben (beleértve az érték szerinti paraméterátvételt is), anélkül, hogy tudnánk a típus méretét. A mutatók és hivatkozások használata azzal jár, hogy a tárkezelés terhe a felhasználó kódra hárul.

Az absztrakt osztályos megközelítés másik korlátja az, hogy az osztályobjektumok rögzített méretűek. Az osztályokat azonban fogalmak ábrázolására használjuk, melyek megvalósításához különböző mennyiségű tárterület kell.

E kérdések kezelésének kedvelt módja két részre osztani azt, amit egyetlen objektumként használunk. Az egyik lesz a felhasználói felületet adó *leíró* (handle), a másik pedig az ábrázolás, amely az objektum állapotának egészét vagy annak java részét tárolja. A leíró és az ábrázolás közötti kapcsolatot általában egy, a leíróban levő mutató biztosítja. A leíróban az ábrázolásra hivatkozó mutatón kívül általában még van egy-két adat, de nem sok. Ebből következik, hogy a leíró szerkezete rendszerint stabil (még akkor is, ha az ábrázolás megváltozik), illetve hogy a leírók meglehetősen kicsik és viszonylag szabadon mozgathatók, így a felhasználónak nem kell mutatókat és referenciákat használnia.



A §11.12 *String*-je a leíró egyszerű példája. A leíró felületet, hozzáférés-szabályozást és tárkezelést biztosít az ábrázolás részére. Ebben az esetben mind a leíró, mind az ábrázolás konkrét típusok, bár az ábrázoló osztály többnyire absztrakt osztály szokott lenni.

Vegyük a §25.3-ból a *Set* absztrakt típust. Hogyan gondoskodhatunk számára egy leíróról és milyen előnyökkel, illetve hátrányokkal járhat ez? Egy adott halmazosztályhoz egyszerűen megadhatunk egy leírót, a -> operátor túlterhelésével:

```
template<class T> class Set_handle {
  Set<T>* rep;
public:
  Set<T>* operator->() { return rep; }

  Set_handle(Set<T>* pp) : rep(pp) { }
};
```

Ez nem befolyásolja jelentősen a *Set*-ek használatának módját; egyszerűen *Set_handle*-eket adunk át *Set&*-ek vagy *Set**-ok helyett:

```
void f(Set_handle<int> s)
{
  for (int* p = s->first(); p; p = s->next())
  {
          // ...
  }
}

void user()
{
  Set_handle<int> sl(new List_set<int>);
  Set_handle<int> v(new Vector_set<int>(100));

  f(sl);
  f(v);
}
```

Gyakran többet kívánunk egy leírótól, mint hogy a hozzáférésről gondoskodjon. Például, ha a *Set* osztályt és a *Set_handle* osztályt együtt tervezték, a hivatkozásokat könnyen megszámlálhatjuk, ha minden *Set*-ben elhelyezünk egy használatszámlálót. Persze a leírót általában nem akarjuk együtt tervezni azzal, aminek a leírója, így önálló objektumban kell tárolnunk minden adatot, amit a leírónak biztosítania kell. Más szóval, szükségünk lenne nem tolakodó (non-intensive) leírókra is a tolakodók mellett. Íme egy leíró, mely felszámol egy objektumot, amikor annak utolsó leírója is megsemmisül:

```
template<class X> class Handle {
  X* rep;
  int* pcount;
public:
  X* operator->() { return rep; }

  Handle(X* pp) : rep(pp), pcount(new int(1)) { }
  Handle(const Handle& r) : rep(r.rep), pcount(r.pcount) { (*pcount)++; }

  Handle& operator=(const Handle& r)
  {
          if (rep == r.rep) return *this;
          if (--(*pcount) == 0) {
                  delete rep;
                  delete pcount;
          }
          rep = r.rep;
          pcount = r.pcount;
          (*pcount)++;
          return *this;
  }

  ~Handle() { if (--(*pcount) == 0) { delete rep; delete pcount; } }

  // ...
};
```

Egy ilyen leíró szabadon átadható:

```
void f1(Handle<Set>);

Handle<Set> f2()
{
  Handle<Set> h(new List_set<int>);
  // ...
  return h;
}

void g()
{
  Handle<Set> hh = f2();
  f1(hh);
  // ...
}
```

Itt az *f2()*-ben létrehozott halmaz törlődik a *g()*-ből való kilépéskor – hacsak *f1()* fenn nem tartja annak egy másolatát. (A programozónak erről nem kell tudnia.)

Természetesen ennek a kényelemnek ára van, de a használatszámláló tárolásának és fenntartásának költsége a legtöbb alkalmazásnál elfogadható.

Néha hasznos az ábrázolásra hivatkozó mutatót a leíróból kinyerni és közvetlenül felhasználni. Erre akkor lehet szükség, ha egy objektumot egy olyan függvénynek kell átadni, amely nem ismeri a leírókat. Ez a megoldás jól működik, feltéve, hogy a hívott függvény nem semmisíti meg a neki átadott objektumot vagy egy arra hivatkozó mutatót nem tárol a hívóhoz való visszatérés utáni használatra. Egy olyan művelet szintén hasznos lehet, amely a leírót egy új ábrázoláshoz kapcsolja:

```
template<class X> class Handle {
  // ...

  X* get_rep() { return rep; }

  void bind(X* pp)
  {
        if (pp != rep) {
              if (--*pcount == 0) {
                    delete rep;
                    *pcount = 1;              // pcount újrahasznosítása
              }
              else
                    pcount = new int(1);      // új pcount
              rep = pp;
        }
  }
};
```

Vegyük észre, hogy *Handle*-ből új osztályokat származtatni nem különösebben hasznos, ez ugyanis egy konkrét típus, virtuális függvények nélkül. Az alapelv, hogy egy bázisosztály által meghatározott teljes osztálycsaládhoz egyetlen leíró osztályunk legyen. Ebből a bázisosztályból származtatni viszont már hasznos lehet. Ez a csomópont-osztályokra éppúgy érvényes, mint az absztrakt típusokra.

Fenti formájában a *Handle* nem foglalkozik örökléssel. Ahhoz, hogy egy olyan osztályunk legyen, mely úgy működik, mint egy valódi használatszámláló, a *Handle*-t együtt kell használni a §13.6.3.1 *Ptr*-jével (lásd §25.10[2]).

Az olyan leírót, melynek felülete közel azonos azon osztályéval, melynek leírója, gyakran nevezzük *proxy*-nak. Ez különösen azokra a leírókra vonatkozik, melyek távoli gépen lévő objektumokra hivatkoznak.

25.7.1. A leírók műveletei

A -> operátor túlterhelése lehetővé teszi, hogy egy leíró minden hozzáféréskor megkapja a vezérlést, és valamilyen műveletet végezzen egy objektumon. A leírón keresztül hozzáférhető objektum felhasználásainak számáról például statisztikát készíthetünk:

```
template <class T> class Xhandle {
   T* rep;
   int no_of_accesses;
public:
   T* operator->() { no_of_accesses++; return rep; }

   // ...
};
```

Azon leírók, melyeknél a hozzáférés előtt és után is valamilyen műveletet kell végezni, kidolgozottabb kódot igényelnek. Tegyük fel például, hogy egy olyan halmazt szeretnénk, amely zárolható, amíg beszúrás vagy eltávolítás folyik. Az ábrázoló osztály felületét lényegében meg kell ismételni a leíró osztályban:

```
template<class T> class Set_controller {
   Set<T>* rep;
   Lock lock;
   // ...
public:
   void insert(T* p) { Lock_ptr x(lock); rep->insert(p); }      // lásd §14.4.1
   void remove(T* p) { Lock_ptr x(lock); rep->remove(p); }

   int is_member(T* p) { return rep->is_member(p); }

   T get_first() { T* p = rep->first(); return p ? *p : T(); }
   T get_next() { T* p = rep->next(); return p ? *p : T(); }

   T first() { Lock_ptr x(lock); T tmp = *rep->first(); return tmp; }
   T next() { Lock_ptr x(lock); T tmp = *rep->next(); return tmp; }

   // ...
};
```

Ezekről a továbbító függvényekről gondoskodni fáradságos (így hibát is véthetünk közben), jóllehet nehézséget nem jelent és a futási időt sem növeli.

Vegyük észre, hogy a *Set*-nek csak némelyik függvénye kíván zárolást. Tapasztalatom szerint általános, hogy egy elő- és utótevékenységeket igénylő osztály a műveleteket csak néhány tagfüggvénynél kívánja meg. A minden műveletnél való zárolás – ahogy egyes rendszerfigyelőknél lenni szokott – felesleges zárolásokhoz vezet és észrevehetően lassíthatja a párhuzamos végrehajtást.

A leírón végzett műveletek alapos kidolgozásának előnye a -> operátor túlterhelésével szemben az, hogy a *Set_controller* osztályból származtathatunk. Sajnos a leírók néhány előnyös tulajdonságát feladjuk, ha a származtatott osztályhoz adattagokat teszünk, mert a közösen használt kód mennyisége az egyes leírókban levő kód mennyiségéhez viszonyítva csökken.

25.8. Keretrendszerek

A §25.2–§25.7-ben leírt osztályfajtákból épített komponensek azáltal támogatják a kódtervezést és -újrahasznosítást, hogy építőkockákat és kombinációs lehetőségeket biztosítanak. A programozók és az alkalmazáskészítő eszközök építik fel azt a vázat, amelybe ezek az építőkockák beleillenek. A tervezés és újrahasznosítás támogatásának egy másik, általában nehezebb módja egy olyan, közös vázat adó kód megírása, melybe az alkalmazáskészítő építőkockákként az adott alkalmazásra jellemző kódokat illeszti be. Ez az, amit általában ***keretrendszernek*** (application framework) hívunk. Az ilyen vázat biztosító osztályok felülete gyakran olyan „kövér”, hogy hagyományos értelemben aligha nevezhetők típusoknak; inkább teljes alkalmazásnak tűnnek, bár nem végeznek semmilyen tevékenységet; azokat az alkalmazásprogramozó biztosítja.

Példaképpen vegyünk egy szűrőt, vagyis egy olyan programot, amely egy bemeneti adatfolyamból olvas, majd annak alapján (esetleg) elvégez néhány műveletet, (esetleg) egy kimeneti adatfolyamot hoz létre, és (esetleg) ad egy végeredményt. Első ötletünk bizonyára az, hogy a programhoz olyan keretrendszert készítsünk, amely olyan művelethalmazt ad meg, melyet egy alkalmazásprogramozó biztosít:

```
class Filter {
public:
  class Retry {
  public:
          virtual const char* message() { return 0; }
  };

  virtual void start() { }
  virtual int read() = 0;
  virtual void write() { }
  virtual void compute() { }
  virtual int result() = 0;

  virtual int retry(Retry& m) { cerr << m.message() << '\n'; return 2; }

  virtual ~Filter() { }
};
```

Azon függvényeket, melyeket a származtatott osztálynak kell biztosítania, tisztán virtuális (pure virtual) függvényekként deklaráltuk; a többit egyszerűen úgy definiáltuk, mint amik nem végeznek műveletet.

A keretrendszer gondoskodik egy főciklusról és egy kezdetleges hibakezelő eljárásról is:

```
int main_loop(Filter* p)
{
  for(;;) {
          try {
                  p->start();
                  while (p->read()) {
                          p->compute();
                          p->write();
                  }
                  return p->result();
          }
          catch (Filter::Retry& m) {
                  if (int i = p->retry(m)) return i;
          }
          catch (...) {
                  cerr << "Végzetes szűrőhiba\n";
                  return 1;
          }
  }
}
```

A programot végül így írhatjuk meg:

```
class My_filter : public Filter {
  istream& is;
  ostream& os;
  int nchar;
public:
  int read() { char c; is.get(c); return is.good(); }
  void compute() { nchar++; }
  int result() { os << nchar << " elolvasott karakter\n"; return 0; }

  My_filter(istream& ii, ostream& oo) : is(ii), os(oo), nchar(0) { }
};
```

És így indíthatjuk el:

```
int main()
{
  My_filter f(cin,cout);
  return main_loop(&f);
}
```

Természetesen ahhoz, hogy igazán hasznát vegyük, a keretrendszernek több szerkezetet és jóval több szolgáltatást kellene nyújtania, mint ebben az egyszerű példában. A keretrendszer általában csomópont-osztályokból álló hierarchia. Ha egy mélyen egymásba ágyazott elemekből álló hierarchiában az alkalmazásprogramozóval íratjuk meg a levél osztályokat, lehetővé válik a közös elemek több program által való használata és a hierarchia által nyújtott szolgáltatások újrahasznosítása. A keretrendszert egy könyvtár is támogathatja, olyan osztályokkal, melyek az alkalmazásprogramozó számára a műveletosztályok meghatározásánál hasznosnak bizonyulhatnak.

25.9. Tanácsok

[1] Az egyes osztályok használatára vonatkozóan hozzunk megfontolt döntéseket. §25.1.
[2] Óvakodjunk a választás kényszerétől, melyet az osztályok eltérő fajtái okoznak. §25.1.
[3] Egyszerű, független fogalmak ábrázolására használjunk konkrét típusokat. §25.2.
[4] Használjunk konkrét típusokat azon fogalmak ábrázolására, melyeknél nélkülözhetetlen az optimálishoz közeli hatékonyság. §25.2.

[5] Konkrét osztályból ne származtassunk. §25.2.
[6] Használjunk absztrakt osztályokat olyan felületek ábrázolására, ahol az objektumok ábrázolása változhat. §25.3.
[7] Használjunk absztrakt osztályokat olyan felületek ábrázolására, ahol az objektumok különböző ábrázolásainak együtt kell létezniük. §25.3.
[8] A létező típusok új felületeinek ábrázolására használjunk absztrakt osztályokat. §25.3.
[9] Ha hasonló fogalmak közösen használják a megvalósítás egyes részeit, használjunk csomópont-osztályokat. §25.4.
[10] A megvalósítás fokozatos kidolgozásához használjunk csomópont-osztályokat. §25.4.
[11] Az objektumok felületének kinyerésére használjunk futási idejű típusazonosítást. §25.4.1.
[12] Az állapothoz kapcsolódó műveletek ábrázolására használjunk osztályokat. §25.5.
[13] Azon műveletek ábrázolására, melyeket tárolni, átvinni, vagy késleltetni kell, használjunk osztályokat. §25.5.
[14] Ha egy osztályt új felhasználáshoz kell igazítani (az osztály módosítása nélkül), használjunk felületosztályokat. §25.6.
[15] Ellenőrzés hozzáadására használjunk felületosztályokat. §25.6.1.
[16] Használjunk leírókat, hogy elkerüljük a mutatók és referenciák közvetlen használatát. §25.7.
[17] A közösen használt ábrázolások kezelésére használjunk leírókat. §25.7.
[18] Ha az alkalmazási terület lehetővé teszi, hogy a vezérlési szerkezetet előre meghatározzuk, használjunk keretrendszert. §25.8.

25.10. Gyakorlatok

1. (*1) A §25.4.1 *Io* sablonja nem működik beépített típusokra. Módosítsuk úgy, hogy működjön.
2. (*1.5) A §25.7 *Handle* sablonja nem tükrözi azon osztályok öröklési kapcsolatait, amelyeknek leírója. Módosítsuk úgy, hogy tükrözze (vagyis lehessen egy *Handle<Circle>*-lel egy *Handle<Shape>*-nek értéket adni, de nem fordítva).
3. (*2.5) Ha adott egy *String* osztály, azt ábrázolásként használva és műveleteit virtuális függvényekként megadva hozzunk létre egy másik karakterlánc-osztályt. Hasonlítsuk össze a két osztály teljesítményét. Próbáljunk találni egy értelmes osztályt, amely a legjobban a virtuális függvényekkel rendelkező karakterlánc-osztályból történő nyilvános származtatással valósítható meg.

4. (*4) Tanulmányozzunk két széles körben használt könyvtárat. Osztályozzuk a könyvtári osztályokat konkrét típusokként, absztrakt típusokként, csomópont-osztályokként, leíró osztályokként, és felületosztályokként. Használnak-e absztrakt és konkrét csomópont-osztályokat? Van-e a könyvtárakban levő osztályokra megfelelőbb osztályozás? Használnak-e kövér felületeket? Milyen tárkezelési módot használnak? Milyen lehetőségek vannak – ha vannak – futási idejű típusinformációra?
5. (*2) A *Filter* váz (§25.8) segítségével írjunk olyan programot, mely egy bemeneti adatfolyamból eltávolítja a szomszédos ismételt szavakat, majd az eredményt átmásolja a kimenetre.
6. (*2) A *Filter* keretrendszer segítségével írjunk olyan programot, mely egy bemeneti adatfolyamban megszámlálja a szavak gyakoriságát és kimenetként gyakorisági sorrendben felsorolja a (szó, szám) párokat.
7. (*1.5) Írjunk egy *Range* sablont, mely sablonparaméterekként veszi át a tartományt és az elemtípust.
8. (*1) Írjunk egy *Range* sablont, mely a tartományt konstruktor-paraméterekként veszi át.
9. (*2) Írjunk egy egyszerű karakterlánc-osztályt, mely nem végez hibaellenőrzést. Írjunk egy másik osztályt, mely ellenőrzi az előbbihez való hozzáférést. Vitassuk meg az alapszolgáltatások és a hibaellenőrzés elválasztásának előnyeit és hátrányait.
10. (*2.5) Készítsük el a §25.4.1 objektum I/O rendszerét néhány típusra, köztük legalább az egészekre, a karakterláncokra és egy tetszőlegesen kiválasztott osztályhierarchiára.
11. (*2.5) Határozzuk meg a *Storable* osztályt, mint absztrakt bázisosztályt a *write_out()* és *read_in()* virtuális függvényekkel. Az egyszerűség kedvéért tételezzük fel, hogy egy perzisztens tárolóhely meghatározásához egy karakterlánc elegendő. Használjuk fel a *Storage* osztályt egy olyan szolgáltatásban, mely a *Storable*-ből származtatott osztályok objektumait írja lemezre és ugyanilyen objektumokat olvas lemezről. Ellenőrizzük néhány tetszés szerint választott osztállyal.
12. (*4) Hozzuk létre a *Persistent* alaposztályt a *save()* és *no_save()* műveletekkel, melyek egy destruktor által ellenőrzik, hogy egy objektum bekerült-e az állandó tárba. A *save()*-en és *no_save()*-en kívül még milyen használható műveleteket nyújthatna a *Persistent*? Teszteljük a *Persistent* osztályt néhány tetszés szerint választott osztállyal. Csomópont-osztály, konkrét típus, vagy absztrakt típus-e *Persistent*? Miért?
13. (*3) Írjunk egy *Stack* osztályt, melynek megvalósítása futási időben módosítható. Tipp: „Egy újabb indirekció minden problémát megold.”

14. (*3.5) Készítsük el az *Oper* osztályt, amely egy *Id* típusú (*string* vagy C stílusú karakterlánc) azonosítót és egy műveletet (függvénymutatót vagy függvényobjektumot) tartalmaz. Határozzuk meg a *Cat_object* osztályt, mely *Oper*-ek listáját és egy *void**-ot tartalmaz. Lássuk el a *Cat_object*-et egy *add_oper(Oper)* művelettel, mely egy *Oper*-t ad a listához; egy *remove_oper(Id)*-del, mely egy *Id*-del azonosított *Oper*-t eltávolít a listából; valamint egy *operator() (Id, arg)*-gal, mely meghívja az *Id*-del azonosított *Oper*-t. Készítsünk egy *Cat*-eket tároló vermet egy *Cat_object* segítségével. Írjunk egy kis programot ezen osztályok használatára.
15. (*3) Készítsünk egy *Object* sablont a *Cat_object* osztály alapján. Használjuk fel az *Object*-et egy *String*-verem megvalósításához. Írjunk egy kis programot a sablon használatára.
16. (*2.5) Határozzuk meg az *Object* osztály *Class* nevű változatát, mely biztosítja, hogy az azonos műveletekkel rendelkező objektumok közös műveletsort használjanak. Írjunk egy kis programot a sablon használatára.
17. (*2) Készítsünk egy olyan *Stack* sablont, mely egy, az *Object* sablon által megvalósított verem részére egy hagyományos, típusbiztos felületről gondoskodik. Hasonlítsuk össze ezt a vermet az előző gyakorlatokban talált veremosztályokkal. Írjunk egy kis programot a sablon használatára.
18. (*3) Írjunk egy osztályt olyan műveletek ábrázolására, melyeket végrehajtásra egy másik gépre kell átvinni. Teszteljük egy másik gépnek ténylegesen elküldött parancsokkal vagy parancsoknak egy fájlba írásával, melyeket ezután a fájlból kiolvasva hajtunk végre.
19. (*2) Írjunk egy osztályt függvényobjektumok alakjában ábrázolt műveletek együttes használatára. Ha adott *f* és *g* függvényobjektum, a *Compose(f,g)* hozzon létre egy olyan objektumot, mely egy *g*-hez illeszkedő *x* paraméterrel meghívható, és *f(g(x))*-et ad vissza, feltéve, hogy a *g()* által visszaadott érték egy, az *f()* által elfogadható paramétertípus.

Függelékek és tárgymutató

A függelékek a C++ nyelvtanával; a C és a C++ között, valamint a szabványos és a szabványosítás előtti C++-változatok között felmerülő kompatibilitási kérdésekkel; illetve a nyelv néhány egyéb szolgáltatásával foglalkoznak. Az igen részletes tárgymutató a könyv lényeges része.

Fejezetek

A Nyelvtan
B Kompatibilitás
C Technikai részletek
D Helyi sajátosságok
E Kivételbiztosság a standard könyvtárban
I Tárgymutató

A

Nyelvtan

„Nincs nagyobb veszély, ami egy tanárra leselkedik,
mint hogy szavakat tanít a dolgok helyett."
(Marc Block)

Bevezetés • Kulcsszavak • Nyelvi egységek • Programok • Kifejezések • Utasítások • Deklarációk • Deklarátorok • Osztályok • Származtatott osztályok • Különleges tagfüggvények • Túlterhelés • Sablonok • Kivételkezelés • Az előfeldolgozó direktívái

A.1. Bevezetés

A C++ szintaxisának (formai követelményeinek) itt található összefoglalása a megértés megkönnyítését célozza. Nem a nyelv pontos leírása a cél; az alább leírtaknál a C++ több érvényes nyelvtani szerkezetet is elfogad. Az egyszerű kifejezéseknek a deklarációktól való megkülönböztetésére a többértelműség-feloldási szabályokat (§A.5, §A.7), a formailag helyes, de értelmetlen szerkezetek kiszűrésére pedig a hozzáférési, többértelműségi és típusszabályokat együttesen kell alkalmaznunk.

A C és C++ szabvány a legkisebb különbségeket is formai különbségekkel és nem megszorításokkal fejezi ki. Ez nagyfokú pontosságot ad, de nem mindig javítja a kód olvashatóságát.

A.2. Kulcsszavak

A *typedef* (§4.9.7), névtér (§8.2), osztály (10. fejezet), felsoroló típus (§4.8), és *template* (13. fejezet) deklarációk új környezetfüggő kulcsszavakat vezetnek be a programba.

typedef-név:
  azonosító

névtér-név:
  eredeti-névtér-név
  névtér-álnév

eredeti-névtér-név:
  azonosító

névtér-álnév:
  azonosító

osztálynév:
  azonosító
  sablon-azonosító

felsorolásnév:
  azonosító

sablonnév:
  azonosító

Jegyezzük meg, hogy egy osztályt megnevező *typedef-név* egyben *osztálynév* is.

Ha nem adjuk meg kifejezetten egy azonosítóról, hogy egy típus neve, a fordító feltételezi, hogy nem típust nevez meg (lásd §C.13.5).

A C++- kulcsszavai a következők:

C++ kulcsszavak					
and	*and_eq*	*asm*	*auto*	*bitand*	*bitor*
bool	*break*	*case*	*catch*	*char*	*class*
compl	*const*	*const_cast*	*continue*	*default*	*delete*
do	*double*	*dynamic_cast*	*else*	*enum*	*explicit*
export	*extern*	*false*	*float*	*for*	*friend*
goto	*if*	*inline*	*int*	*long*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*register*	*reinterpret_cast*
return	*short*	*signed*	*sizeof*	*static*	*static_cast*
struct	*switch*	*template*	*this*	*throw*	*true*
try	*typedef*	*typeid*	*typename*	*union*	*unsigned*
using	*virtual*	*void*	*volatile*	*wchar_t*	*while*
xor	*xor_eq*				

A.3. Nyelvi egységek

A szabványos C és C++ nyelvtanok a nyelvi egységeket nyelvtani szerkezetekként mutatják be. Ez növeli a pontosságot, de a nyelvtan „méretét” is, és nem mindig javítja az olvashatóságot:

hex-quad:
  hexadecimális-számjegy hexadecimális-számjegy hexadecimális-számjegy hexadecimális-számjegy

általános-karakternév:
  `\u` *hex-quad*
  `\U` *hex-quad hex-quad*

előfeldolgozó-szimbólum:
  fejállomány-név
  azonosító
  pp-szám
  karakterliterál
  karakterlánc-literál
  előfeldolgozó-utasítás-vagy-műveleti-jel
  minden nem üreshely karakter, ami nem tartozik a fentiek közé

szimbólum:
  azonosító
  kulcsszó
  literál

operátor
műveleti-jel

fejállomány-név:
<*h-char-sorozat*>
"*q-char-sorozat*"

h-char-sorozat:
h-char
h-char-sorozat h-char

h-char:
a forrás-karakterkészlet bármely tagja, kivéve az újsor és > karaktereket

q-char-sorozat:
q-char
q-char-sorozat q-char

q-char:
a forrás-karakterkészlet bármely tagja, kivéve az újsor és " karaktereket

pp-szám:
számjegy
. *számjegy*
pp-szám számjegy
pp-szám nem-számjegy
pp-szám `e` *előjel*
pp-szám `E` *előjel*
pp-szám .

azonosító:
nem-számjegy
azonosító nem-számjegy
azonosító számjegy

nem-számjegy: a következők egyike
általános-karakternév
`_ a b c d e f g h i j k l m n o p q r s t u v w x y z`
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

számjegy: a következők egyike
`0 1 2 3 4 5 6 7 8 9`

előfeldolgozó-utasítás-vagy-műveleti-jel: a következők egyike

```
{      }      [      ]      #      ##     (      )      <:     :>     <%     %>     %:%:
%:     ;      :      ?      ::     .      .*     +      -      *      /      %      ^
&      |      ~      !      =      <      >      +=     -=     *=     /=     %=     ^=
&=     |=     <<=    >>=    <<     >>     ==     !=     <=     >=     &&     ||     ++
--     ,      ->     ->*    ...    new    delete        and    and_eq        bitand
bitor         compl         not    or     not_eq        xor    or_eq         xor_eq
```

literál:
egészliterál
karakterliterál
lebegőpontos-literál
karakterlánc-literál
logikai-literál

egészliterál:
decimális-literál egész-utótag$_{\textit{nem kötelező}}$
oktális-literál egész-utótag$_{\textit{nem kötelező}}$
hexadecimális-literál egész-utótag$_{\textit{nem kötelező}}$

decimális-literál:
nem-nulla-számjegy
decimális-literál számjegy

oktális-literál:
`0`
oktális-literál oktális-számjegy

hexadecimális-literál:
`0x` *hexadecimális-számjegy*
`0X` *hexadecimális-számjegy*
hexadecimális-literál hexadecimális-számjegy

nem-nulla-számjegy: a következők egyike
`1 2 3 4 5 6 7 8 9`

oktális-számjegy: a következők egyike
`0 1 2 3 4 5 6 7`

hexadecimális-számjegy: a következők egyike
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

egész-utótag:
előjel-nélküli-utótag long-utótag$_{\textit{nem kötelező}}$
long-utótag előjel-nélküli-utótag$_{\textit{nem kötelező}}$

előjel-nélküli-utótag: a következők egyike
`u U`

long-utótag: a következők egyike
`l L`

karakterliterál:
'c-char-sorozat'
L*'c-char-sorozat'*

c-char-sorozat:
c-char
c-char-sorozat c-char

c-char:
a forrás-karakterkészlet bármely tagja, kivéve az egyszeres idézőjelet, a fordított perjelet, és az újsor karaktert
escape-sorozat
általános-karakternév

escape-sorozat:
egyszerű-escape-sorozat
oktális-escape-sorozat
hexadecimális-escape-sorozat

egyszerű-escape-sorozat: a következők egyike
`\' \" \? \\ \a \b \f \n \r \t \v`

oktális-escape-sorozat:
\ *oktális-számjegy*
\ *oktális-számjegy oktális-számjegy*
\ *oktális-számjegy oktális-számjegy oktális-számjegy*

hexadecimális-escape-sorozat:
\x *hexadecimális-számjegy*
hexadecimális-escape-sorozat hexadecimális-számjegy

lebegőpontos-literál:
tört-konstans kitevő-rész$_{nem\ kötelező}$ *lebegőpontos-utótag*$_{nem\ kötelező}$
számjegy-sorozat kitevő-rész lebegőpontos-utótag$_{nem\ kötelező}$

tört-konstans:
számjegy-sorozat$_{nem\ kötelező}$. *számjegy-sorozat*
számjegy-sorozat .

kitevő-rész:
e *előjel*$_{nem\ kötelező}$ *számjegy-sorozat*
E *előjel*$_{nem\ kötelező}$ *számjegy-sorozat*

előjel: a következők egyike
    `+   -`

számjegy-sorozat:
    számjegy
    számjegy-sorozat számjegy

lebegőpontos-utótag: a következők egyike
    `f   l   F   L`

karakterlánc-literál:
    "*s-char-sorozat*$_{nem\ kötelező}$"
    L"*s-char-sorozat*$_{nem\ kötelező}$"

s-char-sorozat:
    s-char
    s-char-sorozat s-char

s-char:
  a forrás-karakterkészlet bármely tagja, kivéve a kettős idézőjelet, a fordított perjelet, és az újsort
    escape-sorozat
    általános-karakternév

logikai-literál:
    `false`
    `true`

A.4. Programok

A programok összeszerkesztés által összeállított *fordítási egységek* (translation unit) gyűjteményei (§9.4). A *fordítási egységek* vagy másképp *forrásfájlok*, *deklarációk* sorozatából állnak:

fordítási-egység:
    deklaráció-sorozat$_{nem\ kötelező}$

A.5. Kifejezések

A kifejezéseket a 6. fejezet írja le és a §6.2 pont összegzi. A *kifejezés-lista* (expression-list) definíciója azonos a *kifejezésével* (expression). A függvényparamétereket elválasztó vesszőnek a vessző operátortól (comma, sequencing operator, §6.2.2) való megkülönböztetésére két szabály szolgál.

elsődleges-kifejezés:
  literál
  `this`
  `::` *azonosító*
  `::` *operátorfüggvény-azonosító*
  `::` *minősített-azonosító*
  `(` *kifejezés* `)`
  azonosító-kifejezés

azonosító-kifejezés:
  nem-minősített-azonosító
  minősített-azonosító

nem-minősített-azonosító:
  azonosító
  operátorfüggvény-azonosító
  átalakítófüggvény-azonosító
  `~` *osztálynév*
  sablon-azonosító

minősített-azonosító:
  beágyazott-név-meghatározás `template`$_{\textit{nem kötelező}}$ *nem-minősített-azonosító*

beágyazott-név-meghatározás:
  osztály-vagy-névtér-név `::` *beágyazott-név-meghatározás*$_{\textit{nem kötelező}}$
  osztály-vagy-névtér-név `::` `template` *beágyazott-név-meghatározás*

osztály-vagy-névtér-név:
  osztálynév
  névtér-név

utótag-kifejezés:
  elsődleges-kifejezés
  utótag-kifejezés `[` *kifejezés* `]`
  utótag-kifejezés `(` *kifejezés-lista*$_{\textit{nem kötelező}}$ `)`
  egyszerű-típus-meghatározás `(` *kifejezés-lista*$_{\textit{nem kötelező}}$ `)`
  `typename` `::`$_{\textit{nem kötelező}}$ *beágyazott-név-meghatározás azonosító* `(` *kifejezés-lista*$_{\textit{nem kötelező}}$ `)`

typename ::$_{\text{nem kötelező}}$ *beágyazott-név-meghatározás* template$_{\text{nem kötelező}}$
sablon-azonosító (*kifejezés-lista*$_{\text{nem kötelező}}$)
utótag-kifejezés . template$_{\text{nem kötelező}}$ ::$_{\text{nem kötelező}}$ *azonosító-kifejezés*
utótag-kifejezés -> template$_{\text{nem kötelező}}$ ::$_{\text{nem kötelező}}$ *azonosító-kifejezés*
utótag-kifejezés . *ál-destruktor-név*
utótag-kifejezés -> *ál-destruktor-név*
utótag-kifejezés ++
utótag-kifejezés --
dynamic-cast < *típusazonosító* > (*kifejezés*)
static-cast < *típusazonosító* > (*kifejezés*)
reinterpret-cast < *típusazonosító* > (*kifejezés*)
const-cast < *típusazonosító* > (*kifejezés*)
typeid (*kifejezés*)
typeid (*típusazonosító*)

kifejezés-lista:
értékadó-kifejezés
kifejezés-lista , *értékadó-kifejezés*

ál-destruktor-név:
::$_{\text{nem kötelező}}$ *beágyazott-név-meghatározás*$_{\text{nem kötelező}}$ *típusnév* :: ~ *típusnév*
::$_{\text{nem kötelező}}$ *beágyazott-név-meghatározás* template *sablon-azonosító* :: ~ *típusnév*
::$_{\text{nem kötelező}}$ *beágyazott-név-meghatározás*$_{\text{nem kötelező}}$ ~ *típusnév*

egyoperandusú-kifejezés:
utótag-kifejezés
++ *cast-kifejezés*
-- *cast-kifejezés*
egyoperandusú-operátor cast-kifejezés
sizeof *egyoperandusú-kifejezés*
sizeof (*típusazonosító*)
new-kifejezés
delete-kifejezés

egyoperandusú-operátor: a következők egyike
* & + - ! ~

new-kifejezés:
::$_{\text{nem kötelező}}$ new *elhelyező-utasítás*$_{\text{nem kötelező}}$ *new-típusazonosító new-kezdőérték-adó*$_{\text{nem kötelező}}$
::$_{\text{nem kötelező}}$ new *elhelyező-utasítás*$_{\text{nem kötelező}}$ (*típusazonosító*) *new-kezdőérték-adó*$_{\text{nem kötelező}}$

elhelyező-utasítás:
(*kifejezés-lista*)

new-típusazonosító:
típus-meghatározó-sorozat new-deklarátor$_{\text{nem kötelező}}$

new-deklarátor:
ptr-operátor new-deklarátor$_{\text{nem kötelező}}$
közvetlen-new-deklarátor

közvetlen-new-deklarátor:
[*kifejezés*]
közvetlen-new-deklarátor [*konstans-kifejezés*]

new-kezdőérték-adó:
(*kifejezés-lista*$_{\text{nem kötelező}}$)

delete-kifejezés:
`::`$_{\text{nem kötelező}}$ `delete` *cast-kifejezés*
`::`$_{\text{nem kötelező}}$ `delete [ ]` *cast-kifejezés*

cast-kifejezés:
egyoperandusú-kifejezés
(*típusazonosító*) *cast-kifejezés*

pm-kifejezés:
cast-kifejezés
pm-kifejezés .* *cast-kifejezés*
pm-kifejezés ->* *cast-kifejezés*

szorzó-kifejezés:
pm-kifejezés
szorzó-kifejezés * *pm-kifejezés*
szorzó-kifejezés / *pm-kifejezés*
szorzó-kifejezés % *pm-kifejezés*

összeadó-kifejezés:
szorzó-kifejezés
összeadó-kifejezés + *szorzó-kifejezés*
összeadó-kifejezés - *szorzó-kifejezés*

eltoló-kifejezés:
összeadó-kifejezés
eltoló-kifejezés << *összeadó-kifejezés*
eltoló-kifejezés >> *összeadó-kifejezés*

viszonyító-kifejezés:
eltoló-kifejezés
viszonyító-kifejezés < *eltoló-kifejezés*
viszonyító-kifejezés > *eltoló-kifejezés*
viszonyító-kifejezés <= *eltoló-kifejezés*
viszonyító-kifejezés >= *eltoló-kifejezés*

egyenértékűség-kifejezés:
  viszonyító-kifejezés
  egyenértékűség-kifejezés == *viszonyító-kifejezés*
  egyenértékűség-kifejezés != *viszonyító-kifejezés*

és-kifejezés:
  egyenértékűség-kifejezés
  és-kifejezés & *egyenértékűség-kifejezés*

kizáró-vagy-kifejezés:
  és-kifejezés
  kizáró-vagy-kifejezés ^ *és-kifejezés*

megengedő-vagy-kifejezés:
  kizáró-vagy-kifejezés
  megengedő-vagy-kifejezés | *kizáró-vagy-kifejezés*

logikai-és-kifejezés:
  megengedő-vagy-kifejezés
  logikai-és-kifejezés && *megengedő-vagy-kifejezés*

logikai-vagy-kifejezés:
  logikai-és-kifejezés
  logikai-vagy-kifejezés || *logikai-és-kifejezés*

feltételes-kifejezés:
  logikai-vagy-kifejezés
  logikai-vagy-kifejezés ? *kifejezés* : *értékadó-kifejezés*

értékadó-kifejezés:
  feltételes-kifejezés
  logikai-vagy-kifejezés értékadó-operátor értékadó-kifejezés
  throw-kifejezés

értékadó-operátor: a következők egyike
  = *= /= %= += -= >>= <<= &= ^= |=

kifejezés:
  értékadó-kifejezés
  kifejezés , *értékadó-kifejezés*

konstans-kifejezés:
  feltételes-kifejezés

A függvény stílusú típusátalakítások (cast) és a deklarációk hasonlóságából többértelműségek adódhatnak. Például:

```
int x;

void f()
{
        char(x); // x átalakítása char típusra
                 // vagy egy x nevű char deklarációja?
}
```

A fordító minden ilyen többértelmű szerkezetet deklarációként értelmez, vagyis a szabály: „ha lehet deklarációként értelmezni, akkor deklaráció". Például:

```
T(a)->m;            // kifejezés-utasítás
T(a)++;             // kifejezés-utasítás

T(*e)(int(3));      // deklaráció
T(f)[4];            // deklaráció

T(a);               // deklaráció
T(a)=m;             // deklaráció
T(*b)();            // deklaráció
T(x),y,z=7;         // deklaráció
```

A többértelműség ilyen feloldása tisztán szintaktikus. A fordító a névről csak annyi információt használ fel, hogy az egy ismert típus- vagy sablonnév-e. Ha nem az, akkor ezektől különböző jelentést tételez fel róla.

A `template` *nem-minősített-azonosító* szerkezet azt jelenti, hogy a *nem-minősített-azonosító* egy sablon neve, mégpedig egy olyan környezetben, ahonnan ezt nem lehetne levezetni (§C.13.6)

A.6. Utasítások

Lásd §6.3-at.

utasítás:
  címkézett-utasítás
  kifejezés-utasítás
  összetett-utasítás
  kiválasztó-utasítás
  ciklus-utasítás
  ugró-utasítás
  deklaráció-utasítás
  try-blokk

címkézett-utasítás:
  azonosító `:` *utasítás*
  `case` *konstans-kifejezés* `:` *utasítás*
  `default :` *utasítás*

kifejezés-utasítás:
  $\textit{kifejezés}_{\textit{nem kötelező}}$ `;`

összetett-utasítás:
  `{` $\textit{utasítás-sorozat}_{\textit{nem kötelező}}$ `}`

utasítás-sorozat:
  utasítás
  utasítás-sorozat utasítás

kiválasztó-utasítás:
  `if (` *feltétel* `)` *utasítás*
  `if (` *feltétel* `)` *utasítás* `else` *utasítás*
  `switch (` *feltétel* `)` *utasítás*

feltétel:
  kifejezés
  típus-meghatározás-sorozat deklarátor = *értékadó-kifejezés*

ciklus-utasítás:
  `while (` *feltétel* `)` *utasítás*
  `do` *utasítás* `while (` *kifejezés* `) ;`
  `for (` *for-init-utasítás* $\textit{feltétel}_{\textit{nem kötelező}}$ `;` $\textit{kifejezés}_{\textit{nem kötelező}}$ `)` *utasítás*

for-init-utasítás:
  kifejezés-utasítás
  egyszerű-deklaráció

ugró-utasítás:
  `break ;`
  `continue ;`
  `return` *kifejezés*$_{\text{nem kötelező}}$ `;`
  `goto` *azonosító* `;`

deklaráció-utasítás:
  blokk-deklaráció

A.7. Deklarációk

A deklarációk szerkezetét a 4. fejezet írja le, a felsoroló típusokat (enumeration) a §4.8, a mutatókat (pointer) és tömböket (array) az 5. fejezet, a függvényeket (function) a 7. fejezet, a névtereket (namespace) a §8.2, az összeszerkesztési direktívákat (linkage directive) a §9.2.4, a tárolási módokat (storage class) pedig a §10.4.

deklaráció-sorozat:
  deklaráció
  deklaráció-sorozat deklaráció

deklaráció:
  blokk-deklaráció
  függvénydefiníció
  sablondeklaráció
  explicit-példányosítás
  explicit-specializáció
  összeszerkesztési-mód
  névtér-definíció

blokk-deklaráció:
  egyszerű-deklaráció
  asm-definíció
  névtér-álnév-definíció
  using-deklaráció
  using-utasítás

egyszerű-deklaráció:
  deklaráció-meghatározás-sorozat$_{\text{nem kötelező}}$ *kezdőérték-deklarátor-lista*$_{\text{nem kötelező}}$ `;`

deklaráció-meghatározás:
  tárolási-mód-meghatározás
  típus-meghatározás

függvény-meghatározás
`friend`
`typedef`

deklaráció-meghatározás-sorozat:
deklaráció-meghatározás-sorozat$_{nem\ kötelező}$ *deklaráció-meghatározás*

tárolási-mód-meghatározás:
`auto`
`register`
`static`
`extern`
`mutable`

függvény-meghatározás:
`inline`
`virtual`
`explicit`

typedef-név:
azonosító

típus-meghatározás:
egyszerű-típus-meghatározás
osztály-meghatározás
felsorolás-meghatározás
összetett-típus-meghatározás
cv-minősítő

egyszerű-típus-meghatározás:
`::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás*$_{nem\ kötelező}$ *típusnév*
`::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás* `template`$_{nem\ kötelező}$ *sablon-azonosító*
char
wchar_t
bool
short
int
long
signed
unsigned
float
double
void

típusnév:
osztálynév
felsorolásnév
typedef-név

összetett-típus-meghatározás:
osztálykulcs `::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás*$_{nem\ kötelező}$ *azonosító*
`enum` `::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás*$_{nem\ kötelező}$ *azonosító*
`typename` `::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás azonosító*
`typename` `::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás* `template`$_{nem\ kötelező}$ *sablon-azonosító*

felsorolásnév:
azonosító

felsorolás-meghatározás:
`enum` *azonosító*$_{nem\ kötelező}$ `{` *felsorolás-lista*$_{nem\ kötelező}$ `}`

felsorolás-lista:
felsoroló-definíció
felsorolás-lista , *felsoroló-definíció*

felsoroló-definíció:
felsoroló
felsoroló = *konstans-kifejezés*

felsoroló:
azonosító

névtér-név:
eredeti-névtér-név
névtér-álnév

eredeti-névtér-név:
azonosító

névtér-definíció:
nevesített-névtér-definíció
nevesítetlen-névtér-definíció

nevesített-névtér-definíció:
eredeti-névtér-definíció
bővített-névtér-definíció

eredeti-névtér-definíció:
`namespace` *azonosító* `{` *névtér-törzs* `}`

bővített-névtér-definíció:
`namespace` *eredeti-névtér-név* `{` *névtér-törzs* `}`

nevesítetlen-névtér-definíció:
`namespace` `{` *névtér-törzs* `}`

névtér-törzs:
  deklaráció-sorozat$_{nem\ kötelező}$

névtér-álnév:
  azonosító

névtér-álnév-definíció:
  `namespace` *azonosító* = *minősített-névtér-meghatározás* `;`

minősített-névtér-meghatározás:
  `::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás$_{nem\ kötelező}$ névtér-név*

using-deklaráció:
  `using typename`$_{nem\ kötelező}$ `::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás nem-minősített-azonosító* `;`
  `using ::` *nem-minősített-azonosító* `;`

using-utasítás:
  `using namespace ::`$_{nem\ kötelező}$ *beágyazott-név-meghatározás$_{nem\ kötelező}$ névtér-név* `;`

asm-definíció:
  `asm (` *karakterlánc-literál* `) ;`

összeszerkesztési-mód:
  `extern` *karakterlánc-literál* `{` *deklaráció-sorozat$_{nem\ kötelező}$* `}`
  `extern` *karakterlánc-literál deklaráció*

A nyelvtan megengedi a deklarációk tetszőleges egymásba ágyazását, de bizonyos megszorítások érvényesek. Például nem szabad függvényeket egymásba ágyazni, azaz egy függvényen belül egy másik függvényt kifejteni.

A deklarációkat kezdő meghatározások („minősítők", specifier) listája nem lehet üres (azaz „nincs implicit int", §B.2) és a meghatározások leghosszabb lehetséges sorozatából áll. Például:

```
typedef int I;
void f(unsigned I) { /* ... */ }
```

Itt *f()* paramétere egy *unsigned int*.

Az *asm()* szerkezet az assembly kód beszúrására szolgál. Jelentését az adott nyelvi változat határozza meg, de célja az, hogy a megadott helyen az adott szövegnek megfelelő assembly kód kerüljön be a fordító által létrehozott kódba.

Ha egy változót *register*-ként vezetünk be, azzal azt jelezzük a fordítóprogram számára, hogy a gyakori hozzáférésekre kell optimalizálnia a kódot. Az újabb fordítóprogramok legtöbbje számára ezt felesleges megadni.

A.7.1. Deklarátorok

Lásd a §4.9.1 pontot, az 5. fejezetet (mutatók és tömbök), a §7.7 pontot (függvénymutatók) és a §15.5 pontot (tagokra hivatkozó mutatók).

kezdőérték-deklarátor-lista:
    kezdőérték-deklarátor
    kezdőérték-deklarátor-lista , *kezdőérték-deklarátor*

kezdőérték-deklarátor:
    deklarátor kezdőérték-adó$_{nem\ kötelező}$

deklarátor:
    közvetlen-deklarátor
    ptr-operátor deklarátor

közvetlen-deklarátor:
    deklarátor-azonosító
    közvetlen-deklarátor (*paraméter-deklaráció-záradék*) *cv-minősítő-sorozat*$_{nem\ kötelező}$
        kivétel-meghatározás$_{nem\ kötelező}$
    közvetlen-deklarátor [*konstans-kifejezés*$_{nem\ kötelező}$]
    (*deklarátor*)

ptr-operátor:
    * *cv-minősítő-sorozat*$_{nem\ kötelező}$
    &
    ::$_{nem\ kötelező}$ *beágyazott-név-meghatározás* * *cv-minősítő-sorozat*$_{nem\ kötelező}$

cv-minősítő-sorozat:
    cv-minősítő cv-minősítő-sorozat$_{nem\ kötelező}$

cv-minősítő:
    `const`
    `volatile`

deklarátor-azonosító:
    ::$_{nem\ kötelező}$ *azonosító-kifejezés*
    ::$_{nem\ kötelező}$ *beágyazott-név-meghatározás*$_{nem\ kötelező}$ *típusnév*

típusazonosító:
típus-meghatározás-sorozat elvont-deklarátor$_{\text{nem kötelező}}$

típus-meghatározás-sorozat:
típus-meghatározás típus-meghatározás-sorozat$_{\text{nem kötelező}}$

elvont-deklarátor:
ptr-operátor elvont-deklarátor$_{\text{nem kötelező}}$
közvetlen-elvont-deklarátor

közvetlen-elvont-deklarátor:
közvetlen-elvont-deklarátor$_{\text{nem kötelező}}$ (*paraméter-deklaráció-záradék*) *cv-minősítő-sorozat*$_{\text{nem kötelező}}$ *kivétel-meghatározás*$_{\text{nem kötelező}}$
közvetlen-elvont-deklarátor$_{\text{nem kötelező}}$ [*konstans-kifejezés*$_{\text{nem kötelező}}$]
(*elvont-deklarátor*)

paraméter-deklaráció-záradék:
paraméter-deklaráció-lista$_{\text{nem kötelező}}$ ... $_{\text{nem kötelező}}$
paraméter-deklaráció-lista , ...

paraméter-deklaráció-lista:
paraméter-deklaráció
paraméter-deklaráció-lista , *paraméter-deklaráció*

paraméter-deklaráció:
deklaráció-meghatározás-sorozat deklarátor
deklaráció-meghatározás-sorozat deklarátor = *értékadó-kifejezés*
deklaráció-meghatározás-sorozat elvont-deklarátor$_{\text{nem kötelező}}$
deklaráció-meghatározás-sorozat elvont-deklarátor$_{\text{nem kötelező}}$ = *értékadó-kifejezés*

függvénydefiníció:
deklaráció-meghatározás-sorozat$_{\text{nem kötelező}}$ *deklarátor ctor-kezdőérték-adó*$_{\text{nem kötelező}}$ *függvénytörzs*
deklaráció-meghatározás-sorozat$_{\text{nem kötelező}}$ *deklarátor függvény-try-blokk*

függvénytörzs:
összetett-utasítás

kezdőérték-adó:
= *kezdőérték-adó-záradék*
(*kifejezés-lista*)

kezdőérték-adó-záradék:
értékadó-kifejezés
{ *kezdőérték-lista* , $_{\text{nem kötelező}}$ }
{ }

kezdőérték-lista:
kezdőérték-adó-záradék
kezdőérték-lista **,** *kezdőérték-adó-záradék*

A *volatile* meghatározás/minősítés azt jelzi a fordítóprogram számára, hogy az objektum a nyelv által nem meghatározott módon változtatja az értékét, így az „agresszív optimalizálás" kerülendő. Egy valósidejű órát például így deklarálhatunk:

extern const volatile clock;

A *clock* két egymást követő leolvasása különböző eredményeket adhat.

A.8. Osztályok

Lásd a 10. fejezetet.

osztálynév:
azonosító
sablon-azonosító

osztály-meghatározás:
osztályfej { *tag-meghatározás*$_{nem\ kötelező}$ }

osztályfej:
osztálykulcs azonosító$_{nem\ kötelező}$ *alap-záradék*$_{nem\ kötelező}$
osztálykulcs beágyazott-név-meghatározás azonosító alap-záradék$_{nem\ kötelező}$
osztálykulcs beágyazott-név-meghatározás `template` *sablon-azonosító alap-záradék*$_{nem\ kötelező}$

osztálykulcs:
`class`
`struct`
`union`

tag-meghatározás:
tag-deklaráció tag-meghatározás$_{nem\ kötelező}$
hozzáférés-meghatározás **:** *tag-meghatározás*$_{nem\ kötelező}$

tag-deklaráció:
deklaráció-meghatározás-sorozat$_{nem\ kötelező}$ *tag-deklarátor-lista*$_{nem\ kötelező}$;
függvénydefiníció ; $_{nem\ kötelező}$
:: $_{nem\ kötelező}$ *beágyazott-név-meghatározás* template$_{nem\ kötelező}$ *nem-minősített-azonosító* ;
using-deklaráció
sablondeklaráció

tag-deklarátor-lista:
tag-deklarátor
tag-deklarátor-lista , *tag-deklarátor*

tag-deklarátor:
deklarátor üres-meghatározás$_{nem\ kötelező}$
deklarátor konstans-kezdőérték-adó$_{nem\ kötelező}$
azonosító$_{nem\ kötelező}$: *konstans-kifejezés*

üres-meghatározás:
= 0

konstans-kezdőérték-adó:
= *konstans-kifejezés*

A C-vel való összeegyeztethetőséget megőrzendő egy azonos nevű osztály és egy nem-osztály ugyanabban a hatókörben is deklarálható (§5.7). Például:

```
struct stat { /* ... */ };
int stat(char* név, struct stat* buf);
```

Ebben az esetben a sima név (*stat*) a nem-osztály neve. Az osztályra egy *osztálykulcs* előtaggal (*class*, *struct* vagy *union*) kell hivatkozni.

A konstans kifejezéseket a §C.5 pont írja le.

A.8.1. Származtatott osztályok

Lásd a 12. és 15. fejezetet.

alap-záradék:
: *alap-meghatározás-lista*

alap-meghatározás-lista:
alap-meghatározás
alap-meghatározás-lista , *alap-meghatározás*

alap-meghatározás:
`::`*nem kötelező* *beágyazott-név-meghatározás*nem kötelező *osztálynév*
`virtual` *hozzáférés-meghatározás*nem kötelező `::`*nem kötelező* *beágyazott-név-meghatározás*nem kötelező *osztálynév*
hozzáférés-meghatározás `virtual`nem kötelező `::`*nem kötelező* *beágyazott-név-meghatározás*nem kötelező *osztálynév*

hozzáférés-meghatározás:
`private`
`protected`
`public`

A.8.2. Különleges tagfüggvények

Lásd a §11.4 (átalakító operátorok), §10.4.6 (osztálytagok kezdeti értékadása) és §12.2.2 (alaposztályok kezdeti értékadása) pontokat.

átalakítófüggvény-azonosító:
`operator` *átalakítás-típusazonosító*

átalakítás-típusazonosító:
*típus-meghatározás-sorozat átalakítás-deklarátor*nem kötelező

átalakítás-deklarátor:
*ptr-operátor átalakítás-deklarátor*nem kötelező

ctor-kezdőérték-adó:
`:` *mem-kezdőérték-lista*

mem-kezdőérték-lista:
mem-kezdőérték-adó
mem-kezdőérték-adó `,` *mem-kezdőrték-lista*

mem-kezdőérték-adó:
mem-kezdőérték-adó-azonosító `(` *kifejezés-lista*nem kötelező `)`

mem-kezdőérték-adó-azonosító:
`::`*nem kötelező* *beágyazott-név-meghatározás*nem kötelező *osztálynév*
azonosító

A.8.3. Túlterhelés

Lásd a 11. fejezetet.

operátorfüggvény-azonosító:
  `operator` *operátor*

operátor: a következők egyike

```
new     delete     new[]     delete[]
+    -    *    /    %    ^    &    |    ~    !    =    <    >
+=   -=   *=   /=   %=   ^=   &=   |=   <<   >>   >>=  <<=  ==
!=   <=   >=   &&   ||   ++   --   ,    ->*  ->   ()   []
```

A.9. Sablonok

A sablonokkal a 13. fejezet és a §C.13. pont foglalkozik részletesen.

sablondeklaráció:
  `export`$_{\text{nem kötelező}}$ `template` < *sablonparaméter-lista* > *deklaráció*

sablonparaméter-lista:
  sablonparaméter
  sablonparaméter-lista `,` *sablonparaméter*

sablonparaméter:
  típus-paraméter
  paraméter-deklaráció

típus-paraméter:
  `class` *azonosító*$_{\text{nem kötelező}}$
  `class` *azonosító*$_{\text{nem kötelező}}$ = *típusazonosító*
  `typename` *azonosító*$_{\text{nem kötelező}}$
  `typename` *azonosító*$_{\text{nem kötelező}}$ = *típusazonosító*
  `template` < *sablonparaméter-lista* > `class` *azonosító*$_{\text{nem kötelező}}$
  `template` < *sablonparaméter-lista* > `class` *azonosító*$_{\text{nem kötelező}}$ = *sablonnév*

sablon-azonosító:
  sablonnév < *sablonargumentum-lista*$_{\text{nem kötelező}}$ >

sablonnév:
  azonosító

sablonargumentum-lista:
  sablonargumentum
  sablonargumentum-lista , *sablonargumentum*

sablonargumentum:
  értékadó-kifejezés
  típusazonosító
  sablonnév

explicit-példányosítás:
  `template` *deklaráció*

explicit-specializáció:
  `template < >` *deklaráció*

Az explicit sablonargumentum-meghatározás egy érdekes többértelműségre ad lehetőséget. Vegyük a következő példát:

```
void h()
{
    f<1>(0);          // többértelmű: ((f)<1) > (0) vagy (f<1>)(0) ?
                      // feloldása: f<1> meghívása a 0 paraméterrel
}
```

A feloldás egyszerű és hatékony: ha *f* egy sablon neve, akkor *f<* egy minősített sablonnév kezdete és az azt követő nyelvi egységek eszerint értelmezendőek; ha nem ez a helyzet, a < jel a „kisebb mint” műveletet jelenti. Ugyanígy az első pár nélküli > jel lezárja a sablonparaméterek listáját. Ha a „nagyobb mint” jelre van szükségünk, zárójelezést kell alkalmaznunk:

```
f< a>b >(0);          // szintaktikus hiba
f< (a>b) >(0);        // rendben
```

Hasonló többértelműség léphet fel, ha a záró > jelek túl közel kerülnek:

```
list<vector<int>> lv1;    // szintaktikus hiba: nem várt >> (jobbra léptetés)
list< vector<int> > lv2;  // helyes: vektorok listája
```

Figyeljünk a szóközre a két > jel között (a >> a jobbra léptető operátor!), mert nagyon könnyű elnézni.

A.10. Kivételkezelés

Lásd a §8.3 pontot és a 14. fejezetet.

try-blokk:
  `try` *összetett-utasítás kezelő-sorozat*

függvény-try-blokk:
  `try` *ctor-kezdőérték-adó*$_{nem\ kötelező}$ *függvénytörzs kezelő-sorozat*

kezelő-sorozat:
  kezelő kezelő-sorozat$_{nem\ kötelező}$

kezelő:
  `catch (` *kivétel-deklaráció* `)` *összetett-utasítás*

kivétel-deklaráció:
  típus-meghatározás-sorozat deklarátor
  típus-meghatározás-sorozat elvont-deklarátor
  típus-meghatározás-sorozat
  `...`

throw-kifejezés:
  `throw` *értékadó-kifejezés*$_{nem\ kötelező}$

kivétel-meghatározás:
  `throw (` *típusazonosító-lista*$_{nem\ kötelező}$ `)`

típusazonosító-lista:
  típusazonosító
  típusazonosító-lista `,` *típusazonosító*

A.11. Az előfeldolgozó direktívái

Az előfeldolgozó (preprocessor, pp) egy viszonylag egyszerű makró-feldolgozó rendszer, amely elsődlegesen nyelvi egységeken és nem egyes karaktereken dolgozik. A makrók meghatározásának és használatának (§7.8) képességén kívül az előfeldolgozó a szövegfájloknak és szabványos fejállományoknak (§9.2.1) a forrásba építésére is lehetőséget ad és makrókon alapuló feltételes fordítást is tud végezni (§9.3.3). Például:

```
#if OPT==4
#include "fejállomány4.h"
#elif 0<OPT
#include "fejállomány.h"
#else
#include<cstdlib>
#endif
```

Az összes előfeldolgozói utasítás (direktíva) a # jellel kezdődik, amelynek az első nem üreshely karakternek kell lennie a sorban.

előfeldolgozó-fájl:
    csoport$_{\text{nem kötelező}}$

csoport:
    csoport-rész
    csoport csoport-rész

csoport-rész:
    pp-szimbólumok$_{\text{nem kötelező}}$ *újsor*
    if-rész
    vezérlősor

if-rész:
    if-csoport elif-csoportok$_{\text{nem kötelező}}$ *else-csoport*$_{\text{nem kötelező}}$ *endif-sor*

if-csoport:
    `# if` *konstans-kifejezés újsor csoport*$_{\text{nem kötelező}}$
    `# ifdef` *azonosító újsor csoport*$_{\text{nem kötelező}}$
    `# ifndef` *azonosító újsor csoport*$_{\text{nem kötelező}}$

elif-csoportok:
    elif-csoport
    elif-csoportok elif-csoport

elif-csoport:
    `# elif` *konstans-kifejezés újsor csoport*$_{\text{nem kötelező}}$

else-csoport:
    `# else` *újsor csoport*$_{\text{nem kötelező}}$

endif-sor:
    `# endif` *újsor*

vezérlősor:
  `#` `include` *pp-szimbólumok újsor*
  `#` `define` *azonosító helyettesítő-lista újsor*
  `#` `define` *azonosító balzárójel azonosító-lista*$_{nem\ kötelező}$ `)` *helyettesítő-lista újsor*
  `#` `undef` *azonosító újsor*
  `#` `line` *pp-szimbólumok újsor*
  `#` `error` *pp-szimbólumok*$_{nem\ kötelező}$ *újsor*
  `#` `pragma` *pp-szimbólumok*$_{nem\ kötelező}$ *újsor*
  `#` *újsor*

balzárójel:
  a bal oldali zárójel karakter megelőző üreshely nélkül

helyettesítő-lista:
  pp-szimbólumok$_{nem\ kötelező}$

pp-szimbólumok:
  előfeldolgozó-szimbólum
  pp-szimbólumok előfeldolgozó-szimbólum

újsor:
  újsor karakter

azonosító-lista:
  azonosító
  azonosító-lista `,` *azonosító*

B

Kompatibilitás

„Te mész a te utadon, a te szokásaid szerint, és én is követem a saját elveimet."
(C. Napier)

C/C++ kompatibilitás • „Észrevétlen" különbségek a C és a C++ között • C program, ami nem C++ program • Elavult szolgáltatások • C++ program, ami nem C program • Régebbi C++-változatok használata • Fejállományok • A standard könyvtár • Névterek • Helyfoglalási hibák • Sablonok • A *for* utasítás kezdőérték-adó része • Tanácsok • Gyakorlatok

B.1. Bevezetés

Ebben a függelékben azokat a különbségeket vizsgáljuk meg, amelyek a C és a C++, illetve a szabványos C++ és a régebbi C++-változatok között állnak fenn. Célunk az, hogy egyrészt leírjuk azokat a különbségeket, amelyek a programozóknak problémát okozhatnak, másrészt módszereket mutassunk ezen problémák kezelésére. A legtöbb kompatibilitási problémával akkor kerülünk szembe, amikor egy C programot a C++-ból akarunk használni, vagy egy nem szabványos C++ rendszerben létrehozott programot egy másik környezetbe akarunk átvinni, esetleg új lehetőségeket akarunk egy régebbi C++-változatban használ-

ni. A cél nem az, hogy részletesen bemutassuk az összes kompatibilitási problémát, ami valaha is előfordulhat, hanem az, hogy a leggyakoribb problémákra szabványos megoldást adjunk.

Amikor kompatibilitási problémákról beszélünk, a legfontosabb kérdés az, hogy programunk a különböző nyelvi változatok milyen széles körében képes működni. A C++ nyelv megismeréséhez érdemes a legteljesebb és legkényelmesebb rendszert használnunk, rendszerfejlesztéskor azonban ennél konzervatívabb stratégiát kell követnünk, hiszen olyan programot szeretnénk, amely a lehető legtöbb környezetben működőképes. Régebben ez nagyon jó kifogás volt arra, hogy a C++ „túl fejlettnek" minősített lehetőségeit elkerüljük. Az egyes változatok azonban közeledtek egymáshoz, így a különböző platformok közötti hordozhatóság követelménye ritkán igényel olyan komoly erőfeszítéseket a programozótól, mint néhány évvel ezelőtt.

B.2. C/C++ kompatibilitás

Kisebb kivételektől eltekintve a C++ a C nyelv továbbfejlesztésének tekinthető. A legtöbb különbség a C++ hangsúlyozottabb típusellenőrzéséből következik. A jól megírt C programok általában C++ programok is. A C és C++ közötti összes különbséget a fordítónak is jeleznie kell.

B.2.1. „Észrevétlen" különbségek

Néhány kivételtől eltekintve azok a programok, melyek C és C++ nyelven is értelmezhetők, ugyanazt jelentik mind a két nyelvben. Szerencsére azok a különbségek, melyek mégis előfordulnak, általában csak árnyalatnyiak:

A C-ben a karakterkonstansok és a felsoroló típusok mérete *sizeof(int)*. A C++-ban *sizeof('a')* egyenértékű *sizeof(char)*-ral, a felsoroló típusok megvalósításához pedig az egyes C++-változatok olyan méretet használhatnak, ami az adott környezetben a legcélszerűbb (§4.8).

A C++ lehetővé teszi a *//* jellel bevezetett megjegyzések használatát; ez a C-ben nem áll rendelkezésünkre (bár nagyon sok C-változat tartalmazza bővítésként). Ezt az eltérést olyan programok készítéséhez használhatjuk fel, melyek a két nyelvben különbözőképpen viselkednek.

Például:

```
int f(int a, int b)
{
  return a //* elég valószínűtlen */ b
          ;        /* pontosvessző a szintaktikus hiba elkerülésére */
}
```

Az ISO C ma már a C++-hoz hasonlóan megengedi a *//* használatát.

Egy belső hatókörben bevezetett adatszerkezet-név elrejtheti egy külső hatókörben deklarált objektum, függvény, felsorolás, vagy típus nevét:

```
int x[99];
void f()
{
        struct x { int a; };
        sizeof(x);          /* a tömb mérete C-ben, a struct mérete C++-ban */
}
```

B.2.2. C program, ami nem C++ program

A leggyakrabban problémát okozó különbségek általában nem túl élesek. A legtöbbet ezek közül a fordítók is könnyen észreveszik. Ebben a pontban olyan C kódokat mutatunk be, amelyek nem C++ kódok. Ezek legtöbbjét már az újabb C-változatok is „rossz stílusúnak", sőt „elavultnak" minősítik.

A C-ben a legtöbb függvényt deklarálásuk előtt is meghívhatjuk:

```
main()    /* rossz stílus C-ben, hibás C++-ban */
{
  double sq2 = sqrt(2);                      /* deklarálatlan függvény meghívása */
  printf("the square root of 2 is %g\n",sq2);     /* deklarálatlan függvény meghívása */
}
```

A függvény-deklarációk (függvény-prototípusok) teljes és következetes használata minden C-megvalósításban ajánlott. Ha ezt a tanácsot megfogadjuk (és ezt a fordítók általában valamilyen kapcsolóval biztosítani is tudják), C programjaink illeszkedni fognak a C++ szabályaihoz. Ha deklarálatlan függvényeket hívunk meg, nagyon pontosan ismernünk kell C rendszerünk függvényhívással kapcsolatos szabályait, hogy eldönthessük, okoztunk-e hibát vagy hordozhatósági problémát. Az előbbi *main()* program például legalább két hibát tartalmaz C programként is.

A C-ben azok a függvények, melyeket paramétertípus megadása nélkül vezetünk be, tetszőleges számú és típusú paraméterrel meghívhatók. Az ilyen függvényhasználat a Standard C-ben is elavult, ennek ellenére sok helyen találkozhatunk vele:

```
void f();    /* a paramétertípusokat elhagyjuk */

void g()
{
  f(2);      /* rossz stílus C-ben, hibás C++-ban */
}
```

A C-ben megengedett az a függvény-meghatározási forma, melyben a paraméterek típusát a paraméterek listájának megadása után rögzítjük:

```
void f(a,p,c) char *p; char c; { /* ... */ } /* helyes C-ben, hibás C++-ban */
```

Ezeket a definíciókat át kell írnunk:

```
void f(int a, char* p, char c) { /* ... */ }
```

A C-ben és a C++ szabványosítás előtti változataiban az *int* alapértelmezett típus volt. Például:

```
const a = 7;          /* C-ben "int" típust feltételez; C++-ban hibás */
```

Az ISO C, a C++-hoz hasonlóan, megtiltotta az „implicit *int*" használatát.

A C megengedi, hogy a visszatérési típusok és a paramétertípusok deklarációiban *struct*-okat adjunk meg:

```
struct S { int x,y; } f();                 /* helyes C-ben, hibás C++-ban */
void g(struct S { int x,y; } y);           /* helyes C-ben, hibás C++-ban */
```

A C++ típus-meghatározásokra vonatkozó szabályai az ilyen deklarációkat feleslegessé teszik és nem is engedik meg.

A C-ben a felsoroló típusú változóknak értékül adhatunk egészeket is:

```
enum Direction { up, down };
Direction d = 1;      /* hiba: int értékadás Direction-nek; C-ben helyes */
```

A C++ sokkal több kulcsszót ismer, mint a C. Ha ezek valamelyikét azonosítóként használjuk egy C programban, kénytelenek leszünk azt lecserélni, hogy C++-ban is értelmezhető programot kapjunk.

C++ kulcsszavak, melyek a C-ben nem kulcsszavak					
and	*and_eq*	*asm*	*bitand*	*bitor*	*bool*
catch	*class*	*compl*	*const_cast*	*delete*	*dynamic_cast*
explicit	*export*	*false*	*friend*	*inline*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*reinterpret_cast*	*static_cast*
template	*this*	*throw*	*true*	*try*	*typeid*
typename	*using*	*virtual*	*wchar_t*	*xor*	*xor_eq*

A C-ben néhány C++ kulcsszó makróként szerepel a szabványos fejállományokban:

C++ kulcsszavak, melyek C makrók					
and	*and_eq*	*bitand*	*bitor*	*compl*	*not*
not_eq	*or*	*or_eq*	*wchar_t*	*xor*	*xor_eq*

Ez azt is jelenti, hogy a C-ben ezeket tesztelhetjük az *#ifdef* segítségével, meghatározásukat felülbírálhatjuk stb.

A C-ben a globális adatobjektumokat többször is deklarálhatjuk egy fordítási egységen belül, anélkül, hogy az *extern* kulcsszót használnánk. Amíg a deklarációk közül csak egy ad kezdőértéket, a fordító az objektumot egyszer meghatározottnak (egyszer definiáltnak) tekinti.

Például:

```
int i; int i;              /* egyetlen 'i' egészet határoz meg vagy vezet be; C++-ban hibás */
```

A C++-ban minden elemet csak egyszer határozhatunk meg (§9.2.3).

A C++-ban egy osztálynak nem lehet ugyanaz a neve, mint egy *typedef* elemnek, amely ugyanabban a hatókörben valamilyen más típusra hivatkozik (§5.7).

A C-ben a *void** használható bármilyen mutató típusú változó egyszerű vagy kezdeti értékadásának jobb oldalán. A C++-ban ezt nem tehetjük meg (§5.6.):

```
void f(int n)
{
  int* p = malloc(n*sizeof(int));          /* C++-ban hibás; használjuk inkább a 'new'

operátort */
}
```

A C megengedi, hogy ugrásokkal kikerüljünk egy kezdeti értékadást, a C++-ban ez sem lehetséges.

A C-ben a globális konstansokat a fordító automatikusan *extern* elemekként kezeli, míg a C++-ban nem. Kötelező vagy kezdőértéket adni, vagy az *extern* kulcsszót használni (§5.4.).

A C-ben a beágyazott szerkezetek nevei ugyanabba a hatókörbe kerülnek, mint a felettük álló szerkezeté:

```
struct S {
  struct T { /* ... */ };
  // ...
};

struct T x;      /* helyes C-ben, jelentése 'S::T x;'; C++-ban hibás */
```

A C-ben egy tömbnek olyan elemmel is adhatunk kezdőértéket, amelynek több eleme van, mint amennyire a deklarált tömbnek szüksége van:

```
char v[5] = "Oscar";              /* helyes C-ben, a lezáró 0-t nem használja; C++-ban hibás */
```

B.2.3. Elavult szolgáltatások

Ha a szabványosító bizottság egy szolgáltatást elavultnak nyilvánít, akkor ezzel azt fejezi ki, hogy a szolgáltatást el kell kerülni. A bizottságnak azonban nincs joga egy korábban gyakran használható lehetőséget teljesen megszüntetni, akkor sem, ha az esetleg felesleges vagy kifejezetten veszélyes. Tehát az „elavultság” kimondása csak egy (nyomatékos) ajánlás a programozóknak arra, hogy ne használják a lehetőséget.

A *static* kulcsszó, melynek alapjelentése: „statikusan lefoglalt”, általában azt jelzi, hogy egy függvény vagy egy objektum helyinek (lokálisnak) számít a fordítási egységre nézve:

```
// fájl1:
  static int glob;

// fájl2:
  static int glob;
```

Ennek a programnak valójában két *glob* nevű, egész típusú változója lesz. Mindkettőt kizárólag azok a függvények fogják használni, amelyekkel megegyező fordítási egységben szerepel.

A *static* kulcsszó használata a „lokális a fordítási egységre nézve” értelemben elavult a C++-ban. Helyettük a névtelen névterek használata javasolt (§8.2.5.1).

A karakterlánc-literálok automatikus átalakítása (nem konstans) *char** típusra szintén elavult. Használjunk inkább névvel rendelkező karaktertömböket, így elkerülhetjük, hogy karakterlánc-literálokat kelljen értékül adnunk *char** változóknak (§5.2.2).

A C stílusú típusátalakítást az új átalakításoknak szintén elavulttá kellett volna tenniük; sajnos még sokan használják, pedig a felhasználónak programjaiban érdemes komolyan megfogadnia a C stílusú átalakítások tilalmát. Ha explicit típusátalakításra van szükségünk, a *static_cast*, a *reinterpret_cast*, illetve a *const_cast* kulcsszóval vagy ezek együttes használatával ugyanazt az eredményt érhetjük el, mint a C stílusú átalakítással. Az új átalakításokat azért is érdemes használnunk, mert ezek szélesebb körben használhatók és pontosabban megfogalmazottak (§6.2.7).

B.2.4. C++ program, ami nem C program

Ebben a pontban azokat a szolgáltatásokat soroljuk fel, melyek a C++-ban megtalálhatók, de a C-ben nem. A lehetőségeket szerepük szerint soroljuk fel. Természetesen számtalan osztályozás lehetséges, mert a legtöbb szolgáltatás több célt is szolgál, tehát az itt bemutatott nem az egyetlen jó csoportosítás:

- Lehetőségek, melyek elsősorban kényelmes jelölésrendszert biztosítanak:
  1. A *//* megjegyzések (§2.3); a C-ben is szerepelni fognak.
  2. Korlátozott karakterkészletek használatának lehetősége (§C.3.1).
  3. Bővített karakterkészletek használatának lehetősége (§C.3.3), a C-ben is megjelenik.
  4. A *static* tárban levő objektumok nem-konstans kezdőértéket is kaphatnak (§9.4.1).
  5. A *const* a konstans kifejezésekben (§5.4, §C.5).
  6. A deklarációk utasításokként szerepelnek.
  7. Deklarációk szerepelhetnek a *for* utasítás kezdőérték-adó elemében és az *if* feltételben is (§6.3.3, §6.3.2.1).
  8. Az adatszerkezetek nevei előtt nem kell szerepelnie a struct kulcsszónak.

- Lehetőségek, melyek elsősorban a típusrendszer erősítését szolgálják:
  1. A függvényparaméterek típusellenőrzése (§7.1); később a C-ben is megjelent (§B.2.2).
  2. Típusbiztos összeszerkesztés (§9.2, §9.2.3).
  3. A szabad tár kezelése a *new* és a *delete* operátor segítségével (§6.2.6, §10.4.5, §15.6).
  4. A *const* (§5.4, §5.4.1); később a C-be is bekerült.
  5. A logikai bool adattípus (§4.2).
  6. Új típusátalakítási forma (§6.2.7).

- Felhasználói típusokat segítő lehetőségek:
  1. Osztályok (10. fejezet).
  2. Tagfüggvények (§10.2.1) és tagosztályok (§11.12).
  3. Konstruktorok és destruktorok (§10.2.3, §10.4.1).
  4. Származtatott osztályok (12. fejezet, 15. fejezet).
  5. Virtuális függvények és elvont osztályok (§12.2.6, §12.3).
  6. A *public/protected/private* hozzáférés-szabályozás (§10.2.2, §15.3, §C.11).
  7. A „barátok” (friend) lehetőségei (§11.5).
  8. Mutatók tagokra (§15.5, §C.12).
  9. *static* tagok (§10.2.4).

10. *mutable* tagok (§10.2.7.2).
11. Operátor-túlterhelés (11. fejezet).
12. Hivatkozások (§5.5).

- Lehetőségek, melyek elsősorban a program rendszerezésére szolgálnak (az osztályokon túl):
  1. Sablonok (13. fejezet, §C.13).
  2. Helyben kifejtett (inline) függvények (§7.1.1).
  3. Alapértelmezett paraméterek (§7.5).
  4. Függvénynév-túlterhelés (§7.4).
  5. Névterek (§8.2).
  6. Explicit hatókör-meghatározás (a *::* operátor, §4.9.4).
  7. Kivételkezelés (§8.3, 14. fejezet).
  8. Futási idejű típus-meghatározás (§15.4).

A C++-ban bevezetett új kulcsszavak (§B.2.2) a legtöbb olyan szolgáltatást bemutatják, melyek kifejezetten a C++-ban jelentek meg. Van azonban néhány nyelvi elem – például a függvény-túlterhelés vagy a *consts* alkalmazása konstans kifejezésekben –, melyeket nem új kulcsszó segítségével valósít meg a nyelv. Az itt felsorolt nyelvi lehetőségek mellett a C++ könyvtár is (§16.1.2) nagy részben csak a C++-ra jellemző.

A *__cplusplus* makró segítségével mindig megállapíthatjuk, hogy programunkat éppen C vagy C++ fordítóval dolgozzuk-e fel (§9.2.4).

B.3. Régebbi C++-változatok használata

A C++ nyelvet 1983 óta folyamatosan használják (§1.4). Azóta számtalan változat és önálló fejlesztőkörnyezet készült belőle. A szabványosítás alapvető célja, hogy a programozók és felhasználók részére egy egységes C++ álljon rendelkezésre. Amíg azonban a szabvány teljes körben el nem terjed a C++-rendszerek készítői körében, mindig figyelembe kell vennünk azt a tényt, hogy nem minden megvalósítás nyújtja az összes szolgáltatást, ami ebben a könyvben szerepel.

Sajnos nem ritka, hogy a programozók első komoly benyomásaikat a C++-ról egy négy-öt éves változat alapján szerzik meg. Ennek oka az, hogy ezek széles körben és olcsón elérhetők. Ha azonban a legkisebb lehetősége is van, egy magára valamit is adó szakember

ilyen antik rendszernek a közelébe sem megy. Egy kezdőnek a régebbi változatok számtalan rejtett problémát okozhatnak. A nyelvi lehetőségek és a könyvtárban megvalósított szolgáltatások hiánya olyan problémák kezelését teszi szükségessé, melyek az újabb változatokban már nem jelentkeznek. A kevés lehetőséget biztosító régebbi változatok a kezdő programozó programozási stílusának is sokat ártanak, ráadásul helytelen képünk alakul ki arról, mi is valójában a C++. Véleményem szerint a C++ első megismerésekor nem az a legmegfelelőbb részhalmaz, amely az alacsonyszintű szolgáltatásokat tartalmazza (tehát semmiképpen sem a C és a C++ közös része). Azt ajánlom, először a standard könyvtárat és a sablonokat ismerjük meg, mert ezekkel egyszerűen megoldhatunk komolyabb feladatokat is és jó ízelítőt kapunk a „valódi C++" szolgáltatásaiból.

A C++ első kereskedelmi kiadása 1985-ben jelent meg; e könyv első kiadása alapján készítették. Akkor a C++ még nem biztosított többszörös öröklődést, sablonokat, futási idejű típusinformációkat, kivételeket és névtereket sem. Mai szemmel semmi értelmét nem látom annak, hogy olyan rendszerrel kezdjünk el dolgozni, amely ezen szolgáltatások legalább egy részét nem biztosítja. A többszörös öröklődés, a sablonok és a kivételek 1989-ben kerültek be a C++-ba. A sablonok és kivételek első megvalósítása még nagyon kiforratlan és szegényes volt. Ha ezek használatakor problémákba ütközünk, sürgősen szerezzünk be egy újabb nyelvi változatot.

Általában igaz, hogy olyan változatot érdemes használnunk, amely minden lehetséges helyen igazodik a szabványhoz, hogy elkerülhessük a megvalósításból eredő különbségeket, illetve az adott nyelvi változatnak a szabvány által nem meghatározott tulajdonságait. Tervezzük programjainkat úgy, mintha a teljes nyelv a rendelkezésünkre állna, majd valósítsuk meg önállóan a hiányzó részletek szolgáltatásait. Így jobban rendszerezett és könnyebben továbbfejleszthető programot kapunk, mint ha a C++ mindenhol megtalálható, közös magjához terveznénk a programot. Arra is mindig figyeljünk, hogy megvalósítás-függő szolgáltatásokat csak akkor használjunk, ha elkerülhetetlen.

B.3.1. Fejállományok

Eredetileg minden fejállomány kiterjesztése *.h* volt, így a C++ egyes változataiban is megjelentek az olyan fejállományok, mint a *<map.h>* vagy az *<iostream.h>*. A kompatibilitás érdekében ezek általában ma is használhatók.

Amikor a szabványosító bizottságnak új fejállományokat kellett bevezetnie a szabványos könyvtárak átírt változatai, illetve az újonnan meghatározott könyvtári szolgáltatások kezeléséhez, a fejállományok elnevezése problémákba ütközött. A régi *.h* kiterjesztés használa-

ta összeegyeztethetőségi problémákat okozott volna. A megoldást a *.h* kiterjesztés elhagyása jelentette az új szabványos fejállomány-nevekben. Az utótag egyébként is felesleges volt, mert a < > enélkül is jelezte, hogy szabványos fejállományt neveztünk meg.

Tehát a standard könyvtár kiterjesztés nélküli fejállományokat biztosít (például *<map>* vagy *<iostream>*). Ezen állományok deklarációi az *std* névtérben szerepelnek. A régebbi fejállományok a globális névtérben kapnak helyet és ezek nevében szerepel a *.h* kiterjesztés.

Például:

```
#include<iostream>

int main()
{
  std::cout << "Helló, világ!\n";
}
```

Ha ezt a programrészletet nem sikerül lefordítanunk, próbálkozzunk a hagyományosabb változattal:

```
#include<iostream.h>

int main()
{
  cout << "Helló, világ!\n";
}
```

A legtöbb hordozhatósági problémát a nem teljesen összeegyeztethető fejállományok okozzák. A szabványos fejállományok ebből a szempontból ritkábban jelentenek gondot. Nagyobb programoknál gyakran előfordul, hogy sok fejállományt használunk, és ezek nem mindegyike szerepel az összes rendszerben vagy sok deklaráció nem ugyanabban a fejállományban jelenik meg. Az is előfordul, hogy bizonyos deklarációk szabványosnak tűnnek (mert szabványos nevű fejállományokban szerepelnek), de valójában semmilyen szabványban nem szerepelnek.

Sajnos nincs teljesen kielégítő módszer a fejállományok által okozott hordozhatósági problémák kezelésére. Egy gyakran alkalmazott megoldás, hogy elkerüljük a fejállományoktól való közvetlen függést és a fennmaradó függőségeket külön fogalmazzuk meg. Így a hordozhatóságot a közvetett hivatkozásokkal és az elkülönített függőségkezeléssel javítjuk.

Például, ha egy szükséges deklaráció a különböző rendszerekben különböző fejállományokon keresztül érhető el, egy alkalmazásfüggő fejállományt használhatunk, amely az

#include segítségével magába foglalja az egyes rendszerek megfelelő állományait. Ugyanígy, ha valamilyen szolgáltatást a különböző rendszerekben másképpen érhetünk el, a szolgáltatáshoz alkalmazásfüggő osztályokat és függvényeket készíthetünk.

B.3.2. A standard könyvtár

Természetesen a C++ szabvány előtti változataiból hiányozhatnak a standard könyvtár bizonyos részei. A legtöbb rendszerben szerepelnek az adatfolyamok, a *complex* adattípus (általában nem sablonként), a különböző *string* osztályok és a C standard könyvtára. A *map*, a *list*, a *valarray* stb. azonban gyakran hiányozik ezekből a megvalósításokból. Ilyenkor próbáljuk az elérhető könyvtárakat úgy használni, hogy később lehetőségünk legyen az átalakításra, ha szabványos környezetbe kerülünk. Általában jobban járunk, ha a nem szabványos *string*, *list* vagy *map* osztályt használjuk, ahelyett, hogy a standard könyvtár osztályainak hiánya miatt visszatérnénk a C stílusú programozáshoz. Egy másik lehetőség, hogy a standard könyvtár STL részének (16., 17., 18. és 19. fejezet) jó megvalósításai tölthetők le ingyenesen az Internetről.

A standard könyvtár régebbi változatai még nem voltak teljesek. Például sokszor jelentek meg tárolók úgy, hogy memóriafoglalók használatára még nem volt lehetőség, vagy éppen ellenkezőleg, mindig kötelező volt megadni a memóriafoglalót. Hasonló problémák fordultak elő az „eljárásmód-paraméterek" esetében is, például az összehasonlítási feltételeknél:

```
list<int> li;                  // rendben, de néhány megvalósítás megköveteli a memóriafoglalót
list<int,allocator<int> > li2; // rendben, de néhány megvalósítás nem ismeri
                               // a memóriafoglalót

map<string,Record> m1;         // rendben, de néhány megvalósítás megkövetel egy
                               // 'kisebb mint' műveletet
map<string,Record,less<string> > m2;
```

Mindig azt a változatot kell használnunk, amelyet az adott fejlesztőkörnyezet elfogad. Szerencsés esetben rendszerünk az összes formát ismeri.

A régebbi C++-változatokban gyakran *istrstream* és *ostrstream* osztály szerepelt, a *<strstream.h>* nevű fejállomány részeként, míg a szabvány az *istringstream* és *ostringstream* osztályokat adja meg, melyeknek helye az *<sstream>* fejállomány. A *strstream* adatfolyamok közvetlenül karaktertömbökön végeztek műveleteket (lásd §21.10 [26]).

A szabvány előtti C++-változatokban az adatfolyamok nem voltak paraméteresek. A *basic_* előtagú sablonok újak a szabványban; a *basic_ios* osztályt régebben *ios* osztálynak hívták. Kissé meglepő módon az *iostate* régi neve viszont *io_state* volt.

B.3.3. Névterek

Ha rendszerünk nem támogatja a névterek használatát, a program logikai szerkezetét kifejezhetjük forrásfájlok segítségével is (9.fejezet). A fejállományok ugyanígy jól használhatók saját vagy C kóddal közösen használt felületeink leírására.

Ha a névterek nem állnak rendelkezésre, a névtelen névterek hiányát a *static* kulcsszó használatával ellensúlyozhatjuk. Szintén sokat segíthet, ha a globális nevekben egy előtaggal jelezzük, hogy milyen logikai egységhez tartoznak:

```
// névtér előtti nyelvi változatokban:

class bs_string { /* ... */ };            // Bjarne saját string típusa
typedef int bs_bool;                      // Bjarne saját bool típusa

class joe_string;                         // Joe saját string típusa
enum joe_bool { joe_false, joe_true };    // Joe saját bool típusa
```

Az előtagok kiválasztásakor azonban legyünk elővigyázatosak, mert a létező C és C++ könyvtárak esetleg ugyanilyen előtagokat használnak.

B.3.4. Helyfoglalási hibák

Mielőtt a kivételkezelés megjelent a C++-ban, a *new* operátor *0* értéket adott vissza, ha a helyfoglalás nem sikerült. A szabványos C++ *new* operátora ma már egy *bad_alloc* kivétellel jelzi a hibát.

Általában érdemes programjainkat a szabványnak megfelelően átírni. Itt ez annyit jelent, hogy nem a *0* visszatérési értéket, hanem a *bad_alloc* kivételt figyeljük. Általában mindkét esetben nagyon nehéz a problémát egy hibaüzenetnél hatékonyabban kezelni.

Ha úgy érezzük, nem érdemes a *0* érték vizsgálatát a *bad_alloc* kivétel figyelésére átalakítani, általában elérhetjük, hogy a program úgy működjön, mintha nem állnának rendelkezésünkre a kivételek. Ha rendszerünkben nincs *_new_handler* telepítve, a *nothrow* memóriafoglaló segítségével kivételek helyett a *0* visszatérési értékkel vizsgálhatjuk a helyfoglalási hibák előfordulását:

```
X* p1 = new X;               // bad_alloc kivételt vált ki, ha nincs memória
X* p2 = new(nothrow) X;      // visszatérési értéke 0, ha nincs memória
```

B.3.5. Sablonok

A szabvány számos új szolgáltatást vezetett be a sablonok körében és a régebbi lehetőségek szabályait is tisztábban fogalmazta meg.

Ha rendszerünk nem támogatja a részlegesen egyedi célú változatok (specializációk) megadását, akkor azokhoz a sablonokhoz, amelyeket egyébként egyszerű szakosítással hoznánk létre, önálló nevet kell rendelnünk:

```
template<class T> class plist : private list<void*> {          // list<T*> kellett volna
  // ...
};
```

Ha az adott nyelvi változat nem támogatja a sablon tagok használatát, néhány módszerről kénytelenek leszünk lemondani. A sablon tagok segítségével például olyan rugalmasan határozhatjuk meg elemek létrehozását, illetve átalakítását, amire ezek nélkül nincs lehetőségünk. (§13.6.2.) Néha kisegítő megoldást jelent az, hogy egy önálló (nem tag) függvényt adunk meg, amely létrehozza a megfelelő objektumot:

```
template<class T> class X {
  // ...
  template<class A> X(const A& a);
};
```

Ha nem használhatunk sablon tagokat, kénytelenek vagyunk konkrét típusokra korlátozni tevékenységünket:

```
template<class T> class X {
  // ...
  X(const A1& a);
  X(const A2& a);
  // ...
};
```

A korábbi C++-rendszerek többsége a sablon osztályok példányosításakor a sablon összes függvényét lemásolja és létrehozza kódjukat. Ez ahhoz vezethet, hogy a nem használt tagfüggvények hibát okoznak (§C.13.9.1.). A megoldást az jelenti, ha a kérdéses tagfüggvények meghatározását az osztályon kívül helyezzük el.

A

```
template<class T> class Container {
  // ...
public:
  void sort() { /* használja < műveletet */ }        // osztályon belüli meghatározás
};

class Glob { /* a Glob-ban nincs < művelet */ };

Container<Glob> cg;           // néhány szabvány előtti változatban
                              // Container<Glob>::sort() szerepel
```

helyett például a következő megoldást adhatjuk:

```
template<class T> class Container {
  // ...
public:
  void sort();
};

template<class T> void Container<T>::sort() { /* a < művelet használata */ }
                                              // osztályon kívüli definíció

class Glob { /* a Glob-ban nincs < művelet */ };

Container<Glob> cg;                           // nincs baj, amíg a cg.sort()-ot meg nem hívjuk
```

A C++ régebbi megvalósításai nem teszik lehetővé az osztályban később bevezetett tagok kezelését:

```
template<class T> class Vector {
public:
  T& operator[](size_t i) { return v[i]; }  // v később bevezetendő
  // ...
private:
  T* v;                                      // hoppá: nem találja!
  size_t sz;
};
```

Ilyenkor vagy úgy rendezzük át a tagokat, hogy elkerüljük az ilyen problémákat, vagy a függvény meghatározását az osztálydeklaráció után helyezzük.

Néhány, szabvány előtti C++-változat a sablonok esetében nem fogadja el az alapértelmezett paraméterek használatát (§13.4.1.). Ez esetben minden sablonparamétert külön meg kell adnunk:

```
template<class Key, class T, class LT = less<T> > class map {
  // ...
};

map<string,int> m;                  // hoppá: nem lehet alapértelmezett sablonparaméter
map< string,int,less<string> > m2;  // megoldás: explicit megadás
```

B.3.6. A for utasítás kezdőérték-adó része

Vizsgáljuk meg az alábbi példát:

```
void f(vector<char>& v, int m)
{
  for (int i= 0; i<v.size() && i<=m; ++i) cout << v[i];

  if (i == m) {      // hiba: hivatkozás 'i'-re a 'for' utasítás után
       // ...
  }
}
```

Az ilyen programok régebben működtek, mert az eredeti C++-ban a változó hatóköre a *for* utasítást tartalmazó hatókör végéig terjedt. Ha ilyen programokkal találkozunk, a változót érdemes egyszerűen a *for* ciklus előtt bevezetnünk:

```
void f2(vector<char>& v, int m)
{
  int i= 0;          // 'i' kell a ciklus után
  for (; i<v.size() && i<=m; ++i) cout << v[i];

  if (i == m) {
       // ...
  }
}
```

B.4. Tanácsok

[1] A C++ megtanulásához használjuk a szabványos C++ legújabb és legteljesebb változatát, amihez csak hozzá tudunk férni. §B.3.

[2] A C és a C++ közös része egyáltalán nem az a része a C++-nak, amit először meg kell tanulnunk. §1.6, §B.3.

[3] Amikor komoly alkalmazást készítünk, gondoljunk rá, hogy nem minden C++-változat tartalmazza a szabvány összes lehetőségét. Mielőtt egy főbb szolgáltatást használunk egy komoly programban, érdemes kitapasztalnunk azt egy-két kisebb program megírásával. Ezzel ellenőrizhetjük, hogy rendszerünk mennyire illeszkedik a szabványhoz és mennyire hatékony a megvalósítás. Példaképpen lásd §8.5[6-7], §16.5[10], §B.5[7]

[4] Kerüljük az elavult szolgáltatásokat, például a *static* kulcsszó globális használatát, vagy a C stílusú típusátalakítást.

[5] A szabvány megtiltotta az „implicit *int*" használatát, tehát mindig pontosan adjuk meg függvényeink, változóink, konstansaink stb. típusát. §B.2.2.

[6] Amikor egy C programot C++ programmá alakítunk, először a függvénydeklarációkat (prototípusokat), és a szabványos fejállományok következetes használatát ellenőrizzük. §B.2.2.

[7] Amikor egy C programot C++ programmá alakítunk, nevezzük át azokat a változókat, melyek C++ kulcsszavak. §B.2.2.

[8] Amikor egy C programot C++ programmá alakítunk, a *malloc()* eredményét mindig megfelelő típusúra kell alakítanunk. Még hasznosabb, ha a *malloc()* összes hívását a *new* operátorral helyettesítjük. §B.2.2.

[9] Ha a *malloc()* és a *free()* függvényeket a *new* és a *delete* operátorra cseréljük, vizsgáljuk meg, hogy a *realloc()* függvény használata helyett nem tudjuk-e a *vector*, a *push_back()* és a *reserve()* szolgáltatásait igénybe venni. §3.8, §16.3.5.

[10] Amikor egy C programot C++ programmá alakítunk, gondoljunk rá, hogy nincs automatikus átalakítás egészekről felsoroló típusokra, tehát meghatározott átalakítást kell alkalmaznunk, ha ilyen átalakításra van szükség. §4.8.

[11] Az *std* névtérben meghatározott szolgáltatások kiterjesztés nélküli fejállományokban szerepelnek. (Az *std::cout* deklarációja például az *<iostream>* fejállomány része.) A régebbi változatokban a standard könyvtár nevei is a globális névtérbe kerültek, a fejállományok pedig *.h* kiterjesztéssel rendelkeztek. (A *::cout* deklarációja például az *<iostream.h>* fejállományban szerepelt.) §9.2.2, §B.3.1.

[12] Ha régebbi C++ programok a *new* visszatérési értékét a *0* értékkel hasonlítják össze, akkor ezt a *bad_alloc* kivétel ellenőrzésére kell cserélnünk, vagy a *new(nothrow)* utasítást kell helyette használnunk. §B.3.4.

[13] Ha fejlesztőkörnyezetünk nem támogatja az alapértelmezett sablonparaméterek használatát, meg kell adnunk minden paramétert. A *typedef* segítségével a sablonparaméterek ismételgetése elkerülhető (ahhoz hasonlóan, ahogy a *string* típus-meghatározás megkímél minket a *basic_string< char, char_traits<char>, allocator<char> >* leírásától). §B.3.5.

[14] Az *std::string* osztály használatához a *<string>* fejállományra van szükségünk. (A *<string.h>* állományban a régi, C stílusú karakterlánc-függvények szerepelnek.) §9.2.2, §B.3.1.

[15] Minden *<X.h>* szabványos C fejállománynak (amely a globális névtérbe vezet be neveket) megfelel egy *<cX>* fejállomány, amely az elemeket az *std* névtérbe helyezi. §B.3.1.

[16] Nagyon sok rendszerben található egy *"String.h"* fejállomány, amely egy karakterlánc-típust ír le. Mindig gondoljunk rá, hogy ezek eltérhetnek a szabványos *string* osztálytól.

[17] Ha lehetőségünk van rá, a szabványos eszközöket használjuk a nem szabványos lehetőségek helyett. §20.1, §B.3, §C.2.

[18] Ha C függvényeket vezetünk be, használjuk az *extern "C"* formát.

B.5. Gyakorlatok

1. (*2.5) Vegyünk egy C programot és alakítsuk át C++-ra. Soroljuk fel a programban használt, a C++-ban nem használható elemeket, és vizsgáljuk meg, érvényesek-e az ANSI C szabvány szerint. Első lépésben a programot csak ANSI C formátumra alakítsuk (prototípusokkal stb.), csak ezután C++-ra. Becsüljük meg, mennyi időt vesz igénybe egy 100 000 soros C program átalakítása C++-ra.
2. (*2.5) Írjunk programot, amely segít átalakítani egy C programot C++-ra. Végezze el azon változók átnevezését, melyek a C++-ban kulcsszavak, a *malloc()* hívásokat helyettesítse a *new* operátorral stb. Ajánlás: ne akarjunk tökéletes programot írni.
3. (*2) Egy C stílusú C++ programban (például amit mostanában alakítottak át egy C programból) a *malloc()* függvény hívásait cseréljük a *new* operátor meghívására. Ötlet: §B.4[8-9]
4. (*2.5) Egy C stílusú C++ programban (például amit mostanában alakítottak át egy C programból) a makrók, globális változók, kezdőérték nélküli változók és típusátalakítások számát csökkentsük a minimumra.

5. (*3) Vegyünk egy C++ programot, amit egy egyszerű eljárással alakítottunk át egy C programból, és bíráljuk azt, mint C++ programot az adatrejtés, az elvont ábrázolás, az olvashatóság, a bővíthetőség és a részek esetleges újrahasznosíthatósága szerint. Végezzünk valamilyen nagyobb átalakítást ezen bírálatok alapján.
6. (*2) Vegyünk egy kicsi (mondjuk 500 soros) C++ programot és alakítsuk azt C programmá. Hasonlítsuk össze a két programot a méret és az elképzelhető továbbfejlesztések szempontjából.
7. (*3) Írjunk néhány kicsi tesztprogramot, mellyel megállapíthatjuk, hogy egy C++-változat rendelkezik-e a legújabb szabványos lehetőségekkel. Például mi a for utasítás kezdőérték-adó részében bevezetett változó hatóköre? (§B.3.6.) Használhatók-e alapértelmezett sablonparaméterek? (§B.3.5.) Használhatók-e sablon tagok? (§8.2.6.) Ajánlás: §B.2.4.
8. (*2.5) Vegyünk egy C++ programot, amely egy <*X.h*> fejállományt használ és alakítsuk át úgy, hogy az <*X*> és <*cX*> fejállományokat használja. Csökkentsük a lehető legkevesebbre a *using* utasítások használatát.

C

Technikai részletek

„Valahol a lélek és az Univer-
zum legmélyén mindennek
megvan a maga oka"
(Slartibartfast)

Mit ígér a szabvány? • Karakterkészletek • Egész literálok • Konstans kifejezések – Kiterjesztések és átalakítások • Többdimenziós tömbök • Mezők és uniók • Memóriakezelés • Szemétgyűjtés • Névterek • Hozzáférés-szabályozás • Mutatók adattagokra • Sablonok • *static* tagok • *friend*-ek • Sablonok, mint sablonparaméterek • Sablonparaméterek levezetése • A *typename* és a *template* minősítők • Példányosítás • Névkötés • Sablonok és névterek • Explicit példányosítás • Tanácsok

C.1. Bevezetés és áttekintés

Ebben a fejezetben olyan technikai részleteket és példákat mutatok be, amelyeket nem lehetett elegánsan beilleszteni abba a szerkezetbe, amit a C++ nyelv főbb lehetőségeinek bemutatására használtam. Az itt közölt részletek fontosak lehetnek programjaink megírásakor is, de elengedhetetlenek, ha ezek felhasználásával készült programot kell megértenünk. Technikai részletekről beszélek, mert nem szabad, hogy a tanulók figyelmét elvonja az el-

sődleges feladatról, a C++ nyelv helyes használatának megismeréséről, vagy hogy a programozókat eltérítse a legfontosabb céltól, gondolataik olyan tiszta és közvetlen megfogalmazásától, amennyire csak a C++ lehetővé teszi.

C.2. A szabvány

Az általános hiedelemmel ellentétben a feltétlen ragaszkodás a C++ nyelvhez és a szabványos könyvtárakhoz önmagában véve nem jelent sem tökéletes programot, sem hordozható kódot. A szabvány nem mondja meg, hogy egy programrészlet jó vagy rossz, mindössze azt árulja el, hogy a programozó mit várhat el egy megvalósítástól és mit nem. Van, aki a szabvány betartásával is borzalmas programokat ír, míg a legtöbb jó program használ olyan lehetőségeket is, melyeket a szabvány nem tartalmaz.

A szabvány nagyon sok fontos dolgot *megvalósítás-függőnek* (implementation-defined) minősít. Ez azt jelenti, hogy minden nyelvi változatnak egy konkrét, pontosan meghatározott megoldást kell adnia az adott kérdésre és ezt a megoldást pontosan dokumentálnia is kell:

```
unsigned char c1 = 64;      // megfelelő meghatározás: egy karakter legalább 8 bit és
                            // mindig képes 64-et tárolni
unsigned char c2 = 1256;    // megvalósítás-függő: csonkol, ha a char csak 8 bites
```

A *c1* kezdeti értékadása megfelelően meghatározott, mert a *char* típusnak legalább 8 bitesnek kell lennie, a c2-é viszont megvalósítás-függő, mert a *char* típus ábrázolásához használt bitek pontos száma is az. Ha a *char* csak 8 bites, az *1256* érték *232*-re csonkul (§C.6.2.1). A legtöbb megvalósítás-függő szolgáltatás a programok futtatásához használt hardverhez igazodik.

Amikor komoly programokat írunk, gyakran szükség van arra, hogy megvalósítás-függő lehetőségeket használjunk. Ez az ára annak, ha számtalan különböző rendszeren hatékonyan futtatható programokat akarunk írni. Sokat egyszerűsített volna a nyelven, ha a karaktereket 8 bitesként, az egészeket pedig 32 bitesként határozzuk meg. Nem ritkák azonban a 16 vagy 32 bites karakterkészletek, sem az olyan egész értékek, melyek nem ábrázolhatók 32 biten. Ma már léteznek 32 gigabájtot meghaladó kapacitású merevlemezek is, így a lemezcímek ábrázolásához érdemes lehet 48 vagy 64 bites egész értékeket használni.

A hordozhatóság lehető leghatékonyabb megvalósításához érdemes pontosan megfogalmaznunk, hogy milyen megvalósítás-függő nyelvi elemeket használtunk a programunkban, és érdemes világosan elkülönítenünk a program azon részeit, ahol ezeket a szolgáltatásokat használjuk. Egy jellemző példa erre a megoldásra, hogy az összes olyan elemet, amely valamilyen módon függ a hardvertől, egy fejállományban, konstansok és típus-meghatározások formájában rögzítjük. Ezt az eljárást támogatja a standard könyvtár a *numeric_limits* osztállyal (§22.2).

A nem meghatározott (definiálatlan) viselkedés kellemetlenebb dolog. Egy nyelvi szerkezetet akkor nevez nem meghatározottnak a szabvány, ha semmilyen értelmes működést nem várhatunk el a megvalósítástól. A szokásos megvalósítási módszerek általában a nem meghatározott lehetőségeket használó programok nagyon rossz viselkedését eredményezik:

```
const int size = 4*1024;
char page[size];

void f()
{
  page[size+size] = 7;          // nem meghatározható
}
```

Egy ilyen programrészlet teljesen kézenfekvő következménye, hogy a memóriában olyan adatokat írunk felül, melyeket nem lenne szabad, vagy hardverhibák, kivételek lépnek fel. A megvalósítástól nem várhatjuk el, hogy ilyen kézenfekvő szabálytalanság esetében is ésszerűen működjön. Ha a programot komolyan optimalizáljuk, a nem meghatározott szerkezetek használatának eredménye teljesen kiszámíthatatlanná válik. Ha létezik kézenfekvő és könnyen megvalósítható megoldása egy problémának, azt inkább megvalósítás-függőnek minősíti a szabvány és nem definiálatlannak.

Érdemes jelentős időt és erőfeszítést szánnunk annak biztosítására, hogy programunk semmi olyat ne használjon, ami a szabvány által nem meghatározott helyzetekhez vezethet. Sok esetben külön eszközök segítik ezt a munkát.

C.3. Karakterkészletek

E könyv példaprogramjai az ASCII-nek (ANSI3.4-1968) nevezett, 7 bites, nemzetközi, ISO 646-1983 karakterkészlet amerikai angol változatának felhasználásával készültek. Ez háromféle problémát jelenthet azoknak, akik más karakterkészletet használó C++-környezetben írják programjaikat:

1. Az ASCII tartalmaz olyan elválasztó karaktereket és szimbólumokat is – például a `]`, a `{` vagy a `!` –, melyek egyes karakterkészletekben nem találhatók meg.
2. Azokhoz a karakterekhez, melyeknek nincs hagyományos ábrázolása, valamilyen jelölésmódot kell választanunk (például az újsorhoz, vagy a 17-es kódú karakterhez).
3. Az ASCII nem tartalmaz bizonyos karaktereket – például a ζ, a Π vagy az æ –, melyeket az angoltól eltérő nyelvekben viszonylag gyakran használnak.

C.3.1. Korlátozott karakterkészletek

A különleges ASCII karakterek – [,], {, }, | és \ – az ISO szerint betűnek minősített karakterpozíciót foglalnak el. A legtöbb európai ISO-646 karakterkészletben ezeken a pozíciókon az angol ábécében nem szereplő betűk szerepelnek. A dán nemzeti karakterkészlet például ezeket a pozíciókat az Æ, æ, Ø, ø, Å, å magánhangzók ábrázolására használja és ezek nélkül nem túl sok dán szöveg írható le.

A trigráf (három jelből álló) karakterek szolgálnak arra, hogy tetszőleges nemzeti karaktereket írhassunk le, „hordozható" formában, a legkisebb szabványos karakterkészlet felhasználásával. Ez a forma hasznos lehet, ha programunkat tényleg át kell vinnünk más rendszerre, de a programok olvashatóságát ronthatja. Természetesen a hosszú távú megoldás erre a problémára a C++ programozók számára az, hogy beszereznek egy olyan rendszert, ahol saját nemzeti karaktereiket és a C++ jeleit is használhatják. Sajnos ez a megoldás nem mindenhol megvalósítható, és egy új rendszer beszerzése idegesítően lassú megoldás lehet. A C++ az alábbi jelölésekkel teszi lehetővé, hogy a programozók hiányos karakterkészlettel is tudjanak programot írni:

Kulcsszavak		Digráf		Trigráf	
and	&&	<%	{	??=	#
and_eq	&=	%>	}	??(	[
bitand	&	<:	[	??<	{
bitor	|	:>	]	??/	\
compl	~	%:	#	??)	]
not	!	%:%:	##	??>	}
or	||			??'	^
or_eq	|=			??!	|
xor	^			??-	~
xor_eq	^=				
not_eq	!=				

Azok a programok melyek ezeket a kulcsszavakat és digráf karaktereket használják, sokkal olvashatóbbak, mint a velük egyenértékű, de trigráf karakterekkel készült programok. Ha azonban az olyan karakterek, mint a { nem érhetők el rendszerünkben, kénytelenek vagyunk a trigráf karaktereket használni a karakterláncokban és karakter-konstansokban a hiányzó jelek helyett. A *'{'* helyett például a *'??<'* karakterkonstanst írhatjuk.

Egyes programozók hagyományos operátor-megfelelőik helyett szívesebben használják az *and* és hasonló kulcsszavakat.

C.3.2. Vezérlőkarakterek

A \ jel segítségével néhány olyan karaktert érhetünk el, melyeknek szabványos neve is van:

Név	ASCII jelölés	C++ jelölés
újsor/sortörés	NL (LF)	`\n`
vízszintes tabulátor	HT	`\t`
függőleges tabulátor	VT	`\v`
visszatörlés	BS	`\b`
kocsivissza	CR	`\r`
lapdobás	FF	`\f`
csengő	BEL	`\a`
fordított perjel	\	`\\`
kérdőjel	?	`\?`
aposztróf	'	`\'`
idézőjel	"	`\"`
oktális szám	*ooo*	*ooo*
hexadecimális szám	*hhh*	*xhhh*...

Megjelenési formájuk ellenére ezek is egyetlen karaktert jelentenek.

Lehetőségünk van arra is, hogy egy karaktert egy-, két- vagy háromjegyű, oktális (a \ után nyolcas számrendszerbeli), vagy hexadecimális (a *x* után 16-os számrendszerbeli) számként adjunk meg. A hexadecimális jegyek száma nem korlátozott. Az oktális vagy hexadecimális számjegyek sorozatának végét az első olyan karakter jelzi, amely nem értelmezhető oktális, illetve hexadecimális jegyként:

Oktális	Hexadecimális	Decimális	ASCII
`'\6'`	`'\x6'`	6	ACK
`'\60'`	`'\x30'`	48	`'0'`
`'\137'`	`'\x05f'`	95	`'_'`

Ez a jelölés lehetővé teszi, hogy az adott rendszer tetszőleges karakterét leírjuk vagy karakterláncokban felhasználjuk azokat (lásd §5.2.2). Viszont ha a karakterek jelölésére számokat használunk, programunk nem lesz átvihető más karakterkészletet használó rendszerre.

Lehetőség van arra is, hogy egy karakterliterálban egynél több karaktert adjunk meg (például *'ab'*). Ez a megoldás azonban elavult és megvalósítás-függő, így érdemes elkerülnünk.

Ha egy karakterláncban oktális konstansokat használunk a karakterek jelölésére, érdemes mindig három jegyet megadnunk. A jelölésrendszer elég kényelmetlen, így könnyen eltéveszthetjük, hogy a konstans utáni karakter számjegy-e vagy sem. A hexadecimális konstansok esetében használjunk két számjegyet:

```
char v1[] = "a\xah\129";     // 6 karakter: 'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";     // 5 karakter: 'a' '\xa' 'h' '\127' '\0'
char v3[] = "a\xad\127";     // 4 karakter: 'a' '\xad' '\127' '\0'
char v4[] = "a\xad\0127";    // 5 karakter: 'a' '\xad' '\012' '7' '\0'
```

C.3.3. Nagy karakterkészletek

C++ programot írhatunk olyan karakterkészletek felhasználásával is, melyek sokkal bővebbek, mint a 127 karakterből álló ASCII készlet. Ha az adott nyelvi változat támogatja a nagyobb karakterkészleteket, az azonosítók, megjegyzések, karakter konstansok és karakterláncok is tartalmazhatnak olyan karaktereket, mint az *å*, a *β* vagy a *Γ*. Ahhoz azonban, hogy az így készült program hordozható legyen, a fordítónak ezeket a különleges karaktereket valahogy olyan karakterekre kell „kódolnia”, melyek minden C++-változatban elérhetők. Ezt az átalakítást a C++ – a könyvben is használt – alap karakterkészletére a rendszer általában még azelőtt végrehajtja, hogy a fordító bármilyen más tevékenységbe kezdene, tehát a programok jelentését ez nem érinti.

A nagy karakterkészletekről a C++ által támogatott kisebb karakterkészletre való szabványos átalakítást négy vagy nyolc jegyből álló hexadecimális számok valósítják meg:

általános karakternév:

```
\U X X X X X X X X
\u X X X X
```

Itt *X* egyetlen, hexadecimális számjegyet jelöl (például *\u1e2b*). A rövidebb *\uXXXX* jelölés egyenértékű a *\U0000XXXX*-szel. Ha a megadott számban a jegyek száma nem négy és nem is nyolc, a fordító hibát jelez.

A programozó ezeket a karakterkódolásokat közvetlenül használhatja, de valójában arra találták ki, hogy a programozó által látható karaktereket a megvalósítás belsőleg egy kisebb karakterkészlettel tárolhassa.

Ha a bővített karakterkészletek azonosítókban való használatához egy adott környezet egyedi szolgáltatásaira támaszkodunk, programunk hordozhatóságát erősen lerontjuk. Ha nem ismerjük azt a természetes nyelvet, amelyet az azonosítók, illetve megjegyzések megfogalmazásához használtak, a program nagyon nehezen olvasható lesz, ezért a nemzetközileg használt programokban érdemes megmaradnunk az angol nyelvnél és az ASCII karakterkészletnél.

C.3.4. Előjeles és előjel nélküli karakterek

Megvalósítás-függő az is, hogy az egyszerű *char* előjeles vagy előjel nélküli típus-e, ami néhány kellemetlen meglepetést és váratlan helyzeteket eredményezhet:

```
char c = 255;       // a 255 "csupa egyes", hexadecimális jelöléssel 0xFF
int i = c;
```

Mi lesz itt az *i* értéke? Sajnos nem meghatározható. A válasz az általam ismert összes nyelvi változatban attól függ, hogy a *char* érték bitmintája milyen értéket eredményez, ha egészként értelmezzük. Egy SGI Challenge gépen a *char* előjel nélküli, így az eredmény *255*. Egy Sun SPARC vagy egy IBM PC gépen viszont a *char* előjeles, aminek következtében az *i* értéke *-1* lesz. Ilyenkor a fordítónak illik figyelmeztetnie, hogy a *255* literál karakterré alakításával a *-1* értéket kapjuk. A C++ nem ad általános megoldást ezen probléma kiküszöbölésére. Az egyik lehetőség, hogy teljesen elkerüljük az egyszerű *char* típus használatát és mindig valamelyik egyedi célú változatot használjuk. Sajnos a standard könyvtár bizonyos függvényei (például a *strcmp()*) egyszerű *char* típusú paramétert várnak (§20.4.1).

Egy *char* értéknek mindenképpen *signed char* vagy *unsigned char* típusúnak kell lennie, de mind a három karaktertípus különböző, így például a különböző *char* típusokra hivatkozó mutatók sem keverhetők össze:

```
void f(char c, signed char sc, unsigned char uc)
{
  char* pc = &uc;                 // hiba: nincs mutató-átalakítás
  signed char* psc = pc;          // hiba: nincs mutató-átalakítás
  unsigned char* puc = pc;        // hiba: nincs mutató-átalakítás
  psc = puc;                      // hiba: nincs mutató-átalakítás
}
```

A különböző *char* típusok változói szabadon értékül adhatók egymásnak, de amikor egy előjeles *char* változóba túl nagy értéket akarunk írni, az eredmény nem meghatározható lesz:

```
void f(char c, signed char sc, unsigned char uc)
{
  c = 255;  // megvalósítás-függő, ha a sima karakterek előjelesek és 8 bitesek

  c = sc;   // rendben
  c = uc;   // megvalósítás-függő, ha a sima karakterek előjelesek és 'uc' értéke túl nagy
  sc = uc;  // megvalósítás-függő, ha 'uc' értéke túl nagy
  uc = sc;  // rendben: átalakítás előjel nélkülire
  sc = c;   // megvalósítás-függő, ha a sima karakterek előjel nélküliek és 'c' értéke túl nagy
  uc = c;   // rendben: átalakítás előjel nélkülire
}
```

Ha mindenhol az egyszerű *char* típust használjuk, ezek a problémák nem jelentkeznek.

C.4. Az egész literálok típusa

Az egész literálok típusa általában alakjuktól, értéküktől és utótagjuktól is függ:

- Ha decimális értékről van szó és a végére nem írunk semmilyen utótagot, típusa az első olyan lesz a következők közül, amelyben ábrázolható: *int, long int, unsigned long int*.
- Ha oktális vagy hexadecimális formában, utótag nélkül adjuk meg az értéket, annak típusa az első olyan lesz a következők közül, amelyben ábrázolható: *int, unsigned int, long int, unsigned long int*.
- Ha az *u* vagy az *U* utótagot használjuk, a típus a következő lista első olyan típusa lesz, amelyben az érték ábrázolható: *unsigned int, unsigned long int*.
- Ha *l* vagy *L* utótagot adunk meg, az érték típusa a következő lista első olyan típusa lesz, amelyben az érték ábrázolható: *long int, unsigned long int*.
- Ha az utótag *ul, lu, uL, Lu, Ul, lU, UL* vagy *LU*, az érték típusa *unsigned long int* lesz.

A *100000* például egy olyan gépen, amelyen az *int* 32 bites, *int* típusú érték, de *long int* egy olyan gépen, ahol az *int* 16, a *long int* pedig 32 bites. Ugyanígy a *0XA000* típusa *int*,

ha az *int* 32 bites, de *unsigned int*, ha 16 bites. Ezeket a „megvalósítás-függőségeket" az utótagok használatával kerülhetjük el: a *100000L* minden gépen *long int* típusú, a *0XA000U* pedig minden gépen *unsigned int*.

C.5. Konstans kifejezések

Az olyan helyeken, mint a tömbhatárok (§5.2.), a *case* címkék (§6.3.2.) vagy a felsoroló elemek kezdőérték-adói (§4.8.), a C++ konstans kifejezéseket használ. A konstans kifejezés értéke egy szám vagy egy felsorolási konstans. Ilyen kifejezéseket literálokból (§4.3.1, §4.4.1, §4.5.1), felsoroló elemekből (§4.8) és konstans kifejezésekkel meghatározott *const* értékekből építhetünk fel. Sablonokban egész típusú sablonparamétereket is használhatunk (§C.13.3). Lebegőpontos literálok (§4.5.1) csak akkor használhatók, ha azokat meghatározott módon egész típusra alakítjuk. Függvényeket, osztálypéldányokat, mutatókat és hivatkozásokat csak a *sizeof* operátor (§6.2) paramétereként használhatunk.

A konstans kifejezés egy olyan egyszerű kifejezés, melyet a fordító ki tud értékelni, mielőtt a programot összeszerkesztenénk és futtatnánk.

C.6. Automatikus típusátalakítás

Az egész és a lebegőpontos típusok (§4.1.1) szabadon keverhetők az értékadásokban és a kifejezésekben. Ha lehetőség van rá, a fordító úgy alakítja át az értékeket, hogy ne veszítsünk információt. Sajnos az információvesztéssel járó átalakítások is automatikusan végrehajtásra kerülnek. Ebben a részben az átalakítási szabályokról, az átalakítások problémáiról és ezek következményeiről lesz szó.

C.6.1. Kiterjesztések

Azokat az automatikus átalakításokat, melyek megőrzik az értékeket, közösen ***kiterjesztéseknek*** (promotion) nevezzük. Mielőtt egy aritmetikai műveletet végrehajtunk, egy ***egész típusú kiterjesztés*** az *int* típusnál kisebb értékeket *int* értékre alakítja. Figyeljünk rá, hogy

ezek a kiterjesztések nem *long* típusra alakítanak (hacsak valamelyik operandus nem *wchar_t* vagy olyan felsoroló érték, amely már eleve nagyobb, mint egy *int*). Ez a kiterjesztés eredeti célját tükrözi a C nyelvben: az operandusokat „természetes” méretűre akarjuk alakítani az aritmetikai műveletek elvégzése előtt.

Az egész típusú kiterjesztések a következők:

- A *char, signed char, unsigned char, short int* és *unsigned short int* értékeket *int* típusúra alakítja a rendszer, ha az *int* az adott típus összes értékét képes ábrázolni; ellenkező esetben a céltípus *unsigned int* lesz.
- A *wchar_t* típust (§4.3) és a felsoroló típusokat (§4.8) a rendszer a következő lista első olyan típusára alakítja, amely ábrázolni tudja a megfelelő típus összes értékét: *int, unsigned int, long, unsigned long*.
- A bitmezők (§C.8.1) *int* típusúak lesznek, ha az képes a bitmező összes értékét ábrázolni. Ha az *int* nem, de az *unsigned int* képes erre, akkor az utóbbi lesz a céltípus. Ha ez sem valósítható meg, akkor nem következik be egész típusú kiterjesztés.
- Logikai értékek *int* típusra alakítása: a *false* értékből *0*, a *true* értékből *1* lesz.

A kiterjesztések a szokásos aritmetikai átalakítások részét képezik. (§C.6.3)

C.6.2. Átalakítások

Az alaptípusok számos módon alakíthatók át: véleményem szerint túl sok is az engedélyezett átalakítás:

```
void f(double d)
{
   char c = d;          // vigyázat: kétszeres pontosságú lebegőpontos érték átalakítása

karakterre
}
```

Amikor programot írunk, mindig figyelnünk kell arra, hogy elkerüljük a nem meghatározott szolgáltatásokat és azokat az átalakításokat, melyek „szó nélkül” tüntetnek el információkat. A fordítók a legtöbb veszélyes átalakításra figyelmeztethetnek és szerencsére általában meg is teszik.

C.6.2.1. Egész átalakítások

Egy egész érték bármely más egész típusra átalakítható, de a felsoroló értékeket is egész típusokra alakíthatjuk.

Ha a céltípus előjel nélküli, az eredmény egyszerűen annyi bit lesz a forrásból, amennyi a céltípusban elfér. (Ha kell, a magas helyiértékékű biteket a rendszer eldobja.) Pontosabban fogalmazva, az eredmény a legkisebb olyan egész, amelynek osztása a 2 *n*-edik hatványával azonos értéket ad, mint a forrásérték hasonló osztása, ahol *n* az előjel nélküli típus ábrázolásához használt bitek száma. Például:

```
unsigned char uc = 1023;    // bináris 1111111111: uc bináris 11111111 lesz; ami 255
```

Ha a céltípus előjeles és az átalakítandó érték ábrázolható vele, az érték nem változik meg. Ha a céltípus nem tudja ábrázolni az értéket, az eredmény az adott nyelvi változattól függő lesz:

```
signed char sc = 1023;      // megvalósítás-függő
```

A lehetséges eredmények: *127* és *-1* (§C.3.4).

A logikai és felsoroló értékek automatikusan a megfelelő egész típusra alakíthatók (§4.2, §4.8).

C.6.2.2. Lebegőpontos átalakítások

Lebegőpontos értékeket más lebegőpontos típusokra alakíthatunk át. Ha a forrásérték pontosan ábrázolható a céltípusban, az eredmény az eredeti számérték lesz. Ha a forrásérték a céltípus két egymást követő ábrázolható értéke között áll, eredményként ezen két érték valamelyikét kapjuk. A többi esetben az eredmény nem meghatározható:

```
float f = FLT_MAX;          // a legnagyobb lebegőpontos érték
double d = f;               // rendben: d == f
float f2 = d;               // rendben: f2 == f
double d3 = DBL_MAX;        // a legnagyobb double érték
float f3 = d3;              // nem meghatározott, ha FLT_MAX<DBL_MAX
```

C.6.2.3. Mutató- és hivatkozás-átalakítások

Az objektumtípusra hivatkozó mutatók mind *void** (§5.6) típusúra alakíthatók; a leszármazott osztályra hivatkozó mutatók és hivatkozások bármelyike átalakítható egy elérhető és egyértelmű alaposztályra hivatkozó mutató, illetve hivatkozás típusára (§12.2). Fontos viszont, hogy egy függvényre vagy egy tagra hivatkozó mutató automatikusan nem alakítható *void** típusra.

A *0* értékű konstans kifejezések (§C.5) bármilyen mutató és tagra hivatkozó mutató (§5.1.1) típusra automatikusan átalakíthatók:

```
int* p =
           !        !     !     !         !          !
         !   !    !       !     !         !          !
       !      !  !        !     !         !          !
     !        !           ! ! ! ! ! !     ! ! ! ! !    ! ! ! ! 1;
```

Egy *T** érték automatikusan *const T** típusra alakítható (§5.4.1) és ugyanígy egy *T&* érték is átalakítható *const T&* típusra.

C.6.2.4. Tagra hivatkozó mutatók átalakítása

A tagokra hivatkozó mutatók és hivatkozások úgy alakíthatók át automatikusan, ahogy a §15.5.1 pontban leírtuk.

C.6.2.5. Logikai értékek átalakítása

A mutatók, egészek és lebegőpontos értékek automatikusan *bool* típusúvá alakíthatók (§4.2). A nem nulla értékek eredménye *true*, a nulla eredménye *false* lesz:

```
void f(int* p, int i)
{
    bool is_not_zero = p;     // igaz, ha p!=0
    bool b2 = i;              // igaz, ha i!=0
}
```

C.6.2.6. Lebegőpontos–egész átalakítások

Amikor egy lebegőpontos értéket egésszé alakítunk, a törtrész elveszik, vagyis a lebegőpontosról egészre való átalakítás csonkolást eredményez. Az *int(1.6)* értéke például *1* lesz. Ha a csonkolással keletkező érték sem ábrázolható a céltípusban, az eredmény nem meghatározhatónak minősül:

```
int i = 2.7;        // i értéke 2 lesz
char b = 2000.7;    // nem meghatározott 8 bites char típusra: a 2000 nem ábrázolható
                    // 8 bites karakterben
```

Az egészről lebegőpontosra való átalakítás matematikailag annyira pontos, amennyire a hardver megengedi. Ha egy egész érték egy lebegőpontos típus értékeként nem ábrázolható, az eredmény nem lesz egészen pontos:

```
int i = float(1234567890);
```

Itt az *i* értéke *1234567936* lesz az olyan gépeken, amely az *int* és a *float* ábrázolására is 32 bitet használnak.

Természetesen célszerű elkerülni az olyan automatikus átalakításokat, ahol információt veszíthetünk. A fordítók képesek észlelni néhány veszélyes átalakítást, például a lebegőpontosról egészre vagy a *long int* típusról *char* típusra valót, ennek ellenére az általános, fordítási időben történő ellenőrzés nem mindig kellően megbízható, ezért a programozónak elővigyázatosnak kell lennie. Amennyiben az elővigyázatosság nem elegendő, a programozónak ellenőrzéseket kell beiktatnia a hibák elkerülésére:

```
class check_failed { };

char checked(int i)
{
  char c = i;                          // figyelem: nem hordozható (§C.6.2.1)
  if (i != c) throw check_failed();
  return c;
}
void my_code(int i)
{
  char c = checked(i);
  // ...
}
```

Ha úgy szeretnénk csonkolást végezni, hogy a program más rendszerre is változtatás nélkül átvihető legyen, vegyük figyelembe a *numeric_limits*-ben (§22.2) megadottakat.

C.6.3. Szokásos aritmetikai átalakítások

Ezek az átalakítások akkor következnek be, ha egy kétoperandusú (bináris) művelet két operandusát közös típusúra kell alakítani. Ez a közös típus határozza meg az eredmény típusát is.

1. Ha az egyik operandus *long double* típusú, a rendszer a másikat is *long double* típusúvá alakítja.
   - Ha egyik sem *long double*, de valamelyik *double*, a másik is *double* típusúvá alakul.
   - Ha egyik sem *double*, de valamelyik *float*, a másik is *float* értékre lesz átalakítva.
   - Ha a fentiek egyike sem teljesül, a rendszer mindkét operandusra egész kiterjesztést (§C.6.1) alkalmaz.
2. Ezután, ha valamelyik operandus *unsigned long*, akkor a másik is *unsigned long* lesz.
   - Ha a fenti nem teljesül, de az egyik operandus *long int*, a másik pedig *unsigned int*, és a *long int* típus képes ábrázolni az összes *unsigned int* értéket, akkor az *unsigned int* értéket a fordító *long int* típusúra alakítja. Ha a *long int* nem elég nagy, mindkét értéket *unsigned long int* típusra kell alakítani.
   - Ha a fenti nem teljesül, de valamelyik operandus *long*, a másik is *long* típusúra alakul.
   - Ha valamelyik operandus *unsigned*, a másik is *unsigned* értékre alakul.
   - Ha a fentiek egyike sem teljesül, mindkét érték *int* lesz.

C.7. Többdimenziós tömbök

Nem ritka, hogy vektorok vektorára van szükségünk, sőt vektorok vektorainak vektorára. A kérdés az, hogy a C++-ban hogyan ábrázolhatjuk ezeket a többdimenziós tömböket. Ebben a pontban először megmutatjuk, hogyan használjuk a standard könyvtár *vector* osztályát erre a célra, majd megvizsgáljuk, hogyan kezelhetők a többdimenziós tömbök a C-ben és a C++-ban akkor, ha csak a beépített lehetőségek állnak rendelkezésünkre.

C.7.1. Vektorok

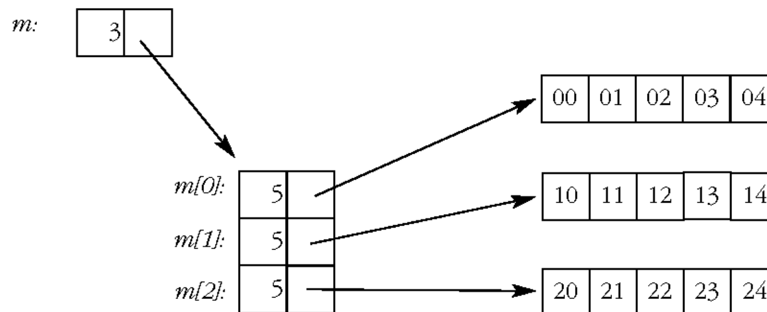
A szabványos *vector* (§16.3) egy nagyon általános megoldást kínál:

```
vector< vector<int> > m(3, vector<int>(5));
```

Ezzel az utasítással egy 3 olyan vektort tartalmazó vektort hozunk létre, melyek mindegyike 5 darab egész érték tárolására képes. Mind a 15 egész elem a *0* alapértelmezett értéket kapja, melyet a következő módon módosíthatunk:

```
void init_m()
{
  for (int i = 0; i<m.size(); i++) {
            for (int j = 0; j<m[i].size(); j++) m[i][j] = 10*i+j;
  }
}
```

Grafikusan ezt így ábrázolhatjuk:



A *vector* objektumot egy, az elemekre hivatkozó mutatóval és az elemek számával ábrázoltuk. Az elemeket általában egy szokásos tömbben tároljuk. Szemléltetésképpen mindegyik egész elembe a koordinátáinak megfelelő értéket írtuk.

Az egyes elemeket kétszeres indexeléssel érhetjük el. Az *m[i][j]* kifejezés például az *i-edik* vektor *j*-edik elemét választja ki. Az *m* elemeit a következőképpen írathatjuk ki:

```
void print_m()
{
  for (int i = 0; i<m.size(); i++) {
              for (int j = 0; j<m[i].size(); j++) cout << m[i][j] << '\t';
              cout << '\n';
  }
}
```

Ennek eredménye a következő lesz:

```
0 1         2          3          4
10          11         12         13         14
20          21         22         23         24
```

Figyeljük meg, hogy az *m* vektorok vektora, nem egyszerű többdimenziós tömb, ezért a gyakorlatban lehetőségünk van arra, hogy egyesével módosítsuk a belső vektorok méretét:

```
void reshape_m(int ns)
{
  for (int i = 0; i<m.size(); i++)  m[i].resize(ns);
}
```

A *vector< vector<int> >* szerkezetben szereplő *vector<int>* vektorok méretének nem feltétlenül kell azonosnak lennie.

C.7.2. Tömbök

A beépített tömbök jelentik a C++-ban a legfőbb hibaforrást, különösen ha többdimenziós tömbök készítésére használjuk fel azokat. Kezdők számára ezek okozzák a legtöbb keveredést is, ezért amikor lehet, használjuk helyettük a *vector*, *list*, *valarray*, *string* stb. osztályokat.

A többdimenziós tömböket tömbök tömbjeként ábrázolhatjuk. Egy 3x5-ös méretű tömböt az alábbi módon vezethetünk be:

```
int ma[3][5];              // 3 tömb, mindegyikben 5 int elem
```

Az *ma* adatszerkezetet a következőképpen tölthetjük fel:

```
void init_ma()
{
  for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) ma[i][j] = 10*i+j;
  }
}
```

Ábrával:

ma:

00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

Az *ma* tömb egyszerűen 15 egész értéket tárol, melyeket úgy érünk el, mintha 3 darab 5 elemű tömbről lenne szó. Tehát a memóriában nincs olyan objektum, amely magát az *ma* mátrixot jelenti, csak az önálló elemeket tároljuk. A *3* és *5* dimenzióértékek csak a forrásban jelennek meg. Amikor ilyen programot írunk, nekünk kell valahogy emlékeznünk az értékekre. Az *ma* tömböt például a következő kódrészlet segítségével írathatjuk ki:

```
void print_ma()
{
  for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) cout << ma[i][j] << '\t';
        cout << '\n';
  }
}
```

Az a vesszős jelölés, amit néhány nyelv a többdimenziós tömbök kezeléséhez biztosít, a C++-ban nem használható, mert a vessző (,) a műveletsorozat-operátor (§6.2.2). Szerencsére a fordító a legtöbb hibára figyelmeztethet:

```
int bad[3,5];             // hiba: a vessző nem megengedett konstans kifejezésben
int good[3][5];           // 3 tömb, mindegyikben 5 int elem
int ouch = good[1,4];     // hiba: int kezdőértéke nem lehet int* típusú (good[1,4]
                          // jelentése good[4], ami int* típus)
int nice = good[1][4];    // helyes
```

C.7.3. Többdimenziós tömbök átadása

Képzeljünk el egy függvényt, mellyel egy többdimenziós tömböt akarunk feldolgozni. Ha a dimenziókat fordításkor ismerjük, nincs probléma:

```
void print_m35(int m[3][5])
{
  for (int i = 0; i<3; i++) {
            for (int j = 0; j<5; j++) cout << m[i][j] << '\t';
            cout << '\n';
  }
}
```

A többdimenziós tömbként megjelenő mátrixokat egyszerűen mutatóként adjuk át (nem készül róla másolat, §5.3). Az első dimenzió érdektelen az egyes elemek helyének meghatározásakor, csak azt rögzíti, hogy az adott típusból (esetünkben az *int[5]*-ből) hány darab áll egymás után (esetünkben *3*). Például, nézzük meg az *ma* előbbi ábrázolását. Megfigyelhetjük, hogy ha csak annyit tudunk, hogy a második dimenzió *5*, akkor is bármely *ma[i][5]* elem helyét meg tudjuk állapítani. Ezért az első dimenziót átadhatjuk külön paraméterként:

```
void print_mi5(int m[][5], int dim1)
{
  for (int i = 0; i<dim1; i++) {
            for (int j = 0; j<5; j++) cout << m[i][j] << '\t';
            cout << '\n';
  }
}
```

A problémás eset az, ha mindkét dimenziót paraméterként akarjuk átadni. A „nyilvánvaló megoldás" nem működik:

```
void print_mij(int m[][], int dim1, int dim2)      // nem úgy viselkedik,
                                                   // ahogy legtöbben gondolnánk
{
  for (int i = 0; i<dim1; i++) {
            for (int j = 0; j<dim2; j++) cout << m[i][j] << '\t';        // meglepetés!
            cout << '\n';
  }
}
```

Először is, az *m[][]* deklaráció hibás, mert egy többdimenziós tömb esetében a második dimenziót ismernünk kell ahhoz, hogy az elemek helyét meg tudjuk határozni. Másrészt, az *m[i][j]* kifejezést (teljesen szabályosan) **(*(m+i)+j)*-ként értelmezi a fordító, ami általában nem azt jelenti, amit a programozó ki akart vele fejezni. A helyes megoldás a következő:

```
void print_mij(int* m, int dim1, int dim2)
{
  for (int i = 0; i<dim1; i++) {
            for (int j = 0; j<dim2; j++) cout << m[i*dim2+j] << '\t'; // zavaros
            cout << '\n';
  }
}
```

A *print_mij()* függvényben az elemek eléréséhez használt kifejezés egyenértékű azzal, amit a fordító állít elő, amikor ismeri az utolsó dimenziót.

Ennek a függvénynek a meghívásához a mátrixot egyszerű mutatóként adjuk át:

```
int main()
{
  int v[3][5] = { {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24} };

  print_m35(v);
  print_mi5(v,3);
  print_mij(&v[0][0],3,5);
}
```

Figyeljük meg az utolsó sorban a *&v[0][0]* kifejezés használatát. A *v[0]* szintén megfelelne, mert az előbbivel teljesen egyenértékű, a *v* viszont típushibát okozna. Az ilyen „csúnya” és körülményes programrészleteket jobb elrejteni. Ha közvetlenül többdimenziós tömbökkel kell foglalkoznunk, próbáljuk különválasztani a vele foglalkozó programrészleteket. Ezzel leegyszerűsíthetjük a következő programozó dolgát, akinek majd a programhoz hozzá kell nyúlnia. Ha külön megadunk egy többdimenziós tömb típust és benne egy megfelelő indexelő operátort, a legtöbb felhasználót megkímélhetjük attól, hogy az adatok fizikai elrendezésével kelljen foglalkoznia (§22.4.6).

A szabványos *vector* (§16.3) osztályban ezek a problémák már nem jelentkeznek.

C.8. Ha kevés a memória...

Ha komolyabb alkalmazást készítünk, gyakran több memóriára lenne szükségünk, mint amennyi rendelkezésünkre áll. Két módszer van arra, hogy több helyet préseljünk ki:

1. Egyetlen bájtban több kisebb objektumot tárolhatunk.
2. Ugyanazt a területet különböző időpontokban különböző objektumok tárolására használhatjuk.

Az előbbit a *mezők*, az utóbbit az *uniók* segítségével valósíthatjuk meg. Ezekről az eszközökről a korábbiakban már volt szó. A mezőket és az uniókat szinte csak optimalizáláshoz használjuk, ami gyakran a memória adott rendszerbeli felépítésén alapul, így a program nem lesz más rendszerre átvihető. Tehát érdemes kétszer is meggondolnunk, mielőtt ezeket a szerkezeteket használjuk. Gyakran jobb megoldást jelent, ha az adatokat „másképp" kezeljük, például több dinamikusan lefoglalt területet (§6.2.6) és kevesebb előre lefoglalt (statikus) memóriát használunk.

C.8.1. Mezők

Gyakran rettentő pazarlásnak tűnik, hogy egy bináris változót – például egy ki/be kapcsolót – egy teljes bájt (*char* vagy *bool*) felhasználásával ábrázolunk, de ez a legkisebb egység, amelyet a memóriában közvetlenül megcímezhetünk a C++ segítségével (§5.1). Viszont, ha a *struct* típusban mezőket használunk, lehetőség van arra, hogy több ilyen kis változót egyetlen csomagba fogjunk össze. Egy tag akkor válik mezővé, ha utána megadjuk az általa elfoglalt bitek számát. Névtelen mezők is megengedettek. Ezek a C++ szempontjából nincsenek hatással az elnevezett mezők jelentésére, segítségükkel azonban pontosan leírhatunk bizonyos gépfüggő adatokat:

```
struct PPN {                    // R6000 fizikai lapszám
    unsigned int PFN : 22;      // lapkeret-szám
    int : 3;                    // nem használt
    unsigned int CCA : 3;       // Cache Coherency Algorithm
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

Ezzel a példával a mezők másik legfontosabb felhasználási területét is bemutattuk: valamilyen külső (nem C++) szerkezet részeit is pontosan elnevezhetjük velük. A mezők egész vagy felsoroló típusúak (§4.1.1). Egy mezőnek nem lehet lekérdezni a címét, viszont ettől

eltekintve teljesen átlagos változóként kezelhetjük. Figyeljük meg, hogy egy *bool* típusú értéket itt tényleg egy bittel ábrázolhatunk. Egy operációs rendszer magjában vagy egy hibakeresőben a *PPN* típus a következőképpen használható:

```
void part_of_VM_system(PPN* p)
{
  // ...

  if (p->dirty) {        // a tartalom megváltozott
                          // lemezre másolás
          p->dirty = 0;
  }

  // ...
}
```

Meglepő módon, ha több változót mezők segítségével egyetlen bájtba sűrítünk össze, akkor sem feltétlenül takarítunk meg memóriát. Az adatterület csökken, de az ilyen adatok kezeléséhez szükséges kód a legtöbb számítógépen jelentősen nagyobb. A programok mérete jelentősen csökkenthető azzal, hogy a bitmezőkként ábrázolt bináris adatokat *char* típusra alakítjuk, ráadásul egy *char* vagy *int* elérése általában sokkal gyorsabb, mint a bitmezők elérése. A mezők csak egy kényelmes rövidítést adnak a bitenkénti logikai műveletekhez (§6.2.4), hogy egy gépi szó részeibe egymástól függetlenül tudjunk adatokat írni, illetve onnan kiolvasni.

C.8.2. Uniók

Az *unió* olyan *struct*, melyben minden tag ugyanarra a címre kerül a memóriában, így az unió csak annyi területet foglal, mint a legnagyobb tagja. Természetesen az unió egyszerre csak egy tagjának értékét tudja tárolni. Képzeljünk el például egy szimbólumtábla-bejegyzést, amely egy nevet és egy értéket tárol:

```
enum Type { S, I };

struct Entry {
  char* name;
  Type t;
  char* s;          // s használata, ha t==S
  int i;            // i használata, ha t==I
};

void f(Entry* p)
{
  if (p->t == S) cout << p->s;
  // ...
}
```

Az *s* és az *i* tagot sohasem fogjuk egyszerre használni, így feleslegesen pazaroltuk a memóriát. Ezen a problémán könnyen segíthetünk a *union* adatszerkezet segítségével:

```
union Value {
  char* s;
  int i;
};
```

A rendszer nem tartja nyilván, hogy a *union* éppen melyik értéket tárolja, tehát ezt továbbra is a programozónak kell megtennie:

```
struct Entry {
  char* name;
  Type t;
  Value v;          // v.s használata, ha t==S; v.i használata, ha t==I
};

void f(Entry* p)
{
  if (p->t == S) cout << p->v.s;
  // ...
}
```

Sajnos a *union* bevezetése arra kényszerít minket, hogy az egyszerű *s* helyett a *v.s* kifejezést használjuk. Ezt a problémát a *névtelen unió* segítségével kerülhetjük el, amely egy olyan unió, amelynek nincs neve, így típust sem ad meg, csak azt biztosítja, hogy tagjai ugyanarra a memóriacímre kerüljenek:

```
struct Entry {
  char* name;
  Type t;
  union {
        char* s;        // s használata, ha t==S
        int i;          // i használata, ha t==I
  };
};

void f(Entry* p)
{
  if (p->t == S) cout << p->s;
  // ...
}
```

Így az *Entry* szerkezetet használó programrészleteken nem kell változtatnunk. A *union* típus olyan felhasználásakor, amikor egy értéket mindig abból a tagból olvasunk vissza, amelyiken keresztül írtuk, csak optimalizálást hajtunk végre. Az unió ilyen jellegű felhasználá-

sát azonban nem mindig könnyű biztosítani és a helytelen használat nagyon kellemetlen hibákat okozhat. A hibák elkerülése érdekében az uniót befoglalhatjuk egy olyan egységbe, amely biztosítja, hogy az érték típusa és a hozzáférés módja egymásnak megfelelő lesz (§10.6[20]). Az uniót néha szándékosan helytelenül használjuk, valamiféle „típusátalakítás" érdekében. Ezt a lehetőséget általában azok a programozók használják, akik az explicit típusátalakítást nem támogató nyelven is írnak programokat. Ott az ilyen „csalásra" tényleg szükség van. Az alábbi szerkezettel az *int* és az *int** típus közötti átalakítást próbáljuk megvalósítani, egyszerű bitenkénti egyenértékűség feltételezésével:

```
union Fudge {
  int i;
  int* p;
};

int* cheat(int i)
{
  Fudge a;
  a.i = i;
  return a.p;          // hibás használat
}
```

Ez valójában nem is igazi átalakítás. Bizonyos gépeken az *int* és az *int** nem ugyanannyi helyet foglal, más gépeken az egész értékeknek nem lehet páratlan címe. Tehát az unió ilyen felhasználása nagyon veszélyes és nem is vihető át más rendszerre, ráadásul ugyanerre a feladatra van egy egyszerűbb, biztonságosabb és „hordozható" megoldás: a meghatározott (explicit) típusátalakítás (§6.2.7).

Az uniót időnként szándékosan a típusátalakítás elkerülésére használják. Például szükségünk lehet arra, hogy a *Fudge* segítségével megállapítsuk, hogy a *0* mutatót hogyan ábrázolja a rendszer:

```
int main()
{
  Fudge foo;
  foo.p = 0;
  cout << "A 0 mutató egész értéke: " << foo.i << '\n';
}
```

C.8.3. Uniók és osztályok

A bonyolultabb uniók esetében gyakran előfordul, hogy egyes tagok jóval nagyobbak, mint a rendszeresen használt tagok. Mivel a union mérete legalább akkora, mint a legnagyobb tagja, sok helyet elpazarolhatunk így. A pazarlást általában elkerülhetjük, ha unió helyett származtatott osztályokat használunk.

Egy konstruktorral, destruktorral és másoló művelettel rendelkező osztály nem tartozhat egy unió tag típusába (§10.4.12), hiszen a fordító nem tudja, melyik tagot kell törölnie.

C.9. Memóriakezelés

A C++-ban a memória kezelésére három alapvető módszer áll rendelkezésünkre:

Statikus memória: Ebben az esetben az objektum számára az összeszerkesztő (linker) foglal helyet a program teljes futási idejére. Statikus memóriában kapnak helyet a globális és névtér-változók, a *static* adattagok (§10.2.4) és a függvényekben szereplő *static* változók (§7.1.2). Azok az objektumok, melyek a statikus memóriában kapnak helyet, egyszer jönnek létre és a program végéig léteznek. Címük a program futása közben nem változik meg. A statikus objektumok a szálakat (osztott című memóriaterületeket párhuzamosan) használó programokban problémát jelenthetnek, mert a megfelelő eléréshez zárolásokat kell alkalmaznunk.

Automatikus memória: Ezen a területen a függvényparaméterek és helyi változók jönnek létre. A függvény vagy blokk minden egyes lefutásakor saját példányok jönnek létre az ilyen változókból. Ezt a típusú memóriát automatikusan foglalja le és szabadítja fel a rendszer, ezért kapta az automatikus memória nevet.
Az automatikus memóriára gyakran hivatkozunk úgy, hogy az elemek „a veremben helyezkednek el". Ha feltétlenül hangsúlyozni akarjuk ezt a típusú helyfoglalást, a C++-ban az *auto* kulcsszót használhatjuk.

Szabad tár: Erről a területről a program bármikor közvetlenül igényelhet memóriát és bármikor felszabadíthatja az itt lefoglalt területeket (a *new*, illetve a *delete* operátor segítségével). Amikor a programnak újabb területekre van szüksége a szabad tárból, a *new* operátorral igényelhet az operációs rendszertől. A szabad tárra gyakran hivatkozunk *dinamikus memória* vagy *kupac* (heap) néven is. A szabad tár a program teljes élettartama alatt csak nőhet, mert az ilyen területeket a program befejeződéséig nem adjuk vissza az operációs rendszernek, így más programok azokat nem használhatják.

A programozó szemszögéből az automatikus és a statikus tár egyszerűen, biztonságosan és automatikusan kezelhető. Az érdekes kérdést a szabad tár kezelése jelenti. A *new* segítségével könnyen foglalhatunk területeket, de ha nincs következetes stratégiánk a memória felszabadítására (a memória visszaadására a szabad tár kezelőjének), a memória előbb-utóbb elfogy, főleg ha programunk elég sokáig fut.

A legegyszerűbb módszer, ha automatikus objektumokat használunk a szabad tárban levő objektumok kezeléséhez. Ennek megfelelően a tárolókat sokszor úgy valósítják meg, hogy a szabad tárban levő elemekre mutatókat állítanak (§25.7). Az automatikus *String*-ek (§11.12) például a szabad tárban levő karaktersorozatokat kezelik és automatikusan felszabadítják a területet, amikor kikerülnek a hatókörből. Az összes szabványos tároló (§16.3, 17. fejezet, 20. fejezet, §22.4) kényelmesen megvalósítható ilyen formában.

C.9.1. Automatikus szemétgyűjtés

Ha ez az egyszerű megoldás nem felel meg igényeinknek, akkor használhatunk valamilyen memóriakezelő rendszert, amely megkeresi az olyan objektumokat, melyekre már nincs hivatkozás, és az ezek által lefoglalt memóriaterületet elérhetővé teszi új objektumok számára. Ezt a rendszert általában ***automatikus szemétgyűjtő algoritmusnak*** vagy egyszerűen ***szemétgyűjtőnek*** (garbage collection) nevezzük.

A szemétgyűjtés alapötlete az, hogy ha egy objektumra már sehonnan sem hivatkozunk a programból, arra a későbbiekben sem fogunk, tehát az általa lefoglalt memóriaterületet biztonságosan felszabadíthatjuk vagy odaadhatjuk más objektumoknak:

```
void f()
{
   int* p = new int;
   p = 0;
   char* q = new char;
}
```

Itt a *p=0* értékadás törli az egyetlen hivatkozást, ami a *p* által eddig meghatározott *int* objektumra mutatott, így ezen *int* érték területét odaadhatjuk egy új objektumnak. A *char* típusú objektum tehát már szerepelhet ugyanezen a memóriaterületen, ami azt jelenti, hogy a *q* ugyanazt az értéket kapja, mint eredetileg a *p*.

A szabvány nem követeli meg, hogy minden nyelvi változat tartalmazza a szemétgyűjtést, de egyre gyakrabban használják az olyan területeken, ahol a C++-ban a szabad tár egyéni kezelése költségesebb, mint a szemétgyűjtés. Amikor a két módszer költségeit összehasonlítjuk, figyelembe kell vennünk a futási időt, a memória-felhasználást, a megbízhatóságot, a hordozhatóságot, a programozás pénzbeli költségeit és a teljesítmény megjósolhatóságát is.

C.9.1.1. Rejtett mutatók

Mikor nevezhetünk egy objektumot hivatkozás nélkülinek? Vizsgáljuk meg például az alábbi programrészletet:

```
void f()
{
  int* p = new int;
  long i1 = reinterpret_cast<long>(p)&0xFFFF0000;
  long i2 = reinterpret_cast<long>(p)&0x0000FFFF;
  p = 0;
  // #1 pont: nem létezik az int-re hivatkozó mutató

  p = reinterpret_cast<int*>(i1 | i2);
  // itt az int-re ismét létezik mutató
}
```

Ha egy programban bizonyos mutatókat időnként nem mutatóként tárolunk, akkor „rejtett mutatókról" beszélünk. Esetünkben azt a mutatót, amit eredetileg a *p* változó tárolt, elrejtettük az *i1* és az *i2* egészekben. A szemétgyűjtő rendszertől azonban nem várhatjuk el, hogy kezelni tudja a rejtett mutatókat, tehát amikor a program a *#1* ponthoz ér, a szemétgyűjtő felszabadítja az *int* érték számára lefoglalt területet. Valójában az ilyen programok helyes működését még az olyan rendszerekben sem garantálhatjuk, ahol nem használunk szemétgyűjtő algoritmust, mert a *reinterpret_cast* használata egészek és mutatók közötti típusátalakításra még a legjobb esetben is megvalósítás-függő.

Az olyan *union* objektumok, melyek lehetővé teszik mutatók és nem mutatók tárolását is, külön problémát jelentenek a szemétgyűjtők számára. Általában nincs mód annak megállapítására, hogy egy ilyen szerkezet éppen mutatót tárol-e:

```
union U {          // unió mutató és nem mutató taggal
  int* p;
  int i;
};

void f(U u, U u2, U u3)
{
  u.p = new int;
  u2.i = 999999;
  u.i = 8;
  // ...
}
```

A biztonságos feltételezés az, hogy minden érték mutató, ami egy ilyen *union* objektumban megjelenik. Egy kellően okos szemétgyűjtő ennél kicsit jobb megoldást is adhat. Például észreveheti, hogy (az adott rendszerben) *int* értékek nem helyezhetők el páratlan memóriacímen és semmilyen objektumot nem hozhatunk létre olyan kicsi memóriacímen, mint a *8*. Ezek figyelembevételével a szemétgyűjtő algoritmusnak nem kell feltételeznie, hogy a *999999* vagy a *8* címen értelmes objektum szerepel, amit az *f()* esetleg felhasznál.

C.9.1.2. A delete

Ha rendszerünk automatikus szemétgyűjtést használ, a memória felszabadításához nincs szükség a *delete* és a *delete[]* operátorokra, tehát a szemétgyűjtést használó programozóknak nem kell ezeket használniuk. A memória felszabadításán kívül azonban a *delete* és a *delete[]* destruktorok meghívására is használatos.

Ha szemétgyűjtő algoritmust használunk, a

```
delete p;
```

utasítás meghívja a *p* által mutatott objektum destruktorát (ha van ilyen), viszont a memória újrafelhasználását elhalaszthatjuk addig, amíg a memóriaterületre szükség lesz. Ha sok objektumot egyszerre szabadítunk fel, csökkenthetjük a memória feltöredeződésének mértékét (§C.9.1.4) Ez a megoldás ártalmatlanná teszi azt az egyébként rendkívül veszélyes hibát is, hogy ugyanazt az objektumot kétszer semmisítjük meg, amikor a destruktor csak a memória törlését végzi.

Az olyan objektumok elérése, melyeket már töröltünk, szokás szerint nem meghatározható eredménnyel jár.

C.9.1.3. Destruktorok

Amikor a szemétgyűjtő algoritmus egy objektumot meg akar semmisíteni, két lehetőség közül választhat:

1. Meghívja az adott objektum destruktorát (ha az létezik).
2. Feltételezi, hogy nyers memóriaterületről van szó és nem hív meg destruktort.

Alapértelmezés szerint a szemétgyűjtő algoritmus a második lehetőséget használja, mert a *new* segítségével létrehozott, de a *delete* operátorral nem törölt objektumokat nem kell megsemmisítenünk. Tehát a szemétgyűjtő algoritmust bizonyos szempontból végtelen mé-

retű memória utánzásának tekinthetjük. Lehetőség van arra is, hogy a szemétgyűjtő algoritmust úgy valósítsuk meg, hogy a destruktor az algoritmus által „bejegyzett” objektumokra fusson le. A „bejegyzésre” azonban nincs szabványos megoldás. Figyeljünk arra, hogy az objektumokat mindig olyan sorrendben kell megsemmisítenünk, hogy az egyik objektum destruktora soha ne hivatkozzon olyan objektumra, amelyet korábban már töröltünk. A programozó segítsége nélkül ilyen sorrendet a szemétgyűjtő algoritmusok nagyon ritkán képesek betartani.

C.9.1.4. A memória feltöredeződése

Ha sok különböző méretű objektumot hozunk létre, illetve semmisítünk meg, a memória „feltöredeződik”. Ez azt jelenti, hogy a memóriát olyan kis darabokban használjuk, amelyek túl kicsik ahhoz, hogy hatékonyan kezelhetők legyenek. Ennek oka az, hogy a memóriakezelő rendszerek nem mindig találnak pontosan olyan méretű memóriadarabot, amilyenre egy adott objektumnak éppen szüksége van. Ha a kelleténél nagyobb területet használunk, a megmaradó memóriatöredék még kisebb lesz. Ha egy ilyen egyszerű memóriakezelővel programunk elég sokáig fut, nem ritka, hogy akár a rendelkezésre álló memória felét olyan memóriatöredékek töltik ki, melyek túl kis méretűek ahhoz, hogy használhatók legyenek.

A memória-feltöredeződés problémájának kezelésére számos módszert dolgoztak ki. A legegyszerűbb megoldás, hogy csak nagyobb memóriadarabok lefoglalását tesszük lehetővé és minden ilyen darabban azonos méretű objektumokat tárolunk (§15.3, §19.4.2). Mivel a legtöbb helyfoglalást kis méretű objektumok igénylik (például a tárolók elemei), ez a módszer nagyon hatékony lehet. Az eljárást akár egy önálló memóriafoglaló is automatikusan megvalósíthatja. A memóriatöredeződést mindkét esetben tovább csökkenthetjük, ha az összes nagyobb memóriadarabot azonos méretűre (például a lapméretre) állítjuk, így ezek lefoglalása sem okoz töredeződést.

A szemétgyűjtő rendszerek két fő típusba tartoznak:

1. A *másoló szemétgyűjtők* az objektumokat áthelyezik a memóriában, ezzel egyesítik a feltöredezett memória darabjait.
2. A *konzervatív szemétgyűjtők* úgy próbálnak helyet foglalni az objektumoknak, hogy a memóriatöredeződés a lehető legkisebb legyen.

A C++ szemszögéből a konzervatív szemétgyűjtők a gyakoribbak, mert rendkívül nehéz (sőt a komoly programokban valószínűleg lehetetlen) az objektumokat úgy áthelyezni, hogy minden mutatót helyesen átállítsunk. A konzervatív szemétgyűjtők azt is lehetővé teszik, hogy a C++ programrészletek együttműködjenek akár a C nyelven készült program-

részletekkel is. A másoló szemétgyűjtőket többnyire csak olyan nyelvekben valósítják meg, melyek az objektumokat mindig közvetve – mutatókon vagy hivatkozásokon keresztül – érik el. (Ilyen nyelv például a Lisp vagy a Smalltalk.) Az olyan nagyobb programok esetében, melyeknél a memóriafoglaló és a lapkezelő rendszer közötti kapcsolattartás, illetve a másolás mennyisége jelentős, az újabb konzervatív szemétgyűjtők legalább olyan hatékonynak tűnnek, mint a másoló szemétgyűjtők. Kisebb programok esetében gyakran az a legjobb megoldás, ha egyáltalán nem használunk szemétgyűjtőt – főleg a C++ esetében, ahol az objektumok többségét természetszerűleg automatikusan kezelhetjük.

C.10. Névterek

Ebben a pontban olyan apróságokat vizsgálunk meg a névterekkel kapcsolatban, melyek csupán technikai részleteknek tűnnek, de a viták során, illetve a komoly programokban gyakran felszínre kerülnek.

C.10.1. Kényelem és biztonság

A *using deklarációk* egy új nevet vezetnek be a helyi hatókörbe, a *using utasítások* (direktívák) azonban nem így működnek, csak elérhetővé teszik a megadott hatókörben szereplő neveket:

```
namespace X {
  int i, j, k;
}

int k;

void f1()
{
  int i = 0;
  using namespace X;          // az X-beli nevek elérhetővé tétele
  i++;                        // helyi i
  j++;                        // X::j
  k++;                        // hiba: X::k vagy globális k ?
  ::k++;                      // a globális k
  X::k++;                     // az X-beli k
}
```

```
void f2()
{
   int i = 0;
   using X::i;          // hiba: i kétszer deklarált f2()-ben
   using X::j;
   using X::k;          // elfedi a globális k nevet

   i++;
   j++;                       // X::j
   k++;                       // X::k
}
```

A helyileg (önálló deklarációval vagy *using deklarációval*) bevezetett nevek elrejtik az ugyanilyen néven szereplő nem helyi neveket; a fordító pedig minden hibás túlterhelést a deklaráció helyén jelez.

Figyeljük meg az *f1()* függvényben a *k++* utasítás kétértelműségét. A globális nevek nem élveznek elsőbbséget azokkal a nevekkel szemben, melyek globális hatókörben elérhető névterekből származnak. Ez jelentős mértékben csökkenti a véletlen névkeveredések lehetőségét és biztosítja, hogy ne a globális névtér „beszennyezésével" jussunk előnyökhöz.

Amikor *using utasítások* segítségével olyan könyvtárakat teszünk elérhetővé, melyek *sok* nevet vezetnek be, komoly könnyítést jelent, hogy a felhasználatlan nevek ütközése nem okoz hibát.

A globális névtér ugyanolyan, mint bármelyik másik, átlagos névtér. A globális névtér egyetlen különlegessége, hogy egy rá vonatkozó explicit minősítés esetében nem kell kiírnunk a nevét. Tehát a *::k* jelölés azt jelenti, hogy „keresd meg a *k* nevet a globális névtérben vagy a globális névtérhez *using utasítással* csatolt névterekben", míg az *X::k* jelentése: „keresd meg a *k* nevet az *X* névtérben, vagy a benne *using utasítással* megemlített névterekben" (§8.2.8).

Reméljük, hogy a névtereket használó új programokban radikálisan csökkenni fog a globális nevek száma a hagyományos C és C++ programokhoz viszonyítva. A névterek szabályait kifejezetten úgy találták ki, hogy ne adjanak előnyöket a globális nevek „lusta" felhasználóinak azokkal szemben, akik vigyáznak rá, hogy a globális hatókört ne „szennyezzék össze".

C.10.2. Névterek egymásba ágyazása

A névterek egyik nyilvánvaló felhasználási területe az, hogy deklarációk és definíciók valamilyen értelemben zárt halmazát egyetlen egységbe fogjuk össze:

```
namespace X {
  // saját deklarációk
}
```

A deklarációk listája általában tartalmazni fog újabb névtereket is. Tehát gyakorlati okokból is szükség van a névterek egymásba ágyazására, azon egyszerű ok mellett, hogy ha valamilyen rendszerben nem kifejezetten képtelenség az egymásba ágyazás megvalósítása, akkor illik azt lehetővé tennünk:

```
void h();

namespace X {
  void g();
  // ...
  namespace Y {
        void f();
        void ff();
        // ...
  }
}
```

Az alábbi jelölések a szokásos, hatókörökre vonatkozó, illetve minősítési szabályokból következnek:

```
void X::Y::ff()
{
  f(); g(); h();
}

void X::g()
{
  f();              // hiba: nincs f() X-ben
  Y::f();           // rendben
}

void h()
{
  f();              // hiba: nincs globális f()
  Y::f();           // hiba: nincs globális Y
  X::f();           // hiba: nincs f() X-ben
  X::Y::f();        // rendben
}
```

C.10.3. Névterek és osztályok

A névterek elnevezett hatókörök, az osztályok pedig típusok, melyeket egy elnevezett hatókörrel határozunk meg, ami azt írja le, hogy a típus objektumait hogyan kell létrehozni és kezelni. Tehát a névtér egyszerűbb fogalom, mint az osztály, és ideális esetben az osztályokat úgy is meghatározhatjuk, mint további lehetőségeket biztosító névtereket. A meghatározás azonban csak majdnem pontos, ugyanis a névtér nyitott szerkezet (§8.2.9.3), míg az osztály teljesen zárt. A különbség abból adódik, hogy az osztályoknak objektumok szerkezetét kell leírniuk, ez pedig nem engedi meg azt a szabadságot, amit a névterek nyújtanak. Ezenkívül a *using deklarációk* és *using utasítások* csak nagyon korlátozott esetekben használhatók az osztályokra (§15.2.2).

Ha csak a nevek egységbe zárására van szükségünk, a névterek jobban használhatók, mint az osztályok, mert ekkor nincs szükségünk az osztályok komoly típusellenőrzési lehetőségeire és az objektumkészítési módszerekre; az egyszerűbb névtér-kezelési elvek is elegendőek.

C.11. Hozzáférés-szabályozás

Ebben a pontban a hozzáférés-szabályozás olyan vonatkozásaira mutatunk példákat, melyekről a §15.3 pontban nem volt szó.

C.11.1. Tagok elérése

Vizsgáljuk meg az alábbi deklarációt:

```
class X {
// az alapértelmezés privát:
   int priv;
protected:
   int prot;
public:
   int publ;
   void m();
};
```

Az *X::m()* függvénynek korlátlan hozzáférése van a tagokhoz:

```
void X::m()
{
  priv = 1;          // rendben
  prot = 2;          // rendben
  publ = 3;          // rendben
}
```

A származtatott osztályok tagjai a *public* és *protected* tagokat érhetik el (§15.3):

```
class Y : public X {
  void mderived();
};

void Y::mderived()
{
  priv = 1;          // hiba: priv privát
  prot = 2;          // rendben: prot védett és mderived() az Y származtatott osztály tagja
  publ = 3;          // rendben: publ nyilvános
}
```

A globális függvények csak a nyilvános tagokat látják:

```
void f(Y* p)
{
  p->priv = 1;       // hiba: priv privát
  p->prot = 2;       // hiba: prot védett és f() se nem barát, se nem X vagy Y tagja
  p->publ = 3;       // rendben: publ nyilvános
}
```

C.11.2. Alaposztályok elérése

A tagokhoz hasonlóan az alaposztályokat is bevezethetjük *private*, *protected* vagy *public* minősítéssel:

```
class X {
public:
  int a;
  // ...
};

class Y1 : public X { };
class Y2 : protected X { };
class Y3 : private X { };
```

Mivel az *X* nyilvános alaposztálya az *Y1* osztálynak, bármely függvény szabadon (és automatikusan) átalakíthat egy *Y1** értéket *X** típusra és elérheti az *X* osztály nyilvános tagjait is:

```
void f(Y1* py1, Y2* py2, Y3* py3)
{
  X* px = py1;        // rendben: X nyilvános alaposztálya Y1 osztálynak
  py1->a = 7;         // rendben

  px = py2;           // hiba: X védett alaposztálya Y2 osztálynak
  py2->a = 7;         // hiba

  px = py3;           // hiba: X privát alaposztálya Y3 osztálynak
  py3->a = 7;         // hiba
}
```

Vegyük az alábbi deklarációkat:

```
class Y2 : protected X { };
class Z2 : public Y2 { void f(Y1*, Y2*, Y3*); };
```

Mivel *X* védett alaposztálya *Y2*-nek, csak *Y2* tagjai és barát (friend) osztályai, függvényei, illetve *Y2* leszármazottainak (például *Z2*-nek) tagjai és ezek barát elemei alakíthatják át (automatikusan) az *Y2** típusú értéket *X**-ra, és csak ezek férhetnek hozzá az *X* osztály *public* és *protected* tagjaihoz:

```
void Z2::f(Y1* py1, Y2* py2, Y3* py3)
{
  X* px = py1;        // rendben: X nyilvános alaposztálya Y1 osztálynak
  py1->a = 7;         // rendben

  px = py2;           // rendben: X védett alaposztálya Y2-nek,
                      // és Z2 osztály Y2-ből származik
  py2->a = 7;         // rendben

  px = py3;           // hiba: X privát alaposztálya Y3 osztálynak
  py3->a = 7;         // hiba
}
```

Végül vizsgáljuk meg az alábbit:

```
class Y3 : private X { void f(Y1*, Y2*, Y3*); };
```

Az *X* az *Y3*-nak privát alaposztálya, így kizárólag *Y3* tagjai és „barátai” alakíthatják át (automatikusan) az *Y3** értékeket *X**-ra, és csak azok érhetik el az *X* nyilvános és védett tagjait:

```
void Y3::f(Y1* py1, Y2* py2, Y3* py3)
{
  X* px = py1;        // rendben: X nyilvános alaposztálya Y1 osztálynak
  py1->a = 7;         // rendben

  px = py2;           // hiba: X védett alaposztálya Y2 osztálynak
  py2->a = 7;         // hiba

  px = py3;           // rendben: X privát alaposztálya Y3-nak, és Y3::f() Y3 tagja
  py3->a = 7;         // rendben
}
```

C.11.3. Tagosztályok elérése

A tagosztályok tagjainak nincs különleges jogosultságuk a befoglaló osztály tagjainak elérésére és ugyanígy a befoglaló osztály tagjai sem rendelkeznek különleges hozzáféréssel a beágyazott osztály tagjaihoz. Minden a szokásos szabályok (§10.2.2) szerint működik:

```
class Outer {
  typedef int T;
  int i;
public:
  int i2;
  static int s;

  class Inner {
        int x;
        T y;                  // hiba: Outer::T privát
  public:
        void f(Outer* p, int v);
  };

  int g(Inner* p);
};

void Outer::Inner::f(Outer* p, int v)
{
  p->i = v;                   // hiba: Outer::i privát
  p->i2 = v;                  // rendben: Outer::i2 nyilvános
}
```

```
int Outer::g(Inner* p)
{
  p->f(this,2);          // rendben: Inner::f() nyilvános
  return p->x;           // hiba: Inner::x privát
}
```

Ennek ellenére gyakran hasznos, ha a beágyazott osztály elérheti a beágyazó osztály tagja-it. Ezt úgy érhetjük el, ha a tagosztályt *friend* minősítéssel vezetjük be:

```
class Outer {
  typedef int T;
  int i;
public:
  class Inner;                  // a tagosztály előzetes bevezetése
  friend class Inner;           // elérési jog Outer::Inner számára

  class Inner {
          int x;
          T y;                  // rendben: Inner egy "barát"
  public:
          void f(Outer* p, int v);
  };
};

void Outer::Inner::f(Outer* p, int v)
{
  p->i = v;                     // rendben: Inner egy "barát"
}
```

C.11.4. Barátok

A *friend* minősítés hatása nem öröklődik és nem is tranzitív:

```
class A {
  friend class B;
  int a;
};

class B {
  friend class C;
};
```

```
class C {
  void f(A* p)
  {
        p->a++;    // hiba: C nem barátja A-nak, bár A barátjának barátja
  }
};

class D : public B {
  void f(A* p)
  {
        p->a++;    // hiba: D nem barátja A-nak, bár A barátjából származik
  }
};
```

C.12. Adattagokra hivatkozó mutatók

Természetesen a tagokra hivatkozó mutatók (§15.5) fogalma az adattagokra, valamint a rögzített paraméterekkel és visszatérési értékkel rendelkező függvényekre vonatkozik:

```
struct C {
  const char* val;
  int i;
  void print(int x) { cout << val << x << '\n'; }
  int f1(int);
  void f2();
  C(const char* v) { val = v; }
};

typedef void (C::*PMFI)(int);      // C osztály egy int paraméterrel meghívható
                                   // tagfüggvényre hivatkozó mutató
typedef const char* C::*PM;        // C osztály egy char* típusú adattagjára
                                   // hivatkozó mutató

void f(C& z1, C& z2)
{
  C* p = &z2;
  PMFI pf = &C::print;
  PM pm = &C::val;

  z1.print(1);
  (z1.*pf)(2);
```

```
    z1.*pm = "nv1 ";
    p->*pm = "nv2 ";
    z2.print(3);
    (p->*pf)(4);

    pf = &C::f1;        // hiba: nem megfelelő visszatérési típus
    pf = &C::f2;        // hiba: nem megfelelő paramétertípus
    pm = &C::i;         // hiba: nem megfelelő típus
    pm = pf;            // hiba: nem megfelelő típus
}
```

A függvényekre hivatkozó mutatók típusát a rendszer ugyanúgy ellenőrzi, mint bármely más típusét.

C.13. Sablonok

A sablon osztályok azt határozzák meg, hogyan hozható létre egy osztály, ha a szükséges sablonparamétereket megadjuk. A sablon függvények ehhez hasonlóan azt rögzítik, hogyan készíthetünk el egy konkrét függvényt a megadott sablonparaméterekkel. Tehát a sablonok típusok és futtatható kódrészletek létrehozásához használhatók fel. Ez a nagy kifejezőerő azonban néhány bonyodalmat okoz. A legtöbb probléma abból adódik, hogy más környezet áll rendelkezésünkre a sablon meghatározásakor és felhasználásakor.

C.13.1. Statikus tagok

Az osztálysablonoknak is lehetnek *static* tagjaik. Ezekből a tagokból minden, a sablonból létrehozott osztály saját példánnyal rendelkezik. A statikus tagokat külön kell meghatároznunk, de egyedi célú változatokat is készíthetünk belőlük:

```
template<class T> class X {
  // ...
  static T def_val;
  static T* new_X(T a = def_val);
};

template<class T> T X<T>::def_val(0,0);
template<class T> T* X<T>::new_X(T a) { /* ... */ }
```

```
template<> int X<int>::def_val<int> = 0;
template<> int* X<int>::new_X<int>(int i) { /* ... */ }
```

Ha egy objektumot vagy függvényt a sablonból létrehozott összes osztály összes példányában közösen szeretnénk használni, bevezethetünk egy alaposztályt, amely nem sablon:

```
struct B {
  static B* nil;          // közös nullpointer minden B-ből származtatott osztálynak
};

template<class T> class X : public B {
  // ...
};

B* B::nil = 0;
```

C.13.2. Barátok és sablonok

Más osztályokhoz hasonlóan a sablon is tartalmazhat *friend* osztályokat és függvényeket. Vizsgáljuk meg például a §11.5 pontban szereplő *Matrix* és *Vector* osztályt. Általában mindkét osztályt sablonként valósítjuk meg:

```
template<class T> class Matrix;

template<class T> class Vector {
   T v[4];
public:
   friend Vector operator* <> (const Matrix<T>&, const Vector&);
   // ...
};

template<class T> class Matrix {
   Vector<T> v[4];
public:
   friend Vector<T> operator* <> (const Matrix&, const Vector<T>&);
   // ...
};
```

A barát függvény neve után szereplő <> jel nem hagyható el, mert ez jelzi, hogy a barát egy sablon függvény. A <> jel nélkül a rendszer nem sablon függvényt feltételezne. A fenti bevezetés után a szorzás operátort úgy határozhatjuk meg, hogy a *Vector* és a *Matrix* tagjaira közvetlenül hivatkozunk:

```
template<class T> Vector<T> operator* <> (const Matrix<T>&, const Vector<T>&)
{
    // ... m.v[i] és v.v[i] használata az elemek közvetlen eléréséhez
}
```

A barát nincs hatással arra a hatókörre, melyben a sablon osztályt meghatároztuk, sem arra, melyben a sablon osztályt felhasználjuk. Ehelyett a barát függvényeket és operátorokat paramétertípusaik alapján találja meg a rendszer (§11.2.4, §11.5.1). A tagfüggvényekhez hasonlóan a barát függvényekből is csak akkor készül példány, ha meghívjuk azokat.

C.13.3. Sablonok, mint sablonparaméterek

Gyakran hasznos, ha sablonparaméterként osztályok vagy objektumok helyett sablonokat adunk át:

```
template<class T, template<class> class C> class Xrefd {
  C<T> mems;
  C<T*> refs;
  // ...
};

Xrefd<Entry,vector> x1;        // az Entry kereszthivatkozásokat vektorban tároljuk

Xrefd<Record,set> x2;          // a Record kereszthivatkozásokat halmazban tároljuk
```

Ahhoz, hogy egy sablont sablonparaméterként használjunk, meg kell adnunk az általa várt paramétereket, a paraméterként használt sablon sablonparamétereit viszont nem kell ismernünk. Sablonokat általában akkor használunk sablonparaméterként, ha különböző paramétertípusokkal akarjuk azokat példányosítani (fent például a *T* és a *T** típussal). Tehát a sablon tagjainak deklarációit egy másik sablon fogalmaival fejezzük ki, de ez utóbbi sablont paraméterként szeretnénk megadni, hogy a felhasználó választhassa meg típusát.

Ha a sablonnak egy tárolóra van szüksége azon elemek tárolásához, melyek típusát sablonparaméterként kapja meg, gyakran egyszerűbb a tárolótípust átadni (§13.6, §17.3.1).

Sablonparaméterek csak sablon osztályok lehetnek.

C.13.4. Sablon függvények paramétereinek levezetése

A fordító képes arra, hogy levezessen egy típus- (*T* vagy *TT*) vagy nem típus sablonparamétert (I) egy olyan függvénysablon-paraméterből, melyet a következő szerkezetek valamelyikével állítottunk elő:

```
T                               const T                       volatile T
T*                              T&                            T[konstans_kifejezés]
típus[I]                        osztálysablon_név<T>          osztálysablon_név<I>
TT<T>                           T<I>                          T<>
T típus::*                      T T::*                        típus T::*
T (*)(paraméterek)              típus (T::*)(paraméterek)     T (típus::*)(paraméterek)
típus (típus::*)(paraméterek_TI) T (T::*)(paraméterek_TI)     típus (T::*)(paraméterek_TI)
T (típus::*)(paraméterek_TI)    típus (*)(paraméterek_TI)
```

Ebben a gyűjteményben a *param_T1* egy olyan paraméterlista, melyből egy *T* vagy egy *I* meghatározható a fenti szabályok többszöri alkalmazásával, míg a *param* egy olyan paraméterlista, amely nem tesz lehetővé következtetést. Ha nem minden paraméter következtethető ki ezzel a megoldással, a hívás többértelműnek minősül:

```
template<class T, class U> void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
  f(p,g);       // T char, U int
  f(p,h);    // hiba: U nem vezethető le
}
```

Ha megnézzük az *f()* első hívásának paramétereit, könnyen kikövetkeztethetjük a sablonparamétereket. Az *f()* második meghívásában viszont láthatjuk, hogy a *h()* nem illeszthető az *U(*)(U)* mintára, mert paramétere és visszatérési értéke különböző.

Ha egy sablonparaméter egynél több függvényparaméterből is kikövetkeztethető, akkor mindegyiknek ugyanazt az eredményt kell adnia, ellenkező esetben hibaüzenetet kapunk:

```
template<class T> void f(T i, T* p);

void g(int i)
{
  f(i,&i);               // rendben
  f(i,"Vigyázat!");      // hiba, többértelmű: T int vagy T const char?
}
```

C.13.5. A typename és a template

Az általánosított (generikus) programozás további egyszerűsítése és még általánosabbá tétele érdekében a standard könyvtár tárolói szabványos függvényeket és típusokat biztosítanak (§16.3.1):

```
template<class T> class vector {
public:
   typedef T value_type;
   typedef T* iterator;

   iterator begin();
   iterator end();

   // ...
};

template<class T> class list {
   class link { /* ... */ };
public:
   typedef link* iterator;

   iterator begin();
   iterator end();

   // ...
};
```

Ez arra ösztönöz minket, hogy az alábbi formát használjuk:

```
template<class C> void f(C& v)
{
    C::iterator i = v.begin();  // formai hiba
}
```

Sajnos a fordító nem gondolatolvasó, így nem tudja, hogy a *C::iterator* egy típus neve. Bizonyos esetekben egy okosabb fordító kitalálhatja, hogy egy nevet típusnévnek szántunk-e vagy valami egészen másnak (például függvény- vagy sablonnévnek), de ez általában nem lehetséges. Figyeljük meg például az alábbit:

```
int y;

template<class T> void g(T& v)
{
   T::x(y);     // függvényhívás vagy változó-deklaráció?
}
```

Feltétlenül igaz-e, hogy ez esetben a *T::x* egy függvény, melyet az *y* paraméterrel hívtunk meg? Ugyanezt a sort tekinthetjük az *y* változó deklarációjának is, ahol a *T::x* egy típus és a zárójelek csak felesleges és természetellenes (de nem tiltott) jelek. Tehát el tudunk képzelni olyan környezetet, melyben *X::x(y)* egy függvényhívás, míg az *Y::x(y)* egy deklaráció.

A megoldás rendkívül egyszerű: ha mást nem mondunk, az azonosítókról a rendszer azt feltételezi, hogy olyasvalamire hivatkoznak, ami nem típus és nem sablon. Ha azt akarjuk kifejezni, hogy egy azonosítót típusként kell értelmezni, a *typename* kulcsszót kell használnunk:

```
template<class C> void h(C& v)
{
  typename C::iterator i = v.begin();
  // ...
}
```

A *typename* kulcsszót a minősített név elé írva azt fejezhetjük ki, hogy a megadott elem egy típus. Ebből a szempontból szerepe hasonlít a *struct* és a *class* szerepére.

A *typename* kulcsszóra mindig szükség van, ha a típus neve egy sablonparamétertől függ:

```
void k(vector<T>& v)
{
   vector<T>::iterator i = v.begin();            // formai hiba: a "typename" hiányzik
   typename vector<T>::iterator i = v.begin();   // rendben
   // ...
}
```

Ebben az esetben a fordító esetleg képes lenne megállapítani, hogy az *iterator* a *vector* sablon minden példányosztályában egy típusnevet ad meg, de a szabvány ezt nem követeli meg. Tehát, ha egy fordító ilyesmire képes, akkor az nem szabványos szolgáltatás és nem tekinthető hordozható nyelvkiterjesztésnek. Az egyetlen környezet, ahol a fordítónak feltételeznie kell, hogy egy sablon-paramétertől függő azonosító típusnév, az a néhány helyzet, amikor a nyelvtan csak típusneveket enged meg. Ilyenek például az *alap-meghatározások* (§A.8.1).

A sablondeklarációkban a *typename* a *class* kulcsszó szinonimájaként is használható:

```
template<typename T> void f(T);
```

Mivel a két változat között nincs érdemi különbség, de a képernyő mindig kicsinek bizonyul, én a rövidebb változatot ajánlom:

```
template<class T> void f(T);
```

C.13.6. Sablonok, mint minősítők

A *typename* kulcsszó bevezetésére azért volt szükség, mert hivatkozhatunk olyan adattagokra is, melyek típusok, és olyanokra is, melyek nem azok. Ugyanilyen problémát jelent a sablontagok nevének megkülönböztetése más tagokétól. Vizsgáljuk meg például egy általános memóriakezelő osztály egy lehetséges felületét:

```
class Memory {        // valamilyen memóriafoglaló
public:
   template<class T> T* get_new();
   template<class T> void release(T&);
   // ...
};

template<class Allocator> void f(Allocator& m)
{
   int* p1 = m.get_new<int>();              // formai hiba: 'int' a 'kisebb mint' operátor után
   int* p2 = m.template get_new<int>();  // explicit minősítés
   // ...
   m.release(p1);                          // sablonparaméter levezetése: nem kell explicit minősítő
   m.release(p2);
}
```

A *get_new()* függvény explicit minősítésére van szükség, mert a sablonparamétert nem lehet levezetni. Esetünkben a *template* előtagra van szükség ahhoz, hogy a fordítónak (és az emberi olvasónak) eláruljuk, hogy a *get_new()* egy sablontag, tehát a minősítés lehetséges a megadott elemtípussal. A *template* minősítés nélkül formai hibát kapunk, mert a < jelet a fordító „kisebb, mint” operátorként próbálja meg értelmezni. A *template* kulcsszóval való minősítésre ritkán van szükség, mert a legtöbb esetben a sablonparamétereket a fordító ki tudja következtetni.

C.13.7. Példányosítás

Ha adott egy sablon-meghatározás és ezt a sablont használni szeretnénk, a fordító feladata, hogy a megfelelő programkódot előállítsa. Egy sablon osztályból és a megadott sablonparaméterekből a fordítónak elő kell állítania egy osztály meghatározását és összes olyan tagfüggvényének definícióját, melyet programunkban felhasználtunk. Egy sablon függvény esetében a megfelelő függvényt kell előállítani. Ezt a folyamatot nevezzük a *sablon példányosításának*.

A létrehozott osztályokat és függvényeket *egyedi célú („szakosított") változatoknak* (specializációknak) nevezzük. Ha meg akarjuk különböztetni a fordító által automatikusan létrehozott változatokat a programozó által megadottaktól (§13.5), akkor *fordítói* (generált) *specializációkról*, illetve *explicit specializációkról* beszélünk. Az explicit specializációkat néha *felhasználói specializációknak* nevezzük.

Ahhoz, hogy a sablonokat bonyolultabb programokban is hatékonyan használhassuk, meg kell értenünk, hogy a sablon-meghatározásokban szereplő neveket hogyan köti a fordító konkrét deklarációkhoz és hogyan tudjuk forrásprogramunkat rendszerezni (§13.7).

Alapértelmezés szerint a fordító olyan sablonokból hoz létre osztályokat és függvényeket, melyeket a szokásos névkötési szabályok szerint használtunk (§C.13.8). Ez azt jelenti, hogy a programozónak nem kell megmondania, mely sablonok mely változatait kell elkészíteni. Ez azért fontos, mert a programozó általában nagyon nehezen tudja megmondani, hogy a sablonoknak pontosan mely változataira is van szüksége. A könyvtárak gyakran olyan sablonokat használnak, melyekről a programozó még nem is hallott, sőt az sem ritka, hogy egy, a programozó által egyáltalán nem ismert sablont annak számára szintén teljesen ismeretlen sablonparaméterekkel példányosítunk. Általában ahhoz, hogy megkeressük az összes olyan függvényt, melynek kódját a fordítónak elő kell állítania, az alkalmazásban és a könyvtárakban használt sablonok többszöri átvizsgálására van szükség. Az ilyen vizsgálatok sokkal jobban illenek a számítógéphez, mint a programozóhoz.

Ennek ellenére bizonyos helyzetekben fontos, hogy a programozó pontosan megmondhassa, a fordítónak hol kell elhelyeznie a sablonból létrehozott programrészleteket (§C.13.10). Ezzel a programozó pontosan szabályozhatja a példány környezetét. A legtöbb fordítási környezetben ez azt is jelenti, hogy pontosan rögzítjük, mikor jöjjön létre a példány. Az explicit példányosítás például jól használható arra, hogy a fordítási hibákat megjósolható helyen és időben kapjuk, ne pedig ott és akkor, amikor az adott fordító úgy dönt, hogy létre kell hoznia egy példányt. Bizonyos körülmények között a tökéletesen megjósolható programkód-létrehozás elengedhetetlenül fontos lehet.

C.13.8. Névkötés

Nagyon fontos, hogy a sablon függvényeket úgy határozzuk meg, hogy a lehető legkisebb mértékben függjenek nem helyi adatoktól. Ennek oka az, hogy a sablonból ismeretlen típusok alapján, ismeretlen helyen fog a fordító függvényt vagy osztályt létrehozni. Minden apró környezetfüggőség esélyes arra, hogy a programozó számára tesztelési problémaként jelentkezzen, pedig a programozó valószínűleg egyáltalán nem is akar tudni a sablon megvalósításának részleteiről. Az az általános szabály, miszerint amennyire csak lehet, el kell ke-

rülnünk a globális nevek használatát, a sablonok esetében még elengedhetetlenebb. Ezért a sablon-meghatározásokat próbáljuk annyira önállóvá tenni, amennyire csak lehet, és minden olyan elemet, amelyet egyébként esetleg globális eszközökkel valósítanánk meg, sablonparaméterként adjunk át (pl. *traits* a §13.4 és a §20.2.1 pontban).

Ennek ellenére kénytelenek vagyunk néhány nem helyi nevet is felhasználni. Sokkal gyakrabban készítünk például egymással együttműködő sablon függvényeket, mint egyetlen nagy, önálló eljárást. Néha ezek a függvények jól összefoghatók egy osztályba, de nem mindig. Időnként a nem helyi függvények használata mellett döntünk. Jellemző példa erre a *sort()* függvényben szereplő *swap()* és *less()* eljáráshívás (§13.5.2). A standard könyvtár algoritmusai nagyméretű példaként tekinthetők (§18. fejezet).

A hagyományos névvel és szereppel megvalósított függvények (például a +, a *, a *[]* vagy a *sort()*) egy más jellegű forrást jelentenek a sablon-meghatározásokban levő nem helyi nevek használatára. Vizsgáljuk meg például az alábbi programrészletet:

```
#include<vector>

bool tracing;

// ...

template<class T> T sum(std::vector<T>& v)
{
   T t = 0;
   if (tracing) cerr << "sum(" << &v << ")\n";
   for (int i = 0; i<v.size(); i++) t = t + v[i];
   return t;
}

// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
   Quad c = sum(v);
}
```

Az ártatlan kinézetű *sum()* sablon függvény a + operátortól függ. Példánkban a + műveletet a *<quad.h>* fejállomány határozza meg:

```
Quad operator+(Quad,Quad);
```

Fontos, hogy a *sum()* kifejtésekor semmilyen, a komplex számokhoz kapcsolódó elem nem elérhető és a *sum()* függvény megalkotója semmit sem feltételezhet a *Quad* osztályról. Így könnyen előfordulhat például, hogy a + operátort jóval később határozzuk meg, mint a *sum()* eljárást, mind a program szövegében, mind időben.

Azt a folyamatot, melynek során a fordító megkeresi a sablonban előforduló nevek deklarációját, ***névkötésnek*** (name binding) nevezzük. A sablonok névkötésénél a legnagyobb problémát az jelenti, hogy példányosításkor három különböző környezetet kell figyelembe venni és ezek nem választhatók el egymástól pontosan. A három környezet a következő:

1. A sablon meghatározásának környezete
2. A paramétertípus bevezetésének környezete
3. A sablon felhasználási helyének környezete

C.13.8.1. Függő nevek

Amikor egy függvény sablont meghatározunk, biztosak szeretnénk lenni abban, hogy rendelkezésünkre áll a megfelelő környezet, melyben az aktuális paraméterek fogalmaival el tudjuk végezni az adott feladatot, anélkül, hogy a felhasználási környezet elemeit „véletlenül" beépítenénk az eljárásba. A nyelv ennek megvalósítását azzal segíti, hogy a sablon definíciójában felhasznált neveket két kategóriába osztja:

1. A sablonparaméterektől függő nevek. Ezeket a neveket a példányosítás helyén köti le a fordító (§C.13.8.3). A *sum()* példafüggvényben a + operátor meghatározása a példányosítási környezetben található, mert operandusaiban felhasználja a sablon típusparaméterét.
2. A sablonparaméterektől független nevek. Ezeket a neveket a sablon meghatározásánál köti le a fordító (§C.13.8.2). A *sum()* példafüggvényben a *vector* sablont a <*vector*> szabványos fejállomány határozza meg, a logikai *tracing* változó pedig akkor is a hatókörben van, amikor a fordító találkozik a *sum()* függvény meghatározásával.

Az „*N* függ a *T* sablonparamétertől" kifejezés legegyszerűbb definíciója az lenne, hogy „*N* tagja a *T* osztálynak". Sajnos ez nem mindig elegendő: A *Quad* elemek összeadása (§C.13.8) jó ellenpélda erre. Tehát egy függvényhívást egy sablonparamétertől ***függőnek*** akkor nevezünk, ha az alábbi feltételek valamelyike teljesül:

1. Az aktuális paraméter típusa függ egy *T* sablonparamétertől a típuslevezetési szabályok szerint. Például *f(T(1))*, *f(t)*, *f(g(t))* vagy *f(&t)*, ha feltételezzük, hogy *t* típusa *T*.

2. A meghívott függvénynek van egy olyan formális paramétere, amely függ a *T* típustól, a típuslevezetési szabályok szerint (§13.3.1). Például *f(T)*, *f(list<T>&)* vagy *f(const T*)*.

Alapjában véve azt mondhatjuk, hogy a meghívott függvény akkor függ egy sablonparamétertől, ha a konkrét és formális paramétereire nézve nyilvánvalóan függ tőle.

Egy olyan hívás, melyben a függvény egyik paraméterének típusa véletlenül éppen megfelel az aktuális sablonparaméternek, nem jelent függőséget:

```
template<class T> T f(T a)
{
  return g(1);          // hiba: nincs g() a hatókörben és g(1) nem függ T-től
}

void g(int);

int z = f(2);
```

Az nem számít, hogy az *f(2)* hívásban a *T* éppen az *int* típust jelöli és a *g()* paramétere is ilyen típusú. Ha a *g(1)* hívást függőnek tekintenénk, a sablon-meghatározás olvasójának a függvény jelentése teljesen érthetetlen lenne. Ha a programozó azt akarja, hogy a *g(int)* függvény fusson le, akkor a *g(int)*-et be kell vezetnie az *f()* kifejtése előtt, hogy a *g(int)* függvény elérhető legyen az *f()* feldolgozásakor. Ez teljesen ugyanaz a szabály, mint a nem sablon függvények esetében.

A függvényneveken kívül a változók, típusok, konstansok stb. neve is függő, ha típusuk függ a sablonparamétertől:

```
template<class T> void fct(const T& a)
{
  typename T::Memtype p = a.p;      // p és Memtype függ T-től
  cout << a.i << ' ' << p->j;       // i és j függ T-től
}
```

C.13.8.2. Kötés meghatározáskor

Amikor a fordító egy sablon meghatározásával találkozik, megállapítja, mely nevek függők (§C.13.8.1). Ha egy név függő, deklarációjának megkeresését el kell halasztanunk a példányosításig (§C.13.8.3).

Azoknak a neveknek, melyek nem függnek sablonparaméterektől, a sablon meghatározásakor (definíciójánál) elérhetőnek kell lenniük (§4.9.4):

```
int x;

template<class T> T f(T a)
{
  x++;               // rendben
  y++;               // hiba: nincs y a hatókörben és y nem függ T-től
  return a;
}

int y;

int z = f(2);
```

Ha a fordító talál megfelelő deklarációt, mindenképpen azt fogja használni, függetlenül attól, hogy később esetleg „jobb” deklarációt is találhatna hozzá:

```
void g(double);

template<class T> class X : public T {
public:
  void f() { g(2); }        // g(double) meghívása
  // ...
};

void g(int);

class Z { };

void h(X<Z> x)
{
  x.f();
}
```

Amikor a fordító elkészíti az *X<Z>::f()* függvényt, nem a *g(int)* függvényt fogja használni, mert azt az *X* után vezettük be. Az nem számít, hogy az *X* sablont nem használjuk a *g(int)* bevezetése előtt. Ehhez hasonlóan egy független hívást sem téríthet el egy alaposztály:

```
class Y { public: void g(int); };

void h(X<Y> x)
{
  x.f();
}
```

Az *X<Y>::f()* most is a *g(double)* függvényt fogja meghívni. Ha a programozó az alaposztály *g()* függvényét akarja használni, az *f()* meghatározását a következőképpen kell megfogalmaznia:

```
template<class T> class XX : public T {
  void f() { T::g(2); }          // T::g() meghívása
  // ...
};
```

Ez természetesen annak az alapszabálynak a megjelenése, hogy a sablon definíciójának a lehető legönállóbbnak kell lennie (§C.13.8).

C.13.8.3. Kötés példányosításkor

A sablon minden különböző sablonparaméter-halmazzal való felhasználásakor új példányosítási pontot határozunk meg. A példányosítási pont helye a legközelebbi globális vagy névtér-hatókörben van, közvetlenül a felhasználást tartalmazó deklaráció előtt:

```
template<class T> void f(T a) { g(a); }

void g(int);

void h()
{
  extern g(double);
  f(2);
}
```

Esetünkben az *f<int>()* megfelelő példánya közvetlenül a *h()* előtt jön létre, tehát az *f()* függvényben meghívott *g()* a globális *g(int)* lesz, nem pedig a helyi *g(double)*. A „példányosítási pont” meghatározásából következik, hogy a sablonparamétereket nem köthetjük helyi nevekhez vagy osztálytagokhoz:

```
void f()
{
  struct X { /* ... */ };        // helyi szerkezet
  vector<X> v;                   // hiba: helyi szerkezet nem használható sablonparaméterként
  // ...
}
```

Ugyanígy a sablonban használt minősítetlen neveket sem köthetjük helyi névhez. A minősítetlen nevek akkor sem fognak egy osztály tagjaihoz kötődni, ha a sablont először ebben az osztályban használjuk. A helyi nevek elkerülése nagyon fontos, ha nem akarunk számtalan, „makrószerű" problémával szembekerülni:

```
template<class T> void sort(vector<T>& v)
{
  sort(v.begin(),v.end());      // standard könyvtárbeli sort() használata
}

class Container {
  vector<int> v;                // elemek
  // ...
public:
  void sort()                   // elemek rendezése
  {
        sort(v);                // a sort(vector<int>&) meghívása a Container::sort() helyett
  }
  // ...
};
```

Ha a *sort(vector<T>&)* a *sort()* függvényt az *std::sort()* jelöléssel hívta volna meg, az eredmény ugyanez lett volna, de program sokkal olvashatóbb lenne.

Ha egy névtérben meghatározott sablon példányosítási pontja egy másik névtérben található, a névkötésnél mindkét névtér nevei rendelkezésre állnak. A fordító szokás szerint a túlterhelési szabályokat használja arra, hogy megállapítsa, melyik névtér nevét kell használnia (§8.2.9.2).

Figyeljünk rá, hogy ha egy sablont többször használunk ugyanazokkal a sablonparaméterekkel, mindig új példányosítási pontot határozunk meg. Ha a független neveket a különböző helyeken különböző nevekhez kötjük, programunk helytelen lesz. Az ilyen hibát nagyon nehéz észrevenni egy alkalmazásban, főleg ha a példányosítási pontok különböző fordítási egységekben vannak. A legjobb, ha a névkötéssel járó problémákat kikerüljük azzal, hogy a lehető legkevesebb nem helyi nevet használjuk a sablonban és fejállományok segítségével biztosítjuk, hogy mindenhol a megfelelő környezet álljon a sablon rendelkezésére.

C.13.8.4. Sablonok és névterek

Amikor egy függvényt meghívunk, annak deklarációját a fordító akkor is megtalálhatja, ha az nincs is az aktuális hatókörben. Ehhez az kell, hogy a függvényt ugyanabban a névtérben vezessük be, mint valamelyik paraméterét (§8.2.6). Ez nagyon fontos a sablon-megha-

tározásokban meghívott függvények szempontjából, mert ez a szolgáltatás teszi lehetővé, hogy a függő függvényeket a fordító a példányosítás közben megtalálja.

A sablon egyedi célú változata a példányosítás tetszőleges pontján létrejöhet (§C.13.8.3), de a példányosítást követően az adott fordítási egységben, esetleg egy olyan fordítási egységben is, mely kifejezetten az egyedi célú változatok létrehozásához készült. Ebből három nyilvánvaló módszer alakítható ki az egyedi célú változatok elkészítéséhez:

1. Akkor hozzuk létre azokat, amikor első meghívásukkal találkozunk.
2. A fordítási egység végén létrehozzuk az összeset, amelyre az adott fordítási egységben szükség van.
3. Miután a program összes fordítási egységét megvizsgáltuk, a programban használt összes egyedi célú változatot egyszerre készítjük el.

Mindhárom módszernek vannak előnyei és hátrányai, ezért gyakran ezen lehetőségek valamilyen párosítását használják.

A független nevek kötését mindenképpen a sablon meghatározásakor végzi el a fordító. A függő nevek kötéséhez két dolgot kell megvizsgálni:

1. A sablon meghatározásakor a hatókörben lévő neveket
2. A függő hívások paramétereinek névterében szereplő neveket (a globális függvényeket úgy tekintjük, hogy a beépített típusok névterében szerepelnek)

Például:

```
namespace N {
  class A { /* ... */ };

  char f(A);
}

char f(int);

template<class T> char g(T t) { return f(t); }

char c = g(N::A());                // N::f(N::A) meghívását okozza
```

Itt az *f(t)* egyértelműen függő, így a definíció feldolgozásakor nem köthetjük sem az *f(int)*, sem az *f(N::A)* függvényhez. A *g<N::A>(N::A)* „szakosításakor” a fordító az *N* névtérben keresi a meghívott *f()* függvény meghatározását és ott az *N::f(N::A)* változatot találja meg.

A program hibás, ha különböző eredményeket kaphatunk azzal, hogy különböző példányosítási pontokat választunk, vagy azzal, ha az egyedi célú változat létrehozásakor használt környezetekben a névterek tartalmát megváltoztatjuk:

```
namespace N {
  class A { /* ... */ };

  char f(A,int);
}

template<class T, class T2> char g(T t, T2 t2) { return f(t,t2); }

char c = g(N::A(),'a');          // hiba: f(t) más feloldása is lehetséges

namespace N {                    // az N névtér kibővítése (§8.2.9.3)
  void f(A,char);
}
```

Ha a példányosítás pillanatában hozzuk létre a szakosított változatot, az *f(N::A,int)* függvény kerül meghívásra, viszont ha elkészítését a fordítási egység végére halasztjuk, a fordító az *f(N::A,char)* függvényt találja meg előbb. Tehát a *g(N::A(),'a')* hívás helytelen.

Nagyon rendetlen programozási stílust mutat, ha egy túlterhelt függvényt két deklarációja között hívunk meg. Egy nagyobb program esetében a programozó nem gyanakodna hibára, ebben az esetben azonban a fordító képes észrevenni a többértelműséget. Hasonló problémák jelentkezhetnek különböző fordítási egységekben is, így a hibák észlelése már sokkal nehezebbé válik. Az egyes C++-változatoknak nem kötelességük az ilyen típusú hibák figyelése.

A függvényhívások többféle feloldási lehetőségének problémája leggyakrabban akkor jelentkezik, amikor beépített típusokat használunk. Tehát a legtöbb hibát kiszűrhetjük azzal, hogy a beépített típusokat nagy körültekintéssel használjuk a paraméterekben.

Szokás szerint a globális függvények csak még bonyolultabbá teszik a dolgokat. A globális névteret a beépített típusok szintjének tekinthetjük, így a globális függvények felhasználhatók olyan függő függvények lekötésére is, melyeket beépített típusokkal hívtunk meg:

```
int f(int);

template<class T> T g(T t) { return f(t); }

char c = g('a');       // hiba: f(t) más feloldása is lehetséges

char f(char);
```

A *g<char>(char)* függvény egyedi célú változatát elkészíthetjük a példányosítási ponton is; ekkor az *f(int)* függvényt fogjuk használni. Ha a változatot a fordítási egység végén állítjuk elő, az *f(char)* függvény fut majd le. Tehát a *g('a')* hívás hibás.

C.13.9. Mikor van szükség egyedi célú változatokra?

Egy sablon osztály egyedi célú változatát csak akkor kell előállítani, ha az adott osztályra tényleg szükség van. Tehát amikor egy, az osztályra hivatkozó mutatót adunk meg, az osztály tényleges meghatározására még nincs szükségünk:

```
class X;
X* p;       // rendben: X meghatározására nincs szükség
X a;        // hiba: X meghatározására szükség van
```

A sablon osztályok meghatározásakor ez a különbség fontos lehet. A sablon osztályt mindaddig nem példányosítjuk, amíg arra nincs szükség:

```
template<class T> class Link {
  Link* suc;          // rendben: Link meghatározására (még) nincs szükség
  // ...
};

Link<int>* pl;        // Link<int> példányosítására nincs szükség

Link<int> lnk;        // most van szükség Link<int> példányosítására
```

A példányosítási pont az a hely, ahol a definícióra először szükség van.

C.13.9.1. Sablon függvények példányosítása

A fordító a sablon függvényeket csak akkor példányosítja, amikor a függvényt felhasználjuk. Ennek megfelelően egy sablon osztály példányosítása nem vonja maga után sem összes tagjának példányosítását, sem a sablon osztály deklarációjában szereplő tagok példányosítását. Ez nagy rugalmasságot biztosít a sablon osztályok meghatározásakor:

```
template<class T> class List {
  // ...
  void sort();
};
```

```
class Glob { /* nincs összehasonlító művelet */ };

void f(List<Glob>& lb, List<string>& ls)
{
  ls.sort();
  // lb műveleteinek használata, kivéve az lb.sort()-ot
}
```

Itt a *List<string>::sort()* függvényt a fordító példányosítja, de a *List<Glob>::sort()* függvényre nincs szükség. Ez egyrészt kevesebb kód létrehozását teszi lehetővé, másrészt megkímél minket attól, hogy az egész programot át kelljen szerveznünk. Ha a *List<Glob>::sort()* függvényt is létrehozná a fordító, akkor a *Glob* osztályban szerepelnie kellene az összes olyan műveletnek, melyet a *List::sort()* felhasznál, a *sort()* függvényt ki kellene vennünk a *List* sablonból, vagy a *Glob* objektumokat kellene valamilyen más tárolóban tárolnunk.

C.13.10. Explicit példányosítás

A példányosítást úgy kényszeríthetjük ki, ha a *template* kulcsszó után (melyet ez esetben nem követ < jel) deklaráljuk a kívánt egyedi célú változatot:

```
template class vector<int>;                    // osztály
template int& vector<int>::operator[](int);    // tag
template int convert<int,double>(double);      // függvény
```

Tehát egy sablon deklarációja a *template<* kifejezéssel kezdődik, míg egy példányosítási kérelmet a *template* kulcsszóval fejezünk ki. Figyeljük meg, hogy a *template* szó után a teljes deklaráció szerepel, nem elég csak a példányosítani kívánt nevet leírnunk:

```
template vector<int>::operator[];     // formai hiba
template convert<int,double>;         // formai hiba
```

A függvényparaméterekből levezethető sablonparaméterek ugyanúgy elhagyhatók, mint a sablon függvények meghívásakor (§13.3.1):

```
template int convert<int,double>(double);     // rendben (felesleges)
template int convert<int>(double);            // rendben
```

Amikor egy sablon osztályt így példányosítunk, a tagfüggvényekből is létrejön egy példány.

Az explicit példányosítás különböző ellenőrzések elvégzésére is felhasználható (§13.6.2):

```
template<class T> class Calls_foo {
  void constraints(T t) { foo(t); }        // minden konstruktorból meghívandó
  // ...
};

template class Calls_foo<int>;                // hiba: foo(int) nem meghatározott
template Calls_foo<Shape*>::constraints();    // hiba: foo(Shape*) nem meghatározott
```

A példányosítási kérelmek hatása az összeszerkesztési időre és az újrafordítás hatékonyságára nézve jelentős lehet. Arra is láttam már példát, hogy a sablon-példányosítások egyetlen fordítási egységbe való összefogásával a fordítási időt néhány óráról néhány percre sikerült csökkenteni.

Hibát jelent, ha ugyanarra az egyedi célú változatra két meghatározás is van. Nem számít, hogy ezek felhasználó által megadottak (§13.5), automatikusan létrehozottak (§C.13.7), vagy példányosítási kérelemmel készültek. A fordító nem köteles észrevenni a többszörös példányosítást, ha az egyes változatok különböző fordítási egységekben fordulnak elő. Ez lehetővé teszi a felesleges példányosítások elegáns elkerülését, így megszabadulhatunk azoktól a problémáktól, melyek a több könyvtárt használó programokban az explicit példányosításból származhatnak (§C.13.7). A szabvány azonban nem követeli meg, hogy a megvalósítások „elegánsak” legyenek. A „kevésbé elegáns” megvalósítások használóinak érdemes elkerülniük a többszörös példányosítást. Ha ezt a tanácsot mégsem fogadjuk meg, programunk – a legrosszabb esetben – nem fog futni, de a programban nem következik be rejtett változás.

A nyelv nem követeli meg, hogy a felhasználók explicit példányosítást használjanak. Ez csupán olyan optimalizációs lehetőség, mellyel saját kezűleg szabályozhatjuk a fordítás és összeszerkesztés menetét (§C.13.7).

C.14. Tanácsok

[1] A technikai részletek helyett összpontosítsunk a programfejlesztés egészére. §C.1.
[2] A szabvány szigorú betartása sem garantálja a hordozhatóságot. §C.2.
[3] Kerüljük a nem meghatározott helyzeteket (a nyelvi bővítésekben is). §C.2.

[4] Legyünk tisztában a megvalósítás-függő szolgáltatásokkal. §C.32.
[5] A {, }, [,], | és ! jelek helyett csak akkor használjunk kulcsszavakat és digráf jeleket, ha ezek nem elérhetőek az adott rendszerben; trigráf karaktereket pedig csak akkor, ha a \ sem állítható elő gépünkön. §C.3.1.
[6] Az egyszerű adatközlés érdekében a programok leírására használjuk az ANSI karaktereket. §C.3.3.
[7] A karakterek számmal jelölése helyett lehetőleg használjuk vezérlőkarakter megfelelőiket. §C.3.2.
[8] Soha ne használjuk ki a *char* típus előjelességét vagy előjel nélküliségét. §C.3.4.
[9] Ha kétségeink vannak egy egész literál típusával kapcsolatban, használjunk utótagokat. §C.4.
[10] Kerüljük az értékvesztő automatikus átalakításokat. §C.6.
[11] Használjuk a *vector* osztályt a beépített tömbök helyett. §C.7.
[12] Kerüljük a *union* típus használatát. §C.8.2.
[13] A nem általunk készített szerkezetek leírására használjunk mezőket. §C.8.1.
[14] Figyeljünk a különböző memóriakezelési stílusok előnyeire és hátrányaira. §C.9.
[15] Ne „szennyezzük be" a globális névteret. §C.10.1.
[16] Ha nem típusra, hanem önálló hatókörre (modulra) van szükségünk, osztályok helyett használjunk névtereket. §C.10.3.
[17] Sablon osztályokban is megadhatunk *static* tagokat. §C.13.1.
[18] Ha egy sablonparaméter tagtípusaira van szükségünk, az egyértelműség érdekében használjuk a *typename* kulcsszót. §C.13.5.
[19] Ha sablonparaméterekkel való kifejezett minősítésre van szükségünk, használjuk a *template* kulcsszót a sablon osztály tagjainak egyértelmű megadásához. §C.13.6.
[20] A sablon meghatározását úgy fogalmazzuk meg, hogy a lehető legkevesebbet használjuk fel a példányosítási környezetből. §C.13.8.
[21] Ha egy sablon példányosítása túl sokáig tart, esetleg érdemes explicit példányosítást alkalmaznunk. §C.13.10.
[22] Ha a fordítás sorrendjének pontosan megjósolhatónak kell lennie, explicit példányosításra lehet szükség. §C.13.10.

D

Helyi sajátosságok

„Ha Rómában jársz, tégy úgy, mint a rómaiak.”

A kulturális eltérések kezelése • A *locale* osztály • Nevesített lokálok • Lokálok létrehozása • Lokálok másolása és összehasonlítása • A *global()* és *classic()* lokálok • Karakterláncok összehasonlítása • A *facet* osztály • Jellemzők elérése a lokálokban • Egyszerű felhasználói *facet*-ek • Szabványos *facet*-ek • Karakterláncok összehasonlítása • Numerikus I/O • Pénz I/O • Dátum és idő I/O • Alacsonyszintű időműveletek • Egy *Date* osztály • A karakterek osztályozása • Karakterkód-átalakítások • Üzenetkatalógusok • Tanácsok • Gyakorlatok

D.1. A kulturális eltérések kezelése

A *locale* (lokál) a helyi sajátosságokat jelképező objektum. Olyan kulturális jellemzőket tartalmaz, mint a karakterek tárolásának és a karakterláncok összehasonlításának módja, illetve a számok megjelenési formája a kimeneten. A helyi sajátosságok köre bővíthető: a programozó a lokálhoz további olyan jellemzőket (*facet*; „arculat”, szempont) adhat, amelyeket a standard könyvtár nem támogat közvetlenül. Ilyenek például az irányítószámok vagy a telefonszámok. A *locale* elsődleges szerepe a standard könyvtárban a kimeneti adatfolyamban (*ostream*) levő adatok megjelenítési formájának, illetve a bemeneti adatfolyamok (*istream*) által elfogadott információ formátumának szabályozása.

A §21.7 pontban már tárgyaltuk, hogyan módosíthatjuk az adatfolyamok formátumát; ez a függelék azt írja le, hogy építhető fel jellemzőkből (*facet*) egy *locale*, illetve azt magyarázza el, hogyan befolyásolja a lokál az adatfolyamot. Ismerteti a *facet*-ek definiálásának módját is, felsorolja a szabványos jellemzőket, amelyekkel az adatfolyamok tulajdonságai beállíthatók és módszereket mutat be mindezek létrehozására és használatára. Az adat- és időábrázolást segítő standard könyvtárbeli eszközöket a dátumok be- és kivitelének tárgyalásakor mutatjuk be.

A lokálok és jellemzők tárgyalását a következő részekre bontottam:

§D.1 A kulturális eltérések lokálokkal való ábrázolása mögött rejlő alapgondolatok bemutatása
§D.2 A *locale* osztály
§D.3 A *facet* osztály
§D.4 A szabványos jellemzők áttekintése és részletes ismertetése:
- §D.4.1 Karakterláncok összehasonlítása
- §D.4.2 Számértékek bevitele és kivitele
- §D.4.3 Pénzértékek bevitele és kivitele
- §D.4.4 Dátum és idő bevitele és kivitele
- §D.4.5 A karakterek osztályozása
- §D.4.6 Karakterkód-konverziók
- §D.4.7 Üzenetkatalógusok

A *locale* nem a C++ fogalma, a legtöbb operációs rendszerben és felhasználói környezetben létezik. A helyi sajátosságokon („területi beállításokon") – elvileg – az adott rendszerben megtalálható valamennyi program osztozik, függetlenül attól, hogy a programok milyen programnyelven íródtak. Ezért a C++ standard könyvtárának *locale*-jét úgy tekinthetjük, mint amelynek segítségével a programok szabványos és „hordozható" módon férhetnek hozzá olyan adatokhoz, melyek ábrázolása a különböző rendszerekben jelentősen eltér. Így a C++ *locale* közös felületet biztosít azon rendszeradatok eléréséhez, melyeket az egyes rendszerek más és más módon tárolnak.

D.1.1. A kulturális eltérések programozása

Képzeljük el, hogy olyan programot írunk, melyet sok országban fognak használni. Azt, hogy a programot olyan stílusban írjuk meg, ami ezt lehetővé teszi, gyakran hívják ***nemzetközi támogatásnak*** („internacionalizáció"; ez kihangsúlyozza, hogy a programot több országban használják) vagy ***honosításnak*** („lokalizációnak", kihangsúlyozva a program helyi

viszonyokhoz igazítását). A program által kezelt adatok némelyike – a szokásoknak megfelelően – különbözőképpen jelenik meg az egyes országokban. A problémát úgy kezelhetjük, hogy a bemeneti/kimeneti eljárások megírásánál ezt figyelembe vesszük:

```
void print_date(const Date& d) // kiírás a megfelelő formában
{
    switch(where_am_I) // felhasználói jelzőérték
    {
    case DK:    // pl. 7. marts 1999
        cout << d.day() << ". " << dk_month[d.month()] << " " << d.year();
        break;
    case UK:    // pl. 7 / 3 / 1999
        cout << d.day() << " / " << d.month() << " / " << d.year();
        break;
    case US:    // pl. 3/7/1999
        cout << d.month() << "/" << d.day() << "/" << d.year();
        break;
    // ...
    }
}
```

Egy ilyen kóddal a feladat megoldható, de a kód elég csúnya és csak következetes használatával biztosíthatjuk, hogy minden kimenet megfelelően igazodjon a helyi szokásokhoz. Ami még ennél is rosszabb: ha újabb dátumformátummal szeretnénk kiegészíteni, módosítanunk kell a kódot. Felmerülhet az ötlet, hogy a problémát egy osztályhierarchia létrehozásával oldjuk meg (§12.2.4). A *Date*-ben tárolt adatok azonban függetlenek a megjelenítés módjától, így nem *Date* típusok (például *US_date*, *UK_date*, és *JP_date*) rendszerét kell létrehoznunk, hanem a dátumok megjelenítésére kell megadnunk többféle módszert (mondjuk amerikai, brit és japán stílusú kimenetet). Lásd még: §D.4.4.5.

Az „engedjük meg a felhasználónak, hogy bemeneti/kimeneti függvényeket írjon és kezeljék azok a helyi sajátosságokat" megközelítésnek is vannak buktatói:

1. Egy rendszerfejlesztő programozó nem tudja könnyen, más rendszerre átültethető módon és hatékonyan módosítani a beépített típusok megjelenését a standard könyvtár segítsége nélkül.
2. Egy nagy programban nem mindig lehet az összes I/O műveletet (és minden olyan műveletet, amely a helyi sajátosságokhoz igazítva készít elő adatot I/O-hoz) megtalálni.
3. A programot néha nem igazíthatjuk az új szabályokhoz – és még ha lehetséges is lenne, jobban szeretnénk egy olyan megoldást, amely nem igényel újraírást.

4. Pazarlás lenne minden felhasználóval megterveztetni és elkészíttetni az eltérő helyi sajátosságokat kezelő programelemeket.
5. A különböző programozók különféleképpen kezelik a kulturális eltéréseket, ezért a valójában ugyanazokkal az adatokkal dolgozó programok is különbözni fognak, noha alapvetően azonosnak kellene lenniük. Így azoknak a programozóknak, akik több forrásfájlban lévő kódot „tartanak karban", többfajta programozási megközelítést kell megtanulniuk, ami fárasztó és hibalehetőséget rejt magában.

Következésképpen a standard könyvtár a helyi sajátosságok kezeléséhez bővíthető eljárásgyűjteményt kínál. Az *iostream* könyvtár (§21.7) mind a beépített, mind a felhasználói típusok kezeléséhez ezekre az eljárásokra támaszkodik. Vegyünk például egy egyszerű ciklust, amely mérések sorozatát vagy tranzakciók egy halmazát ábrázoló (*Date, double*) párokat másol:

```
void cpy(istream& is, ostream& os)      // (Date,double) adatfolyamot másol
{
    Date d;
    double volume;
    while (is >> d >> volume) os << d << ' ' << volume << '\n';
}
```

Egy valódi program természetesen csinálna valamit a rekordokkal és egy kicsit jobban törődne a hibakezeléssel is.

Hogyan készíthetünk ebből egy olyan programot, amely a francia szokásoknak megfelelő fájlt olvas be (ahol – a magyarhoz hasonlóan – vesszőt használnak a „tizedespont" jelölésére a lebegőpontos számokban; például a *12,5* jelentése tizenkettő és fél) és az amerikai formának megfelelően írja azt ki? Lokálok és I/O műveletek meghatározásával a *cpy()*-t használhatjuk az átalakításra:

```
void f(istream& fin, ostream& fout, istream& fin2, ostream& fout2)
{
    fin.imbue(locale("en_US"));         // amerikai angol
    fout.imbue(locale("fr"));           // francia
    cpy(fin,fout);                      // amerikai angol olvasás, francia írás
    fin2.imbue(locale("fr"));           // francia
    fout2.imbue(locale("en_US"));       // amerikai angol
    cpy(fin2,fout2);                    // francia olvasás, amerikai angol írás
}
```

Ha adottak a következő adatfolyamok,

Apr 12, 1999 1000.3
Apr 13, 1999 345.45
Apr 14, 1999 9688.321
...
3 juillet 1950 10,3
3 juillet 1951 134,45
3 juillet 1952 67,9
...

a program a következőt fogja kiírni:

12 avril 1999 1000,3
13 avril 1999 345,45
14 avril 1999 9688,321
...
July 3, 1950 10.3
July 3, 1951 134.45
July 3, 1952 67.9
...

A függelék további részének legjavát annak szenteljük, hogy leírjuk, milyen nyelvi tulajdonságok teszik ezt lehetővé, illetve hogy elmagyarázzuk, hogyan használjuk azokat. Jegyezzük meg, hogy a programozók többségének nem kell részletekbe menően foglalkoznia a lokálokkal vagy kifejezetten azokat kezelő kódot írnia. Akik mégis megteszik, azok is leginkább egy szabványos *locale*-t fognak előkeresni, hogy az adott adatfolyamnak előírják annak használatát (§21.7). Azok az eljárások azonban, melyekkel a lokálokat létrehozhatjuk és használatukat egyszerűvé tehetjük, „saját" programnyelvet alkotnak.

Ha egy program vagy rendszer sikeres, olyanok is használni fogják, akiknek az igénye és ízlése eltér attól, amire az eredeti tervezők és programozók számítottak. A legtöbb sikeres programot használni fogják azokban az országokban is, ahol a (természetes) nyelvek és karakterkészletek eltérnek az eredeti tervezők és programozók által ismertektől. Egy program széleskörű használata a siker jele, ezért a nyelvi és kulturális határok között átvihető programok tervezése és írása a sikerre való felkészülést jelenti.

A „nemzetközi támogatás" fogalma egyszerű, a gyakorlati megszorítások azonban a *locale* objektumok elkészítését meglehetősen bonyolulttá teszik:

1. A lokálok az olyan helyi sajátosságokat tartalmazzák, mint a dátumok megjelenési formája. A szabályok azonban egy adott kultúrán belül is számos apró és

nem rendszerezhető módon eltérhetnek. A helyi szokásoknak semmi közük a programnyelvekhez, ezért egy programnyelv nem szabványosíthatja azokat.
2. A lokáloknak bővíthetőnek kell lenniük, mert nem lehetséges az összes helyi sajátosságot felsorolni, amely minden C++ felhasználónak fontos.
3. A *locale* objektumokat olyan I/O műveletekben használjuk, melyeknél a futási idő igen fontos.
4. A *locale* objektumoknak láthatatlannak kell lenniük a legtöbb programozó számára, hiszen ők anélkül szeretnék kihasználni a „megfelelő dolgot végző" adatfolyam-bemenetet és -kimenetet, hogy pontosan ismernék annak felépítését, megvalósítását.
5. A lokáloknak elérhetőnek kell lenniük azok számára, akik olyan eszközöket terveznek, amelyek helyi sajátosságoktól függő adatokat kezelnek az adatfolyam I/O könyvtár keretein kívül.

Egy bemeneti és kimeneti műveleteket végző program tervezésekor választanunk kell, hogy a kimenet formátumát „szokásos kóddal" vagy *locale*-ek felhasználásával vezéreljük-e. Az előbbi (hagyományos) megközelítés ott célszerű, ahol biztosítani tudjuk, hogy minden bemeneti műveletet könnyen át lehet alakítani a helyi sajátosságoknak megfelelően. Ha azonban a beépített típusok megjelenésének kell változnia, ha különböző karakterkészletekre van szükség, vagy ha a bemenetre/kimenetre vonatkozó szabályok halmazai között kell választanunk, a *locale* objektumok használata tűnik ésszerűbbnek.

A *locale* objektumok úgynevezett *facet*-ekből állnak, amelyek az egyes jellemzőket (lebegőpontos érték kiírásakor használt elválasztó karakter (*decimal_point()*, §D.4.2), pénzérték beolvasásakor használt formátum (*moneypunct*, §D.4.3) stb.) szabályozzák. A *facet* egy, a *locale::facet* osztályból származó osztály objektuma (§D.3); leginkább úgy képzelhetjük el, hogy a *locale* *facet*-ek tárolója (§D.2, §D.3.1).

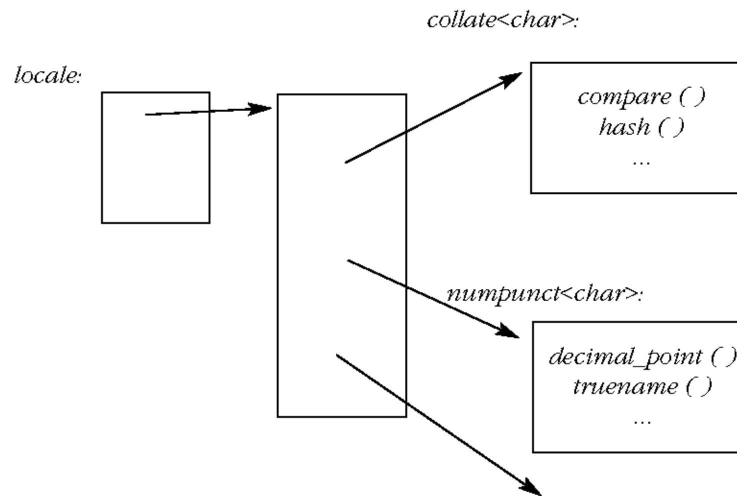
D.2. A locale osztály

A *locale* osztály és a hozzá tartozó szolgáltatások a <*locale*> fejállományban találhatók:

```
class std::locale {
public:
    class facet;              // lokáljellemzőket tartalmazó típus, §D.3
    class id;                 // lokált azonosító típus, §D.3
    typedef int category;     // facet-ek csoportosítására szolgáló típus
```

```
    static const category          // a tényleges értékek megvalósításfüggőek
        none = 0,
        collate = 1,
        ctype = 1<<1,
        monetary = 1<<2,
        numeric = 1<<3,
        time = 1<<4,
        messages = 1<<5,
        all = collate | ctype | monetary | numeric | time | messages;

    locale() throw();                       // a globális lokál másolata (§D.2.1)
    locale(const locale& x) throw();        // x másolata
    explicit locale(const char* p);         // a p nevű lokál másolata (§D.2.1)

    ~locale() throw();

    locale(const locale& x, const char* p, category c);     // x másolata, plusz p-beli c facet-ek
    locale(const locale& x, const locale& y, category c);  // x másolata, plusz y-beli c facet-ek

    template <class Facet> locale(const locale& x, Facet* f);  // x másolata, plusz az f facet
    template <class Facet> locale combine(const locale& x); // *this másolata,
                                                             // plusz az x-beli Facet

    const locale& operator=(const locale& x) throw();

    bool operator==(const locale&) const;           // lokálok összehasonlítása
    bool operator!=(const locale&) const;

    string name() const;                    // az adott lokál neve (§D.2.1)

    template <class Ch, class Tr, class A>  // karakterláncok összehasonlítása
                                            // az adott lokál segítségével
    bool operator()(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&) const;

    static locale global(const locale&);    // a globális lokál beállítása és visszatérés a régivel
    static const locale& classic();         // a "klasszikus" C-stílusú lokál
private:
    // ábrázolás
};
```

A *locale*-ekre úgy gondolhatunk, mint *map<id,facet*>*-ek felületére, vagyis valami olyasmire, ami lehetővé teszi, hogy egy *locale::id* segítségével megtaláljuk a megfelelő objektumot, amelynek osztálya a *locale::facet*-ből származik. A *locale* megvalósítása alatt az e gondolat alapján elkészített (hatékony) szerkezeteket értjük. Az elrendezés ilyesmi lesz:



Itt a *collate<char>* és a *numpunct<char>* a standard könyvtár *facet*-jei (§D.4). Mint mindegyik *facet*, ezek is a *locale::facet*-ből származnak.

A *locale*-nek szabadon és „olcsón" másolhatónak kell lennie. Következésképpen a *locale*-t majdnem mindig úgy valósítják meg, mint egy leírót arra a „szakosított" *map<id,facet*>*-re, amely a szolgáltatások legtöbbjét tartalmazza. A lokál jellemzőinek (a *facet*-eknek) gyorsan elérhetőnek kell lenniük, ezért az egyedi célú *map<id,facet*>* a tömbökhöz hasonló gyors hozzáférést kell, hogy nyújtson. A *locale* jellemzőinek elérése a *use_facet<Facet>(loc)* jelöléssel történik (lásd §D.3.1-et).

A standard könyvtár a *facet*-ek gazdag választékát nyújtja. A programozó ezeket logikai csoportokban kezelheti, mert a szabványos *facet*-ek kategóriákat alkotnak (pl. *numeric* és *collate*, §D.4).

A programozó az egyes kategóriákban levő jellemzőket kicserélheti (§D.4, §D.4.2.1), de nem adhat meg új kategóriákat. A „kategória" fogalma csak a standard könyvtárbeli *facet*-ekre vonatkozik, nem terjeszthető ki a felhasználói jellemzőkre. Ezért nem szükséges, hogy egy *facet* kategóriába tartozzon és sok felhasználói *facet* nem is tartozik ilyenbe.

A lokálokat messze a leggyakrabban az adatfolyamok bemeneti és kimeneti műveletinél használjuk, még ha nem is tudunk róla. Minden *istream* és *ostream* rendelkezik saját lokállal. Az adatfolyamok lokálja a folyam létrehozásának pillanatában alapértelmezés szerint a globális *locale* (§D.2.1) lesz. A folyam lokálját az *imbue()* („megtöltés") művelettel lehet beállítani, a *locale* másolatát pedig a *getloc()* függvénnyel kaphatjuk meg (§21.6.3).

D.2.1. Nevesített lokálok

A lokálok másik *locale*-ből és jellemzőkből hozhatók létre. A legegyszerűbb egy már létező *locale* lemásolása:

```
locale loc0;                      // az érvényes globális lokál másolata (§D.2.3)

locale loc1 = locale();           // az érvényes globális lokál másolata (§D.2.3)
locale loc2("");                  // a felhasználó által előnyben részesített lokál

másolata

locale loc3("C");                 // a "C" lokál másolata
locale loc4 = locale::classic();  // a "C" lokál másolata

locale loc5("POSIX");             // az adott fejlesztőkörnyezet által meghatározott
                                  // "POSIX" lokál másolata
```

A *locale("C")* jelentését a szabvány „klasszikus C lokálként" határozza meg; ebben a könyvben végig ezt a lokált használjuk. A többi *locale* neve a használt C++-változattól függ.

A *locale("")* a „felhasználó által előnyben részesített" lokál, melyet a program végrehajtási környezetében a nyelven kívüli eszközök állítanak be.

A legtöbb operációs rendszer biztosít valamilyen eszközt a programok „területi beállításainak" megadására. A beállításra legtöbbször akkor kerül sor, amikor a felhasználó először találkozik a rendszerrel. Egy olyan gépen például, ahol a rendszer alapértelmezett nyelveként az argentin spanyolt adták meg, a *locale("")* valószínűleg a *locale("es_AR")*-t jelenti. Az egyik rendszerem gyors ellenőrzése 51 megjegyezhető névvel rendelkező lokált mutatott ki (például *POSIX*, *de*, *en_UK*, *en_US*, *es*, *es_AR*, *fr*, *sv*, *da*, *pl*, és *iso_8859_1*). A POSIX által ajánlott formátum: kisbetűs nyelvnév, amit nagybetűs országnév követ (ez nem kötelező), valamint egy kódjelző (ez sem kötelező); például *jp_JP.jit*. Ezek a nevek azonban nem szabványosítottak a különböző platformok között. Egy másik rendszerben egyéb *locale* nevek mellett a következőket találtam: *g*, *uk*, *us*, *s*, *fr*, *sw*, és *da*. A C++ szabvány nem adja meg az országok és nyelvek *locale*-jét, bár az egyes platformokra létezhetnek szabványok. Következésképpen, ha a programozó nevesített lokálokat akar használni, a rendszer dokumentációjára és tapasztalataira kell hagyatkoznia.

Általában célszerű elkerülni a lokálneveket jelző karakterláncok programszövegbe ágyazását. A fájlnevek és „rendszerállandók" programban való szerepeltetése korlátozza a program hordozhatóságát, a programot új környezetbe beilleszteni kívánó programozó pedig gyakran arra kényszerül, hogy megkeresse ezeket az értékeket, hogy módosíthassa azokat.

A lokálnevek megemlítése is hasonló kellemetlen következményekkel jár. Jobb, ha a lokálokat kivesszük a program végrehajtási környezetéből (például a *locale("")* felhasználásával) vagy a programra bízzuk, hogy a tapasztaltabb felhasználóktól a lokál meghatározását kérje, mondjuk egy karakterlánc bekérésével:

```
void user_set_locale(const string& question_string)
{
    cout << question_string;             // pl. "Ha más lokált kíván használni, adja meg a nevét"
    string s;
    cin >> s;
    locale::global(locale(s.c_str()));  // a felhasználó által megadott globális lokál beállítása
}
```

A kezdő felhasználók számára általában jobb, ha lehetővé tesszük, hogy listából választhassanak. Az ezt kezelő eljárásnak viszont tudnia kell, hol és hogyan tárolja a rendszer a lokálokat.

Ha a paraméterként megadott karakterlánc nem definiált *locale*-re hivatkozik, a konstruktor *runtime_error* kivételt vált ki (§14.10):

```
void set_loc(locale& loc, const char* name)
try
{
    loc = locale(name);
}
catch (runtime_error) {
    cerr << "A \"" << name << "\" lokál nem definiált.\n";
    // ...
}
```

Ha a *locale* nevesített, a *name()* visszaadja annak nevét, ha nem, a *string("*")*-gal tér vissza. A név elsősorban arra való, hogy hivatkozhassunk a végrehajtási környezetben tárolt lokálokra, de a hibakeresésben is segíthet:

```
void print_locale_names(const locale& my_loc)
{
    cout << "name of current global locale: " << locale().name() << "\n";
    cout << "name of classic C locale: " << locale::classic().name() << "\n";
    cout << "name of "user's preferred locale": " << locale("").name() << "\n";
    cout << "name of my locale: " << my_loc.name() << "\n";
}
```

Az alapértelmezett *string("*")*-tól eltérő, de azonos nevű lokálok összehasonlításkor egyenértékűnek minősülnek, az == vagy != operátorokkal azonban az összehasonlítás közvetlenebb módon is elvégezhető.

A névvel rendelkező *locale*-ek másolatai ugyanazt a nevet kapják, mint az eredeti *locale* (ha annak van neve), így azonos néven több *locale* is szerepelhet. Ez logikus, mert a lokálok nem módosíthatók, így ezen objektumok mindegyike a helyi sajátosságok ugyanazon halmazát írja le.

A *locale(loc,"Foo",cat)* hívás a *loc*-hoz hasonló lokált hoz létre, de annak jellemzőit (a *facet*-eket) a *locale("Foo") cat* kategóriájából veszi. Az eredményül kapott lokálnak kizárólag akkor lesz neve, ha a *loc*-nak is volt. A szabvány nem határozza meg pontosan, milyen nevet kap az új *locale*, de feltehető, hogy különbözni fog a *loc*-étól. A legegyszerűbb, ha a nevet a *loc* nevéből és a *"Foo"*-ból építjük fel. Például ha a *loc* neve *en_UK*, az új lokál neve *"en_UK:Foo"* lesz.

Az új lokálok elnevezésére vonatkozó szabályok a következőképpen foglalhatók össze:

Lokál	Név
locale("Foo")	*"Foo"*
locale(loc)	*loc.name()*
locale(loc,"Foo",cat)	Új név, ha a *loc*-nak van neve; egyébként *string("*")*
locale(loc,loc2,cat)	Új név, ha a *loc*-nak és a *loc2*-nek is van neve; egyébként *string("*")*
locale(loc,Facet)	*string("*")*
loc.combine(loc2)	*string("*")*

A programozó az újonnan létrehozott *locale*-ek neveként nem adhat meg C stílusú karakterláncot. A neveket a program végrehajtási környezete határozza meg vagy a *locale* konstruktorok építik fel azokat a nevek párosításából.

D.2.1.1. Új lokálok létrehozása

Új *locale* objektumot úgy készíthetünk, hogy veszünk egy már létező *locale*-t és ehhez jellemzőket adunk vagy kicserélünk benne néhányat. Az új *locale*-ek jellemzően egy már létező *locale* kissé eltérő változatai:

```
void f(const locale& loc, const My_money_io* mio)        // A §D.4.3.1-ben leírt

"My_money_io"
{
    locale loc1(locale("POSIX"), loc, locale::monetary);   // A loc-beli pénzformátum-
                                                            // jellemzők használata
    locale loc2 = locale(locale::classic(), mio);           // a klasszikus, plusz "mio"
    // ...
}
```

Itt *loc1* a *POSIX* lokál másolata, amit úgy módosítottunk, hogy a *loc* pénzformátum-jellemzőit használja (§D.4.3). Ehhez hasonlóan, *loc2* a *C* lokál másolata, amely a *My_money_io*-t használja (§D.4.3.1). Ha a *Facet** paraméter (itt a *My_money_io*) értéke *0*, az eredményül kapott lokál egyszerűen a *locale* paraméter másolata lesz.

Ha a következőt írjuk,

```
locale(const locale& x, Facet* f);
```

az *f* paraméternek egy meghatározott *facet* típust kell jelölnie. Egy egyszerű *facet** nem elegendő:

```
void g(const locale::facet* mio1, const My_money_io* mio2)
{
    locale loc3 = locale(locale::classic(), mio1);    // hiba: a facet típusa nem ismert
    locale loc4 = locale(locale::classic(), mio2);    // rendben: a facet típusa ismert
    // ...
}
```

Ennek az az oka, hogy a *locale* a *Facet** paraméter típusát használja arra, hogy fordítási időben megállapítsa a *facet* típusát. Pontosabban, a *locale* megvalósítása a jellemző azonosítóját, a *facet::id*-t (§D.3.3) használja, hogy a jellemzőt megtalálja a lokálban (§D.3.1).

Jegyezzük meg, hogy a

```
template <class Facet> locale(const locale& x, Facet* f);
```

konstruktor a nyelv által nyújtott egyetlen eljárás a programozó számára, hogy *facet*-eket adhasson meg, melyeket egy *locale*-en keresztül használni lehet. Az egyéb lokálokat a megvalósító programozóknak kell megadniuk, nevesített *locale*-ek formájában (§D.2.1), melyeket a program végrehajtási környezetéből lehet megszerezni. Ha a programozó érti az erre használatos – a fejlesztőkörnyezettől függő – eljárást, a meglevő lokálok körét újakkal bővítheti (§D.6[11,12]).

A lokálok konstruktorainak halmazát úgy tervezték, hogy minden jellemző típusa megállapítható legyen, vagy típuslevezetés útján (a *Facet* sablonparaméter típusából), vagy azért, mert egy másik lokálból származik, amely ismerte a jellemző típusát. A *category* paraméter megadásával a *facet*-ek típusát közvetett módon is meghatározhatjuk, hiszen a lokálok ismerik a kategóriákban lévő jellemzők típusát. Ebből következik, hogy a *locale* osztály nyomon követheti a *facet*-ek típusát, így azokat különösebb többletterhelés nélkül módosíthatja is.

A lokál a *facet* típusok azonosítására a *locale::id* tagtípust használja (§D.3).

Néha hasznos lehet olyan lokált létrehozni, amely egy másik lokál másolata, de az egyik jellemzője egy harmadik lokálból származik. A *combine()* sablon tagfüggvény erre való:

```
void f(const locale& loc, const locale& loc2)
{
    locale loc3 = loc.combine< My_money_io >(loc2);
    // ...
}
```

Az eredményül kapott *loc3* úgy viselkedik, mint a *loc*, de a pénzformátumot a *loc2*-ben levő *My_money_io* (§D.4.3.1) másolata alapján állítja be. Ha a *loc2* nem rendelkezik a *My_money_io*-val, hogy átadhassa azt az új lokálnak, a *combine()* *runtime_error*-t (§14.10) vált ki. A *combine()* eredményeként kapott lokál nem nevesített.

D.2.2. Lokálok másolása és összehasonlítása

A lokálok kezdeti vagy egyszerű értékadással másolhatók:

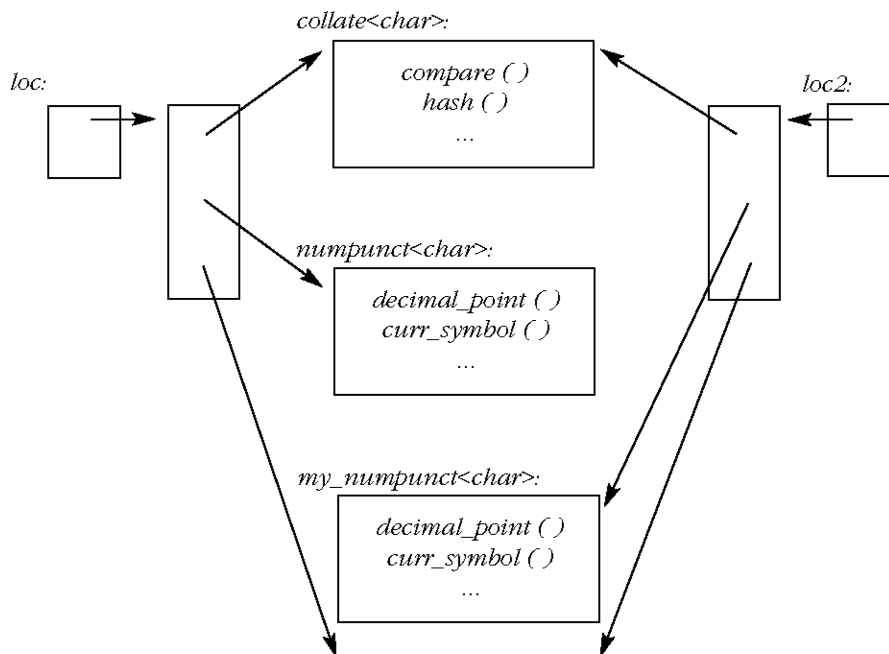
```
void swap(locale& x, locale& y)     // ugyanaz, mint az std::swap()
{
    locale temp = x;
    x = y;
    y = temp;
}
```

A másolat az eredetivel egyenértékű, de független, önálló objektum:

```
void f(locale* my_locale)
{
    locale loc = locale::classic(); // "C" lokál

    if (loc != locale::classic()) {
        cerr << "Hiba a megvalósításban: értesítse a készítőt.\n";
```

```
        exit(1);
    }

    if (&loc != &locale::classic()) cout << "Nem meglepő: a címek különböznek.\n";

    locale loc2 = locale(loc, my_locale, locale::numeric);

    if (loc == loc2) {
        cout << "a classic() hasonló facet-jében levőkkel.\n";
        // ...
    }
    // ...
}
```

Ha a *my_locale* rendelkezik olyan jellemzővel, amely a *classic()* lokál szabványos *numpunct<char>*-jától eltérően adja meg a számok elválasztó írásjeleit (*my_numpunct<char>*), az eredményül kapott lokálok a következőképpen lesznek ábrázolhatók:



A *locale* objektumok nem módosíthatók, műveleteik viszont lehetővé teszik új lokálok létrehozását a már meglevőkből. A létrehozás utáni módosítás tiltása (a lokál „nem változékony”, *immutable* természete) alapvető fontosságú a futási idejű hatékonyság érdekében, ez teszi ugyanis lehetővé a felhasználó számára, hogy meghívja a *facet*-ek virtuális függvényeit és tárolja a visszaadott értékeket. A bemeneti adatfolyamok például anélkül tudhatják, milyen karakter használatos a tizedesvessző (pontosabban „tizedespont”) jelzésére, illetve anélkül ismerhetik a *true* ábrázolását, hogy minden egyes alkalommal meghívnák – szám beolvasásakor – a *decimal_point()* függvényt vagy – logikai érték beolvasásakor – a *truename()*-et (§D.4.2). Csak az *imbue()* meghívása az adatfolyamra (§21.6.3) okozhatja, hogy ezek a hívások különböző értékeket adjanak vissza.

D.2.3. A global() és classic() lokálok

A programban érvényben levő lokál másolatát a *locale()* adja vissza, a *locale::global(x)* pedig *x*-re állítja azt. Az érvényes lokált gyakran „globális lokálnak” hívják, utalva arra, hogy valószínűleg globális (vagy statikus) objektum.

Az adatfolyamok létrehozásukkor automatikusan „feltöltődnek” (imbue; §21.1, §21.6.3) a globális lokállal, vagyis a *locale()* másolatával, ami először mindig a szabványos C *locale::classic()*.

A *locale::global()* statikus tagfüggvény megengedi, hogy a programozó meghatározza, melyik lokál legyen globális. Az előző másolatát a *global()* függvény adja vissza; ennek segítségével a felhasználó visszaállíthatja az eredeti globális lokált:

```
void f(const locale& my_loc)
{
    ifstream fin1(some_name);                       // fin1 feltöltése a globális lokállal
    locale& old_global = locale::global(my_loc);    // új globális lokál beállítása
    ifstream fin2(some_other_name);                 // fin2 feltöltése a my_loc lokállal
    // ...
    locale::global(old_global);                     // a régi globális lokál visszaállítása
}
```

Ha az *x* lokál rendelkezik névvel, a *locale::global(x)* szintén a C globális lokált állítja be. Ebből következik, hogy egy vegyes (C és C++) programban egységesen és következetesen kezelhetjük a lokált, ha meghívjuk a C standard könyvtárának valamelyik lokálfüggvényét.

Ha az *x* lokálnak nincs neve, akkor nem meghatározható, hogy a *locale::global(x)* befolyásolja-e a C globális lokált, vagyis a C++ programok nem képesek megbízhatóan (és „hordozható módon”) átállítani a C lokált egy olyan lokálra, amely nem a végrehajtási környe-

zetből való. Nincs szabványos mód arra sem, hogy egy C program beállíthassa a C++ globális lokált (kivéve ha meghív egy C++ függvényt, ami ezt megteszi), ezért a hibák elkerülése érdekében a vegyes programokban nem célszerű a *global()*-tól eltérő C globális lokált használni.

A globális lokál beállítása nincs hatással a már létező I/O adatfolyamokra; azok ugyanazt a lokált fogják használni, amivel eredetileg feltöltődtek. A *fin1*-re például nem hat a globális lokál módosítása, de a *fin2*-t a művelet a *my_loc*-kal tölti fel.

A globális lokál módosításával ugyanaz a probléma, mint a globális adatokat megváltoztató egyéb eljárásokkal: lényegében nem tudható, mire van hatással a változtatás. Ezért a legjobb, ha a *global()*-t a lehető legkevesebbszer használjuk és a módosítást olyan kódrészletekre korlátozzuk, ahol annak hatása pontosan nyomon követhető. Szerencsére ezt segíti az a lehetőség, hogy az adatfolyamokat meghatározott lokálokkal tölthetjük meg (imbue, §21.6.3). Azért vigyázzunk: ha a programban elszórva számos explicit *locale*- és *facet*-kezelő kód található, a program nehezen lesz fenntartható és módosítható.

D.2.4. Karakterláncok összehasonlítása

A lokálokat többnyire arra használjuk, hogy összehasonlítsunk két karakterláncot egy *locale* alapján. Ezt a műveletet maga a *locale* nyújtja, a felhasználóknak nem kell saját összehasonlító eljárást írniuk a *collate* jellemzőből (§D.4.1). Az eljárás a lokál *operator() ()* összehasonlító függvénye, így közvetlenül használható predikátumként (§18.4.2):

```
void f(vector<string>& v, const locale& my_locale)
{
    sort(v.begin(), v.end());                // rendezés a globális lokál szerint
    // ...
    sort(v.begin(), v.end(), my_locale);     // rendezés a my_locale szabályai szerint
    // ...
}
```

Alapértelmezés szerint a standard könyvtárbeli *sort()* a < műveletet alkalmazza a karakterek számértékére, hogy eldöntse a rendezési sorrendet (§18.7, §18.6.3.1).

Jegyezzük meg, hogy a lokálok *basic_string*-eket hasonlítanak össze, nem C stílusúakat.

D.3. Jellemzők

A jellemzők (*facet*-ek) a lokál *facet* tagosztályából származtatott osztályok objektumai:

```
class std::locale::facet {
protected:
    explicit facet(size_t r = 0);        // "r==0": a lokál szabályozza a facet élettartamát
    virtual ~facet();                    // figyelem: védett destruktor
private:
    facet(const facet&);                 // nem meghatározott
    void operator=(const facet&);        // nem meghatározott
    // ábrázolás
};
```

A másoló műveletek privát tagok és szándékosan nincsenek definiálva, hogy a másolást megakadályozzák (§11.2.2).

A *facet* osztály bázisosztály szerepet tölt be, nyilvános függvénye nincs. Konstruktora védett, hogy megakadályozza az „egyszerű *facet*" objektumok létrehozását, destruktora pedig virtuális, hogy biztosítsa a származott osztálybeli objektumok megfelelő megsemmisítését.

A jellemzők kezelését a lokálok elvileg mutatókon keresztül végzik. A *facet* konstruktorának *0* értékű paramétere azt jelenti, hogy a lokálnak törölnie kell az adott jellemzőt, ha már nincs rá hivatkozás. Ezzel szemben a nem nulla konstruktor-paraméter azt biztosítja, hogy a *locale* sohasem törli a jellemzőt. Ez az a ritka eset, amikor a *facet* élettartamát a programozó közvetlenül, nem a lokálon keresztül szabályozza. A *collate_byname<char>* szabványos *facet* típusú objektumokat például így hozhatjuk létre (§D.4.1.1):

```
void f(const string& s1, const string& s2)
{
    // szokásos eset: a 0 (alapértelmezett) paraméter azt jelzi,
    // hogy a lokál felel a felszámolásért
    collate<char>* p = new collate_byname<char>("pl");
    locale loc(locale(), p);

    // ritka eset: a paraméter értéke 1, tehát a felhasználó felel a felszámolásért
    collate<char>* q = new collate_byname<char>("ge",1);

    collate_byname<char> bug1("sw");    // hiba: lokális változót nem lehet felszámolni
    collate_byname<char> bug2("no",1);  // hiba: lokális változót nem lehet felszámolni
    // ...
```

```
    // q nem törölhető: a collate_byname<char> destruktora védett
    // nincs "delete p", mert a lokál intézi *p felszámolását
}
```

Azaz a szabványos *facet*-ek a lokál által kezelt bázisosztályként hasznosak, ritkán kell más módon kezelnünk azokat.

A *_byname()* végződésű jellemzők a végrehajtási környezetben található nevesített lokál *facet*-jei (§D.2.1).

Minden *facet*-nek rendelkeznie kell azonosítóval (*id*), hogy a lokálban a *has_facet()* és *use_facet()* függvényekkel megtalálható legyen (§D.3.1):

```
class std::locale::id {
public:
    id();
private:
    id(const id&);                    // nem definiált
    void operator=(const id&);        // nem definiált
    // ábrázolás
};
```

A másoló műveletek privátok és nincsenek kifejtve, hogy a másolást megakadályozzák (§11.2.2).

Az azonosító arra való, hogy a jellemzők számára új felületet adó osztályokban (például lásd §D.4.1-et) *id* típusú statikus tagokat hozhassunk létre. A lokálok eljárásai az *id*-t használják a *facet*-ek azonosítására (§D.2, §D.3.1). Az azonosítók többnyire egy *facet*-ekre hivatkozó mutatókból álló vektor indexértékei (vagyis *map<id,facet*>*, ami igen hatékony).

A (származtatott) jellemzőket meghatározó adatokat a származtatott osztály írja le, nem maga a *facet* bázisosztály. Ebből következik, hogy a programozó tetszőleges mennyiségű adatot használhat a *facet* fogalmának ábrázolására és korlátozás nélkül módosíthatja azokat.

Jegyezzük meg, hogy a felhasználói *facet*-ek minden függvényét *const* tagként kell meghatározni, mert a *facet*-eknek állandónak kell lenniük (§D.2.2).

D.3.1. Jellemzők elérése a lokálokban

A lokálok *facet*-jei a *use_facet* sablon függvényen keresztül érhetők el és a *has_facet* sablon függvénnyel kérdezhetjük le, hogy a lokál rendelkezik-e egy adott jellemzővel:

```
template <class Facet> bool has_facet(const locale&) throw();
template <class Facet> const Facet& use_facet(const locale&);   // bad_cast kivételt válthat
ki
```

Ezekre a függvényekre úgy kell gondolnunk, mintha azok *locale* paraméterükben keresnék *Facet* sablonparaméterüket. Más megközelítésben a *use_facet* egyfajta típuskényszerítés (cast), egy lokál konvertálása egy meghatározott jellemzőre. Ez azért lehetséges, mert a *locale* objektumok egy adott típusú *facet*-ből csak egy példányt tartalmazhatnak:

```
void f(const locale& my_locale)
{
    char c = use_facet< numpunct<char> >(my_locale).decimal_point()   // a szabványos
                                                                       // facet használata
    // ...

    if (has_facet<Encrypt>(my_locale)) {           // tartalmaz-e a my_locale Encrypt facet-et?
        const Encrypt& f = use_facet<Encrypt>(my_locale);   // az Encrypt facet kinyerése
                                                             // a my_locale-ból
        const Crypto c = f.get_crypto();                     // az Encrypt facet használata
        // ...
    }
    // ...
}
```

Jegyezzük meg, hogy a *use_facet* egy konstans *facet*-re való referenciát ad vissza, így az eredményt nem adhatjuk értékül egy nem konstans változónak. Ez azért logikus, mert a jellemzőknek elvileg nem módosíthatóknak kell lenniük és csak *const* tagokat tartalmazhatnak.

Ha meghívjuk a *use_facet<X>(loc)* függvényt és *loc* nem rendelkezik az *X* jellemzővel, a *use_facet() bad_cast* kivételt vált ki (§14.10). A szabványos *facet*-ek garantáltan hozzáférhetőek minden *locale* számára (§D.4), így az ő esetükben nem kell a *has_facet*-et használnunk, ezekre a *use_facet* nem fog *bad_cast* kivételt kiváltani.

Hogyan lehetne a *use_facet* és *has_facet* függvényeket megvalósítani? Emlékezzünk, hogy egy lokálra úgy gondolhatunk, mintha *map<id,facet*>* lenne (§D.2). Ha a *Facet* sablonparaméterként adott egy *facet* típus, a *has_facet* vagy *use_facet* a *Facet::id*-re hivatkozhat és ezt használhatja a megfelelő jellemző megkeresésére. A *has_facet* és *use_facet* nagyon egyszerű változata így nézhetne ki:

```
// ál-megvalósítás: képzeljük úgy, hogy a lokál rendelkezik egy facet_map-nek nevezett
// map<id,facet*>-tel

template <class Facet> bool has_facet(const locale& loc) throw()
{
    const locale::facet* f = loc.facet_map[Facet::id];
    return f ? true : false;
}

template <class Facet> const Facet& use_facet(const locale& loc)
{
    const locale::facet* f = loc.facet_map[Facet::id];
    if (f) return static_cast<const Facet&>(*f);
    throw bad_cast();
}
```

A *facet::id* használatát úgy is tekinthetjük, mint a fordítási idejű többalakúság (parametrikus polimorfizmus) egy formáját. A *dynamic_cast* a *use_facet*-hez hasonló eredményt adna, de az utóbbi sokkal hatékonyabb, mert kevésbé általános.

Az *id* valójában inkább egy felületet és viselkedést azonosít, mint osztályt. Azaz ha két *facet* osztálynak pontosan ugyanaz a felülete és (a *locale* szempontjából) ugyanaz a szerepük, akkor ugyanaz az *id* kell, hogy azonosítsa őket. A *collate<char>* és a *collate_byname<char>* például felcserélhető egy lokálban, így mindkettőt a *collate<char>::id* azonosítja (§D.4.1).

Ha egy jellemzőnek új felületet adunk – mint az *f()*-ben az *Encrypt*-nek –, definiálnunk kell számára az azonosítót (lásd §D.3.2-t és §D.4.1-et).

D.3.2. Egyszerű felhasználói facet-ek

A standard könyvtár a kulturális eltérések leglényegesebb területeihez – mint a karakterkészletek kezelése és a számok be- és kivitele – szabványos *facet*-eket nyújt. Ahhoz, hogy az általuk nyújtott szolgáltatásokat a széles körben használatos típusok bonyolultságától és a velük járó hatékonysági problémáktól elkülönítve vizsgálhassuk, hadd mutassak be először egy egyszerű felhasználói típusra vonatkozó *facet*-et:

```
enum Season { tavasz, nyár, ősz, tél };   // évszakok
```

Ez volt a legegyszerűbb felhasználói típus, ami éppen eszembe jutott. Az itt felvázolt I/O kis módosításokkal a legtöbb egyszerű felhasználói típus esetében felhasználható.

```
class Season_io : public locale::facet {
public:
    Season_io(int i = 0) : locale::facet(i) { }
    ~Season_io() { }    // lehetővé teszi a Season_io objektumok felszámolását (§D.3)

    // x ábrázolása karakterláncként
    virtual const string& to_str(Season x) const = 0;

    // az s karakterláncnak megfelelő évszak elhelyezése x-be:
    virtual bool from_str(const string& s, Season& x) const = 0;

    static locale::id id;          // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
};
locale::id Season_io::id;          // az azonosító objektum meghatározása
```

Az egyszerűség kedvéért a *facet* csak a *char* típust használó megvalósításokra korlátozódik.

A *Season_io* osztály általános, elvont felületet nyújt minden *Season_io facet* számára.

Ha a *Season* be- és kimenetét egy adott lokálra szeretnénk definiálni, a *Season_io*-ból származtatunk egy osztályt, amelyben megfelelően kifejtjük a *to_str()* és *from_str()* függvényeket.

A *Season* értékét könnyű kiírni. Ha az adatfolyam rendelkezik *Season_io* jellemzővel, azt használva az értéket karakterlánccá alakíthatjuk, ha nem, kiírhatjuk a *Season* egész értékét:

```
ostream& operator<<(ostream& s, Season x)
{
    const locale& loc = s.getloc();    // az adatfolyam lokáljának kinyerése (§21.7.1)
    if (has_facet<Season_io>(loc)) return s << use_facet<Season_io>(loc).to_str(x);
    return s << int(x);
}
```

Észrevehetjük, hogy a << műveletet úgy definiáltuk, hogy más típusokra hívtuk meg a <<-t. Ennek számos előnye van: egyszerűbb a <<-t használnunk, mint a kimeneti adatfolyam átmeneti táraihoz közvetlenül hozzáférnünk, a << műveletet kifejezetten a lokálhoz igazíthatjuk és a művelet hibakezelést is biztosít. A szabványos *facet*-ek a legnagyobb hatékonyság és rugalmasság elérése érdekében többnyire közvetlenül az adatfolyam átmeneti tárát kezelik (§D.4.2.2, §D.4.2.3), de sok felhasználói típus esetében nincs szükség arra, hogy a *streambuf* absztrakciós szintjére süllyedjünk.

Ahogy lenni szokott, a bemenet kezelése némileg bonyolultabb, mint a kimeneté:

```
istream& operator>>(istream& s, Season& x)
{
    const locale& loc = s.getloc();          // az adatfolyam lokáljának kinyerése (§21.7.1)

    if (has_facet<Season_io>(loc)) {         // a szöveges ábrázolás beolvasása
        const Season_io& f = use_facet<Season_io>(loc);
        string buf;
        if (!(s>>buf && f.from_str(buf,x))) s.setstate(ios_base::failbit);
        return s;
    }
    int i;                  // a számábrázolás beolvasása
    s >> i;
    x = Season(i);
    return s;
}
```

A hibakezelés egyszerű, a beépített típusok hibakezelésének stílusát követi. Azaz, ha a bemenő karakterlánc nem a választott lokál valamelyik *Season*-ját jelöli, az adatfolyam hibás (*failure*) állapotba kerül. Ha a kivételek megengedettek, *ios_base::failure* kivétel kiváltására kerülhet sor (§21.3.6).

Vegyünk egy egyszerű tesztprogramot:

```
int main()      // egyszerű teszt
{
    Season x;

    // az alapértelmezett lokál használata (nincs Season_io facet);
    // egész értékű I/O-t eredményez:
    cin >> x;
    cout << x << endl;

    locale loc(locale(), new US_season_io);
    cout.imbue(loc);    // Season_io facet-tel rendelkező lokál használata
    cin.imbue(loc);     // Season_io facet-tel rendelkező lokál használata

    cin >> x;
    cout << x << endl;
}
```

A

```
2
summer
```

bemenetre a program válasza:

```
2
summer
```

Ennek eléréséhez definiálnunk kell a *US_season_io* osztályt, amelyben megadjuk az évszakok karakterlánc-ábrázolását és felülírjuk a *Season_io* azon függvényeit, amelyek a karakterláncokat a felsorolt elemekre alakítják:

```
class US_season_io : public Season_io {
    static const string seasons[];
public:
    const string& to_str(Season) const;
    bool from_str(const string&, Season&) const;

    // figyelem: nincs US_season_io::id
};

const string US_season_io::seasons[] = { "spring", "summer", "fall", "winter" };

const string& US_season_io::to_str(Season x) const
{
    if (x<spring || winter<x) {
        static const string ss = "Nincs ilyen évszak";
        return ss;
    }
    return seasons[x];
}

bool US_season_io::from_str(const string& s, Season& x) const
{
    const string* beg = &seasons[spring];
    const string* end = &seasons[winter]+1;
    const string* p = find(beg,end,s);        // §3.8.1, §18.5.2

    if (p==end) return false;
    x = Season(p-beg);
    return true;
}
```

Vegyük észre, hogy mivel a *US_season_io* csupán a *Season_io* felület megvalósítása, nem adunk meg azonosítót a *US_season_io* számára. Sőt, ha a *US_season_io*-t *Season_io*-ként akarjuk használni, nem is szabad ilyet tennünk. A lokálok műveletei (például a *has_facet*, §D.3.1) arra támaszkodnak, hogy az azonos fogalmakat ábrázoló *facet*-eket ugyanaz az *id* azonosítja (§D.3).

A megvalósítással kapcsolatos egyetlen érdekes kérdés az, hogy mit kell tenni, ha érvénytelen *Season* kiírását kérik? Természetesen ennek nem lenne szabad megtörténnie. Az egyszerű felhasználói típusoknál azonban nem ritka, hogy érvénytelen értéket találunk, így számításba kell vennünk ezt a lehetőséget is. Kiválthatnánk egy kivételt, de miután olyan egyszerű kimenettel foglalkozunk, amelyet emberek fognak olvasni, hasznos, ha a tartományon kívüli értékeket az „értéktartományon kívüli" szöveg is jelzi. A bemenetnél itt a kivételkezelés a >> műveletre hárul, míg a kimenet esetében ezt a *facet to_str()* függvénye végzi (hogy bemutathassuk a lehetőségeket). Valódi programoknál a *facet* függvényei a be- és kimeneti hibák kezelésével egyaránt foglalkoznak, vagy csak jelentik a hibákat, a << és >> műveletekre bízva azok kezelését.

A *Season_io* ezen változata arra támaszkodott, hogy a származtatott osztályok adják meg a lokálra jellemző karakterláncokat. Egy másik megoldás, hogy a *Season_io* maga szerzi meg ezeket egy, a lokálhoz kapcsolódó adattárból (lásd §D.4.7). Gyakorlatként hagytuk annak kidolgozását, hogy egyetlen *Season_io* osztályunk van, amelynek az évszakokat jelző karakterláncok a konstruktor paramétereként adódnak át (§D.6[2]).

D.3.3. A lokálok és jellemzők használata

A lokálok elsődlegesen a standard könyvtáron belül, az I/O adatfolyamokban használatosak, de a *locale* a helyi sajátosságok ábrázolásának ennél általánosabb eszköze. A *messages* (§D.4.7) például olyan *facet*, amelynek semmi köze a be- és kimeneti adatfolyamokhoz. Az *iostream* könyvtár esetleges bővítései, sőt, a nem adatfolyamokkal dolgozó be- és kimeneti eszközök is kihasználhatják a lokálok adta lehetőségeket, a felhasználó pedig a *locale* objektumok segítségével tetszőleges módon rendezheti a helyi sajátosságokat.

A lokálokon és jellemzőkön alapuló eljárások általánossága révén a felhasználói *facet*-ek adta lehetőségek korlátlanok. A dátumok, időzónák, telefonszámok, társadalombiztosítási számok (személyi számok), gyártási számok, hőmérsékletek, általános (mértékegység, érték) párok, irányítószámok, ruhaméretek, és ISBN számok mind megadhatók *facet*-ként.

Mint minden „erős" szolgáltatással, a *facet*-ekkel is óvatosan kell bánni. Az, hogy valamit lehet jellemzőként ábrázolni, még nem jelenti azt, hogy ez a legjobb megoldás. A kulturális eltérések ábrázolásának kiválasztásakor a kulcskérdés – mint mindig – az, hogy milyen nehéz a kód megírása, mennyire könnyű a kapott kódot olvasni, valamint hogy hogyan befolyásolják a döntések a kapott program fenntarthatóságát, illetve az I/O műveletek idő- és tárbeli hatékonyságát.

D.4. Szabványos facet-ek

A standard könyvtár <*locale*> fejállománya a következő *facet*-eket nyújtja a *classic()* lokálhoz:

Szabványos facet-ek (a classic() lokálban)			
	Kategória	Rendeltetés	Facet-ek
§D.4.1	*collate*	Karakterláncok összehasonlítása	*collate<Ch>*
§D.4.2	*numeric*	Számok be- és kivitele	*numpunct<Ch>* *num_get<Ch>* *num_put<Ch>*
§D.4.3	*monetary*	Pénz I/O	*moneypunct<Ch>* *moneypunct<Ch,true>* *money_get<Ch>* *money_put<Ch>*
§D.4.4	*time*	Idő I/O	*time_get<Ch>* *time_put<Ch>*
§D.4.5	*ctype*	Karakterek osztályozása	*ctype<Ch>* *codecvt<Ch,char,mbstate_t>*
§D.4.7	*messages*	Üzenet-visszakeresés	*messages<Ch>*

A táblázatban a *Ch* helyén *char* vagy *wchar_t* típus szerepelhet. Ha a felhasználónak arra van szüksége, hogy a szabványos I/O másfajta *X* karaktertípust kezeljen, a megfelelő *facet*-eket meg kell adnia az *X* számára. A *codecvt<X,char,mbstate_t>* (§D.4.6) például szükséges lehet az *X* és *char* típusok közötti átalakításokhoz. Az *mbstate_t* típus arra való, hogy egy többájtos karakterábrázolás léptetési állapotait jelölje (§D.4.6); definiciója a <*cwchar*> és a <*wchar.h*> fejállományokban található. Tetszőleges *X* karaktertípus esetében az *mbstate_t*-nek a *char_traits<X>::state_type* felel meg.

A standard könyvtár további *facet*-jei a *<locale>* fejállományban a következők:

Szabványos facet-ek			
	Kategória	Rendeltetés	Facet-ek
§D.4.1	*collate*	Karakterláncok összehasonlítása	*collate_byname<Ch>*
§D.4.2	*numeric*	Számok be- és kivitele	*numpunct_byname<Ch>* *num_get<C,In>* *num_put<C,Out>*
§D.4.3	*monetary*	Pénz I/O	*moneypunct_byname<Ch,International>* *money_get<C,In>* *money_put<C,Out>*
§D.4.4	*time*	Idő I/O	*time_put_byname<Ch>*
§D.4.5	*ctype*	Karakterek osztályozása	*ctype_byname<Ch>*
§D.4.7	*messages*	Üzenet-visszakeresés	*messages_byname<Ch>*

A táblázatban szereplő jellemzők példányosításakor a *Ch char* vagy *wchar_t* lehet; a *C* bármilyen karaktertípus (§20.1), az *International* értéke *true* vagy *false*, ahol a *true* azt jelenti, hogy a valuta-szimbólum négykarakteres „nemzetközi" ábrázolását használjuk (§D.4.3.1). Az *mbstate_t* típus a többájtos karakter-ábrázolások léptetési állapotait jelöli (§D.4.6), meghatározása a *<cwchar>* és a *<wchar.h>* fejállományokban található.

Az *In* és az *Out* bemeneti és kimeneti bejárók (iterátorok, §19.1, §19.2.1). Ha a *_put* és *_get* jellemzőket ellátjuk ezekkel a sablonparaméterekkel, olyan *facet*-eket hozhatunk létre, amelyek nem szabványos átmeneti tárakhoz férnek hozzá (§D.4.2.2). Az *iostream*-ek átmeneti tárai (pufferei) adatfolyam átmeneti tárak, így bejáróik *ostreambuf_iterator*-ok (§19.2.6.1, §D.4.2.2), vagyis a hibakezeléshez rendelkezésünkre áll a *failed()* függvény.

Az *F_byname* az *F facet*-ből származik; ugyanazt a felületet nyújtja mint az *F*, de hozzáad egy konstruktort, amelynek egy lokált megnevező karakterlánc paramétere van (lásd §D.4.1-et). Az *F_byname(név)* jelentése ugyanaz, mint az *F locale(név)* szerkezeté. Az elgondolás az, hogy a program végrehajtási környezetében egy nevesített lokálból (§D.2.1) kivesszük a szabványos *facet* egy adott változatát:

```
void f(vector<string>& v, const locale& loc)
{
    locale d1(loc, new collate_byname<char>("da")); // dán karakterlánc-összehasonlítás
                                                    // használata
    locale dk(d1, new ctype_byname<char>("da"));    // dán karakterosztályozás használata
    sort(v.begin(), v.end(), dk);
    // ...
}
```

Az új *dk* lokál „dán stílusú" karakterláncokat fog használni, de megtartja a számokra vonatkozó alapértelmezett szabályokat. Mivel a *facet* második paramétere alapértelmezés szerint *0*, a *new* művelettel létrehozott *facet* élettartamát a lokál fogja kezelni (§D.3).

A karakterlánc paraméterekkel rendelkező *locale*-konstruktorokhoz hasonlóan a *_byname*-konstruktorok is hozzáférnek a program végrehajtási környezetéhez. Ebből az következik, hogy nagyon lassúak azokhoz a konstruktorokhoz képest, amelyeknek nem kell a környezethez fordulniuk információért. Majdnem mindig gyorsabb, ha létrehozunk egy lokált és azután férünk hozzá annak jellemzőihez, mintha *_byname facet*-eket használnánk több helyen a programban. Ezért jó ötlet, ha a *facet*-et egyszer olvassuk be a környezetből, majd később mindig a memóriában lévő másolatát használjuk:

```
locale dk("da");   // a dán lokál beolvasása (beleértve összes facet-jét) egyszer,
                   // majd a dk lokál és jellemzőinek használata igény szerint

void f(vector<string>& v, const locale& loc)
{
    const collate<char>& col = use_facet< collate<char> >(dk);
    const collate<char>& ctyp = use_facet< ctype<char> >(dk);

    locale d1(loc,col);   // dán karakterlánc-összehasonlítás használata
    locale d2(d1,ctyp);   // dán karakterosztályozás és karakterlánc-
                          // összehasonlítás használata

    sort(v.begin(), v.end(), d2);
    // ...
}
```

A kategóriák egyszerűbbé teszik a lokálok szabványos *facet*-jeinek kezelését. Például ha adott a *dk* lokál, létrehozhatunk belőle egy másikat, amely a dán nyelv szabályainak megfelelően (ez az angolhoz képest három további magánhangzót jelent) olvas be és hasonlít össze karakterláncokat, de megtartja a C++-ban használatos számformátumot:

```
locale dk_us(locale::classic(), dk, collate|ctype);       // dán betűk, amerikai számok
```

Az egyes szabványos *facet*-ek bemutatásánál további példákat nézünk meg a jellemzők használatára. A *collate* (§D.4.1) tárgyalásakor például a *facet*-ek sok közös szerkezetbeli tulajdonsága előkerül.

Jegyezzük meg, hogy a szabványos *facet*-ek gyakran függnek egymástól. A *num_put* például a *numpunct*-ra támaszkodik. Csak ha az egyes jellemzőket már részletesen ismerjük, akkor lehetünk képesek azokat sikeresen együtt használni, egyeztetni vagy új változataikat elkészíteni. Más szavakkal, a §21.7 pontban említett egyszerű műveleteken túl a lokálok nem arra valók, hogy a kezdők közvetlenül használják azokat.

A jellemzők megtervezése gyakran nagyon körülményes. Ennek oka részben az, hogy a jellemzők nem rendszerezhető helyi sajátosságokat kell, hogy tükrözzenek, melyekre a könyvtár tervezője nincs befolyással; másrészt pedig az, hogy a C++ standard könyvtárbeli eszközeinek nagyrészt összeegyeztethetőnek kell maradniuk azzal, amit a C standard könyvtára és az egyes platformok szabványai nyújtanak. A POSIX például olyan eszközökkel támogatja a lokálok használatát, melyeket a könyvtárak tervezői nem szabad, hogy figyelmen kívül hagyjanak.

Másfelől a lokálok és jellemzők által nyújtott szerkezet nagyon általános és rugalmas. A *facet*-ek bármilyen adatot tárolhatnak és azokon bármilyen műveletet végezhetnek. Ha az új *facet* viselkedését a szabályok nem korlátozzák túlságosan, szerkezete egyszerű és tiszta lehet (§D.3.2).

D.4.1. Karakterláncok összehasonlítása

A szabványos *collate* jellemző *Ch* típusú karakterekből álló tömbök összehasonlítását teszi lehetővé:

```
template <class Ch>
class std::collate : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit collate(size_t r = 0);

    int compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const
        { return do_compare(b,e,b2,e2); }

    long hash(const Ch* b, const Ch* e) const { return do_hash(b,e); }
    string_type transform(const Ch* b, const Ch* e) const { return do_transform(b,e); }
```

```
    static locale::id id;          // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~collate();                    // figyelem: védett destruktor

    virtual int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    virtual string_type do_transform(const Ch* b, const Ch* e) const;
    virtual long do_hash(const Ch* b, const Ch* e) const;
};
```

A többi *facet*-hez hasonlóan a *collate* is nyilvános módon származik a *facet* osztályból és olyan konstruktorral rendelkezik, amelynek egyetlen paramétere azt mondja meg, hogy a *locale* osztály vezérli-e a jellemző élettartamát (§D.3).

Jegyezzük meg, hogy a destruktor védett (protected). A *collate* nem közvetlen használatra való, inkább arra szánták, hogy minden (származtatott) összehasonlító osztály alapja legyen és a *locale* kezelje (§D.3). A rendszerfejlesztőknek és a könyvtárak készítőinek a karakterláncokat összehasonlító *facet*-eket úgy kell megírniuk, hogy a *collate* nyújtotta felületen keresztül lehessen használni azokat.

Az alapvető karakterlánc-összehasonlítást a *compare()* függvény végzi, az adott *collate*-re vonatkozó szabályok szerint; *1*-et ad vissza, ha az első karakterlánc több karakterből áll, mint a második, *0*-át, ha a karakterláncok megegyeznek, és *-1*-et, ha a második karakterlánc „nagyobb”, mint az első:

```
void f(const string& s1, const string& s2, collate<char>& cmp)
{
    const char* cs1 = s1.data(); // mivel a compare() művelet char[] tömbön dolgozik
    const char* cs2 = s2.data();

    switch ( cmp.compare(cs1,cs1+s1.size(), cs2,cs2+s2.size()) )
    {
    case 0:        // a cmp szerint azonos karakterláncok
        // ...
        break;
    case -1:       // s1 < s2
        // ...
        break;
    case 1:        // s1 > s2
        // ...
        break;
    }
}
```

Vegyük észre, hogy a *collate* tagfüggvények *Ch* típusú elemekből álló tömböket hasonlítanak össze, nem *basic_string*-eket vagy nulla végződésű C stílusú karakterláncokat, vagyis a *0* számértékű *Ch* közönséges karakternek minősül, nem végződésnek. A *compare()* abban is különbözik a *strcmp()*-től, hogy pontosan a *-1, 0, 1* értékeket adja vissza, nem egyszerűen *0*-át és (tetszőleges) pozitív és negatív értékeket (§20.4.1).

A standard könyvtárbeli *string* nem függ a lokáloktól, azaz a karakterláncok összehasonlítása az adott nyelvi változat karakterkészlete alapján történik (§C.2). Ezenkívül a szabványos *string* nem biztosít közvetlen módot arra, hogy meghatározzuk az összehasonlítási feltételt (20. fejezet). A lokáltól függő összehasonlításhoz a *collate* kategória *compare()* függvényét használhatjuk. Még kényelmesebb, ha a függvényt a lokál *operator()* operátorán keresztül, közvetett módon hívjuk meg (§D.2.4):

```
void f(const string& s1, const string& s2, const char* n)
{
    bool b = s1 == s2; // összehasonlítás a megvalósítás karakterkészletének értékei szerint

    const char* cs1 = s1.data();    // mivel a compare() művelet char[] tömbön dolgozik
    const char* cs2 = s2.data();

    typedef collate<char> Col;

    const Col& glob = use_facet<Col>(locale());      // az érvényes globális lokálból
    int i0 = glob.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

    const Col& my_coll = use_facet<Col>(locale(""));   // az előnyben részesített lokálból
    int i1 = my_coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

    const Col& coll = use_facet<Col>(locale(n));      // az "n" nevű lokálból
    int i2 = coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

    int i3 = locale()(s1,s2);    // összehasonlítás az érvényes globális lokál alapján
    int i4 = locale("")(s1,s2);  // összehasonlítás az előnyben részesített lokál alapján
    int i5 = locale(n)(s1,s2);   // összehasonlítás az "n" lokál alapján
    // ...
}
```

Itt *i0==i3*, *i1==i4*, és *i2==i5*, de könnyű olyan eseteket elképzelni, ahol *i2*, *i3*, és *i4* értéke más. Vegyük a következő szavakból álló sorozatot egy német szótárból:

Dialekt, Diät, dich, dichten, Dichtung

A nyelv szabályainak megfelelően a főnevek (és csak azok) nagy kezdőbetűsek, de a rendezés nem különbözteti meg a kis- és nagybetűket.

Egy kis- és nagybetűket megkülönböztető német nyelvű rendezés minden *D*-vel kezdődő szót a *d* elé tenne:

Dialekt, Diät, Dichtung, dich, dichten

Az *ä* (umlautos a) „egyfajta a"-nak minősül, így a *c* elé kerül. A legtöbb karakterkészletben azonban az *ä* számértéke nagyobb a *c* számértékénél, következésképpen *int('c') < int('ä')*, a számértékeken alapuló egyszerű alapértelmezett rendezés pedig a következőket adja:

Dialekt, Dichtung, Diät, dich, dichten

Érdekes feladat lehet egy olyan függvényt írni, amely a szótárnak megfelelően helyesen rendezi ezt a sorozatot (§D.6[3]).

A *hash()* függvény egy hasítóértéket számít ki (§17.6.2.3), ami magától értetődően hasítótáblák létrehozásakor lehet hasznos.

A *transform()* függvény egy olyan karakterláncot állít elő, amelyet más karakterláncokkal összehasonlítva ugyanazt az eredményt kapjuk, mint amit a paraméter-karakterlánccal való összehasonlítás eredményezne. A *transform()* célja az, hogy optimális kódot készíthessünk az olyan programrészekből, ahol egy karakterláncot számos másikkal hasonlítunk össze. Ez akkor hasznos, ha karakterláncok halmazában egy vagy több karakterláncot szeretnénk megkeresni.

A *public compare()*, a *hash()* és a *transform()* függvények megvalósítását a *do_compare()*, *do_hash()* és *do_transform()* nyilvános virtuális függvények meghívása biztosítja. Ezeket a „*do_* függvényeket" a származtatott osztályokban felül lehet írni. A kétfüggvényes megoldás lehetővé teszi a könyvtár azon készítőjének, aki a nem virtuális függvényeket írja, hogy valamilyen közös szerepet biztosítson minden hívásnak, függetlenül attól, hogy mit csinálnának a felhasználó által megadott *do_* függvények.

A virtuális függvények használata megőrzi a *facet*-ek többalakú (polimorfikus) természetét, de költséges lehet. A túl sok függvényhívás elkerüléséhez a *locale* pontosan meghatározhatja a használatos jellemzőket és bármennyi értéket a gyorsítótárba tehet, amennyire csak szüksége van a hatékony végrehajtáshoz (§D.2.2).

A jellemzőket *locale::id* típusú statikus *id* tagok azonosítják (§D.3). A szabványos *has_facet* és *use_facet* függvények az azonosítók és jellemzők közötti összefüggéseken alapulnak (§D.3.1). Az azonos felületű és szerepű *facet*-eknek ugyanazzal az azonosítóval kell rendelkezniük, így a *collate<char>* és a *collate_byname<char>* (§D.4.1.1) azonosítója is megegyezik. Következésképpen két *facet*-nek biztosan különböző az azonosítója, ha (a *locale* szempontjából nézve) különböző függvényeket hajtanak végre, így ez a helyzet a *numpunct<char>* és a *num_put<char>* esetében is (§D.4.2).

D.4.1.1. Nevesített Collate

A *collate_byname* olyan jellemző, amely a *collate* azon változatát nyújtja az adott lokálnak, amelyet a konstruktor karakterlánc-paramétere nevez meg:

```
template <class Ch>
class std::collate_byname : public collate<Ch> {
public:
    typedef basic_string<Ch> string_type;

    // létrehozás névvel rendelkező lokálból
    explicit collate_byname(const char*, size_t r = 0);

    // figyelem: nincs azonosító és nincsenek új függvények

protected:
    ~collate_byname(); // figyelem: védett destruktor

    // a collate<Ch> virtuális függvényeinek felülírása

    int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    string_type do_transform(const Ch* b, const Ch* e) const;
    long do_hash(const Ch* b, const Ch* e) const;
};
```

Így a *collate_byname* arra használható, hogy kivegyünk egy *collate*-et egy, a program végrehajtási környezetében levő nevesített lokálból (§D.4). A végrehajtási környezetben a *facet*-eket egyszerűen fájlban, adatként is tárolhatjuk. Egy kevésbé rugalmas megoldás, ha a jellemzőket programszövegként és adatként ábrázoljuk egy *_byname facet*-ben.

A *collate_byname<char>* osztály olyan *facet*, amelynek nincs saját azonosítója (§D.3). A lokálokban a *collate_byname<Ch>* és a *collate<Ch>* felcserélhetők. Azonos lokál esetében a *collate* és a *collate_byname* csak az utóbbi szerepében és a *collate_byname* által felkínált további konstruktorban különbözik.

Jegyezzük meg, hogy a *_byname* destruktor védett. Ebből következik, hogy lokális (helyi) változóként nem használhatunk *_byname facet*-et:

```
void f()
{
    collate_byname<char> my_coll("");  // hiba: a my_coll nem számolható fel
    // ...
}
```

Ez azt a nézőpontot tükrözi, hogy a lokálok és jellemzők olyasmik, amiket a legjobb elég-gé magas szinten használni a programban, hogy a program minél nagyobb részére legyünk hatással. Erre példa a globális lokál beállítása (§D.2.3) vagy egy adatfolyam megtöltése (§21.6.3, §D.1). Ha szükséges, egy *_byname* osztályból egy nyilvános destruktorral rendelkező osztályt származtathatunk és ebből az osztályból lokális változókat hozhatunk létre.

D.4.2. Számok be- és kivitele

A szám-kimenetet a *num_put facet* kezeli, amely egy adatfolyam átmeneti tárba ír (§21.6.4). A bemenet kezelése a *num_get* dolga; ez is átmeneti tárból olvas. A *num_put* és *num_get* által használt formátumot a *numpunct* jellemző határozza meg.

D.4.2.1. Számjegy-formátumok

A beépített típusok (mint a *bool*, az *int*, és a *double*) be- és kimeneti formátumát a *numpunct* jellemző írja le:

```
template <class Ch>
class std::numpunct : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit numpunct(size_t r = 0);

    Ch decimal_point() const; // '.' a classic() lokálban
    Ch thousands_sep() const; // ',' a classic() lokálban
    string grouping() const;     // ""  a classic() lokálban, jelentése: nincs csoportosítás

    string_type truename() const;          // "true"  a classic() lokálban
    string_type falsename() const;         // "false" a classic() lokálban
```

```
    static locale::id id;          // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~numpunct();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

A *grouping()* által visszaadott karakterlánc karaktereinek beolvasása kis egész értékek sorozataként történik. Minden szám a számjegyek számát határozza meg egy csoport számára. A 0. karakter a jobb szélső csoportot adja meg (ezek a legkisebb helyiértékű számjegyek), az 1. az attól balra levő csoportot és így tovább. Így a *"\004\002\003"* egy számot ír le (pl. *123-45-6789*, feltéve, hogy a *'-'* az elválasztásra használt karakter). Ha szükséges, a csoportosító minta utolsó karaktere ismételten használható, így a *"\003"* egyenértékű a *"\003\003\003"*-mal. Ahogy az elválasztó karakter neve, a *thousands_sep()* mutatja is, a csoportosítást leggyakrabban arra használják, hogy a nagy egészeket olvashatóbbá tegyék. A *grouping()* és *thousands_sep()* függvények az egészek be- és kimeneti formátumát is megadják, a lebegőpontos számok szabványos be- és kimenetéhez azonban nem használatosak, így nem tudjuk kiíratni az *1234567.89*-et *1,234,567.89*-ként csupán azáltal, hogy megadjuk a *grouping()* és *thousands_sep()* függvényeket.

A *numpunct* osztályból származtatással új formátumot adhatunk meg. A *My_punct* jellemzőben például leírhatjuk, hogy az egész értékek számjegyeit szóközökkel elválasztva és hármasával csoportosítva, a lebegőpontos értékeket pedig európai stílus szerint, tizedesvesszővel elválasztva kell kiírni:

```
class My_punct : public std::numpunct<char> {
public:
    typedef char char_type;
    typedef string string_type;

    explicit My_punct(size_t r = 0) : std::numpunct<char>(r) { }
protected:
    char do_decimal_point() const { return ','; }  // vessző
    char do_thousands_sep() const { return ' '; }  // szóköz
    string do_grouping() const { return "\003"; }  // 3 számjegyű csoportok
};

void f()
{
    cout << "Első stílus: " << 12345678 << " *** " << 1234567.8 << '\n';

    locale loc(locale(), new My_punct);
    cout.imbue(loc);
    cout << "Második stílus: " << 12345678 << " *** " << 1234567.8 << '\n';
}
```

Ez a következő eredményt adja:

*Első stílus: 12345678 *** 1.23457e+06*
*Második stílus: 12 345 678 *** 1,23457e+06*

Jegyezzük meg, hogy az *imbue()* az adatfolyamában másolatot tárol paraméteréről. Következésképpen az adatfolyam akkor is támaszkodhat egy megtöltött lokálra, ha annak eredeti példánya már nem létezik. Ha a bemeneti adatfolyam számára be van állítva a *boolalpha* jelzőbit (§21.2.2, §21.4.1), a *true* és *false* értékeket a *truename()* és a *falsename()* által visszaadott karakterláncok jelölhetik, más esetben a *0* és az *1*.

A *numpunct _byname* változata (§D.4, §D.4.1) is adott:

```
template <class Ch>
class std::numpunct_byname : public numpunct<Ch> { /* ... */ };
```

D.4.2.2. Számok kiírása

Az átmeneti tárba való íráskor (§21.6.4) a kimeneti adatfolyamok (*ostream*) a *num_put* jellemzőre támaszkodnak:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::num_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;

    explicit num_put(size_t r = 0);

    // a "v" érték elhelyezése az "s" adatfolyam átmeneti tárának "b" pozíciójára:
    Out put(Out b, ios_base& s, Ch fill, bool v) const;
    Out put(Out b, ios_base& s, Ch fill, long v) const;
    Out put(Out b, ios_base& s, Ch fill, unsigned long v) const;
    Out put(Out b, ios_base& s, Ch fill, double v) const;
    Out put(Out b, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, ios_base& s, Ch fill, const void* v) const;

    static locale::id id; // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~num_put();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

Az *Out* kimeneti bejáró (iterátor) paraméter azonosítja, hogy a *put()* a számértéket jelölő karaktereket hol helyezi el a kimeneti adatfolyam átmeneti tárában (§21.6.4). A *put()* értéke az a bejáró (iterator), amely egy hellyel az utolsó karakter mögé mutat.

Jegyezzük meg, hogy a *num_put* alapértelmezett változata (amelynek bejárójával *ostreambuf_iterator<Ch>* típusú karakterekhez lehet hozzáférni) a szabványos *locale*-ek (§D.4) része. Ha más változatot akarunk használni, akkor azt magunknak kell elkészítenünk:

```
template<class Ch>
class String_numput : public std::num_put<Ch, typename basic_string<Ch>::iterator> {
public:
    String_numput() : num_put<Ch, typename basic_string<Ch>::iterator>(1) { }
};

void f(int i, string& s, int pos)       // "i" formázása "s"-be, a "pos" pozíciótól kezdve
{
    String_numput<char> f;
    ios_base& xxx = cout;               // a cout formázási szabályainak használata
    f.put(s.begin()+pos, xxx, ' ', i);  // "i" formázása "s"-be
}
```

Az *ios_base* paraméterrel a formátumról és a lokálról kaphatunk információt. Például ha üres helyeket kell kitöltenünk, az *ios_base* paraméter által megadott *fill* karakter lesz felhasználva. Az átmeneti tár, amelybe *b*-n keresztül írunk, általában ahhoz az *ostream*-hez kapcsolódik, amelynek *s* a bázisosztálya. Az *ios_base* objektumokat nem könnyű létrehozni, mert a formátummal kapcsolatban több dolgot is szabályoznak és ezeknek egységesnek kell lenniük, hogy a kimenet elfogadható legyen. Következésképpen az *ios_base* osztálynak nincs nyilvános konstruktora (§21.3.3).

A *put()* függvények szintén *ios_base* paramétereket használnak az adatfolyam lokáljának lekérdezéséhez. A lokál határozza meg az elválasztó karaktereket (§D.4.2.1), a logikai értékek szöveges ábrázolását és a *Ch*-ra való átalakítást. Például ha feltesszük, hogy a *put()* függvény *ios_base* paramétere *s*, a *put()* függvényben ehhez hasonló kódot találhatunk:

```
const locale& loc = s.getloc();
// ...
wchar_t w = use_facet< ctype<char> >(loc).widen(c);          // átalakítás char-ról Ch-ra
// ...
string pnt = use_facet< numpunct<char> >(loc).decimal_point(); // alapértelmezés: '.'
// ...
string flse = use_facet< numpunct<char> >(loc).falsename();    // alapértelmezés: "false"
```

A *num_put<char>*-hoz hasonló szabványos *facet*-eket a szabványos I/O adatfolyam-függvények általában automatikusan használják, így a legtöbb programozónak nem is kell tudnia róluk. Érdemes azonban szemügyre venni, hogy a standard könyvtár függvényei hogyan használják a jellemzőket, mert jól mutatja, hogyan működnek a be- és kimeneti adatfolyamok, illetve a *facet*-ek. Mint mindig, a standard könyvtár most is érdekes programozási eljárásokra mutat példákat.

A *num_put* felhasználásával az *ostream* készítője a következőket írhatja:

```
template<class Ch, class Tr>
ostream& std::basic_ostream<Ch,Tr>::operator<<(double d)
{
    sentry guard(*this);        // lásd §21.3.8
    if (!guard) return *this;

    try {
        if (use_facet< num_put<Ch> >(getloc()).put(*this,*this,this->fill(),d).failed())
            setstate(badbit);
    }
    catch (...) {
        handle_ioexception(*this);
    }
    return *this;
}
```

Itt sok minden történik. Az „őrszem" (sentry) biztosítja, hogy minden művelet végrehajtódik (§21.3.8). Az *ostream* lokálját a *getloc()* tagfüggvény meghívásával kapjuk meg, majd a lokálból a *use_facet* sablon függvénnyel (§D.3.1) kiszedjük a *num_put* jellemzőt. Miután ezt megtettük, meghívjuk a megfelelő *put()* függvényeket az igazi munka elvégzéséhez. A *put()* első két paraméterét könnyen megadhatjuk, hiszen az *ostream*-ből létrehozhatjuk az *ostream_buf* bejárót (§19.2.6), az adatfolyamot pedig automatikusan *ios_base* bázisosztályára alakíthatjuk (§21.2.1).

A *put()* kimeneti bejáró paraméterét adja vissza. A bejárót egy *basic_ostream*-ből szerzi meg, így annak típusa *ostreambuf_iterator*. Következésképpen a *failed()* (§19.2.6.1) rendelkezésünkre áll a hibaellenőrzéshez és lehetővé teszi számunkra, hogy megfelelően beállíthassuk az adatfolyam állapotát.

Nem használtuk a *has_facet* függvényt, mert a szabványos *facet*-ek (§D.4) garantáltan ott vannak minden lokálban. Ha ez a garancia nem áll fenn, *bad_cast* kivétel kiváltására kerül sor (§D.3.1).

A *put()* a *do_put* virtuális függvényt hívja meg. Következésképpen lehet, hogy felhasználói kód hajtódik végre és az *operator<<()*-nek fel kell készülnie arra, hogy a felülírt *do_put()* által kiváltott kivételt kezelje. Továbbá lehet, hogy a *num_put* nem létezik valamilyen karaktertípusra, így a *use_facet()* az *std::bad_cast* kivételt válthatja ki (§D.3.1). A beépített típusokra, mint amilyen a *double*, a << viselkedését a C++ szabvány írja le. Következésképpen nem az a kérdés, hogy a *handle_ioexception()* függvénynek mit kell csinálnia, hanem az, hogyan csinálja azt, amit a szabvány előír. Ha a *badbit* jelzőbit az *ostream* kivétel állapotára van állítva (§21.3.6), a kivétel továbbdobására kerül sor, más esetben a kivétel kezelése az adatfolyam állapotának beállítását és a végrehajtás folytatását jelenti. A *badbit* jelzőbitet mindkét esetben az adatfolyam állapotára kell állítani (§21.3.3):

```
template<class Ch, class Tr>
void handle_ioexception(std::basic_ostream<Ch,Tr>& s)   // a catch részből meghívva
{
    if (s.exceptions()&ios_base::badbit) {
        try {
            s.setstate(ios_base::badbit); } catch(...) { }
        throw;              // továbbdobás
    }
    s.setstate(ios_base::badbit);   // basic_ios::failure kivételt válthat ki
}
```

A *try* blokk azért szükséges, mert a *setstate() basic_ios::failure* kivételt válthat ki (§21.3.3, §21.3.6). Ha azonban a *badbit* a kivétel állapotra állított, az *operator<<()*-nek tovább kell dobnia azt a kivételt, amely a *handle_ioexception()* meghívását okozta (nem pedig egyszerűen *basic_ios::failure* kivételt kell kiváltania).

A <<-t úgy kell megvalósítani a beépített típusokra, például a *double*-ra, hogy közvetlenül az adatfolyam átmeneti tárába írjunk. Ha a <<-t felhasználói típusra írjuk meg, az ebből eredő nehézségeket úgy kerülhetjük el, hogy a felhasználói típusok kimenetét már meglévő típusok kimenetével fejezzük ki (§D.3.2).

D.4.2.3. Számok bevitele

A bemeneti adatfolyamok (*istream*) a *num_get* jellemzőre támaszkodnak az átmeneti tárból (§21.6.4) való olvasáshoz:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::num_get : public locale::facet {
public:
    typedef Ch char_type;
    typedef In iter_type;
```

```
    explicit num_get(size_t r = 0);

    // olvasás [b:e)-ból v-be, az s-beli formázási szabályok használatával;
    // hibajelzés az r beállításával:
    In get(In b, In e, ios_base& s, ios_base::iostate& r, bool& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned short& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned int& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, float& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, void*& v) const;

    static locale::id id;       // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~num_get();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

A *num_get* szerkezete alapvetően a *num_put*-éhoz (§D.4.2.2) hasonló. Mivel inkább olvas, mint ír, a *get()*-nek egy bejárópárra van szüksége, az olvasás célpontját meghatározó paraméter pedig egy referencia. Az *iostate* típusú *r* változó úgy van beállítva, hogy tükrözze az adatfolyam állapotát. Ha a kívánt típusú értéket nem lehet beolvasni, az *r*-ben a *failbit* beállítására kerül sor, ha elértük a bemenet végét, az *eofbit*-ére. A bemeneti műveletek az *r*-t arra használják, hogy eldöntsék, hogyan állítsák be az adatfolyam állapotát. Ha nem történt hiba, a beolvasott érték a *v*-n keresztül értékül adódik; egyébként a *v* változatlan marad.

Az *istream* készítője a következőket írhatja:

```
template<class Ch, class Tr>
istream& std::basic_istream<Ch,Tr>::operator>>(double& d)
{
    sentry guard(*this);        // lásd §21.3.8
    if (!guard) return *this;

    iostate state = 0;          // jó
    istreambuf_iterator<Ch> eos;
    double dd;

    try {
        use_facet< num_get<Ch> >(getloc()).get(*this,eos,*this,state,dd);
    }
```

```
    catch (...) {
        handle_ioexception(*this); // lásd §D.4.2.2
        return *this;
    }
    if (state==0 || state==eofbit) d = dd; // d értékének beállítása csak akkor,
                                           // ha a get() sikerrel járt
    setstate(state);
    return *this;
}
```

Az *istream* számára megengedett kivételeket hiba esetén a *setstate()* függvény váltja ki (§21.3.6).

Egy *numpunct* jellemzőt – mint amilyen a *my_numpunct* a §D.4.2 pontban – megadva nem szabványos elválasztó karaktereket használva is olvashatunk:

```
void f()
{
    cout << "Első stílus: "
    int i1;
    double d1;
    cin >> i1 >> d1; // beolvasás a szabványos "12345678" forma használatával

    locale loc(locale::classic(), new My_punct);
    cin.imbue(loc);
    cout << "Második stílus: "
    int i2;
    double d2;
    cin >> i1 >> d2; // beolvasás a "12 345 678" forma használatával
}
```

Ha igazán ritkán használt számformátumokat szeretnénk beolvasni, felül kell írnunk a *do_get()* függvényt. Megadhatunk például egy *num_get facet*-et, amely római számokat olvas be (*XXI* vagy *MM*, §D.6[15]).

D.4.3. Pénzértékek be- és kivitele

A pénzösszegek formázásának módja az „egyszerű" számok formázásához hasonlít (§D.4.2), az előbbinél azonban a kulturális eltérések jelentősége nagyobb. A negatív összegeket (veszteség, tartozás, mondjuk -1,25) például egyes helyeken pozitív számokként, zárójelben kell feltüntetni (1,25). Az is előfordulhat, hogy a negatív összegek felismerésének megkönnyítésére színeket kell használnunk.

Nincs szabványos „pénz típus". Ehelyett kifejezetten a „pénz" *facet*-eket kell használnunk az olyan számértékeknél, amelyekről tudjuk, hogy pénzösszegeket jelentenek:

```
class Money {      // egyszerű típus pénzösszegek tárolására
    long int amount;
public:
    Money(long int i) : amount(i) { }
    operator long int() const { return amount; }
};
// ...

void f(long int i)
{
    cout << "Érték= " << i << " Összeg= " << Money(i) << endl;
}
```

Ezen *facet*-ek feladata az, hogy jelentősen megkönnyítsék az olyan kimeneti műveletek megírását a *Money* típusra, melyek az összeget a helyi szokásoknak megfelelően írják ki (lásd §D.4.3.2-t). A kimenet a *cout* lokáljától függően változik:

```
Érték= 1234567 Összeg= $12345.67
Érték= 1234567 Összeg= 12345,67 DKK
Érték= -1234567 Összeg= $-12345.67
Érték= -1234567 Összeg= -$12345.67
Érték= -1234567 Összeg= (CHF12345,67)
```

A pénzértékek esetében rendszerint alapvető a legkisebb pénzegységre is kiterjedő pontosság. Következésképpen én azt a szokást követem, hogy inkább a „fillérek" (penny, øre, cent stb.) számát ábrázolom egész értékekkel, nem a „forintokét" (font, korona, dínár, euró stb.). Ezt a megoldást a *money_punct frac_digits()* függvénye támogatja (§D.4.3.1). A „tizedes-elválasztót" a *decimal_point()* adja meg.

A *money_base* jellemző által leírt formátumon alapuló bemenetet és kimenetet kezelő függvényeket a *money_get* és *money_put facet*-ek biztosítják.

A be- és kimeneti formátum szabályozására és a pénzértékek tárolására egy egyszerű *Money* típust használhatunk. Az első esetben a pénzösszegek tárolására használt (más) típusokat kiírás előtt a *Money* típusra alakítjuk, a beolvasást pedig szintén *Money* típusú változókba végezzük, mielőtt az értékeket más típusra alakítanánk. Kevesebb hibával jár, ha a pénzösszegeket következetesen a *Money* típusban tároljuk: így nem feledkezhetünk meg arról, hogy egy értéket *Money* típusra alakítsunk, mielőtt kiírnánk és nem kapunk bemeneti hibákat, ha a lokáltól függetlenül próbálunk pénzösszegeket beolvasni. Lehetséges azonban, hogy a *Money* típus bevezetése kivitelezhetetlen, ha a rendszer nincs felkészítve rá. Ilyen esetekben szükséges a *Money*-konverziókat alkalmazni az olvasó és író műveleteknél.

D.4.3.1. A pénzértékek formátuma

A pénzösszegek megjelenítését szabályozó *moneypunct* természetesen a közönséges számok formátumát megadó *numpunct facet*-re (§D.4.2.1) hasonlít:

```
class std::money_base {
public:
    enum part { none, space, symbol, sign, value };          // az elrendezés részei
    struct pattern { char field[4]; };                       // elrendezés
};

template <class Ch, bool International = false>
class std::moneypunct : public locale::facet, public money_base {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit moneypunct(size_t r = 0);

    Ch decimal_point() const;              // '.' a classic() lokálban
    Ch thousands_sep() const;              // ',' a classic() lokálban
    string grouping() const;     // "" a classic() lokálban, jelentése: nincs csoportosítás

    string_type curr_symbol() const;       // "$" a classic() lokálban
    string_type positive_sign() const;     // "" a classic() lokálban
    string_type negative_sign() const;     // "-" a classic() lokálban

    int frac_digits() const;   // számjegyek száma a tizedesvessző után; 2 a classic() lokálban
    pattern pos_format() const;      // { symbol, sign, none, value } a classic() lokálban
    pattern neg_format() const;      // { symbol, sign, none, value } a classic() lokálban

    static const bool intl = International;    // a "nemzetközi" pénzformátum használata

    static locale::id id; // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~moneypunct();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

A *moneypunct* szolgáltatásait elsősorban a *money_put* és *money_get* jellemzők készítőinek szánták (§D.4.3.2, §D.4.3.3).

A *decimal_point()*, *thousands_sep()*, és *grouping()* tagok úgy viselkednek, mint a velük egyenértékű függvények a *numpunct facet*-ben.

A *curr_symbol()*, *positive_sign()* és *negative_sign()* tagok rendre a valutajelet (*$*, *¥*, *FrF*, *DKr*), a pluszjelet és a mínuszjelet jelölő karakterláncot adják vissza. Ha az *International* sablonparaméter értéke *true* volt, az *intl* tag szintén *true* lesz és a valutajelek „nemzetközi" ábrázolása lesz használatos. A nemzetközi ábrázolás egy négy karakterből álló karakterlánc:

```
"USD "
"DKK "
"EUR "
```

Az utolsó karakter általában szóköz. A három betűs valuta-azonosítót az ISO-4217 szabvány írja le. Ha az *International* értéke *false*, „helyi" valutajelet – *$*, *£* vagy *¥* – lehet használni.

A *pos_format()* és *neg_format()* által visszaadott minta (*pattern*) négy részből (*part*) áll, amelyek a számérték, a valutajel, az előjel és az üreshely megjelenítésének sorrendjét adják meg. A leggyakoribb formátumok ezt a mintát követik:

```
+$ 123.45     // { sign, symbol, space, value }, ahol a positive_sign() visszatérési értéke "+"
$+123.45      // { symbol, sign, value, none }, ahol a positive_sign() visszatérési értéke "+"
$123.45       // { symbol, sign, value, none }, ahol a positive_sign() visszatérési értéke ""
$123.45-      // { symbol, value, sign, none }
-123.45 DKK   // { sign, value, space, symbol }
($123.45)     // { sign, symbol, value, none }, ahol a negative_sign() visszatérési értéke "()"
(123.45DKK)   // { sign, value, symbol, none }, ahol a negative_sign() visszatérési értéke "()"
```

A negatív számok zárójeles ábrázolását a *negative_sign()* függvény visszatérési értékeként a *()* karakterekből álló karakterláncot definiálva biztosítjuk. Az előjel-karakterlánc első karaktere oda kerül, ahol a *sign* (előjel) található a mintában, a maradék pedig a minta többi része után következik. Ezt a megoldást leggyakrabban ahhoz a szokásos pénzügyi jelöléshez használják, miszerint a negatív összegeket zárójelben tüntetik fel, de másra is fel lehet használni:

```
-$123.45         // { sign, symbol, value, none }, ahol a negative_sign() visszatérési értéke "-"
*$123.45 silly   // { sign, symbol, value, none }, ahol a negative_sign()
                 // visszatérési értéke "* silly"
```

Az előjel, érték és szimbólum (*sign*, *value*, *symbol*) értékek csak egyszer szerepelhetnek a mintában. A maradék érték *space* (üreshely) vagy *none* (semmi) lehet. Ahol *space* szerepel, oda legalább egy üreshely karakternek kell kerülnie, a *none* nulla vagy több üreshely karaktert jelent (kivéve ha a *none* a minta végén szerepel).

Ezek a szigorú szabályok néhány látszólag ésszerű mintát is megtiltanak:

```
pattern pat = { sign, value, none, none }; // hiba: a symbol nincs megadva
```

A *decimal_point()* helyét a *frac_digits()* függvény jelöli ki. A pénzösszegeket általában a legkisebb valutaegységgel ábrázolják (§D.4.3), ami jellemzően a fő egység századrésze (például cent és dollár), így a *frac_digits()* értéke általában *2*.

Következzen egy egyszerű formátum, *facet*-ként megadva:

```
class My_money_io : public moneypunct<char,true> {
public:
    explicit My_money_io(size_t r = 0) : moneypunct<char,true>(r) { }

    char_type do_decimal_point() const { return '.'; }
    char_type do_thousands_sep() const { return ','; }
    string do_grouping() const { return "\003\003\003"; }

    string_type do_curr_symbol() const { return "USD "; }
    string_type do_positive_sign() const { return ""; }
    string_type do_negative_sign() const { return "()"; }

    int do_frac_digits() const { return 2; }        // 2 számjegy a tizedesvessző után

    pattern do_pos_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }
    pattern do_neg_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }
};
```

Ezt a *facet*-et használjuk a *Money* alábbi be- és kimeneti műveleteiben is (§D.4.3.2 és §D.4.3.3).

A *moneypunct _byname* változata (§D.4, §D.4.1) is adott:

```
template <class Ch, bool Intl = false>
class std::moneypunct_byname : public moneypunct<Ch, Intl> { /* ... */ };
```

D.4.3.2. Pénzértékek kiírása

A *money_put* a *moneypunct* által meghatározott formátumban írja ki a pénzösszegeket. Pontosabban, a *money_put* olyan *put()* függvényeket nyújt, amelyek megfelelően formázott karakter-ábrázolásokat tesznek egy adatfolyam átmeneti tárába:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::money_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_put(size_t r = 0);

    // a "v" érték elhelyezése az átmeneti tár "b" pozíciójára:
    Out put(Out b, bool intl, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, bool intl, ios_base& s, Ch fill, const string_type& v) const;

    static locale::id id; // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~money_put();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

A *b*, *s*, *fill* és *v* paraméterek ugyanarra használatosak, mint a *num_put* jellemző *put()* függvényeiben (§D.4.3.2.2). Az *intl* paraméter jelzi, hogy a szabványos négykarakteres „nemzetközi" valutaszimbólum vagy „helyi" valutajel használatos-e (§D.4.3.1).

Ha adott a *money_put* jellemző, a *Money* osztály számára(§D.4.3) kimeneti műveletet írhatunk:

```
ostream& operator<<(ostream& s, Money m)
{
    ostream::sentry guard(s);        // lásd §21.3.8
    if (!guard) return s;

    try {
        const money_put<char>& f = use_facet< money_put<char> >(s.getloc());
        if (m==static_cast<long double>(m)) {      // m long double-ként ábrázolható
            if (f.put(s,true,s,s.fill(),m).failed()) s.setstate(ios_base::badbit);
        }
        else {
            ostringstream v;
            v << m;             // karakterlánc-ábrázolásra alakít
```

```
            if (f.put(s,true,s,s.fill(),v.str()).failed()) s.setstate(ios_base::badbit);
        }
    }
    catch (...) {
        handle_ioexception(s);      // lásd §D.4.2.2
    }
    return s;
}
```

Ha a *long double* nem elég pontos a pénzérték ábrázolásához, az értéket karakterlánccá alakítjuk és azt írjuk ki az ilyen paramétert váró *put()* függvénnyel.

D.4.3.3. Pénzértékek beolvasása

A *money_get* a *moneypunct* által meghatározott formátumnak megfelelően olvassa be a pénzösszegeket. Pontosabban, a *money_get* olyan *get()* függvényeket nyújt, amelyek a megfelelően formázott karakteres ábrázolást olvassák be egy adatfolyam átmeneti tárából:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::money_get : public locale::facet {
public:
    typedef Ch char_type;
    typedef In iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_get(size_t r = 0);

    // olvasás [b:e)-ból v-be, az s-beli formázási szabályok használatával;
    // hibajelzés az r beállításával:
    In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, string_type& v) const;

    static locale::id id;      // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~money_get();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

A *b*, *s*, *fill*, és *v* paraméterek ugyanarra használatosak, mint a *num_get* jellemző *get()* függvényeiben (§D.4.3.2.2). Az *intl* paraméter jelzi, hogy a szabványos négykarakteres „nemzetközi” valutaszimbólum vagy „helyi” valutajel használatos-e (§D.4.3.1).

Megfelelő *money_get* és *money_put* jellemzőpárral olyan formában adhatunk kimenetet, amelyet hiba és adatvesztés nélkül lehet visszaolvasni:

```
int main()
{
    Money m;
    while (cin>>m) cout << m << "\n";
}
```

Ez az egyszerű program kimenetét el kell, hogy fogadja bemenetként is. Sőt, ha a programot másodszor is lefuttatjuk az első futtatás eredményével, a kimenetnek meg kell egyeznie a program eredeti bemenetével.

A *Money* osztály számára a következő megfelelő bemeneti művelet lehet:

```
istream& operator>>(istream& s, Money& m)
{
    istream::sentry guard(s);          // lásd §21.3.8
    if (guard) try {
        ios_base::iostate state = 0;   // jó
        istreambuf_iterator<char> eos;
        string str;

        use_facet< money_get<char> >(s.getloc()).get(s,eos,true,state,str);

        if (state==0 || state==ios_base::eofbit) {    // csak akkor állít be értéket,
                                                      // ha a get() sikerrel járt
            long int i = strtol(str.c_str(),0,0);
            if (errno==ERANGE)
                state |= ios_base::failbit;
            else
                m = i; // csak akkor állítja be m értékét, ha az átalakítás long int-re sikerült
            s.setstate(state);
        }
    }
    catch (...) {
        handle_ioexception(s);       // lásd §D.4.2.2
    }
    return s;
}
```

D.4.4. Dátum és idő beolvasása és kiírása

Sajnos a C++ standard könyvtára nem nyújt megfelelő dátum típust, de a C standard könyvtárából alacsonyszintű eszközöket örökölt a dátumok és időtartományok kezeléséhez. Ezek a C-beli eszközök szolgálnak a C++ rendszerfüggetlen időkezelő eszközeinek alapjául.

A következőkben azt mutatjuk be, hogyan igazítható a dátum és idő megjelenítése a lokálhoz, továbbá példákat adunk arra, hogyan illeszthető be egy felhasználói típus (*Date*) az *iostream* (21. fejezet) és *locale* (§D.2) által nyújtott szerkezetbe. A *Date* típuson keresztül olyan eljárásokkal is megismerkedünk, amelyek segíthetik az időkezelést, ha nem áll rendelkezésünkre a *Date* típus.

D.4.4.1. Órák és időzítők

A legtöbb rendszer a legalacsonyabb szinten rendelkezik egy finom időzítővel. A standard könyvtár a *clock()* függvényt bocsátja rendelkezésünkre, amely megvalósítás-függő *clock_t* típusú értékkel tér vissza. A *clock()* eredménye a *CLOCK_PER_SEC* makró segítségével szabályozható. Ha nincs hozzáférésünk megbízható időmérő eszközhöz, akkor így mérhetjük meg egy ciklus idejét:

```
int main(int argc, char* argv[])          // §6.1.7
{
    int n = atoi(argv[1]);                 // §20.4.1

    clock_t t1 = clock();
    if (t1 == clock_t(-1)) {               // clock_t(-1) jelentése: "a clock() nem működik"
        cerr << "Sajnos nincs óránk.\n";
        exit(1);
    }

    for (int i = 0; i < n; i++) do_something();      // időzítő ciklus

    clock_t t2 = clock();
    if (t2 == clock_t(-1)) {
        cerr << "Túlcsordulás.\n";
        exit(2);
    }
    cout << "A do_something() " << n << " alkalommal való végrehajtása "
         << double(t2-t1)/CLOCKS_PER_SEC << " másodpercet vett igénybe"
         << " (mérési érzékenység: " << CLOCKS_PER_SEC << " per másodperc).\n";
}
```

Az osztás előtt végrehajtott *double(t2-t1)* átalakítás azért szükséges, mert a *clock_t* lehet, hogy egész típusú. Az, hogy a *clock()* mikor kezd futni, az adott nyelvi változattól függ; a függvény az egyes programrészek futási idejének mérésére való. A *clock()* által visszaadott *t1* és *t2* értékeket alapul véve a *double(t2-t1)/CLOCK_PER_SEC* a rendszer legjobb közelítése két hívás közt eltelt időre (másodpercben).

Ha az adott feldolgozóegységre nincs megadva a *clock()* függvény vagy ha az idő túl hosszú ahhoz, hogy mérhető legyen, a *clock()* a *clock_t(-1)* értéket adja vissza.

A *clock()* függvény a másodperc töredékétől legfeljebb néhány másodpercig terjedő időtartományok mérésére való. Például ha a *clock_t* egy 32 bites előjeles egész és a *CLOCK_PER_SEC* értéke 1 000 000, a *clock()* függvénnyel az időt 0-tól valamivel több mint 2000 másodpercig (körülbelül fél óráig) mérhetjük (a másodperc milliomod részében).

A programokról lényegi méréseket készíteni nem könnyű. A gépen futó többi program jelentősen befolyásolhatja az adott program futási idejét, nehéz megjósolni a gyorsítótárazás és az utasításcsövek hatásait és az algoritmusok nagymértékben függhetnek az adatoktól is. Ha meg akarjuk mérni egy program futási idejét, futtassuk többször és vegyük hibásnak azokat az eredményeket, amelyek jelentősen eltérnek a többitől.

A hosszabb időtartományok és a naptári idő kezelésére a standard könyvtár a *time_t* típust nyújtja, amely egy időpontot ábrázol; valamint a *tm* szerkezetet, amely az időpontokat „részeikre” bontja:

```
typedef megvalósítás_függő time_t;   // megvalósítás-függő aritmetikai típus (§4.1.1)
                                     // képes időtartomány ábrázolására,
                                     // általában 32 bites egész
struct tm {
    int tm_sec;     // a perc másodpercei [0,61]; a 60 és a 61 "szökő-másodpercek"
    int tm_min;     // az óra percei [0,59]
    int tm_hour;    // a nap órái [0,23]
    int tm_mday;    // a hónap napjai [1,31]
    int tm_mon;     // az év hónapjai [0,11]; a 0 jelentése "január" (figyelem: NEM [1:12])
    int tm_year;    // évek 1900 óta; a 0 jelentése "1900", a 102-é "2002"
    int tm_wday;    // napok vasárnap óta [0,6]; a 0 jelentése "vasárnap"
    int tm_yday;    // napok január 1 óta [0,365]; a 0 jelentése "január 1."
    int tm_isdst;   // nyári időszámítás jelző
};
```

A szabvány csak azt biztosítja, hogy a *tm* rendelkezik a fenti említett *int* típusú tagokkal, azt nem, hogy a tagok ilyen sorrendben szerepelnek vagy hogy nincsenek más tagok.

A *time_t* és *tm* típusok, valamint a hozzájuk kapcsolódó szolgáltatások a <*ctime*> és <*time.h*> fejállományokban találhatók:

```
clock_t clock();                        // órajelek száma a program indulása óta

time_t time(time_t* pt);                // érvényes naptári idő
double difftime(time_t t2, time_t t1);  // t2–t1 másodpercben

tm* localtime(const time_t* pt);        // *pt értéke helyi idő szerint
tm* gmtime(const time_t* pt);           // *pt értéke greenwichi középidő (GMT) szerint, vagy 0
                                        // (hivatalos neve: Coordinated Universal Time, UTC)

time_t mktime(tm* ptm);                 // *ptm értéke time_t formában, vagy time_t(-1)

char* asctime(const tm* ptm);           // *ptm egy C stílusú karakterlánccal ábrázolva
                                        // pl. "Sun Sep 16 01:03:52 1973\n"

char* ctime(const time_t* t) { return asctime(localtime(t)); }
```

Vigyázzunk: mind a *localtime()*, mind a *gmtime()* statikusan lefoglalt objektumra mutató **tm* típusú értéket ad vissza, vagyis mindkét függvény következő meghívása meg fogja változtatni az objektum értékét. Ezért rögtön használjuk fel a visszatérési értéket vagy másoljuk a *tm*-et olyan memóriahelyre, amit mi felügyelünk. Ugyanígy az *asctime()* függvény is statikusan lefoglalt karaktertömbre hivatkozó mutatót ad vissza.

A *tm* típus legalább tízezer évnyi dátumot (ez a legkisebb egész esetében körülbelül a [-32000,32000] tartomány) képes ábrázolni. A *time_t* azonban általában 32 bites (előjeles) *long int*. Ha másodperceket számolunk, a *time_t* nem sokkal több, mint 68 évet tud ábrázolni valamilyen „0. évtől” mindkét „irányban”. Az „alapév” rendszerint 1970, a hozzá tartozó alapidő pedig január 1., 0:00 GMT (UTC). Ha a *time_t* 32 bites előjeles egész, akkor 2038-ban futunk ki az „időből”, hacsak nem terjesztjük ki a *time_t*-t egy nagyobb egész típusra, ahogy azt már néhány rendszeren meg is tették.

A *time_t* alapvetően arra való, hogy a „jelenhez közeli időt” ábrázoljuk, ezért nem szabad elvárnunk, hogy képes legyen az [1902,2038] tartományon kívüli dátumokat is jelölni. Ami még ennél is rosszabb, nem minden időkezelő függvény kezeli azonos módon a negatív számokat. A „hordozhatóság” miatt az olyan értékeknek, amelyeket *tm*-ként és *time_t*-ként is ábrázolni kell, az [1920,2038] tartományba kell esniük. Azoknak, akik az 1970-től 2038-ig terjedő időkereten kívüli dátumokat szeretnének ábrázolni, további eljárásokat kell kidolgozniuk ahhoz, hogy ezt megtehessék.

Ennek egyik következménye, hogy az *mktime()* hibát eredményezhet. Ha az *mktime()* paraméterét nem lehet *time_t*-ként ábrázolni, a függvény a *time_t(-1)* hibajelzést adja vissza.

Ha van egy sokáig futó programunk, így mérhetjük le futási idejét:

```
int main(int argc, char* argv[])    // §6.1.7
{
    time_t t1 = time(0);
    do_a_lot(argc,argv);
    time_t t2 = time(0);
    double d = difftime(t2,t1);
    cout << "A do_a_lot() végrehajtása" << d << " másodpercig tartott.\n";
}
```

Ha a *time()* paramétere nem *0*, a függvény eredményeként kapott idő értékül adódik a függvény *time_t* típusra mutató paraméterének is. Ha a naptári idő nem elérhető (mondjuk valamilyen egyedi processzoron), akkor a *time_t(-1)* érték adódik vissza. A mai dátumot a következőképpen próbálhatjuk meg óvatosan megállapítani:

```
int main()
{
    time_t t;

    if (time(&t) == time_t(-1)) { // a time_t(–1) jelentése: "a time() nem működik"
        cerr << "Az idő nem állapítható meg.\n";
        exit(1);
    }

    tm* gt = gmtime(&t);
    cout << gt->tm_mon+1 << '/' << gt->tm_mday << '/' << 1900+gt->tm_year << endl;
}
```

D.4.4.2. Egy dátum osztály

Amint a §10.3 pontban említettük, nem valószínű, hogy egyetlen *Date* típus minden igényt ki tud szolgálni. A dátum felhasználása többféle megvalósítást igényel és a XIX. század előtt a naptárak nagyban függtek a történelem szeszélyeitől. Példaként azonban a §10.3-hoz hasonlóan készíthetünk egy *Date* típust, a *time_t* típust felhasználva:

```
class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
```

```
    class Bad_date {};

    Date(int dd, Month mm, int yy);
    Date();

    friend ostream& operator<<(ostream& s, const Date& d);
    // ...
private:
    time_t d;   // szabványos dátum- és időábrázolás
};

Date::Date(int dd, Month mm, int yy)
{
    tm x = { 0 };
    if (dd<0 || 31<dd) throw Bad_date();        // túlegyszerűsített: lásd §10.3.1
    x.tm_mday = dd;
    if (mm<jan || dec<mm) throw Bad_date();
    x.tm_mon = mm-1;          // a tm_mon 0 alapú
    x.tm_year = yy-1900;      // a tm_year 1900 alapú
    d = mktime(&x);
}

Date::Date()
{
    d = time(0);                  // alapértelmezett Date: a mai nap
    if (d == time_t(-1)) throw Bad_date();
}
```

A feladat az, hogy a << és >> operátorokat a *Date* típusra lokálfüggő módon valósítsuk meg.

D.4.4.3. Dátum és idő kiírása

A *num_put facet*-hez (§D.4.2) hasonlóan a *time_put* is *put()* függvényeket ad, hogy bejárókon keresztül az átmeneti tárakba írhassunk:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;

    explicit time_put(size_t r = 0);
```

```
    // írás az s adatfolyam átmeneti tárába b-n keresztül, az fmt formátum használatával
    Out put(Out b, ios_base& s, Ch fill, const tm* t,
                const Ch* fmt_b, const Ch* fmt_e) const;

    Out put(Out b, ios_base& s, Ch fill, const tm* t, char fmt, char mod = 0) const
       { return do_put(b,s,fill,t,fmt,mod); }

    static locale::id id; // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~time_put();

    virtual Out do_put(Out, ios_base&, Ch, const tm*, char, char) const;
};
```

A *put(b,s,fill,t,fmt_b,fmt_e)* a *t*-ben lévő dátumot *b*-n keresztül az *s* adatfolyam átmeneti tárába helyezi. A *fill* karakterek a kitöltéshez használatosak, ha az szükséges. A kimeneti formátumot a *printf()* formázóhoz hasonló *(fmt_b,fmt_e)* formázási karakterlánc (vagy formátumvezérlő) határozza meg. A tényleges kimenethez a *printf()*-hez hasonló (§21.8) formátum használatos, ami a következő különleges formátumvezérlőket tartalmazhatja:

%a	A hét napjának rövidített neve (pl. Szo)
%A	A hét napjának teljes neve (pl. Szombat)
%b	A hónap rövidített neve (pl. Feb)
%B	A hónap teljes neve (pl. Február)
%c	A dátum és idő (pl. Szo Feb 06 21:46:05 1999)
%d	A hónap napja [01,31] (pl. 06)
%H	Óra [00,23] (pl. 21)
%I	Óra [01,12] (pl. 09)
%j	Az év napja [001,366] (pl. 037)
%m	Az év hónapja [01,12] (pl. 02)
%M	Perc [00,59] (pl. 48)
%p	Délelőtt/délután (am./pm. – de./du.) jelzése a 12 órás órához (pl. PM)
%S	Másodperc [00,61] (pl. 48)
%U	Az év hete [00,53] vasárnappal kezdődően (pl. 05): az 1. hét az első vasárnappal kezdődik
%w	A hét napja [0,6]: a 0 jelenti a vasárnapot (pl. 6)
%W	Az év hete [00,53] hétfővel kezdődően (pl. 05): az 1. hét az első hétfővel kezdődik
%x	Dátum (pl. 02/06/99)
%X	Idő (pl.21:48:40)
%y	Év az évszázad nélkül [00,99] (pl. 99)
%Y	Év (pl.1999)
%Z	Az időzóna jelzése (pl. EST), ha az időzóna ismert

Ezeket az igen részletes formázó szabályokat a bővíthető I/O rendszer paramétereiként használhatjuk, de mint minden egyedi jelölést, ezeket is célszerűbb (és kényelmesebb) csak eredeti feladatuk ellátására alkalmazni.

A fenti formázó utasításokon túl a legtöbb C++-változat támogatja az olyan „módosítókat” (modifier), mint amilyen a mezőszélességet (§21.8) meghatározó egész szám (*%10X*). Az idő- és dátumformátumok módosítói nem képezik részét a C++ szabványnak, de néhány platformszabvány, mint a POSIX, igényelheti azokat. Következésképpen a módosítókat nehéz elkerülni, még akkor is, ha nem tökéletesen „hordozhatók”.

A *<ctime>* vagy *<time.h>* fejállományban található – az *sprintf()*-hez (§21.8) hasonló – *strftime()* függvény a kimenetet az idő- és dátumformázó utasítások felhasználásával hozza létre:

```
size_t strftime(char* s, size_t max, const char* format, const tm* tmp);
```

A függvény legfeljebb *max* karaktert tesz a **tmp*-ből és a *format*-ból a **s*-be, a *format* formázónak megfelelően:

```
int main()
{
    const int max = 20; // hanyag: abban bízik, hogy az strftime()
                        // soha sem eredményez 20-nál több karaktert
    char buf[max];
    time_t t = time(0);
    strftime(buf,max,"%A\n",localtime(&t));
    cout << buf;
}
```

A fenti program az alapértelmezett *classic()* lokálban (§D.2.3) egy szerdai napon *Wednesday*-t fog kiírni, dán lokálban *onsdag*-ot.

Azok a karakterek, melyek nem részei a meghatározott formátumnak, mint a példában az újsor karakter, egyszerűen bemásolódnak az első *(s)* paraméterbe.

Amikor a *put()* azonosít egy *f* formátumkaraktert (és egy nem kötelező *m* módosítót), meghívja a *do_put()* virtuális függvényt, hogy az végezze el a tényleges formázást: *do_put(b,s,fill,t,f,m)*.

A *put(b,s,fill,t,f,m)* hívás a *put()* egyszerűsített formája, ahol a formátumkarakter (*f*) és a módosító (*m*) pontosan meghatározott. Ezért a

```
const char fmt[] = "%10X";
put(b, s, fill, t, fmt, fmt+sizeof(fmt));
```

hívást a következő alakra lehet rövidíteni:

```
put(b, s, fill, t, 'X', 10);
```

Ha egy formátum többájtos karaktereket tartalmaz, annak az alapértelmezett állapotban (§D.4.6) kell kezdődnie és végződnie.

A *put()* függvényt felhasználhatjuk arra is, hogy a *Date* számára lokáltól függő kimeneti műveletet adjunk meg:

```
ostream& operator<<(ostream& s, const Date& d)
{
    ostream::sentry guard(s);   // lásd §21.3.8
    if (!guard) return s;

    tm* tmp = localtime(&d.d);
    try {
        if (use_facet< time_put<char> >(s.getloc()).put(s,s,s.fill(),tmp,'x').failed())
            s.setstate(ios_base::failbit);
    }
    catch (...) {
        handle_ioexception(s); // lásd §D.4.2.2
    }
    return s;
}
```

Mivel nincs szabványos *Date* típus, nem létezik alapértelmezett formátum a dátumok be- és kivitelére. Itt úgy határoztam meg a *%x* formátumot, hogy formázóként az *'x'* karaktert adtam át. Mivel a *get_time()* függvény (§D.4.4.4) alapértelmezett formátuma a *%x*, valószínűleg ez áll legközelebb a szabványhoz. A §D.4.4.5 pontban példát is láthatunk arra, hogyan használhatunk más formátumokat.

D.4.4.4. Dátum és idő beolvasása

Mint mindig, a bemenet „trükkösebb”, mint a kimenet. Amikor érték kiírására írunk kódot, gyakran különböző formátumok közül választhatunk. A beolvasásnál emellett a hibákkal is foglalkoznunk kell és néha számítanunk kell arra, hogy számos formátum lehetséges.

A dátum és idő beolvasását a *time_get* jellemzőn keresztül kezeljük. Az alapötlet az, hogy egy lokál *time_get facet*-je el tudja olvasni a *time_put* által létrehozott időket és dátumokat. Szabványos dátum- és időtípusok azonban nincsenek, ezért a programozó egy *locale*-t használhat arra, hogy a kimenetet különféle formátumoknak megfelelően hozza létre. A következő ábrázolásokat például mind létre lehet hozni egyetlen kimeneti utasítással, úgy, hogy a *time_put* jellemzőt (§D.4.4.5) különböző lokálokból használjuk:

January 15th 1999
Thursday 15th January 1999
15 Jan 1999AD
Thurs 15/1/99

A C++ szabvány arra biztat, hogy a *time_get* elkészítésénél úgy fogadjuk el a dátum- és időformátumokat, ahogy azt a POSIX és más szabványok előírják. A probléma az, hogy nehéz szabványosítani a dátum és idő beolvasásának bármilyen módját, amely egy adott kultúrában szabályos. Bölcsebb kísérletezni és megnézni, mit nyújt egy adott *locale* (§D.6[8]). Ha egy formátum nem elfogadott, a programozó másik, megfelelő *time_get facet*-et készíthet.

Az idő beolvasására használt szabványos *time_get* a *time_base* osztályból származik:

```
class std::time_base {
public:
    enum dateorder {
        no_order,     // nincs sorrend, esetleg több elem is van (pl. a hét napja)
        dmy,          // nap, hónap, év sorrend
        mdy,          // hónap, nap, év sorrend
        ymd,          // év, hónap, nap sorrend
        ydm           // év, nap, hónap sorrend
    };
};
```

Ezt a felsorolást arra használhatjuk, hogy egyszerűsítsük a dátumformátumok elemzését.

A *num_get* -hez hasonlóan a *time_get* is egy bemeneti bejárópáron keresztül fér hozzá átmeneti tárához:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class time_get : public locale::facet, public time_base {
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit time_get(size_t r = 0);

    dateorder date_order() const { return do_date_order(); }

    // olvasás [b:e)-ből d-be, az s-beli formázási szabályok használatával; hibajelzés az r
    // beállításával:
    In get_time(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_year(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

    In get_weekday(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_monthname(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

    static locale::id id;     // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~time_get();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

A *get_time()* a *do_get_time()* függvényt hívja meg. Az alapértelmezett *get_time()* a *%X* formátum (§D.4.4) felhasználásával úgy olvassa be az időt, ahogy a lokál *time_put()* függvénye létrehozza azt. Ugyanígy a *get_date()* függvény a *do_get_date()*-et hívja meg és az alapértelmezett *get_time()* a *%x* formátum felhasználásával (§D.4.4) a lokál *time_put()* függvényének eredménye szerint olvas.

A *Date* típusok legegyszerűbb bemeneti művelete valami ilyesmi:

```
istream& operator>>(istream& s, Date& d)
{
    istream::sentry guard(s);   // lásd §21.3.8
    if (!guard) return s;

    ios_base::iostate res = 0;
    tm x = { 0 };
    istreambuf_iterator<char,char_traits<char> > end;
```

```
    try {
        use_facet< time_get<char> >(s.getloc()).get_date(s,end,s,res,&x);
        if (res==0 || res==ios_base::eofbit)
            d = Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900);
        else
            s.setstate(res);
    }
    catch (...) {
        handle_ioexception(s); // lásd §D.4.2.2
    }
    return s;
}
```

A *get_date(s,end,s,res,&x)* hívás az *istream*-ről való két automatikus átalakításon alapul: az első *s* paraméter egy *istreambuf_iterator* létrehozására használatos, a harmadik paraméterként szereplő *s* pedig az *istream* bázisosztályára, az *ios_base*-re alakul át.

A fenti bemeneti művelet azon dátumok esetében működik helyesen, amelyek a *time_t* típussal ábrázolhatók.

Egy egyszerű próba a következő lenne:

```
int main()
try {
    Date today;
    cout << today << endl;              // írás %x formátummal
    Date d(12, Date::may, 1998);

    cout << d << endl;
    Date dd;
    while (cin >> dd) cout << dd << endl;  // az %x formátummal létrehozott
                                            // dátumok olvasása
}
catch (Date::Bad_date) {
    cout << "Rossz dátum: a program kilép.\n";
}
```

A *put_time _byname* változata (§D.4, §D.4.1) szintén adott:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put_byname : public time_put<Ch,Out> { /* ... */ };
```

D.4.4.5. Egy rugalmasabb Date osztály

Ha megpróbálnánk felhasználni a §D.4.4.2 pontból a *Date* osztályt a §D.4.4.3-ban és §D.4.4.4-ben szereplő be- és kimenettel, hamarosan korlátokba ütköznénk:

1. A *Date* csak olyan dátumokat tud kezelni, amelyek a *time_t* típussal ábrázolhatók: ez általában az [1970,2038] tartományt jelenti.
2. A *Date* csak a szabványos formátumban fogadja el a dátumokat – bármilyen legyen is az.
3. A *Date*-nél a bemeneti hibák jelzése elfogadhatatlan.
4. A *Date* csak *char* típusú adatfolyamokat támogat, nem tetszőleges karakter típusúakat.

Egy hasznosabb bemeneti művelet a dátumok szélesebb tartományát fogadná el, felismerne néhány gyakori formátumot és megbízhatóan jelentené a hibákat valamilyen jól használható formában. Ahhoz, hogy ezt elérjük, el kell térnünk a *time_t* ábrázolástól:

```
class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    struct Bad_date {
        const char* why;
        Bad_date(const char* p) : why(p) { }
    };

    Date(int dd, Month mm, int yy, int day_of_week = 0);
    Date();

    void make_tm(tm* t) const;          // a Date tm ábrázolását *t-be teszi
    time_t make_time_t() const;         // a Date time_t ábrázolásával tér vissza

    int year() const { return y; }
    Month month() const { return m; }
    int day() const { return d; }
    // ...
private:
    char d;
    Month m;
    int y;
};
```

Itt az egyszerűség kedvéért a *(d,m,y)* ábrázoláshoz (§10.2) tértünk vissza.

A konstruktort így adhatjuk meg:

```
Date::Date(int dd, Month mm, int yy, int day_of_week)
      : d(dd), m(mm), y(yy)
{
   if (d==0 && m==Month(0) && y==0) return;          // Date(0,0,0) a "null dátum"
   if (mm<jan || dec<mm) throw Bad_date("rossz a hónap");
   if (dd<1 || 31<dd)                                 // jócskán leegyszerűsítve; lásd §10.3.1
      throw Bad_date("rossz a hónap napja");
   if (day_of_week && day_in_week(yy,mm,dd)!=day_of_week)
      throw Bad_date("rossz a hét napja");
}
Date::Date() : d(0), m(0), y(0) { }                   // egy "null dátum"
```

A *day_in_week()* kiszámítása nem könnyű és nem kapcsolódik a lokálokhoz, ezért kihagytam. Ha szükségünk van rá, rendszerünkben biztosan megtalálhatjuk valahol.

Az összehasonlító műveletek mindig hasznosak az olyan típusok esetében, mint a *Date*:

```
bool operator==(const Date& x, const Date& y)
{
   return x.year()==y.year() && x.month()==y.month() && x.day()==y.day();
}
bool operator!=(const Date& x, const Date& y)
{
   return !(x==y);
}
```

Mivel eltértünk a szabványos *tm* és *time_t* formátumoktól, szükségünk van konverziós függvényekre is, amelyek együtt tudnak működni azokkal a programokkal, amelyek ezeket a típusokat várják el:

```
void Date::make_tm(tm* p) const // a dátum *p-be helyezése
{
   tm x = { 0 };
   *p = x;
   p->tm_year = y-1900;
   p->tm_mday = d;
   p->tm_mon = m-1;
}

time_t Date::make_time_t() const
{
   if (y<1970 || 2038<y)      // túlegyszerűsített
```

```
        throw Bad_date("A dátum a time_t tartományán kívül esik.");
    tm x;
    make_tm(&x);
    return mktime(&x);
}
```

D.4.4.6. Dátumformátumok megadása

A C++ nem határozza meg a dátumok kiírásának szabványos formátumát (a *%x* közelíti meg a legjobban; §D.4.4.3), de még ha létezne is ilyen, valószínűleg szeretnénk más formátumokat is használni. Ezt úgy tehetjük meg, hogy meghatározunk egy „alapértelmezett formátumot" és lehetőséget adunk annak módosítására:

```
class Date_format {
    static char fmt[];     // alapértelmezett formátum
    const char* curr;   // az éppen érvényes formátum
    const char* curr_end;
public:
    Date_format() : curr(fmt), curr_end(fmt+strlen(fmt)) { }

    const char* begin() const { return curr; }
    const char* end() const { return curr_end; }

    void set(const char* p, const char* q) { curr=p; curr_end=q; }
    void set(const char* p) { curr=p; curr_end=curr+strlen(p); }

    static const char* default_fmt() { return fmt; }
};

const char Date_format<char>::fmt[] = "%A, %B %d, %Y"; // pl. Friday, February 5, 1999

Date_format date_fmt;
```

Hogy az *strftime()* formátumot (§D.4.4.3) használhassuk, tartózkodtam attól, hogy a *Date_format* osztályt a használt karaktertípussal paraméterezzem. Ebből következik, hogy ez a megoldás csak olyan dátumjelölést enged meg, amelynek formátumát ki lehet fejezni a *char[]* típussal. Ezenkívül felhasználtam egy globális formátum-objektumot is (*date_fmt*), az alapértelmezett dátumformátum megadásához. Mivel a *date_fmt* értékét meg lehet változtatni, a *Date* formázásához rendelkezésünkre áll egy – meglehetősen nyers – módszer, hasonlóan ahhoz, ahogy a *global()* lokált (§D.2.3) lehet használni a formázásra.

Sokkal általánosabb megoldás, ha létrehozzuk a *Date_in* és *Date_out facet*-eket az adatfolyamból történő olvasás és írás vezérléséhez. Ezt a megközelítést a §D.4.4.7 pontban mutatjuk be. Ha adott a *Date_format*, a *Date::operator<<()*-t így írhatjuk meg:

```
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>& s, const Date& d)
// írás felhasználói formátummal
{
   typename basic_ostream<Ch,Tr>::sentry guard(s);      // lásd §21.3.8
   if (!guard) return s;

   tm t;
   d.make_tm(&t);
   try {
      const time_put<Ch>& f = use_facet< time_put<Ch> >(s.getloc());
      if (f.put(s,s,s.fill(),&t,date_fmt.begin(),date_fmt.end()).failed())
         s.setstate(ios_base::failbit);
   }
   catch (...) {
      handle_ioexception(s);            // lásd §D.4.2.2
   }
   return s;
}
```

A *has_facet* függvénnyel ellenőrizhetnénk, hogy az *s* lokálja rendelkezik-e a *time_put<Ch>* jellemzővel, itt azonban egyszerűbb úgy kezelni a problémát, hogy minden kivételt elkapunk, amit a *use_facet* kivált.

Íme egy egyszerű tesztprogram, ami a kimenet formátumát a *date_fmt*-n keresztül vezérli:

```
int main()
try {
   while (cin >> dd && dd != Date()) cout << dd << endl; // írás az alapértelmezett
                                                          // date_fmt használatával

   date_fmt.set("%Y/%m/%d");

   while (cin >> dd && dd != Date()) cout << dd << endl;  // írás az "%Y/%m/%d"
                                                           // használatával
}
catch (Date::Bad_date e) {
   cout << "Rossz dátum: " << e.why << endl;
}
```

D.4.4.7. Bemeneti facet-ek dátumokhoz

Mint mindig, a bemenet kezelése kicsit nehezebb, mint a kimeneté. Mégis, mivel a *get_date()* javított a felületen az alacsonyszintű bemenethez és mert a §D.4.4.4 pontban a *Date* típushoz megadott *operator>>()* nem fért hozzá közvetlen módon a *Date* ábrázolásához, az *operator>>()*-t változatlanul használhatjuk. Íme az *operator>>()* sablon függvényként megírt változata:

```
template<class Ch, class Tr>
istream<Ch,Tr>& operator>>(istream<Ch,Tr>& s, Date& d)
{
    typename istream<Ch,Tr>::sentry guard(s);
    if (guard) try {
        ios_base::iostate res = 0;
        tm x = { 0 };
        istreambuf_iterator<Ch,Tr> end;

        use_facet< time_get<Ch> >(s.getloc()).get_date(s,end,s,res,&x);

        if (res==0 || res==ios_base::eofbit)
            d = Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900,x.tm_wday);
        else
            s.setstate(res);
    }
    catch (...) {
        handle_ioexception(s);          // lásd §D.4.2.2
    }
    return s;
}
```

Ez a bemeneti művelet az *istream time_get* jellemzőjének *get_date()* függvényét hívja meg, így a bemenet új, rugalmasabb formáját adhatjuk meg, úgy, hogy származtatással egy új *facet*-et készítünk a *time_get*-ből:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class Date_in : public std::time_get<Ch,In> {
public:
    Date_in(size_t r = 0) : std::time_get<Ch>(r) { }
protected:
    In do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const;
private:
    enum Vtype { novalue, unknown, dayofweek, month };
    In getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const;
};
```

A *getval()* függvénynek egy évet, egy hónapot, a hónap napját és a hét egy napját kell beolvasnia (ez utóbbi nem kötelező), az eredményt pedig egy *tm* szerkezetbe kell tennie.

A hónapok és a hét napjainak nevei a lokáltól függnek, következésképpen nem említhetjük meg azokat közvetlenül a bemeneti függvényben. Ehelyett a hónapokat és napokat úgy ismerjük fel, hogy meghívjuk azokat a függvényeket, amelyeket a *time_get* ad erre a célra: a *get_monthname()* és *get_weekday()* függvényeket (§D.4.4.4).

Az év, a hónap napja és valószínűleg a hónap is egészként van ábrázolva. Sajnos egy szám nem jelzi, hogy egy hónap napját jelöli-e vagy valami egészen mást. A *7* például jelölhet júliust, egy hónap 7. napját vagy éppen a 2007-es évet is. A *time_get date_order()* függvényének valódi célja az, hogy feloldja az efféle többértelműségeket.

A *Date_in* értékeket olvas be, osztályozza azokat, majd a *date_order()* függvénnyel megnézi, a beírt adatok értelmesek-e (illetve „hogyan értelmesek"). A tényleges olvasást az adatfolyam átmeneti tárából, illetve a kezdeti osztályozást a privát *getval()* függvény végzi:

```
template<class Ch, class In>
In Date_in<Ch,In>::getval(In b,In e,ios_base& s,ios_base::iostate& r,int* v,Vtype* res)

const
// A Date részének olvasása: szám, a hét napja vagy hónap. Üreshelyek és írásjelek átlépése.
{
    const ctype<Ch>& ct = use_facet< ctype<Ch> >(s.getloc());        // a ctype leírását
                                                                     // lásd §D.4.5.-ben
    Ch c;

    *res = novalue;                 // nem talált értéket

    for (;;) {                      // üreshelyek és írásjelek átlépése
        if (b == e) return e;
        c = *b;
        if (!(ct.is(ctype_base::space,c) || ct.is(ctype_base::punct,c))) break;
        ++b;
    }
    if (ct.is(ctype_base::digit,c)) { // egész olvasása a numpunct figyelmen kívül hagyásával
        int i = 0;

        do {            // tetszőleges karakterkészlet számjegyének decimális értékké alakítása
            static char const digits[] = "0123456789";
            i = i*10 + find(digits,digits+10,ct.narrow(c,' '))-digits;
            c = *++b;
        } while (ct.is(ctype_base::digit,c));
        *v = i;
```

```
        *res = unknown;                  // egy egész, de nem tudjuk, mit ábrázol
        return b;
    }
    if (ct.is(ctype_base::alpha,c)) {         // hónap nevének vagy a hét napjának keresése
        basic_string<Ch> str;
        while (ct.is(ctype_base::alpha,c)) {           // karakterek olvasása karakterláncba
            str += c;
            if (++b == e) break;
            c = *b;
        }

        tm t;
        basic_stringstream<Ch> ss(str);
        typedef istreambuf_iterator<Ch> SI;          // bejárótípus az ss átmeneti tárához
        get_monthname(ss.rdbuf(),SI(),s,r,&t);       // olvasás memóriabeli adatfolyam
                                                     // átmeneti tárából

        if ((r&(ios_base::badbit|ios_base::failbit))==0) {
            *v= t.tm_mon;
            *res = month;
            return b;
        }

        r = 0;              // az adatfolyam-állapot törlése a második olvasás előtt
        get_weekday(ss.rdbuf(),SI(),s,r,&t);         // olvasás memóriabeli adatfolyam
                                                     // átmeneti tárából
        if ((r&(ios_base::badbit|ios_base::failbit))==0) {
            *v = t.tm_wday;
            *res = dayofweek;
            return b;
        }
    }
    r |= ios_base::failbit;
    return b;
}
```

Itt a „trükkös" rész a hét napjainak és a hónapoknak a megkülönböztetése. Mivel bemeneti bejárókon keresztül olvasunk, nem olvashatjuk be *[b,e)*-t kétszer, először egy hónapot, másodszor pedig egy napot keresve. Másfelől nem tudjuk megkülönböztetni a hónapokat sem a napoktól, ha egyszerre csak egy karaktert olvasunk be, mert csak a *get_monthname()* és a *get_weekday()* függvények tudják, hogy az adott lokálban a hónapok és a hét napjainak nevei mely karaktersorozatokból épülnek fel. Azt a megoldást választottam, hogy az alfabetikus karakterekből álló sorozatokat beolvastam egy karakterláncba, ebből egy *stringstream*-et készítettem, majd ismételten olvastam az adatfolyam *streambuf*-jából.

A hibajelző rendszer közvetlenül használja az olyan állapotbiteket, mint az *ios_base::badbit*. Ez azért szükséges, mert az adatfolyam állapotát kezelő kényelmesebb függvényeket, mint a *clear()* és a *setstate()*, a *basic_ios* osztály határozza meg, nem pedig annak bázisosztálya, az *ios_base* (§21.3.3). Ha szükséges, a >> operátor felhasználja a *get_date()* által jelentett hibákat, hogy visszaállítsa a bemeneti adatfolyam állapotát.

Ha a *getval()* adott, először beolvashatjuk az értékeket, majd később megnézhetjük, hogy értelmesek-e. A *dateorder()* szerepe döntő lehet:

```
template<class Ch, class In>
In Date_in<Ch,In>::do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const
// a "hét napja" (nem kötelező), melyet md, dmy, mdy, vagy ydm formátum követ
{
    int val[3];                    // nap, hónap, és év értékek számára
    Vtype res[3] = { novalue };   // értékosztályozáshoz

    for (int i=0; b!=e && i<3; ++i) {       // nap, hónap, év beolvasása
        b = getval(b,e,s,r,&val[i],&res[i]);
        if (r) return b;            // hoppá: hiba
        if (res[i]==novalue) {      // nem tudjuk befejezni a dátumot
            r |= ios_base::badbit;
            return b;
        }
        if (res[i]==dayofweek) {
            tmp->tm_wday = val[i];
            --i;    // hoppá: nem nap, hónap, vagy év
        }
    }

    time_base::dateorder order = dateorder();      // most megpróbálunk rájönni
                                                    // a beolvasott adat jelentésére

    if (res[0] == month) {         // mdy formátum vagy hiba
        // ...
    }
    else if (res[1] == month) {   // dmy vagy ymd vagy hiba
        tmp->tm_mon = val[1];
        switch (order)
        {
        case dmy:
            tmp->tm_mday = val[0];
            tmp->tm_year = val[2];
            break;
        case ymd:
            tmp->tm_year = val[0];
            tmp->tm_mday = val[1];
```

```
                break;
            default:
                r |= ios_base::badbit;
                return b;
            }
        }
        else if (res[2] == month) {           // ydm vagy hiba
            // ...
        }
        else {                                // megbízunk a dateorder-ben vagy hiba
            // ...
        }
        tmp->tm_year -= 1900;                 // az alapév igazítása a tm-hez
        return b;
    }
```

Azokat a kódrészleteket, amelyek nem járulnak hozzá a lokálok, dátumok és a bemenet kezelésének megértéséhez, kihagytam. A jobb és általánosabb dátumformátumok beolvasására alkalmas függvények megírását feladatként tűztem ki (§D.6[9-10]).

Íme egy egyszerű tesztprogram:

```
int main()
try {
    cin.imbue(loc(locale(), new Date_in));   // dátumok olvasása a Date_in használatával

    while (cin >> dd && dd != Date()) cout << dd << endl;
}
catch (Date::Bad_date e) {
    cout << "Rossz dátum: " << e.why << endl;
}
```

Vegyük észre, hogy a *do_get_date()* értelmetlen dátumokat is elfogad, mint amilyen a

```
Thursday October 7, 1998
```

és az

```
1999/Feb/31
```

Az év, hónap, nap és (nem kötelezően) a hét napja egységességének ellenőrzése a *Date* konstruktorában történik. A *Date* osztálynak kell tudnia, mi alkot helyes dátumot, a *Date_in* osztállyal pedig nem kell megosztania ezt a tudást.

Meg lehetne oldani azt is, hogy a *get_val()* vagy a *get_date()* függvények próbálják kitalálni a számértékek jelentését. Például a

12 May 1922

világos, hogy nem a 12-es év 1922. napja. Azaz „gyaníthatnánk”, hogy egy olyan számérték, ami nem lehet a megadott hónap napja, biztosan évet jelöl. Az ilyen feltevések bizonyos esetekben hasznosak lehetnek, de nem jó ötlet ezeket általánosabb környezetben használni. Például a

12 May 15

a 12., 15., 1912.,1915., 2012., vagy 2015. évben jelölhet egy dátumot. Néha jobb megközelítés, ha kiegészítjük a jelölést valamivel, ami segít az évek és napok megkülönböztetésében. Az angolban a *1st* és *15th* például biztosan egy hónap napjait jelölik. Ugyanígy a *751BC*-t és az *1453AD*-t (i.e.751 és i.sz.1453) nyilvánvalóan évként lehetne azonosítani.

D.4.5. Karakterek osztályozása

Amikor a bemenetről karaktereket olvasunk be, gyakran van szükség arra, hogy osztályozzuk azokat, hogy értelmezhessük, mit is olvastunk. Egy szám beolvasásához például egy bemeneti eljárásnak tudnia kell, hogy a számjegyeket mely karakterek jelölik. A §6.1.2 pontban is bemutattuk a szabványos karakterosztályozó függvények egy felhasználását a bemenet elemzésére.

Természetesen a karakterek osztályozása függ az éppen használatos ábécétől. Ezért rendelkezésünkre áll a *ctype* jellemző, amely az adott lokálban a karakterek osztályozását ábrázolja.

A karakterosztályokat a *mask* felsoroló típus határozza meg:

```
class std::ctype_base {
public:
    enum mask {              // a tényleges értékek megvalósításfüggőek
        space = 1,           // üreshely (a "C" lokálban ' ', '\n', '\t', ...)
        print = 1<<1,        // nyomtatható karakterek
        cntrl = 1<<2,        // vezérlőkarakterek
        upper = 1<<3,        // nagybetűk
        lower = 1<<4,        // kisbetűk
        alpha = 1<<5,        // alfabetikus karakterek
        digit = 1<<6,        // decimális számjegyek
        punct = 1<<7,        // írásjelek
```

```
        xdigit = 1<<8,          // hexadecimális számjegyek
        alnum=alpha|digit,      // alfanumerikus karakterek
        graph=alnum|punct
    };
};
```

A *mask* nem függ egyetlen karaktertípustól sem, következésképpen a felsorolás egy (nem sablon) bázisosztályban található.

Világos, hogy a *mask* a hagyományos C és C++-beli osztályozást tükrözi (§20.4.1). Eltérő karakterkészletek esetében azonban a különböző karakterértékek különböző osztályokba esnek. Az ASCII karakterkészletben például a *125* egész érték a *'}'* karaktert jelöli, ami az írásjelekhez (*punct*) tartozik, a dán karakterkészletben viszont az *'å'* magánhangzót, amit egy dán *locale*-ben az *alpha* osztályba kell sorolni.

Az osztályozást „maszknak" nevezik, mert a kis karakterkészletek hagyományos és hatékony osztályozása egy táblával történik, amelyben minden bejegyzés az osztály(oka)t ábrázoló biteket tárolja:

```
table['a'] == lower|alpha|xdigit
table['1'] == digit
table[' '] == space
```

Ha a fenti adott, a *table[c]&m* nem nulla, ha a *c* karakter az *m* osztályba tartozik („*c* egy *m*"), egyébként pedig *0*.

A *ctype* definíciója így fest:

```
template <class Ch>
class std::ctype : public locale::facet, public ctype_base {
public:
    typedef Ch char_type;
    explicit ctype(size_t r = 0);

    bool is(mask m, Ch c) const;    // "c" az "m" osztályba tartozik?

    // minden [b:e)-beli Ch osztályozása, az eredmény a v-be kerül
    const Ch* is(const Ch* b, const Ch* e, mask* v) const;

    const Ch* scan_is(mask m, const Ch* b, const Ch* e) const;    // m keresése
    const Ch* scan_not(mask m, const Ch* b, const Ch* e) const; // nem m keresése
```

```
    Ch toupper(Ch c) const;
    const Ch* toupper(Ch* b, const Ch* e) const;         // [b:e) átalakítása
    Ch tolower(Ch c) const;
    const Ch* tolower(Ch* b, const Ch* e) const;

    Ch widen(char c) const;
    const char* widen(const char* b, const char* e, Ch* b2) const;
    char narrow(Ch c, char def) const;
    const Ch* narrow(const Ch* b, const Ch* e, char def, char* b2) const;

    static locale::id id;        // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~ctype();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

Az *is(c,m)* hívással vizsgálhatjuk meg, hogy a *c* karakter az *m* osztályba tartozik-e:

```
int count_spaces(const string& s, const locale& loc)
{
    const ctype<char>& ct = use_facet< ctype<char> >(loc);
    int i = 0;
    for(string::const_iterator p = s.begin(); p != s.end(); ++p)
        if (ct.is(ctype_base::space,*p)) ++i;        // üreshely a ct meghatározása alapján
    return i;
}
```

Az *is()* függvénnyel azt is megnézhetjük, hogy egy karakter adott osztályok valamelyikébe tartozik-e:

```
ct.is(ctype_base::space|ctype_base::punct,c);    // c üreshely vagy írásjel ct alapján?
```

Az *is(b,e,v)* hívás a *[b,e)*-ben szereplő minden karakter osztályát eldönti és a megfelelő pozícióra helyezi azokat a *v* tömbben.

A *scan_is(m,b,e)* egy mutatót ad vissza a *[b,e)*-ben lévő első olyan karakterre, amely az *m* osztályba tartozik. Ha egyetlen karakter sem tartozik az *m* osztályba, a függvény az *e*-t adja vissza. Mint mindig a szabványos *facet*-ek esetében, a nyilvános tagfüggvény saját „*do_*" virtuális függvényét hívja meg. Egy egyszerű változat a következő lenne:

```
template <class Ch>
const Ch* std::ctype<Ch>::do_scan_is(mask m, const Ch* b, const Ch* e) const
{
    while (b!=e && !is(m,*b)) ++b;
    return b;
}
```

A *scan_not(m,b,e)* a *[b,e)*-ben lévő első olyan karakterre ad vissza mutatót, amely nem tartozik az *m* osztályba. Ha minden karakter az *m* osztályba tartozik, a függvény az *e*-t adja vissza.

A *toupper(c)* a *c* nagybetűs változatát adja vissza, ha az létezik a használatban lévő karakterkészletben; különben pedig magát a *c*-t.

A *toupper(b,e)* a *[b,e)* tartományban minden karaktert nagybetűsre alakít és az *e*-t adja vissza. Ennek egy egyszerű megvalósítása a következő lehet:

```
template <class Ch>
const Ch* std::ctype<Ch>::to_upper(Ch* b, const Ch* e)
{
    for ( ; b!=e; ++b) *b = toupper(*b);
    return e;
}
```

A *tolower()* függvények hasonlóak a *toupper()*-hez, azzal az eltéréssel, hogy kisbetűsre alakítanak.

A *widen(c)* a *c* karaktert a neki megfelelő *Ch* típusú értékre alakítja. Ha a *Ch* karakterkészlete több *c*-nek megfelelő karaktert nyújt, a szabvány „a legegyszerűbb ésszerű átalakítás" alkalmazását írja elő. Például a

```
wcout << use_facet< ctype<wchar_t> >(wcout.getloc()).widen('e');
```

az *e* karakternek a *wcout* lokálban lévő ésszerű megfelelőjét fogja kiírni.

A különböző karakterábrázolások, mint az ASCII és az EBCDIC közötti átalakítást szintén el lehet végezni a *widen()* függvénnyel. Például tegyük fel, hogy létezik az *ebcdic* lokál:

```
char EBCDIC_e = use_facet< ctype<char> >(ebcdic).widen('e');
```

A *widen(b,e,v)* hívás a *[b,e)* tartományban lévő minden karakter „szélesített" változatát helyezi a megfelelő pozícióra a *v* tömbben.

A *narrow(ch,def)* egy *char* típusú értéket ad a *Ch* típusú *ch* karakternek. Itt is a „legegyszerűbb ésszerű átalakítást” kell alkalmazni. Ha nem létezik megfelelő *char* típusú érték, a függvény a *def*-et adja vissza.

A *narrow(b,e,v)* hívás a *[b,e)* tartományban lévő minden karakter „leszűkített” változatát helyezi a megfelelő pozícióra a *v* tömbben.

Az általános elgondolás az, hogy a *narrow()* a nagyobb karakterkészletet alakítja egy kisebbre, míg a *widen()* ennek ellenkezőjét végzi. Egy kisebb karakterkészletben szereplő *c* karakter esetében elvárnánk a következőt:

```
c == narrow(widen(c), 0)    // nem garantált
```

Ez akkor igaz, ha a *c* által ábrázolt karakternek csak egyetlen jelölése van a „kisebbik” karakterkészletben, ez azonban nem biztos. Ha a *char* típussal ábrázolt karakterek nem képezik részhalmazát azoknak, amelyeket a nagyobb (*Ch*) karakterkészlet ábrázol, rendellenességekre és problémákra számíthatunk a karaktereket általánosságban kezelő programokban.

A nagyobb karakterkészletben lévő *ch* karakterre ugyanígy elvárnánk az alábbit:

```
widen(narrow(ch,def)) == ch || widen(narrow(ch,def)) == widen(def)   // nem garantált
```

Ez gyakran teljesül, de a nagyobb karakterkészletben több értékkel jelölt karakterek esetében nem mindig biztosítható, hogy csak egyszer legyenek ábrázolva a kisebbik készletben. Egy számjegynek (pl. a 7-nek) például gyakran több eltérő ábrázolása létezik egy nagy karakterkészletben. Ennek jellemzően az az oka, hogy egy nagy karakterkészletnek általában számos hagyományos karakterkészlet-részhalmaza van és a kisebb karakterkészletek karaktereiből másodpéldányok léteznek, hogy megkönnyítsék az átalakítást.

Az alap karakterkészletben (§C.3.3) szereplő minden karakterre biztosított a következő:

```
widen(narrow(ch_lit,0)) == ch_lit
```

Például:

```
widen(narrow('x')) == 'x'
```

A *narrow()* és *widen()* függvények mindenütt figyelembe veszik a karakterek osztályozását, ahol lehetséges. Például ha *is(alpha,c)* igaz, akkor az *is(alpha,narrow(c,'a'))* és az *is(alpha,widen(c))* is igaz lesz ott, ahol az *alpha* érvényes maszk a használatban lévő lokálra nézve.

A *ctype facet*-et – különösen a *narrow()* és *widen()* függvényeket – általában arra használjuk, hogy olyan kódot írjunk, amely tetszőleges karakterkészleten végzi a be- és kimenet, illetve a karakterláncok kezelését; azaz, hogy az ilyen kódot általánossá tegyük a karakterkészletek szempontjából. Ebből következik, hogy az *iostream*-megvalósítások alapvetően függnek ezektől az eszközöktől. A felhasználónak legtöbbször nem kell közvetlenül használnia a *ctype* jellemzőt, ha az *<iostream>*-re és a *<string>*-re támaszkodik.

A *ctype _byname* változata (§D.4, §D.4.1) szintén adott:

```
template <class Ch> class std::ctype_byname : public ctype<Ch> { /* ... */ };
```

D.4.5.1. Kényelmet szolgáló felületek

A *ctype facet*-et leggyakrabban arra használjuk, hogy lekérdezzük, hogy egy karakter egy adott osztályba tartozik-e. Következésképpen erre a célra a következő függvények adottak:

```
template <class Ch> bool isspace(Ch c, const locale& loc);
template <class Ch> bool isprint(Ch c, const locale& loc);
template <class Ch> bool iscntrl(Ch c, const locale& loc);
template <class Ch> bool isupper(Ch c, const locale& loc);
template <class Ch> bool islower(Ch c, const locale& loc);
template <class Ch> bool isalpha(Ch c, const locale& loc);
template <class Ch> bool isdigit(Ch c, const locale& loc);
template <class Ch> bool ispunct(Ch c, const locale& loc);
template <class Ch> bool isxdigit(Ch c, const locale& loc);
template <class Ch> bool isalnum(Ch c, const locale& loc);
template <class Ch> bool isgraph(Ch c, const locale& loc);
```

A fenti függvények a *use_facet*-et használják:

```
template <class Ch>
inline bool isspace(Ch c, const locale& loc)
{
    return use_facet< ctype<Ch> >(loc).is(space, c);
}
```

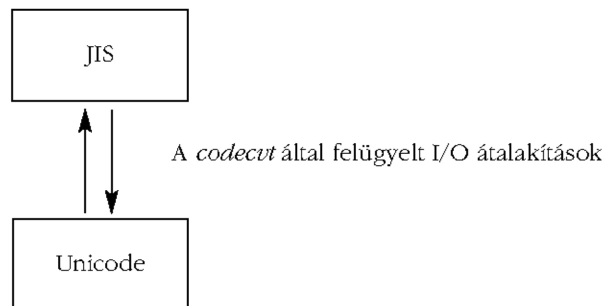
Ezen függvények egyparaméterű változatai (§20.4.2) az aktuális C globális lokálhoz, nem a C++ globális lokáljához, a *locale()*-hez készültek. Azokat a ritka eseteket kivéve, amikor a C és a C++ globális lokál különbözik (§D.2.3), ezeket a változatokat úgy tekinthetjük, mintha a kétparaméterű változatokat a *locale()*-re alkalmaztuk volna:

```
inline int isspace(int i)
{
   return isspace(i, locale());    // majdnem
}
```

D.4.6. Karakterkód-átalakítások

A fájlban tárolt karakterek ábrázolása néha különbözik ugyanazoknak a karaktereknek az elsődleges memóriában kívánatos ábrázolásától. A japán karaktereket például gyakran olyan fájlokban tárolják, amelyekben „léptetések" jelzik, hogy az adott karaktersorozat a négy legelterjedtebb karakterkészlet (kandzsi, katakana, hiragana és romadzsi) közül melyikhez tartozik. Ez kissé esetlen megoldás, mert minden bájt jelentése a „léptetési állapottól" függ, de memóriát takaríthat meg, mert csak egy kandzsi karakter ábrázolása igényel egy bájtnál többet. Az elsődleges memóriában ezek a karakterek könnyebben kezelhetők, ha többbájtos karakterként ábrázoltak, ahol minden karakter mérete megegyezik. Az ilyen karakterek (például a Unicode karakterek) általában széles karakterekben vannak elhelyezve (*wchar_t*; §4.3). Mindezek miatt a *codecvt facet* eszközöket nyújt a karakterek egyik ábrázolásról a másikra való átalakítására; miközben azokat olvassuk vagy írjuk:

Ábrázolás lemezen:



Ábrázolás az elsődleges memóriában:

Ez a kódátalakító rendszer elég általános ahhoz, hogy a karakterábrázolások között tetsző-leges átalakítást tegyen lehetővé, így a bemeneti adatfolyamok által használt *locale* beállításával olyan programot írhatunk, amely (*char*, *wchar_t* vagy más típusban tárolt) alkalmas belső karakterábrázolást használ és többféle karakter-adatfolyam ábrázolást elfogad bemenetként. A másik lehetőség az lenne, hogy magát a programot módosítjuk vagy a beolvasott és kiírt fájlok formátumát alakítjuk oda-vissza.

A *codecvt* a különböző karakterkészletek közötti átalakításra akkor ad lehetőséget, amikor a karaktert az adatfolyam átmeneti tára és egy külső tár között mozgatjuk:

```
class std::codecvt_base {
public:
    enum result { ok, partial, error, noconv };      // eredményjelző
};

template <class I, class E, class State>
class std::codecvt : public locale::facet, public codecvt_base {
public:
    typedef I intern_type;
    typedef E extern_type;
    typedef State state_type;

    explicit codecvt(size_t r = 0);

    result in(State&, const E* from, const E* from_end, const E*& from_next,  // olvasás
                I* to, I* to_end, I*& to_next) const;
    result out(State&, const I* from, const I* from_end, const I*& from_next, // írás
                E* to, E* to_end, E*& to_next) const;

    result unshift(State&, E* to, E* to_end, E*& to_next) const; // karaktersorozat vége

    int encoding() const throw();                    // alapvető kódolási tulajdonságok
    bool always_noconv() const throw();              // mehet az I/O kódátalakítás nélkül?

    int length(const State&, const E* from, const E* from_end, size_t max) const;
    int max_length() const throw();      // a lehetséges legnagyobb length()

    static locale::id id; // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~codecvt();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

A *codecvt facet*-et a *basic_filebuf* (§21.5) karakterek olvasásához és írásához használja és az adatfolyam lokáljából olvassa ki (§21.7.1).

A *State* sablonparaméter az éppen átalakított adatfolyam léptetési állapotának tárolására szolgáló típus. Ha specializációt készítünk belőle, a *State*-et a különböző átalakítások azonosítására is felhasználhatjuk. Ez utóbbi azért hasznos, mert így többfajta karakterkódoláshoz (karakterkészlethez) tartozó karaktereket lehet ugyanolyan típusú objektumokban tárolni:

```
class JISstate { /* .. */ };
p = new codecvt<wchar_t,char,mbstate_t>;     // szabványos char-ról széles karakterre
q = new codecvt<wchar_t,char,JISstate>;      // JIS-ről széles karakterre
```

A különböző *State* paraméterek nélkül a *facet* nem tudná megállapítani, melyik kódolást tételezze fel egy adott *char* adatfolyamra. A rendszer szabványos átalakítását a *char* és a *wchar_t* típusok között a *<cwchar>* vagy *<wchar.h>* fejállományban lévő *mbstate_t* típus írja le.

Származtatott osztályként, névvel azonosítva új *codecvt*-t is létrehozhatunk:

```
class JIScvt : public codecvt<wchar_t,char,mbstate_t> { /* ... */ };
```

Az *in(s,from,from_end,from_next,to,to_end,to_next)* hívás minden karaktert beolvas a *[from,from_next)* tartományból és megpróbálja átalakítani azokat. Ha egy karakterrel végzett, akkor azt új alakjában a *[to,to_end)* tartomány megfelelő helyére írja; ha átalakításra nem kerül sor, a függvény ezen a ponton megáll. A visszatéréskor az *in()* a *from_next*-ben az utolsó beolvasott karakter utáni pozíciót, a *to_next*-ben pedig az utolsó írott karakter utáni pozíciót tárolja. Az *in()* által visszaadott *result* érték jelzi, hogy a függvény mennyire tudta elvégezni a munkát:

ok:	A *[from,from_end)* tartományban minden karakter átalakításra került.
partial:	A *[from,from_end)* tartományban nem minden karakter került átalakításra.
error:	Az *out()* olyan karakterrel találkozott, amit nem tudott átalakítani.
noconv:	Nem volt szükség átalakításra.

A *partial* (részleges) átalakítás nem biztos, hogy hiba. Elképzelhető, hogy több karaktert kell beolvasni, mielőtt egy többájtos karakter teljes lesz és ki lehet írni, esetleg a kimeneti átmeneti tárat kell kiüríteni, hogy helyet csináljunk a további karaktereknek.

A bemeneti karaktersorozat állapotát az *in()* meghívásának kezdetén a *State* típusú *s* paraméter jelzi. Ennek akkor van jelentősége, ha a külső karakterábrázolás léptetési állapotokat használ. Jegyezzük meg, hogy az *s* (nem konstans) referencia paraméter: a hívás végén a bemeneti sorozat léptetésének állapotát tárolja, ami lehetővé teszi, hogy a programozó kezelhesse a részleges átalakításokat és hosszú sorozatokat is átalakíthasson az *in()* többszöri meghívásával. Az *out(s,from,from_end,from_next,to,to_end,to_next)* ugyanúgy alakítja át a *[from,from_end)*-et a belső ábrázolásról a külsőre, ahogy az *in()* a külső ábrázolást a belsőre.

A karakterfolyamoknak „semleges" („nem léptetett") állapotban kell kezdődniük és végződniük. Ez az állapot általában a *State()*. Az *unshift(s,to,to_end,to_next)* hívás megnézi az *s*-t és a *[to,to_end)*-be annyi karaktert tesz, amennyire szükség van ahhoz, hogy a karaktersorozatot visszaállítsa a nem léptetett állapotba. Az *unshift()* eredménye és a *to_next* felhasználási módja az *out()* függvényével azonos.

A *length(s,from,from_end,max)* azoknak a karaktereknek a számát adja vissza, amelyeket az *in()* át tudott alakítani a *[from,from_end)*-ből.

Az *encoding()* visszatérési értékei a következők lehetnek:

- *-1*, ha a külső karakterkészlet kódolása állapotot (például léptetett és nem léptetett karaktersorozatokat) használ,
- *0*, ha a kódolás az egyes karakterek ábrázolására különböző számú bájtot használ (például egy bájtban egy bitet annak a jelölésére, hogy egy vagy két bájtos-e a karakter ábrázolása.),
- *n*, ha a külső karakterkészlet minden karakterének ábrázolása pontosan *n* bájtos.

Az *always_noconv()* hívás *true*-t ad vissza, ha a belső és a külső karakterkészletek között nincs szükség átalakításra, különben pedig *false*-t. Világos, hogy az *always_noconv()==true* megnyitja a lehetőséget az adott nyelvi változat számára, hogy a lehető leghatékonyabb megvalósítást nyújtsa; azáltal, hogy egyáltalán nem hívja meg az átalakító függvényeket.

A *max_length()* hívás azt a legnagyobb értéket adja vissza, amit a *length()* visszaadhat egy adott érvényes paraméterhalmazra.

A bemenet nagybetűsre alakítása a legegyszerűbb kódátalakítás. Annyira egyszerű, amennyire csak egy *codecvt* lehet, mégis ellátja a feladatát:

```
class Cvt_to_upper : public codecvt<char,char,mbstate_t> {     // nagybetűssé alakítás
    explicit Cvt_to_upper(size_t r = 0) : codecvt(r) { }
```

```
protected:
    // külső ábrázolás olvasása, belső ábrázolás írása:
    result do_in(State& s, const char* from, const char* from_end, const char*& from_next,
                char* to, char* to_end, char*& to_next) const;

    // belső ábrázolás olvasása, külső ábrázolás írása:
    result do_out(State& s, const char* from, const char* from_end, const char*&
from_next,
                char* to, char* to_end, char*& to_next) const
    {
        return codecvt<char,char,mbstate_t>::do_out
                (s,from,from_end,from_next,to,to_end,to_next);
    }

    result do_unshift(State&, E* to, E* to_end, E*& to_next) const { return ok; }

    int do_encoding() const throw() { return 1; }
    bool do_always_noconv() const throw() { return false; }

    int do_length(const State&, const E* from, const E* from_end, size_t max) const;
    int do_max_length() const throw(); // a lehetséges legnagyobb length()
};

codecvt<char,char,mbstate_t>::result
Cvt_to_upper::do_in(State& s, const char* from, const char* from_end,
            const char*& from_next, char* to, char* to_end, char*& to_next) const
{
    // ... §D.6[16] ...
}

int main()      // egyszerű teszt
{
    locale ulocale(locale(), new Cvt_to_upper);

    cin.imbue(ulocale);

    while (cin>>ch) cout << ch;
}
```

A *codecvt _byname* változata (§D.4, §D.4.1) is adott:

```
template <class I, class E, class State>
class std::codecvt_byname : public codecvt<I,E,State> { /* ... */ };
```

D.4.7. Üzenetek

Természetesen minden felhasználó a saját anyanyelvét szereti használni, amikor „társalog" a programmal, a lokáltól függő üzenetek kifejezésére azonban nem tudunk szabványos eljárást biztosítani. A könyvtár csupán egyszerű eszközöket nyújt ahhoz, hogy lokálfüggő karakterlánc-halmazokat tárolhassunk, melyekből egyszerű üzeneteket (message) hozhatunk létre. A *messages* osztály alapjában véve egy igen egyszerű, csak olvasható adatbázist ír le:

```
class std::messages_base {
public:
    typedef int catalog;         // katalógus-azonosító típus
};

template <class Ch>
class std::messages : public locale::facet, public messages_base {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit messages(size_t r = 0);

    catalog open(const basic_string<char>& fn, const locale&) const;
    string_type get(catalog c, int set, int msgid, const string_type& d) const;
    void close(catalog c) const;

    static locale::id id;        // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
protected:
    ~messages();

    // virtuális "do_" függvények a nyilvános függvények számára (lásd §D.4.1)
};
```

Az *open(s,loc)* hívás megnyitja az *s* nevű „üzenetkatalógust" a *loc* lokálban. A katalógus egy megvalósítástól függően elrendezett karakterlánc-halmaz, amelyhez a *messages::get()* függvénnyel férhetünk hozzá. Ha az *s* nevű katalógust nem lehet megnyitni, az *open()* negatív értéket ad vissza. A katalógust meg kell nyitni a *get()* első használata előtt.

A *close(cat)* hívás bezárja a *cat* által azonosított katalógust és minden vele kapcsolatos erőforrást felszabadít.

A *get(cat,set,id,"foo")* a *(set,id)*-vel azonosított üzenetet keresi meg a *cat* katalógusban. Ha megtalálja a karakterláncot, akkor a *get()* ezzel tér vissza; ha nem, az alapértelmezett karakterlánccal (itt ez a *"foo"*).

Íme egy példa a *messages facet*-re, ahol az üzenetkatalógus „üzenetek" halmazából álló vektor, az „üzenet" pedig egy karakterlánc:

```
struct Set {
    vector<string> msgs;
};

struct Cat {
    vector<Set> sets;
};

class My_messages : public messages<char> {
    vector<Cat>& catalogs;
public:
    explicit My_messages(size_t = 0) : catalogs(*new vector<Cat>) { }

    catalog do_open(const string& s, const locale& loc) const;    // az s katalógus megnyitása
    string do_get(catalog c, int s, int m, const string&) const;   // az (s,m) üzenet
                                                                    // megszerzése c-ből
    void do_close(catalog cat) const
    {
        if (catalogs.size()<=cat) catalogs.erase(catalogs.begin()+cat);
    }
    ~My_messages() { delete &catalogs; }
};
```

A *messages* minden tagfüggvénye *const*, így a katalógus adatszerkezete (a *vector<Set>*) a *facet*-en kívül tárolódik.

Az üzeneteket egy katalógus, azon belül egy halmaz, majd a halmazon belül egy üzenet-karakterlánc megadásával jelölhetjük ki. Egy karakterláncot paraméterként adunk meg; ezt a függvény alapértelmezett visszatérési értékként használja, ha nem találja az üzenetet a katalógusban:

```
string My_messages::do_get(catalog cat, int set, int msg, const string& def) const
{
    if (catalogs.size()<=cat) return def;
    Cat& c = catalogs[cat];
    if (c.sets.size()<=set) return def;
    Set& s = c.sets[set];
    if (s.msgs.size()<=msg) return def;
    return s.msgs[msg];
}
```

A katalógus megnyitásakor a lemezről egy szöveges ábrázolást olvasunk egy *Cat* szerkezetbe. Itt olyan ábrázolást választottam, ami könnyen olvasható: a <<< és >>> jelek egy halmazt határolnak, az üzenetek pedig szövegsorok:

```
messages<char>::catalog My_messages::do_open(const string& n, const locale& loc) const
{
    string nn = n + locale().name();
    ifstream f(nn.c_str());
    if (!f) return -1;

    catalogs.push_back(Cat());            // a katalógus memóriába helyezése
    Cat& c = catalogs.back();
    string s;
    while (f>>s && s=="<<<") {            // halmaz olvasása
        c.sets.push_back(Set());
        Set& ss = c.sets.back();
        while (getline(f,s) && s != ">>>") ss.msgs.push_back(s);        // üzenetek beolvasása
    }
    return catalogs.size()-1;
}
```

Íme egy egyszerű felhasználás:

```
int main()
{
    if (!has_facet< My_messages >(locale())) {
        cerr << "A " << locale().name() << "lokálban nincs messages facet.\n";
        exit(1);
    }

    const messages<char>& m = use_facet< My_messages >(locale());
    extern string message_directory;    // itt tartom az üzeneteimet
    int cat = m.open(message_directory,locale());
    if (cat<0) {
        cerr << "Nincs katalógus.\n";
        exit(1);
    }

    cout << m.get(cat,0,0,"Megint téves!") << endl;
    cout << m.get(cat,1,2,"Megint téves!") << endl;
    cout << m.get(cat,1,3,"Megint téves!") << endl;
    cout << m.get(cat,3,0,"Megint téves!") << endl;
}
```

Ha a katalógus

```
<<<
helló
viszlát
>>>
<<<
igen
nem
talán
>>>
```

a program a következőt írja ki:

```
helló
talán
Megint téves!
Megint téves!
```

D.4.7.1. Más facet-ekben lévő üzenetek használata

Azon túl, hogy a felhasználókkal való kapcsolattartásra alkalmas lokálfüggő karakterláncokat tárolnak, az üzenetek arra is felhasználhatók, hogy más *facet*-ek számára tároljanak karakterláncokat. A *Season_io* jellemzőt (§D.3.2) például így is megírhattuk volna:

```
class Season_io : public locale::facet {
   const messages<char>& m;            // üzenetkönyvtár
   int cat;                            // üzenetkatalógus
public:
   class Missing_messages { };

   Season_io(int i = 0)
      : locale::facet(i),
        m(use_facet<Season_messages>(locale())),
        cat(m.open(message_directory,locale()))
   { if (cat<0) throw Missing_messages(); }

   ~Season_io() { }           // lehetővé teszi a Season_io objektumok felszámolását (§D.3)

   const string& to_str(Season x) const;          // x ábrázolása karakterlánccal

   bool from_str(const string& s, Season& x) const;        // az s-nek megfelelő
                                                           // Season objektumot x-be helyezi
```

```
    static locale::id id;           // facet-azonosító objektum (§D.2, §D.3, §D.3.1)
};

locale::id Season_io::id;           // az azonosító objektum meghatározása

const string& Season_io::to_str(Season x) const
{
    return m->get(cat, x, "Nincs ilyen évszak");
}

bool Season_io::from_str(const string& s, Season& x) const
{
    for (int i = Season::spring; i<=Season::winter; i++)
        if (m->get(cat, i, "Nincs ilyen évszak") == s) {
            x = Season(i);
            return true;
        }
    return false;
}
```

Ez az üzenetekre épülő megoldás abban különbözik az eredetitől (§D.3.2), hogy a *Season* karakterláncok halmazát egy adott lokálhoz megíró programozónak a karakterláncokat hozzá kell tudnia adni egy *messages* könyvtárhoz. Ezt az teheti meg könnyen, aki új lokált ad a végrehajtási környezethez. Mégis, mivel a *messages* csak olvasható felületet nyújt, az évszakok egy újabb halmazának megadása a rendszerprogramozók hatáskörén kívül esik.

A *messages _byname* változata (§D.4, §D.4.1) is adott:

```
template <class Ch>
class std::messages_byname : public messages<Ch> { /* ... */ };
```

D.5. Tanácsok

[1] Számítsunk rá, hogy a felhasználókkal közvetlenül társalgó programokat és rendszereket több országban is használni fogják. §D.1.
[2] Ne tételezzük fel, hogy mindenki ugyanazt a karakterkészletet használja, amit mi. §D.4.1.
[3] Ha a bemenet és kimenet függ a helyi sajátosságoktól, lokálokat használjuk és ne alkalmi kódot írjunk. §D.1.
[4] Kerüljük a lokálnevek beágyazását a programszövegbe. §D.2.1.

[5] Használjuk a lehető legkevesebb globális formázást. §D.2.3, §D.4.4.7.
[6] Részesítsük előnyben a lokálhoz igazodó karakterlánc-összehasonlításokat és -rendezéseket. §D.2.4, §D.4.1.
[7] A *facet*-ek legyenek nem módosíthatók. §D.2.2, §D.3.
[8] Csak kevés helyen változtassuk meg a lokált egy programban. §D.2.3.
[9] Hagyjuk, hogy a lokál szabályozza a *facet*-ek élettartamát. §D.3.
[10] Amikor lokálfüggő I/O függvényeket írunk, ne felejtsük el kezelni a felhasználói (felülírt) függvények okozta kivételeket. §D.4.2.2.
[11] A pénzértékek tárolására használjunk egy egyszerű *Money* típust. §D.4.3.
[12] Használjunk egyszerű felhasználói típusokat az olyan értékek tárolására, amelyek lokáltól függő I/O-t igényelnek (és ne a beépített típusok értékeit alakítsuk oda-vissza). §D.4.3.
[13] Ne higgyünk az időértékeknek, amíg nincs valamilyen jó módszerünk arra, hogy beleszámítsuk az összes lehetséges módosító tényezőt. §D.4.4.1.
[14] Legyünk tisztában a *time_t* típus korlátaival. §D.4.4.1, §D.4.4.5.
[15] Használjunk olyan dátumbeviteli eljárást, amely többféle formátumot is elfogad. §D.4.4.5.
[16] Részesítsük előnyben az olyan karakterosztályozó függvényeket, amelyekben a lokál kifejezetten megadott. §D.4.5, §D.4.5.1.

D.6. Gyakorlatok

1. (*2,5) Készítsük el a *Season_io*-t (§D.3.2) az amerikai angoltól eltérő nyelvre.
2. (*2) Hozzunk létre egy *Season_io* osztályt (§D.3.2), amelynek konstruktora nevek halmazát veszi paraméterként, hogy a különböző lokálokban lévő évszakok neveit ezen osztály objektumaiként lehessen ábrázolni.
3. (*3) Írjunk egy *collate<char>::compare* függvényt, amely ábécésorrendet állít fel. Lehetőleg olyan nyelvre készítsük el, mint a német, a francia vagy a magyar, mert ezek ábécéje az angolnál több betűt tartalmaz.
4. (*2) Írjunk egy programot, ami logikai értékeket számokként, angol szavakként és egy tetszőleges nyelv szavaiként olvas be és ír ki.
5. (*2,5) Határozzuk meg a *Time* típust a pontos idő ábrázolására. Határozzuk meg a *Date_and_time* típust is, a *Time* és valamilyen *Date* típus felhasználásával. Fejtsük ki ennek a megközelítésnek a §D.4.4. pontban szereplő *Date* típussal szembeni előnyeit és hátrányait. Írjuk meg a *Time* és a *Date_and_time* típusok lokálfüggő be- és kimeneti eljárásait.

6. (*2,5) Tervezzünk meg és készítsünk el egy „postai irányítószám" *facet*-et. Írjuk meg legalább két, eltérő címzési szokásokat használó országra.
7. (*2,5) Készítsünk egy „telefonszám" *facet*-et. Írjuk meg legalább két, eltérő telefonszám-formát (például (973) 360-8000 és 1223 343000) használó országra.
8. (*2,5) Próbáljuk meg kitalálni, milyen be- és kimeneti formátumokat használ C++-változatunk a dátumokhoz.
9. (*2,5) Írjunk olyan *get_time()* függvényt, amely megpróbálja kitalálni a többértelmű dátumok – például a *12/05/1995* – jelentését, de elutasít minden vagy majdnem minden hibát. Határozzuk meg pontosan, melyik „találgatást" fogadjuk el és gondoljuk át a hibák valószínűségét.
10. (*2) Írjunk olyan *get_time()* függvényt, ami többfajta bemeneti formátumot fogad el, mint a §D.4.4.5 pontban szereplő változat.
11. (*2) Készítsünk listát a rendszerünk által támogatott lokálokról.
12. (*2,5) Találjuk meg, rendszerünk hol tárolja a nevesített lokálokat. Ha van hozzáférésünk a rendszer azon részéhez, ahol a lokálok tárolódnak, készítsünk egy új, névvel rendelkező *locale*-t. Legyünk óvatosak, nehogy tönkretegyük a már létező *locale*-eket.
13. (*2) Hasonlítsuk össze a *Season_io* két megvalósítását (§D.3.2 és §D.4.7.1).
14. (*2) Írjuk meg, és teszteljük le a *Date_out facet*-et, ami *Date*-eket ír ki a konstruktorának megadott paraméter által meghatározott formában. Fejtsük ki ennek a megközelítésnek az előnyeit és hátrányait a *date_fmt* által nyújtott globális adatformátummal (§D.4.4.6) szemben.
15. (*2,5) Írjuk meg a római számok (például *XI* és *MDCLII*) be- és kimeneti eljárásait.
16. (*2,5) Készítsük el és ellenőrizzük a *Cvt_to_upper*-t (§D.4.6).
17. (*2,5) Állapítsuk meg a következők átlagos költségét a *clock()* függvény felhasználásával: (1) függvényhívás, (2) virtuális függvényhívás, (3) egy *char* beolvasása, (4) egy 1 számjegyű *int* beolvasása, (5) egy 5 számjegyű *int* beolvasása (6) egy 5 számjegyű *double*, (7) egy 1 karakteres *string*, (8) egy 5 karakteres *string*, és (9) egy 40 karakteres *string* beolvasása.
18. (*6,5) Tanuljunk meg egy másik természetes nyelvet.

E

Kivételbiztosság a standard könyvtárban

„Minden úgy működik, ahogy
elvárjuk – feltéve, hogy
elvárásaink helyesek."
(Hyman Rosen)

Kivételbiztosság • Kivételbiztos eljárások • Erőforrások ábrázolása • Értékadás • *push_back()* • Konstruktorok és invariánsok • A szabványos tárolók szolgáltatásai • Elemek beillesztése és eltávolítása • Garanciák és kompromisszumok • *swap()* • A kezdeti értékadás és a bejárók • Hivatkozás elemekre • Predikátumok • Karakterláncok, adatfolyamok, algoritmusok, a *valarray* és a *complex* • A C standard könyvtára • Javaslatok a könyvtárak felhasználói számára • Tanácsok • Gyakorlatok

E.1. Bevezetés

A standard könyvtár függvényei gyakran olyan függvényeket hívnak meg, melyeket a felhasználó ad meg akár függvény-, akár sablonparaméterek formájában. Természetesen ezek a felhasználói eljárások néha kivételeket váltanak ki. Egyes függvények (például a memóriafoglaló eljárások) önmagukban is képesek kivétel kiváltására:

```
void f(vector<X>& v, const X& g)
{
    v[2] = g;                   // X típus értékadása kivételt válthat ki
    v.push_back(g);             // vector<X> memóriafoglalója kivételt válthat ki
    sort(v.begin(), v.end());   // X "kisebb mint" operátora kivételt válthat ki
    vector<X> u = v;            // X másoló konstruktora kivételt válthat ki
    // ...

    // u itt semmisül meg: biztosítanunk kell, hogy X destruktora helyesen működjön
}
```

Mi történik, ha az értékadás kivételt vált ki, miközben a *g* értékét próbáljuk lemásolni? A *v* ezután egy érvénytelen elemet fog tartalmazni? Mi történik, ha az a konstruktor, amit a *v.push_back()* használ a *g* lemásolásához, *std::bad_alloc* kivételt vált ki? Megváltozik az elemek száma? Bekerülhet érvénytelen elem a tárolóba? Mi történik, ha az *X* osztály „kisebb, mint" operátora eredményez kivételt a rendezés közben? Az elemek ezután részben rendezettekké válnak? Elképzelhető, hogy a rendező eljárás eltávolít egy elemet a tárolóból és egy hiba miatt nem teszi vissza?

A fenti példában szereplő összes kivétel-lehetőség megtalálása az §E.8.[1] feladat célja, ezen függeléké az, hogy elmagyarázzuk, hogyan illik viselkednie a fenti programnak az összes megfelelően meghatározott *X* típus esetében (beleértve azt az esetet is, amikor az *X* kivételeket válthat ki). Természetesen a függelék nagy részében azt fogjuk tisztázni, mit is nevezünk helyes viselkedésnek, illetve megfelelően meghatározottnak a kivételekkel kapcsolatban.

A függelékben a következőkkel foglalkozunk:

1. Megvizsgáljuk, hogyan adhat meg a felhasználó olyan típusokat, amelyek megfelelnek a standard könyvtár elvárásainak.
2. Rögzítjük, milyen garanciákat biztosít a standard könyvtár.
3. Megnézzük, milyen elvárásokat állít a standard könyvtár a felhasználó által megadott programrészekkel szemben.
4. Bemutatunk néhány hatékony módszert, mellyel kivételbiztos és hatékony tárolókat készíthetünk.
5. Megemlítjük a kivételbiztos programozás néhány általános szabályát.

A kivételbiztosság vizsgálata értelemszerűen a legrosszabb esetben tapasztalható viselkedéssel foglalkozik. Hol jelentheti egy kivétel a legtöbb problémát? Hogyan tudja a standard könyvtár megvédeni magát és a felhasználót a lehetséges problémáktól? Hogyan segíthet a felhasználó a problémák elkerülésében? Ne hagyjuk, hogy az itt bemutatott kivétel-

kezelési módszerek elfeledtessék velünk, hogy a kivételek kiváltása a legjobb módszer a hibák jelzésére (§14.1, §14.9). Az elveket, a módszereket és a standard könyvtár szolgáltatásait a következő rendszerben tárgyaljuk:

§E.2. Ebben a pontban a kivételbiztosság fogalmát vizsgáljuk meg.
§E.3. Itt módszereket mutatunk arra, hogyan írhatunk hatékony, kivételbiztos tárolókat és műveleteket.
§E.4. Ebben a részben körvonalazzuk a standard könyvtár tárolói és műveletei által biztosított garanciákat.
§E.5. Itt összefoglaljuk a kivételbiztosság problémáját a standard könyvtár nem tárolókkal foglalkozó részében.
§E.6. Végül újból áttekintjük a kivételbiztosság kérdését a standard könyvtár felhasználóinak szemszögéből.

Szokás szerint a standard könyvtár példákat mutat azokra a problématípusokra, amelyekre egy alkalmazás fejlesztésekor oda kell figyelnünk. A standard könyvtárban a kivételbiztosság megvalósítására alkalmazott eljárások számos más területen is felhasználhatók.

E.2. Kivételbiztosság

Egy objektumon végrehajtott műveletet akkor nevezünk *kivételbiztosnak*, ha a művelet az objektumot akkor is érvényes állapotban hagyja, ha kivétel kiváltásával ért véget. Ez az érvényes állapot lehet egy hibaállapot is, amit esetleg csak teljes újbóli létrehozással lehet megszüntetni, de mindenképpen pontosan meghatározottnak kell lennie, hogy az objektumhoz logikus hibakezelő eljárást adhassunk meg. A kivételkezelő például törölheti az objektumot, kijavíthatja azt, megpróbálkozhat a művelet egy másik változatával vagy megpróbálhat egyszerűen továbbhaladni a hiba figyelmen kívül hagyásával.

Más szavakkal, az objektum rendelkezni fog egy *invariánssal* (állapotbiztosító, nem változó állapot §24.3.7.1), melyet konstruktorai állítanak elő, és minden további műveletnek meg tartania az általa leírt állapotot, még akkor is, ha kivételek következtek be. A végső rendrakást a destruktorok végzik el. Minden műveletnek figyelnie kell arra, hogy a kivételek kiváltása előtt visszaállítsák az invariánst, hogy az objektum érvényes állapotban lépjen ki a műveletből. Természetesen nagy valószínűséggel ez az érvényes állapot nem az az állapot lesz, amire az alkalmazásnak éppen szüksége lenne. Egy karakterlánc egy kivétel következtében például üressé válhat, egy tároló pedig rendezetlen maradhat. Tehát a „kijaví-

tás" csak azt jelenti, hogy az objektumnak olyan értéket adunk, ami elfogadhatóbb a program számára, mint az, amit a félbeszakadt művelet benne hagyott. A standard könyvtár szempontjából a legérdekesebb objektumok a tárolók.

Az alábbiakban azt vizsgáljuk meg, milyen feltételek mellett nevezhetjük a szabványos tárolókat feldolgozó műveleteket kivételbiztosnak. Elvileg mindössze két egyszerű megközelítés közül választhatunk:

1. „*Nincs biztosítás*": Ha kivétel lép fel, a művelet által használt valamennyi tárolót „valószínűleg sérültnek" tételezzük fel.
2. „*Erős biztosítás*": Ha kivétel lép fel, minden tároló pontosan abba az állapotba kerül vissza, mint amiben a standard könyvtár műveletének megkezdése előtt volt.

Sajnos ez a két megoldás annyira egyszerű, hogy igazán nem is alkalmazható. Az első azért elfogadhatatlan, mert ha ezen megoldás mellett egy tárolóművelet kivételt vált ki, a tárolót többet nem érhetjük el, sőt még nem is törölhetjük anélkül, hogy futási idejű hibáktól kéne rettegnünk. A második lehetőség azért nem használható, mert ekkor a visszaállítás megvalósításának költségei a standard könyvtár minden műveletében jelentkeznének.

A probléma megoldására a C++ standard könyvtára kivételbiztosítási szinteket kínál, melyek a helyes program készítésének terheit megosztják a standard könyvtár megvalósítói és felhasználói között:

3a *Alapbiztosítás* az összes műveletre: A standard könyvtár alapvető invariánsai mindenképpen fennmaradnak és semmilyen erőforrás (például memóriatartomány) nem maradhat lefoglalt.

3b *Erős biztosítás* a legfontosabb műveletekhez: Az alapbiztosításon túl, a művelet vagy sikeresen végrehajtásra kerül, vagy semmilyen változást nem eredményez. Ilyen szintű biztosítás csak a könyvtár legfontosabb műveleteit illeti meg, például a *push_back()*,a *list* osztályban az egyelemű *insert()*, illetve az *uninitialized_copy()* függvényeket (§E.3.1, §E.4.1).

3c *„Nncs kivétel" biztosítás* bizonyos műveletekre Az alapbiztosítás mellett néhány művelet egyáltalán nem válthat ki kivételeket. Ez a típusú biztosítás csak néhány egyszerű műveletre valósítható meg, például a *swap()* vagy a *pop_back()* függvényre (§E.4.1).

Az alapbiztosítás és az erős biztosítás is feltételezi, hogy a felhasználó által megadott műveletek (például az értékadások és a *swap()* függvények) nem hagyják a tároló elemeit érvénytelen állapotban és minden általuk lefoglalt erőforrást felszabadítanak. Ezenkívül a destruktoroknak nem szabad kivételt kiváltaniuk. Például vizsgáljuk meg az alábbi osztályokat (§25.7):

```
template<class T> class Safe {
   T* p;                  // p egy T típusú, new-val lefoglalt objektumra mutat
public:
   Safe() : p(new T) { }
   ~Safe() { delete p; }
   Safe& operator=(const Safe& a) { *p = *a.p; return *this; }
   // ...
};

template<class T> class Unsafe {          // hanyag és veszélyes kód
   T* p;      // egy T-re mutat
public:
   Unsafe(T* pp) : p(pp) { }
   ~Unsafe() { if (!p->destructible()) throw E(); delete p; }

   Unsafe& operator=(const Unsafe& a)
   {
      p->~T();                      // a régi érték törlése (§10.4.11)
      new(p) T(a.p);                // T létrehozása *p-ben a.p alapján (§10.4.11)
      return *this;
   }
   // ...
};

void f(vector< Safe<Some_type> >&vg, vector< Unsafe<Some_type> >&vb)
{
   vg.at(1) = Safe<Some_type>();
   vb.at(1) = Unsafe<Some_type>(new Some_type);
   // ...
}
```

Ebben a példában a *Safe* osztály létrehozása csak akkor sikeres, ha a *T* osztály is sikeresen létrejön. Egy *T* objektum létrehozása meghiúsulhat azért, mert nem sikerül a memóriafoglalás (ilyenkor *std::bad_alloc* kivétel jelentkezik), vagy azért, mert a *T* konstruktora bármilyen okból kivételt vált ki. Ettől függetlenül, egy sikeresen létrehozott *Safe* esetében a *p* egy hibátlan *T* objektumra mutat, míg ha a konstruktor sikertelen, sem *T*, sem *Safe* objektum nem jön létre. Ugyanígy kivételt válthat ki a *T* értékadó művelete is; ekkor a *Safe* értékadó művelete (a C++ működésének megfelelően) továbbdobja a kivételt. Mindez nem jelent problémát mindaddig, amíg a *T* értékadó operátora saját operandusát megfelelő állapotban hagyja. Tehát a *Safe* követelményeinknek megfelelően működik, így a standard könyvtár műveletei *Safe* objektumokra alkalmazva logikus és megfelelően meghatározott eredményt fognak adni.

Ezzel szemben az *Unsafe()* konstruktort figyelmetlenül írtuk meg (pontosabban figyelmesen úgy írtuk meg, hogy a helytelen formát bemutassa). Egy *Unsafe* objektum létrehozása sohasem fog meghiúsulni. Ehelyett az objektumot használó műveletekre (például az értékadásra és a megsemmisítésre) bízzuk, hogy törődjenek saját belátásuk szerint az összes lehetséges problémával. Az értékadás meghiúsulhat úgy, hogy a *T* másoló konstruktora kivált egy kivételt. Ilyenkor a *T* típusú objektumot nem meghatározott állapotban hagyjuk, hiszen a **p* régi értékét töröltük, de nem helyettesítettük érvényes értékkel. Az ilyen működés következményei általában megjósolhatatlanok. Az *Unsafe* destruktora tartalmaz egy reménytelen próbálkozást az érvénytelen megsemmisítés elkerülésére, egy kivétel meghívása egy másik kivétel kezelése közben azonban a *terminate()* (§14.7) meghívását eredményezi, míg a standard könyvtár elvárja, hogy a destruktor szabályosan visszatérjen az objektum törlése után. A standard könyvtár nem készült fel és nem is tud felkészülni arra, hogy garanciákat adjon olyan felhasználói objektumokra, melyek nem megfelelően működnek. A kivételkezelés szempontjából a *Safe* és az *Unsafe* abban különbözik, hogy a *Safe* a konstruktort használja az invariáns (§24.3.7.1) létrehozásához, ami lehetővé teszi, hogy műveleteit egyszerűen és biztonságosan valósítsuk meg. Ha az állapotbiztosító feltétel nem elégíthető ki, kivételt kapunk, mielőtt még az érvénytelen objektum létrejönne. Ugyanakkor az *Unsafe* nem rendelkezik értelmes invariánssal és a különálló műveletek mindenféle kivételeket váltanak ki, anélkül, hogy egy „központi" hibakezelő eljárás rendelkezésükre állna. Természetesen ez gyakran vezet a standard könyvtár (ésszerű) feltételezéseinek megsértéséhez. Az *Unsafe* esetében például érvénytelen elemek maradhatnak egy tárolóban, miután a *T::operator=()* egy kivételt váltott ki, és a destruktor is eredményezhet kivételeket.

A standard könyvtár biztosításainak viszonya a rossz viselkedésű felhasználói műveletekhez ugyanolyan, mint a nyelv biztosításainak viszonya a típusrendszer szabályait megsértő műveletekhez. Ha egy alapműveletet nem meghatározott szabályai szerint használunk, az eredmény nem meghatározható lesz. Ha egy *vector* elemeinek destruktora kivételt vált ki, ugyanúgy nincs jogunk logikus viselkedést elvárni, mint amikor egy kezdőértékként véletlenszámmal feltöltött mutatóval szeretnénk adatokat elérni:

```
class Bomb {
public:
    // ...
    ~Bomb() { throw Trouble(); }
};

vector<Bomb> b(10);                    // nem meghatározható viselkedéshez vezet

void f()
{
    int* p = reinterpret_cast<int*>(rand()); // nem meghatározható viselkedéshez vezet
    *p = 7;
}
```

De nézzük a dolgok jó oldalát: ha betartjuk a nyelv és a standard könyvtár alapvető szabályait, a könyvtár jól fog viselkedni úgy is, ha kivételeket váltunk ki.

Amellett, hogy rendelkezésünkre áll a tiszta kivételbiztosság, szeretjük elkerülni az erőforrásokban előforduló „lyukakat"; azaz egy olyan műveletnek, amely kivételt vált ki, nem elég az operandusait meghatározott állapotban hagynia, biztosítania kell azt is, hogy minden igényelt erőforrás valamikor felszabaduljon. Egy kivétel kiváltásának helyén például minden korábban lefoglalt memóriaterületet fel kell szabadítanunk vagy azokat valamilyen objektumhoz kell kötnünk, hogy később a memória szabályosan felszabadítható legyen.

A standard könyvtár biztosítja, hogy nem lesznek erőforrás-lyukak, ha azok a felhasználói műveletek, melyeket a könyvtár használ, maguk sem hoznak létre ilyeneket:

```
void leak(bool abort)
{
    vector<int> v(10);                       // nincs memória-elszivárgás
    vector<int>* p = new vector<int>(10);    // memória-elszivárgás lehetséges
    auto_ptr< vector<int> > q(new vector<int>(10)); // nincs memória-elszivárgás (§14.4.2)

    if (abort) throw Up();
    // ...
    delete p;
}
```

Ha kivétel következik be, a *v vector* és a *q* által mutatott *vector* helyesen lesz törölve, így minden általuk lefoglalt erőforrás felszabadul. Ezzel szemben a *p* által mutatott *vector* objektumot nem védjük a kivételektől, így az nem fog törlődni. Ha a kódrészletet biztonságossá akarjuk tenni, vagy magunknak kell felszabadítani a *p* által mutatott területet a kivétel kiváltása előtt, vagy biztosítanunk kell, hogy valamilyen objektum – például egy *auto_ptr* (§14.4.2) – birtokolja azt és szabályosan felszabadítsa akkor is, ha kivétel következett be.

Figyeljük meg, hogy a nyelv szabályai a részleges létrehozással, illetve megsemmisítéssel kapcsolatban biztosítják, hogy a részobjektumok és az adattagok létrehozásakor bekövetkező kivételeket a standard könyvtár eljárásai minden külön erőfeszítés nélkül helyesen kezeljék (§14.4.1). Ez minden kivételekkel kapcsolatos eljárás esetében nagyon fontos alapelv.

Gondoljunk arra is, hogy nem a memória az egyetlen olyan erőforrás, amelyben lyukak fordulhatnak elő. A megnyitott fájlok, zárolások, hálózati kapcsolatok és a szálak is olyan rendszererőforrások, melyeket egy függvénynek fel kell szabadítania vagy át kell adnia valamely objektumnak egy kivétel kiváltása előtt.

E.3. A kivételbiztosságot megvalósító eljárások

A standard könyvtár – szokás szerint – bemutatja azokat a problémákat, amelyek sok más helyzetben is előfordulhatnak és ezekre olyan megoldást ad, ami széles körben felhasználható. A kivételbiztos programok írásának alapvető eszközei a következők:

1. A *try* blokk (§8.3.1)
2. A „kezdeti értékadás az erőforrás megszerzésével” eljárás (§14.4)

A követendő általános elvek:

3. Soha ne „dobjunk ki” adatokat úgy, hogy nem tudjuk pontosan, mi kerül a helyükre.
4. Az objektumokat mindig érvényes állapotban hagyjuk, amikor kivételt váltunk ki.

Ha betartjuk ezeket a szabályokat, minden hibát megfelelően kezelhetünk. Ezen elvek követése a gyakorlatban azért jelent problémát, mert még a legártalmatlanabbnak tűnő eljárások (például a <, az = vagy a *sort()*) is kiválthatnak kivételeket. Annak megállapításához, hogy egy programban mit kell megvizsgálnunk, nagy tapasztalatra van szükség.

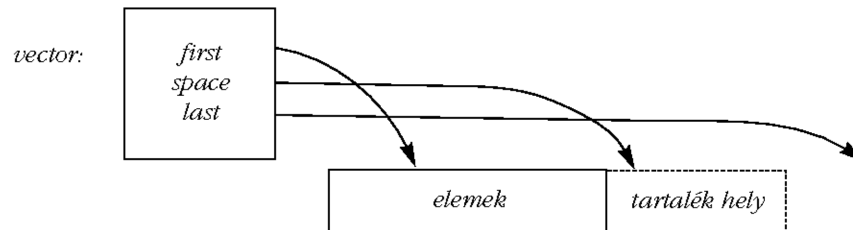
Ha könyvtárat írunk, célszerű az erős kivételbiztosságot (§E.2) célul kitűznünk és mindenhol meg kell valósítanunk az alapbiztosítást. Programok írásakor a kivételbiztosság kevésbé fontos szempont. Például, amikor egy egyszerű adatfeldolgozó programot készítünk saját magunknak, általában nem baj, ha a program egyszerűen befejeződik, amikor a virtuális memória valamilyen hiba miatt elfogy. Ugyanakkor a helyesség és az alapvető kivételbiztosság szorosan kapcsolódik egymáshoz.

Az alapvető kivételbiztosság megvalósítására szolgáló eljárások – például az állapotbiztosítók megadása és fenntartása (§24.3.7.1) – nagyon hasonlítanak azokhoz, melyek segítségével kicsi és helyesen működő programokat hozhatunk létre. Ebből következik, hogy az alapvető kivételbiztosság (az alapbiztosítás; §E.2) – vagy akár az erős biztosítás – megvalósítása csupán jelentéktelen teljesítményromlással jár. Lásd: §E.8[17]

Az alábbiakban a *vector* szabványos tároló (§16.3) egy megvalósítását adjuk meg, hogy bemutassuk, milyen feladatokat kell megoldanunk az ideális kivételbiztosság megvalósításához és hol érdemes alaposabb felügyeletet biztosítanunk.

E.3.1. Egy egyszerű vektor

A *vector* (§16.3) leggyakoribb megvalósításában három mutató (vagy az ezzel egyenértékű mutató–eltolás pár) szerepel: egy az első elemre, egy az utolsó utáni elemre és egy az utolsó utáni lefoglalt helyre hivatkozik (§17.1.3):



Lássuk, mire van szükségünk a *vector* deklarációjában, ha csak a kivételbiztosság és az erőforrás-lyukak elkerülése szempontjából vizsgáljuk az osztályt:

```
template<class T, class A = allocator<T> >
class vector {
private:
    T* v;              // a lefoglalt terület eleje
    T* space;          // az elemsorozat vége, bővítés számára tartalékolt terület kezdete
    T* last;           // a lefoglalt terület vége
    A alloc;           // memóriafoglaló
public:
    explicit vector(size_type n, const T& val = T(), const A& = A());

    vector(const vector& a);                    // másoló konstruktor
    vector& operator=(const vector& a);         // másoló értékadás

    ~vector();

    size_type size() const { return space-v; }
    size_type capacity() const { return last-v; }

    void push_back(const T&);
    // ...
};
```

Vizsgáljunk először egy meggondolatlan konstruktor-megvalósítást:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a) // vigyázat: naív megvalósítás
   : alloc(a)                                               // memóriafoglaló másolása
{
   v = alloc.allocate(n);            // memória lefoglalása az elemek számára (§19.4.1)
   space = last = v+n;
   for (T* p = v; p!=last; ++p) a.construct(p,val);   // val létrehozó másolása *p-be (§19.4.1)
}
```

Itt három helyen keletkezhet kivétel:

1. Az *allocate()* függvény kivételt válthat ki, ha nincs elegendő memória.
2. A memóriafoglaló (allokátor) másoló konstruktora is eredményezhet kivételt.
3. A *T* elemtípus másoló konstruktora is kiválthat kivételt, ha nem tudja lemásolni a *val* értéket.

Egyik esetben sem jön létre új objektum, tehát a *vector* konstruktora sem fut le (§14.4.1).

Ha az *allocate()* végrehajtása sikertelen, a *throw* már akkor kilép a konstruktorból, amikor még semmilyen erőforrást nem foglaltunk le, így ezzel minden rendben.

Ha a *T* másoló konstruktora eredményez hibát, már lefoglaltunk valamennyi memóriát, így azt fel kell szabadítanunk, ha el akarjuk kerülni a memória-elszivárgást. Az igazán nagy problémát az jelenti, hogy a *T* másoló konstruktora esetleg akkor vált ki kivételt, amikor már néhány objektumot sikeresen létrehozott, de még nem az összeset.

Ezen probléma kezeléséhez nyilván kell tartanunk, hogy eddig mely elemeket hoztuk létre, és amikor hiba következik be, ezeket (és csak ezeket) törölnünk kell:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)    // jól kidolgozott megvalósítás
   : alloc(a)                         // memóriafoglaló másolása
{
   v = alloc.allocate(n);         // memória lefoglalása az elemek számára

   iterator p;

   try {
      iterator end = v+n;
      for (p=v; p!=end; ++p) alloc.construct(p,val);      // elemek létrehozása (§19.4.1)
      last = space = p;
   }
```

```
    catch (...) {
        for (iterator q = v; q!=p; ++q) alloc.destroy(q);      // létrehozott elemek megsemmisítése
        alloc.deallocate(v,n);                  // memória felszabadítása
        throw;                                  // továbbdobás
    }
}
```

A pluszköltség ez esetben is csak a *try blokk* költsége. Egy jó C++-változatban ez elhanyagolható a memóriafoglaláshoz és az elemek kezdeti értékadásához képest. Ahol a *try blokk* költsége magas, esetleg beilleszthetjük az *if(n)* vizsgálatot a *try* elé, ezzel az üres vektort külön esetként kezelhetjük.

A konstruktor nagy részét az *uninitialized_fill()* függvény kivételbiztos megvalósítása tölti ki:

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
    For p;
    try {
        for (p=beg; p!=end; ++p)
            new(static_cast<void*>(&*p)) T(x);  // x létrehozó másolása *p-ben (§10.4.11)
    }
    catch (...) {                               // a létrehozott elemek törlése és továbbdobás:
        for (For q = beg; q!=p; ++q) (&*q)->~T();   // (§10.4.11)
        throw;
    }
}
```

A furcsa *&*p* kifejezésre azért volt szükség, hogy a nem mutató bejárókat (iterátorokat) is kezelhessük. Ilyenkor ahhoz, hogy mutatót kapjunk, a hivatkozással meghatározott elem címét kell vennünk. A *void** átalakítás biztosítja, hogy a standard könyvtár elhelyező függvényét használjuk (§19.4.5), nem a felhasználó által a *T** típusra megadott *operator new()* függvényt. Ez az eljárás olyan alacsony szinten működik, ahol a teljes általánosságot már elég nehéz biztosítani.

Szerencsére nem kell újraírnunk az *uninitialized_fill()* függvényt, mert a standard könyvtár biztosítja hozzá a megkívánt erős biztosítást (§E.2). Gyakran elengedhetetlen , hogy olyan kezdőérték-adó művelet álljon rendelkezésünkre, amely vagy minden elemet hibátlanul tölt fel kezdőértékkel, vagy – hiba esetén – egyáltalán nem ad vissza elemeket. Éppen ezért a standard könyvtárban szereplő *uninitialized_fill()*, *uninitialized_fill_n()* és *uninitialized_copy()* függvény (§19.4.4) biztosítja ezt az erős kivételbiztosságot (§E.4.4).

Figyeljük meg, hogy az *uninitialized_fill()* nem védekezik az elemek destruktora vagy a bejáróműveletek által kiváltott kivételek ellen (§E.4.4), ez ugyanis elviselhetetlenül nagy költséget jelentene (lásd §E.8[16-17]).

Az *uninitialized_fill()* nagyon sokféle sorozatra alkalmazható, ezért csak előre haladó bejárókat vesz át (§19.2.1), amivel viszont nem tudja garantálni, hogy az elemeket létrehozásukkal fordított sorrendben törli.

Az *uninitialized_fill()* felhasználásával az alábbiakat írhatjuk:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)          // zavaros megvalósítás
   :alloc(a)                                     // memóriafoglaló másolása
{
   v = alloc.allocate(n);                        // memória lefoglalása az elemek számára
   try {
      uninitialized_fill(v,v+n,val);             // elemek másolása
      space = last = v+n;
   }
   catch (...) {
      alloc.deallocate(v,n);                     // memória felszabadítása
      throw;                                     // továbbdobás
   }
}
```

Ennek ellenére ezt a programot nem nevezhetjük szépnek. A következőkben bemutatjuk, hogyan tehetjük a programot sokkal egyszerűbbé.

Figyeljük meg, hogy a konstruktor újra kiváltja azt a kivételt, amit elkapott. A cél az, hogy a *vector* osztályt „átlátszóvá” tegyük a kivételek előtt, mert így a felhasználó pontosan megállapíthatja a hiba okát. A standard könyvtár összes tárolója rendelkezik ezzel a tulajdonsággal. A kivételekkel szembeni átlátszóság gyakran a legjobb lehetőség a sablonok és a hasonló „vékony” rétegek számára. Ez ellentétben áll a programrendszerek nagyobb részeinek („moduljainak”) irányvonalával, hiszen ezeknek általában önállóan kell kezelniük minden hibát. Pontosabban, az ilyen modulok készítőinek fel kell tudniuk sorolni az összes kivételt, amit a modul kiválthat. Ennek megvalósításához vagy a kivételek csoportosítására (§14.2), vagy az alacsonyszintű eljárások és a modul saját kivételeinek összekapcsolására (§14.6.3), esetleg a kivételek meghatározására (§14.6) van szükség.

E.3.2. A memória ábrázolása

A tapasztalatok bebizonyították, hogy helyes, kivételbiztos programok készítése *try blokkok* segítségével bonyolultabb annál, mint amit egy átlagos programozó még elfogad. Valójában feleslegesen bonyolult, hiszen van egy másik lehetőség: a „kezdeti értékadás az erőforrás megszerzésével" (§14.4), melynek segítségével csökkenthetjük azon programsorok számát, melyek egy „stílusos" program megvalósításához szükségesek. Esetünkben a legfontosabb erőforrás, amire a *vector* osztálynak szüksége van, egyértelműen a memória, melyben az elemeket tároljuk. Ha bevezetünk egy segédosztályt, amely a *vector* által használt memóriát ábrázolja, leegyszerűsíthetjük programunkat és csökkenthetjük annak esélyét, hogy véletlenül elfelejtjük felszabadítani a memóriát.

```
template<class T, class A = allocator<T> >
struct vector_base {
    A alloc;          // memóriafoglaló
    T* v;             // a lefoglalt terület eleje
    T* space;         // az elemsorozat vége, bővítés számára tartalékolt terület kezdete
    T* last;          // a lefoglalt terület vége

    vector_base(const A& a, typename A::size_type n)
        : alloc(a), v(a.allocate(n)), space(v+n), last(v+n) { }
    ~vector_base() { alloc.deallocate(v,last-v); }
};
```

Amíg a *v* és a *last* mutató helyes, a *vector_base* objektum megsemmisíthető. A *vector_base* osztály a *T* típus számára lefoglalt memóriát kezeli és nem *T* típusú objektumokat, ezért mielőtt egy *vector_base* objektumot törlünk, minden ennek segítségével létrehozott objektumot is törölnünk kell.

Természetesen magát a *vector_base* osztályt is úgy kell megírni, hogy ha kivétel következik be (a memóriafoglaló másoló konstruktorában vagy az *allocate()* függvényben), ne jöjjön létre *vector_base* objektum, így memória-elszivárgás sem következik be.

A *vector_base* felhasználásával a *vector* osztályt a következőképpen határozhatjuk meg:

```
template<class T, class A = allocator<T> >
class vector : private vector_base<T,A> {
    void destroy_elements() { for (T* p = v; p!=space; ++p) p->~T(); } // §10.4.11
public:
    explicit vector(size_type n, const T& val = T(), const A& = A());
```

```
    vector(const vector& a);                // másoló konstruktor
    vector& operator=(const vector& a); // másoló értékadás

    ~vector() { destroy_elements(); }

    size_type size() const { return space-v; }
    size_type capacity() const { return last-v; }

    void push_back(const T&);
    // ...
};
```

A *vector* destruktora az összes elemre egymás után meghívja a *T* típus destruktorát. Ebből következik, hogy ha egy elem destruktora kivételt vált ki, a *vector* destruktora sem tud hibátlanul lefutni. Ez igen nagy problémát okoz, ha egy másik kivétel miatt kezdeményezett „verem-visszatekerés" közben következik be, hiszen ekkor a *terminate()* függvény fut le (§14.7). Ha a destruktor kivételt vált ki, általában erőforrás-lyukak keletkeznek és azok az eljárások, melyeket szabályos viselkedésű objektumokhoz fejlesztettek ki, megjósolhatatlanul fognak működni. Nem igazán van használható megoldás a destruktorokban keletkező kivételek kezelésére, ezért a könyvtár semmilyen biztosítást nem vállal, ha az elemek ilyen viselkedésűek lehetnek (§E.4).

A konstruktort az alábbi egyszerű formában adhatjuk meg:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    : vector_base<T,A>(a,n)         // terület lefoglalása n elem számára
{
    uninitialized_fill(v,v+n,val);  // elemek másolása
}
```

A másoló konstruktor mindössze abban különbözik ettől, hogy az *uninitialized_fill()* helyett az *uninitialized_copy()* függvényt használja:

```
template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
    : vector_base<T,A>(a.alloc,a.size())
{
    uninitialized_copy(a.begin(),a.end(),v);
}
```

Figyeljük meg, hogy az ilyen stílusú konstruktor kihasználja a nyelvnek azon alapszabályát, hogy ha a konstruktorban kivétel keletkezik, azokra a részobjektumokra (és alapobjektumokra), melyek sikeresen létrejöttek, a destruktor is szabályosan lefut (§14.4.1). Az *uninitialized_fill()* és testvérei (§E.4.4) ugyanilyen szolgáltatást nyújtanak a félig létrehozott sorozatok esetében.

E.3.3. Értékadás

Szokás szerint az értékadás abban különbözik a létrehozástól, hogy a korábbi értékekre is figyelnünk kell. Vizsgáljuk meg az alábbi megvalósítást:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)     // erős biztosítást ad (§E.2)
{
    vector_base<T,A> b(alloc,a.size());                  // memória lefoglalása
    uninitialized_copy(a.begin(),a.end(),b.v);           // elemek másolása
    destroy_elements();
    alloc.deallocate(v,last-v);                          // a régi memóriaterület felszabadítása
    vector_base::operator=(b);                           // ábrázolás elhelyezése
    b.v = 0;                                             // felszabadítás megelőzése
    return *this;
}
```

Ez az értékadás szép és kivételbiztos is, de túl sok mindent ismétel meg a konstruktorból és a destruktorból. Ennek elkerülésére a következő eljárást írhatjuk:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)     // erős biztosítást ad (§E.2)
{
    vector temp(a);                                      // "a" másolása
    swap< vector_base<T,A> >(*this,temp);                // ábrázolások felcserélése
    return *this;
}
```

A régi elemeket a *temp* destruktora törli, az általuk lefoglalt területet pedig a *temp* változó *vector_base* objektumának destruktora szabadítja fel.

A két változat teljesítménye szinte teljesen egyenlő. Valójában csak két különböző formában adjuk meg ugyanazokat a műveleteket. A második megvalósítás viszont rövidebb és nem ismétli a *vector* osztály egyéb függvényeiben már szereplő kódrészleteket, így az értékadás ezen változata kevesebb hibalehetőséget tartalmaz és egyszerűbb rendben tartani is.

Figyeljük meg, hogy az önértékadás szokásos vizsgálata hiányzik az eljárásból (§10.4.4):

```
if (this == &a) return *this;
```

Ezek az értékadó műveletek először létrehoznak egy másolatot, majd lecserélik az elemeket. Ez a megoldás automatikusan kezeli az önértékadás problémáját. Úgy döntöttem, hogy a ritkán előforduló önértékadás külön vizsgálata nem jár annyi haszonnal, hogy érdemes legyen ezzel lassítani az általános esetet, amikor egy másik *vector* objektumot adunk értékül.

Mindkét esetben két, esetleg jelentős optimalizálási lehetőség hiányzik:

1. Ha az értéket kapó vektor mérete elég nagy ahhoz, hogy az értékül adott vektort tárolja, nincs szükség új memória lefoglalására.
2. Az elemek közötti értékadás hatékonyabb lehet, mint egy elem törlése, majd külön létrehozása.

Ha ezeket a javításokat is beépítjük, a következő eredményt kapjuk:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a) // optimalizált, alapbiztosítás (§E.2)
{
    if (capacity() < a.size()) {              // új vektorábrázolás számára terület lefoglalása
        vector temp(a);                      // "a" másolása
        swap< vector_base<T,A> >(*this,temp);  // az ábrázolások felcserélése
        return *this;
    }
    if (this == &a) return *this;            // védelem az önértékadás ellen (§10.4.4)

                                             // a régi elemek értékadása
    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator();               // memóriafoglaló másolása
    if (asz<=sz) {
        copy(a.begin(),a.begin()+asz,v);
        for (T* p = v+asz; p!=space; ++p) p->~T();  // felesleges elemek felszámolása (§10.4.11)
    }
    else {
        copy(a.begin(),a.begin()+sz,v);
        uninitialized_copy(a.begin()+sz,a.end(),space);     // további elemek létrehozása
    }
    space = v+asz;
    return *this;
}
```

Ezek az optimalizációk nem valósíthatók meg büntetlenül. A *copy()* eljárás nem nyújt erős kivételbiztosítást, mert nem garantálja, hogy a cél változatlan marad, ha másolás közben kivétel következik be. Tehát ha a *T::operator=()* kivételt vált ki a *copy()* művelet közben, előfordulhat, hogy az értéket kapó vektor megváltozik, de nem lesz az értékül adott vektor pontos másolata. Lehetséges például, hogy az első öt elemet sikerül lemásolni az értékül adott vektorból, de a többi változatlan marad, sőt akár az is elképzelhető, hogy egy elem (az, amelyiket éppen másoltuk, amikor a *T::operator=()* kivételt váltott ki) olyan értéket fog tartalmazni, amely nem egyezik meg sem az eredetivel, sem a másolandóval. Azt azért elmondhatjuk, hogy ha a *T::operator=()* érvényes állapotban hagyja operandusát egy kivétel kiváltásakor is, a teljes *vector* is érvényes állapotban marad, bár nem abban az állapotban, amit szerettünk volna.

A fentiekben a memóriafoglalót is értékadással másoltuk le. Valójában a memóriafoglalóktól nem mindig követeljük meg, hogy rendelkezzenek értékadással (§19.4.3, lásd még:§E.8[9]).

A standard könyvtár *vector* értékadásának legutóbbi megvalósítása gyengébb kivételbiztosságot, de nagyobb hatékonyságot biztosít. Csak az alapbiztosítást nyújtja, ami megfelel a legtöbb programozó kivételbiztosságról alkotott fogalmának, de nem áll rendelkezésünkre erős biztosítás (§E.2). Ha olyan értékadásra van szükségünk, amely kivétel felléptekor a *vector*-t változatlanul hagyja, olyan könyvtár-megvalósítást kell használnunk, amely erős biztosítást nyújt az ilyen helyzetekben is vagy saját magunknak kell megírni az értékadó műveletet:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b)   // "magától értendő" a = b
{
    vector<T,A> temp(a.get_allocator());
    temp.reserve(b.size());
    for (typename vector<T,A>::iterator p = b.begin(); p!=b.end(); ++p)
        temp.push_back(*p);
    swap(a,temp);
}
```

Ha nincs elegendő memória ahhoz, hogy létrehozzuk a *temp* változót *b.size()* elem számára, az *std:bad_alloc* kivételt váltjuk ki, mielőtt bármilyen változtatást végeznénk az a vektoron. Ehhez hasonlóan, ha a *push_back()* végrehajtása nem sikerül, az a akkor is érintetlen marad, hiszen minden *push_back()* műveletet a *temp* objektumon hajtunk végre az a helyett. Ezzel a megoldással azt is biztosítjuk, hogy felszabaduljon a *temp* minden eleme, amit a *push_back()* segítségével létrehoztunk, mielőtt a hibát okozó kivételt újra kiváltanánk.

A *swap()* nem másolja a vektorok elemeit, csupán lecseréli a *vector* adattagjait, így a *vector_base* objektumot is. Ennek következtében a *swap()* akkor sem válthat ki kivételt, ha az elemek műveletei képesek erre (§E.4.3). A *safe_assign()* nem készít felesleges másolatokat az elemekről, tehát elég hatékony tud lenni.

Szokás szerint vannak más lehetőségek is a nyilvánvaló megvalósítás mellett, például rábízhatjuk magára a könyvtárra, hogy az elemeket az ideiglenes vektorba másolja:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b)   // egyszerű a = b
{
    vector<T,A> temp(b);        // b elemeinek másolása az ideiglenes változóba
    swap(a,temp);
}
```

Sőt, egyszerű érték szerinti paraméterátadást (§7.2) is használhatunk:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, vector<T,A> b)   // egyszerű a = b
                                                  // (figyelem: b érték szerint átadva)
{
    swap(a,b);
}
```

A *safe_assign()* ez utóbbi két változata a *vector* memóriafoglalóját nem másolja le, ami megengedett optimalizáció (lásd §19.4.3).

E.3.4. A push_back()

A kivételbiztosság szemszögéből nézve a *push_back()* nagyon hasonlít az értékadásra abban, hogy nem szabad a *vector* objektumot megváltoztatnunk, ha az új elem beillesztése valamilyen problémába ütközik:

```
template< class T, class A>
void vector<T,A>::push_back(const T& x)
{
    if (space == last) {    // nincs több hely; áthelyezés
        vector_base b(alloc,size()?2*size():2); // a lefoglalás megkettozése
        uninitialized_copy(v,space,b.v);
        new(b.space) T(x);              // x másolatának *b.space-be helyezése (§10.4.11)
        ++b.space;
        destroy_elements();
```

```
        swap<vector_base<T,A> >(b,*this);       // az ábrázolások felcserélése
        return;
    }
    new(space) T(x);                      // x másolatának *space-be helyezése (§10.4.11)
    ++space;
}
```

Természetesen, a **space* kezdeti értékadására szolgáló másoló konstruktor válthat ki kivételt. Ha ez történik, a *vector* változatlan marad és a *space* értékét sem növeljük meg. Ilyenkor a vektor elemeit sem kell áthelyeznünk a memóriában, tehát az eddigi bejárók is érvényben maradnak. Így ez a megvalósítás erős biztosítást ad: ha egy memóriafoglaló vagy egy felhasználói eljárás kivételt vált ki, a *vector* nem változik meg. A standard könyvtár ilyen biztosítást nyújt a *push_back()* függvényhez (§E.4.1).

Figyeljük meg, hogy az eljárásban nincs egyetlen *try blokk* sem (attól eltekintve, ami az *uninitialized_copy()* függvényben, rejtve szerepel). A módosítást a műveletek sorrendjének pontos megválasztásával hajtjuk végre, így ha kivétel keletkezik, a vektor értéke nem változik meg.

Ez a megközelítés – miszerint az utasítások sorrendje adja a kivételbiztosságot – és a „kezdeti értékadás az erőforrás megszerzésével" (§14.4) alkalmazása elegánsabb és hatékonyabb megoldást kínál, mint a hibák kifejezett kezelése *try blokkok* segítségével. Ha utasításainkat szerencsétlen sorrendben írjuk le, sokkal több kivételbiztossági problémával kell megküzdenünk, mint a kivételkezelő eljárások elhagyása mellett. A sorrend meghatározásakor az alapszabály az, hogy ne semmisítsünk meg információt, mielőtt az azt helyettesítő adatokat létre nem hoztuk és nem biztosítottuk, hogy azokat kivételek veszélye nélkül átírhassuk a helyükre.

A kivételkezelés egyik következménye, hogy a program végrehajtása során a vezérlés meglepő helyekre kerülhet. Az olyan egyszerű, helyi vezérléssel rendelkező függvényekben, mint az *operator=()*, a *safe_assign()* vagy a *push_back()*, meglepetések ritkábban fordulnak elő. Ha ránézünk egy kódrészletre, viszonylag egyszerűen megállapíthatjuk, hogy egy adott sor válthat-e ki kivételt és mi lesz annak következménye. A nagyobb függvényekben, melyekben bonyolult vezérlési szerkezeteket (például bonyolult feltételes utasításokat és egymásba ágyazott ciklusokat) használunk, az ilyen kérdésekre nehéz választ adni. A *try blokkok* alkalmazása a helyi vezérlést még tovább bonyolítja, így újabb keveredéseket és hibákat eredményezhet (§14.4). Azt hiszem, az utasításrendezés és a „kezdeti értékadás az erőforrás megszerzésével" hatékonysága a nagyobb méretű *try blokkokkal* szemben éppen a helyi vezérlés egyszerűsítésében rejlik. Egyszerűbben fogalmazva, a „stílusos" programokat egyszerűbb megérteni és egyszerűbb helyesen megírni.

Gondoljunk rá, hogy az itt szereplő *vector* csak a kivételek által okozott problémák és az ezen problémák megoldására kidolgozott eljárások bemutatására szolgál. A szabvány nem követeli meg, hogy minden megvalósítás pontosan úgy nézzen ki, ahogy itt bemutatjuk. A szabvány által biztosított garanciákat az §E.4. pontban tárgyaljuk.

E.3.5. Konstruktorok és invariánsok

A kivételbiztosság szemszögéből nézve a *vector* többi művelete vagy ugyanúgy viselkedik, mint az eddig bemutatottak (mivel hasonló módon foglalnak le és szabadítanak fel erőforrásokat), vagy megvalósításuk egyszerű (mert nem végeznek olyan tevékenységet, amelyben az érvényes állapot fenntartása problémát okozhatna). A legtöbb osztályban ezek a „triviális" függvények jelentik a döntő többséget. Az ilyen eljárások bonyolultsága elsősorban attól függ, hogy a konstruktor milyen működési környezetet biztosít számukra. Másképpen fogalmazva, az általános tagfüggvények bonyolultsága elsősorban a jó osztályinvariáns megválasztásán múlik (§24.3.7.1). Az egyszerű vektorműveletek vizsgálatával megérthetjük, mitől lesz jó egy osztályinvariáns és hogyan kell megírnunk a konstruktorokat ahhoz, hogy ezeket az invariánsokat biztosítsák.

Az olyan műveleteket, mint a vektorok indexelése (§16.3.3) azért könnyű megvalósítani, mert erősen építenek arra az invariánsra, amit a konstruktor hoz létre és az erőforrásokat lefoglaló, illetve felszabadító függvények tartanak fenn. Az indexelő művelet például hivatkozhat a *v* tömbre, amely az elemeket tárolja:

```
template< class T, class A>
T& vector<T,A>::operator[](size_type i)
{
   return v[i];
}
```

Nagyon fontos és alapvető szabály, hogy a konstruktornak kell lefoglalnia az erőforrásokat és biztosítania kell egy egyszerű invariánst. Ahhoz, hogy ennek fontosságát megértsük, nézzük meg a *vector_base* definíciójának alábbi változatát:

```
template<class T, class A = allocator<T> > // a konstruktor esetlen használata
class vector_base {
public:
   A alloc;          // memóriafoglaló
   T* v;             // a lefoglalt terület eleje
   T* space;         // az elemsorozat vége, bővítés számára tartalékolt terület kezdete
   T* last;          // a lefoglalt terület vége
```

```
    vector_base(const A& a, typename A::size_type n) : alloc(a), v(0), space(0), last(0)
    {
        v = alloc.allocate(n);
        space = last = v+n;
    }

    ~vector_base() { if (v) alloc.deallocate(v,last-v); }
};
```

Itt a *vector_base* objektumot két lépésben hozzuk létre. Először egy „biztonsági állapotot" alakítunk ki, amely a *v*, a *space* és a *last* változót *0*-ra állítja, és csak ezután próbálunk memóriát foglalni. Ez arra az indokolatlan félelemre vezethető vissza, hogy az elemek lefoglalása közben keletkező kivételek miatt félig létrejött objektumot hozhatunk létre. A félelem azért indokolatlan, mert ilyen objektumot egyáltalán nem hozhatunk létre. A szabályok, melyek a statikus, automatikus, illetve tagobjektumok és a standard könyvtár tárolóinak elemeire vonatkoznak, megakadályozzák ezt. Azokban a szabvány előtti könyvtárakban azonban, melyek a tárolókban elhelyező *new* operátort (§10.4.11) használtak (használnak) az objektumok létrehozására és nem foglalkoztak (foglalkoznak) a kivételbiztossággal, ez előfordulhatott (előfordulhat). A megrögzött szokásokon nehéz változtatni.

Figyeljük meg, hogy a biztonságosabb program előállítására irányuló próbálkozás tovább bonyolítja az osztályinvariánst: nem lehetünk biztosak abban, hogy a *v* egy létező, lefoglalt memóriaterületre mutat, mert szerepelhet benne a *0* érték is. Ez a hiba azonnal megbosszulja magát. A standard könyvtár nem követeli meg a memóriafoglalóktól, hogy egy *0* értéket tartalmazó mutatót biztonságosan szabadítsanak fel (§19.4.1). A memóriafoglalók ebben eltérnek a *delete* művelettől (§6.2.6). Ebből következik, hogy a destruktorban külön ellenőrzést kell végeznünk. Ezenkívül minden elemnek kezdőértéket adunk és csak később értelmes értéket. Ezen költségek egy olyan elemtípus esetében lehetnek jelentősek, ahol az értékadás bonyolult, például a *string* vagy a *list* osztálynál.

A kétlépéses létrehozás nem szokatlan megoldás. Gyakran olyan formában jelenik meg, hogy a konstruktor csak egy „egyszerű és biztonságos" kezdeti értékadást végez, mellyel az objektum törölhető állapotba kerül. A valódi kezdeti értékadást egy *init()* függvény végzi el, melyet a felhasználónak külön meg kell hívnia:

```
template<class T>              // régies (szabvány és kivételkezelés előtti) stílus
class vector_base {
public:
    T* v;              // a lefoglalt terület eleje
    T* space;          // az elemsorozat vége, bővítés számára tartalékolt terület kezdete
    T* last;           // a lefoglalt terület vége
```

```
    vector_base() : v(0), space(0), last(0) { }
    ~vector_base() { free(v); }

    bool init(size_t n)     // igazat ad vissza, ha a kezdeti értékadás sikerült
    {
        if (v = (T*)malloc(sizeof(T)*n)) {
            uninitialized_fill(v,v+n,T());
            space = last = v+n;
            return true;
        }
        return false;
    }
};
```

Ezen stílus látható előnyei a következők:

1. A konstruktor nem válthat ki kivételt és az *init()* segítségével megvalósított kezdeti értékadás sikerességét a „szokásos" módszerekkel (azaz nem kivételekkel) ellenőrizhetjük.
2. Érvényes állapot áll rendelkezésünkre, melyet bármely művelet komoly probléma esetén is biztosítani tud.
3. Az erőforrások lefoglalását mindaddig halaszthatjuk, amíg ténylegesen szükségünk nincs kezdőértékkel rendelkező objektumokra.

A következő részfejezetekben ezeket a szempontokat vizsgáljuk meg és bemutatjuk, hogy a kétlépéses létrehozás miért nem biztosítja az elvárt előnyöket, amellett, hogy más problémákat is felvet.

E.3.5.1. Az init() függvény használata

Az első pont (az *init()* eljárás használata a konstruktor helyett) valójában nem is előny. A konstruktorok és kivételek használata sokkal általánosabb és rendezettebb megoldás az erőforrás-lefoglalási hibák és a kezdeti értékadással kapcsolatos problémák kezelésére (§14.1, §14.4). Ez a stílus a kivételek nélküli C++ maradványa.

Ha a két stílusban ugyanolyan figyelmesen írunk meg egy programot, azt tapasztalhatjuk, hogy két, szinte teljesen egyenértékű eredményt kapunk. Az egyik:

```
int f1(int n)
{
    vector<X> v;
    // ...
```

```
    if (v.init(n)) {
        // "v" n elem vektora
    }
    else {
        // a probléma kezelése
    }
}
```

Míg másik változat:

```
int f2(int n)
try {
    vector v<X> v(n);
    // ...
    // "v" n elem vektora
}
catch (...) {
    // a probléma kezelése
}
```

A külön *init()* függvény használata viszont „lehetőséget” ad az alábbi hibák elkövetésére:

1. Elfelejtjük meghívni az *init()* függvényt (§10.2.3).
2. Elfelejtjük megvizsgálni, hogy az *init()* sikerrel járt-e.
3. Elfelejtjük, hogy az *init()* kivételeket válthat ki.
4. Használni kezdünk egy objektumot, mielőtt meghívnánk az *init()* eljárást.

A *vector<T>::init()* definíciója a *[3]* pontra mutat példát.

Egy jó C++-változatban az *f2()* egy kicsit gyorsabb is, mint az *f1()*, mert az általános esetben nem végez ellenőrzést.

E.3.5.2. Alapértelmezett érvényes állapot

A második pont (miszerint egy könnyen előállítható, „alapértelmezett” érvényes állapot áll rendelkezésünkre) általában tényleg előny, de a *vector* esetében ez felesleges költségeket jelent, ugyanis elképzelhető olyan *vector_base*, ahol *v==0* és a *vector* megvalósításának mindenhol védekeznie kell ez ellen:

```
template< class T>
T& vector<T>::operator[](size_t i)
{
```

```
    if (v) return v[i];
    // hibakezelés
}
```

Ha megengedjük a *v==0* állapotot, a tartomány-ellenőrzés nélküli indexelés ugyanolyan lassú lesz, mint az ellenőrzött hozzáférés:

```
template< class T>
T& vector<T>::at(size_t i)
{
    if (i<v.size()) return v[i];
    throw out_of_range("vector index");
}
```

Itt alapjában véve annyi történt, hogy a *vector_base* eredeti invariánsát túlbonyolítottuk, azzal, hogy bevezettük a *v==0* lehetőséget. Ennek következtében a *vector* eredeti invariánsát is ugyanígy kellett módosítanunk, tehát a *vector* és a *vector_base* minden eljárását bonyolultabban kell megfogalmaznunk. Ez számtalan hiba forrása lehet, például nehezebb lesz a kód módosítása és a program lassabban fog futni. Gondoljunk arra, hogy a modern kiépítésű számítógépeknél a feltételes utasítások meglepően költségesek lehetnek. Ha fontos a hatékonyság, a kulcsműveletek – például a vektorindexelés – feltételes utasítások nélküli megvalósítása elsőrendű követelmény lehet.

Érdekes módon, már a *vector_base* eredeti meghatározása is biztosít egy könnyen létrehozható érvényes állapotot. Csak akkor létezhet egy *vector_base* objektum, ha a kezdeti helyfoglalás sikeres volt. Ebből következik, hogy a *vector* írójának biztosítania kell egy „vészkijárat" függvényt, például a következő formában:

```
template< class T, class A>
void vector<T,A>::emergency_exit()
{
    space = v;       // *this méretének 0-ra állítása
    throw Total_failure();
}
```

Ez a megoldás túlságosan drasztikus, mert nem hívja meg az elemek destruktorait és nem szabadítja fel a *vector_base* objektumban az elemek által elfoglalt területet. Röviden fogalmazva, nem nyújt alapbiztosítást (§E.2). Ha figyelünk a *v* és a *space* adattag tartalmára és az elemek destruktoraira, elkerülhetjük az erőforrás-lyukak kialakulását:

```
template< class T, class A>
void vector<T,A>::emergency_exit()
{
```

```
    destroy_elements();    // takarítás
    throw Total_failure();
}
```

Figyeljük meg, hogy a szabványos *vector* olyan egyszerű szerkezet, amely a lehető legkisebbre csökkenti a kétlépéses létrehozás miatt jelentkező problémák lehetőségét. Az *init()* függvény szinte egyenértékű a *resize()* eljárással, a *v==0* lehetőséget pedig a legtöbb esetben a *size()==0* vizsgálat elvégzésével is kezelhetjük. A kétlépéses létrehozás eddig bemutatott negatív hatásai még erősebben jelentkeznek, ha programunkban szerepel egy olyan osztály, amelynek jelentős erőforrásokra – például hálózati kapcsolatra vagy külső állományokra – van szüksége. Ezek az osztályok ritkán képezik egy olyan keretrendszer részét, amely felügyeli használatukat és megvalósításukat, olyan formában, ahogy a standard könyvtár követelményei felügyelik a *vector* használatát. A problémák száma még tovább növekszik, ha az alkalmazás céljai és a megvalósításukhoz szükséges erőforrások kapcsolata bonyolult. Nagyon kevés olyan osztály van, amely annyira közvetlenül kapcsolódik a rendszer erőforrásaihoz, mint a *vector*.

Az egyszerű „biztonságos állapot" létezésének elve alapjában véve nagyon hasznos. Ha egy objektumot nem tudunk érvényes állapotba állítani anélkül, hogy kivételektől kellene tartanunk a művelet befejezése előtt, valóban problémáink lehetnek. A „biztonságos állapotnak" viszont az osztály szerepéhez természetesen kell kapcsolódnia, nem erőltetett módon, az osztály invariánsát bonyolítva.

E.3.5.3. Az erőforrás-lefoglalás késleltetése

A második ponthoz hasonlóan (§E.3.5.2) a harmadik is egy jó ötlet rossz megvalósítása, ami nyereség helyett inkább veszteséget eredményez. A legtöbb esetben – különösen az olyan tárolókban, mint a *vector* – az erőforrás-lefoglalás késleltetésének legjobb módja a programozó számára az, hogy magát az objektumot hozza létre, amikor szüksége van rá. Nézzük meg például a *vector* objektum alábbi felhasználását:

```
void f(int n)
{
    vector<X> v(n);    // n darab alapértelmezett X típusú objektum létrehozása
    // ...
    v[3] = X(99);      // v[3] igazi "kezdeti értékadása"
    // ...
}
```

Nagy pazarlás egy *X* típusú objektumot csak azért létrehozni, mert valamikor, később értéket fogunk adni neki. Különösen nagy a veszteség, ha az *X* osztályra az értékadás költséges művelet. Ezért az *X* kétlépéses létrehozása előnyösnek tűnhet. Az *X* maga is lehet egy *vector*, ezért a *vector* kétlépéses létrehozásától az üres vektorok létrehozási költségeinek csökkentését remélhetjük, az alapértelmezett (üres) vektorok létrehozása azonban már egyébként is elég hatékony, ezért felesleges a megvalósítást azzal bonyolítanunk, hogy az üres vektort külön esetként kezeljük. Általánosabban fogalmazva, a felesleges kezdeti értékadások elkerülésére ritkán jelent tökéletes megoldást az, hogy a konstruktorból kiemeljük az összetettebb kezdeti értékadásokat:

```
void f2(int n)
{
    vector<X> v;              // üres vektor létrehozása
    // ...
    v.push_back(X(99));       // elemek létrehozása, amikor szükséges
    // ...
}
```

Összefoglalva: a kétlépéses létrehozás sokkal bonyolultabb osztályinvariánshoz és általában kevésbé elegáns, több hibalehetőséget tartalmazó és nehezebben kezelhető programhoz vezet. Ezért a nyelv által támogatott „konstruktor elv" jobban használható, mint az „*init()* függvényes megoldás". Tehát az erőforrásokat mindig a konstruktorban foglaljuk le, ha a késleltetett erőforrás-lefoglalást nem teszi kötelezővé maga az osztály természete.

E.4. A szabványos tárolók garanciái

Ha a könyvtár valamelyik művelete önmaga vált ki kivételt, akkor biztosítani tudja – és biztosítja is –, hogy az általa használt objektumok érvényes állapotban maradnak. A *vector* esetében például az *at()* függvény (§16.3.3) képes kiváltani egy *out_of_range* kivételt, ez azonban nem jelent problémát a vektor kivételbiztossága szempontjából. Az *at()* függvény megírójának nem jelent problémát, hogy a vektort érvényes állapotba állítsa a kivétel kiváltása előtt. Problémák csak akkor jelentkeznek – a könyvtár megvalósítói, a könyvtár felhasználói, illetve azok számára, akik megpróbálják megérteni a programot –, amikor felhasználói eljárások váltanak ki kivételt.

A standard könyvtár tárolói alapbiztosítást nyújtanak (§E.2): a könyvtár alap invariánsai mindig megmaradnak és ha a felhasználó a követelményeknek megfelelően jár el, nem keletkeznek erőforrás-lyukak sem. A felhasználói eljárásoktól azt követeljük meg, hogy ne

hagyják a tárolók elemeit érvénytelen állapotban és a destruktorok ne váltsanak ki kivételt. Az eljárásokon most azokat a függvényeket értjük, melyeket a standard könyvtár megvalósításában felhasználunk, tehát a konstruktorokat, az értékadásokat, a destruktorokat, illetve a bejárók műveleteit (§E.4.4).

A programozó ezeket a műveleteket könnyen megírhatja a könyvtár elvárásainak megfelelően. A követelményeket általában akkor is kielégítik eljárásaink, ha nem tudatosan figyelünk rájuk. A következő típusok biztosan kielégítik a standard könyvtár követelményeit a tárolók elemtípusaira vonatkozóan:

1. A beépített típusok, köztük a mutatók
2. Azok a típusok, melyek nem tartalmaznak felhasználói műveleteket
3. Az olyan műveletekkel rendelkező osztályok, melyek nem váltanak ki kivételeket és nem hagyják operandusaikat érvénytelen állapotban
4. Azok az osztályok, melyek destruktora nem vált ki kivételt, és amelyeknél könnyen ellenőrizhető, hogy a standard könyvtár által használt eljárások (a konstruktorok, az értékadások, a <, az == és a *swap()* függvény) nem hagyják operandusaikat érvénytelen állapotban

Azt is ellenőriznünk kell minden esetben, hogy a műveletek ne hozzanak létre erőforrás-lyukakat:

```
void f(Circle* pc, Triangle* pt, vector<Shape*>& v2)
{
    vector<Shape*> v(10);       // vektor létrehozása vagy bad_alloc kivétel kiváltása
    v[3] = pc;                  // nem vált ki kivételt
    v.insert(v.begin()+4,pt);   // vagy beszúrja a pt elemet, vagy nincs hatása v-re
    v2.erase(v2.begin()+3);     // vagy törli v2[3]-t, vagy nincs hatása v2-re
    v2 = v;                     // vagy átmásolja v-t, vagy nincs hatása v2-re
    // ...
}
```

Amikor az *f()* futása véget ér, *v* szabályosan törlődni fog, míg *v2* érvényes állapotban lesz. A fenti részlet nem mutatja, ki felel a *pc* és a *pt* törléséért. Ha *f()* a felelős, akkor vagy el kell kapnia a kivételeket és így kezelni a szükséges törléseket, vagy a mutatókat lokális *auto_ptr* változókhoz kell kötnie.

Ennél érdekesebb kérdés, mikor ad a könyvtár erős biztosítást, azaz mely műveletek működnek úgy, hogy vagy sikeresen futnak le, vagy semmilyen változtatást nem hajtanak végre operandusaikon.

Például:

```
void f(vector<X>& vx)
{
    vx.insert(vx.begin()+4,X(7));   // elem hozzáadása
}
```

Általában az *X* műveletei és a *vector<X>* osztály memóriafoglalója válthat ki kivételt. Mit mondhatunk a *vx* elemeiről, ha az *f()* függvény futása kivétel következtében szakad meg? Az alapbiztosítás garantálja, hogy erőforrás-lyukak nem keletkeznek és a *vx* elemei érvényes állapotban maradnak. De pontosan milyen elemekről van szó? Elképzelhető, hogy egy elem azért törlődik, mert az *insert()* csak így tudja az alapbiztosítás követelményeit visszaállítani? Gyakran nem elég annyit tudnunk, hogy a tároló jó állapotban van, pontosan tudni akarjuk azt is, milyen állapotról van szó. A kivétel kezelése után általában tisztában szeretnénk lenni azzal, hogy milyen elemek szerepelnek a vektorban, mert ellenkező esetben komolyabb hibakezelést kellene végeznünk.

E.4.1. Elemek beszúrása és törlése

Az elemek beszúrása egy tárolóba, illetve az elemek törlése onnan nyilvánvaló példája azon műveleteknek, melyek a tárolót megjósolhatatlan állapotban hagyhatnák egy kivétel bekövetkezésekor. Ennek oka leginkább az, hogy a beszúrás és a törlés során sok olyan műveletet hajtunk végre, amely kivételt válthat ki:

1. Új értéket másolunk a tárolóba.
2. A tárolóból eltávolított elemet meg is kell semmisítenünk.
3. Az új elem tárolásához néha memóriát is kell foglalnunk.
4. A *vector* és a *deque* elemeit néha új helyre kell áthelyeznünk.
5. Az asszociatív tárolók összehasonlító eljárásokat alkalmaznak az elemekre.
6. Sok beszúrás és törlés esetében bejáró műveleteket is végre kell hajtanunk.

Ezek a műveletek mind okozhatnak kivételeket, ezt semmilyen biztosítás (§E.2) nem akadályozza meg. Ahhoz, hogy ezekre az esetekre valamilyen biztosítást nyújtsunk, elviselhetetlenül költséges eljárásokra lenne szükség. Ennek ellenére a könyvtár védi magát – és a felhasználókat – a többi, felhasználói függvény kivételeitől.

Amikor láncolt adatszerkezeteken végzünk műveleteket (például egy *list*-en vagy *map*-en), úgy szúrhatunk be és távolíthatunk el elemeket, hogy a tároló többi elemére nem vagyunk

hatással. Ugyanez nem valósítható meg az olyan tárolókban, ahol több elem számára egyetlen, folytonos memóriaterületet foglalunk le (például a *vector* és a *deque* esetében). Ilyenkor az elemeket néha új helyre kell mozgatnunk.

Az alapbiztosításon túl a standard könyvtár erős biztosítást is nyújt néhány olyan művelethez, amely elemeket szúr be vagy töröl. Mivel a láncolt adatszerkezetekkel megvalósítható tárolók ebből a szempontból jelentősen eltérnek az elemek tárolásához folytonos memóriaterületet használóktól, a standard könyvtár teljesen más garanciákat ad a különböző tárolófajtákhoz:

1. Garanciák a *vector* (§16.3) és a *deque* (§17.2.3) osztályra:
   - Ha egy *push_back()* vagy egy *push_front()* művelet okoz kivételt, akkor az nem változtatja meg operandusait.
   - Ha egy *insert()* utasítás vált ki kivételt és azt nem egy elem másoló konstruktora vagy értékadó művelete okozta, akkor az sem változtat operandusain.
   - Az *erase()* művelet csak akkor vált ki kivételt, ha azt az elemek másoló konstruktora vagy értékadó művelete okozza.
   - A *pop_back()* és a *pop_front()* nem okoz kivételt.
2. A *list* (§17.2.2) garanciái:
   - Ha egy *push_back()* vagy *push_front()* művelet vált ki kivételt, akkor a függvény hatástalan.
   - Ha az *insert()* okoz kivételt, akkor az nem változtatja meg operandusait.
   - Az *erase()*, a *pop_back()*, a *pop_front()*, a *splice()* és a *reverse()* sohasem vált ki kivételt.
   - Ha a predikátumok és az összehasonlító függvények nem okoznak kivételt, akkor a *list* osztály *remove()*, *remove_if()*, *unique()*, *sort()* és *merge()* eljárásai sem válthatnak ki kivételt.
3. Garanciák asszociatív tárolókra (§17.4):
   - Ha egy elem beszúrása közben az *insert()* kivételt vált ki, akkor a függvény hatástalan.
   - Az *erase()* nem okozhat kivételt.

Jegyezzük meg, hogy ha erős biztosítás áll rendelkezésünkre egy tároló valamelyik műveletében, akkor minden bejáró, az elemekre hivatkozó összes mutató és hivatkozás (referencia) érvényes marad kivétel bekövetkezése esetén is.

A szabályokat egy táblázatban foglalhatjuk össze:

A tárolóműveletek garanciái				
	vector	deque	list	map
clear()	nem lehet kivétel (másolás)	nem lehet kivétel (másolás)	nem lehet kivétel	nem lehet kivétel
erase()	nem lehet kivétel (másolás)	nem lehet kivétel (másolás)	nem lehet kivétel	nem lehet kivétel
1 elemű *insert()*	erős (másolás)	erős (másolás)	erős	erős
N elemű *insert()*	erős (másolás)	erős (másolás)	erős	alap
merge()	—	—	nem lehet kivétel (összehasonlítás)	—
push_back()	erős	erős	erős	—
push_front()	—	erős	erős	—
pop_back()	nem lehet kivétel	nem lehet kivétel	nem lehet kivétel	—
pop_front()	—	nem lehet kivétel	nem lehet kivétel	—
remove()	—	—	nem lehet kivétel (összehasonlítás)	—
remove_if()	—	—	nem lehet kivétel (predikátum)	—
reverse()	—	—	nem lehet kivétel	—
splice()	—	—	nem lehet kivétel	—
swap()	nem lehet kivétel	nem lehet kivétel	nem lehet kivétel	nem lehet kivétel (összehasonlítás másolása)
unique()	—	—	nem lehet kivétel (összehasonlítás)	—

A táblázat elemeinek jelentése:

alap	A művelet csak alapbiztosítást nyújt (§E.2).
erős	A művelet erős biztosítást nyújt (§E.2).
nem lehet kivétel	A művelet nem válthat ki kivételt (§E.2).
—	A művelet ebben a tárolóban nem szerepel tagfüggvényként.

Ahol a biztosítás megköveteli, hogy a felhasználó által megadott bizonyos műveletek ne váltsanak ki kivételt, ott a biztosítás alatt zárójelben feltüntettük, milyen műveletekre kell figyelnünk. Ezek a követelmények pontosan megegyeznek a táblázat előtt, szövegesen megfogalmazott feltételekkel.

A *swap()* függvények abban különböznek a többi eljárástól, hogy nem tagfüggvények. A *clear()* függvényre vonatkozó garancia az *erase()* biztosításából következik. (§16.3.6) A táblázatban az alapbiztosításon túli szolgáltatásokat tüntettük fel, tehát nem szerepelnek azok az eljárások (például a *reverse()* vagy a *unique()* a *vector* osztályra), melyek további biztosítás nélkül valósítanak meg valamilyen algoritmust az összes sorozatra.

A „majdnem-tároló" *basic_string* (§17.5, §20.3) minden műveletére garantálja az alapbiztosítást (§E.5.1). A szabvány azt is biztosítja, hogy a *basic_string* osztály *erase()* és *swap()* eljárása nem okoz kivételt, az *insert()* és a *push_back()* függvényre pedig erős biztosítást kapunk.

Az erős biztosítást nyújtó eljárásokban amellett, hogy a tároló változatlan marad, az összes bejáró, mutató és referencia is érvényes marad:

```
void update(map<string,X>& m, map<string,X>::iterator current)
{
    X x;
    string s;
    while (cin>>s>>x)
    try {
        current = m.insert(current,make_pair(s,x));
    }
    catch(...) {
        // itt a "current" még mindig az aktuális elemet jelöli
    }
}
```

E.4.2. Garanciák és kompromisszumok

Az alapbiztosításon túli szolgáltatások összevisszaságai a megvalósítási lehetőségekkel magyarázhatók. A programozók azt szeretnék leginkább, hogy mindenhol erős biztosítás álljon rendelkezésükre a lehető legkevesebb korlátozás mellett, de ugyanakkor azt is elvárják, hogy a standard könyvtár minden művelete optimálisan hatékony legyen. Mindkét elvárás jogos, de sok művelet esetében lehetetlen egymással párhuzamosan megvalósítani. Ahhoz, hogy jobban megvilágítsuk az elkerülhetetlen kompromisszumokat, megvizsgáljuk, milyen módokon lehet egy vagy több elemet felvenni egy listába, vektorba vagy *map*-be.

Nézzük először, hogy egy elemet hogyan vihetünk be egy listába vagy egy vektorba. Szokás szerint, a *push_back()* nyújtja a legegyszerűbb lehetőséget:

```
void f(list<X>& lst, vector<X>& vec, const X& x)
{
    try {
        lst.push_back(x);        // hozzáadás a listához
    }
    catch (...) {
        // lst változatlan
        return;
    }
    try {
        vec.push_back(x);        // hozzáadás a vektorhoz
    }
    catch (...) {
        // vec változatlan
        return;
    }
    // lst és vec egy-egy x értékű új elemmel rendelkezik
}
```

Az erős biztosítás megvalósítása ez esetben egyszerű és „olcsó”. Az eljárás azért is hasznos, mert teljesen kivételbiztos megoldást ad az elemek felvételére. A *push_back()* azonban asszociatív tárolókra nem meghatározott: a *map* osztályban nincs *back()*. Egy asszociatív tároló esetében az utolsó elemet a rendezés határozza meg, nem a pozíció.

Az *insert()* függvény garanciái már kicsit bonyolultabbak. A gondot az jelenti, hogy az *insert()* műveletnek gyakran kell egy elemet a tároló „közepén” elhelyeznie. Láncolt adatszerkezeteknél ez nem jelent problémát, tehát a *list* és a *map* egyszerűen megvalósítható, a *vector* esetében azonban előre lefoglalt terület áll rendelkezésünkre, a *vector<X>::insert()* függvény egy átlagos megvalósítása pedig a beszúrási pont utáni elemeket áthelyezi, hogy helyet csináljon az új elem számára. Ez az optimális megoldás, de arra nincs egyszerű mód-

szer, hogy a vektort visszaállítsuk eredeti állapotába, ha valamelyik elem másoló értékadása vagy másoló konstruktora kivételt vált ki (lásd §E.8[10-11]), ezért a *vector* azzal a feltétellel ad biztosításokat, hogy az elemek másoló konstruktora nem vált ki kivételt. A *list* és a *map* osztálynak nincs szüksége ilyen korlátozásra, ezek könnyedén be tudják illeszteni az új elemet a szükséges másolások elvégzése után.

Példaképpen tételezzük fel, hogy az *X* másoló konstruktora és másoló értékadása egy *X::cannot_copy* kivételt vált ki, ha valamilyen okból nem sikerül létrehoznia a másolatot:

```
void f(list<X>& lst, vector<X>& vec, map<string,X>& m, const X& x, const string& s)
{
  try {
    lst.insert(lst.begin(),x);          // hozzáadás a listához
  }
  catch (...) {
    // lst változatlan
    return;
  }
  try {
    vec.insert(vec.begin(),x);          // hozzáadás a vektorhoz
  }
  catch (X::cannot_copy) {
    // hoppá: vec vagy rendelkezik, vagy nem rendelkezik új elemmel
    return;
  }
  catch (...) {
    // vec változatlan
    return;
  }
  try {
    m.insert(make_pair(s,x));           // hozzáadás az asszociatív tömbhöz
  }
  catch (...) {
    // m változatlan
    return;
  }
  // lst és vec egy-egy x értékű új elemmel rendelkezik
  // m egy új (s,x) értékű elemmel rendelkezik
}
```

Ha *X::cannot_copy* kivételt kapunk, nem tudhatjuk, hogy az új elem bekerült-e a *vec* tárolóba. Ha sikerült beilleszteni az elemet, az érvényes állapotban lesz, de pontos értékét nem ismerjük. Az is elképzelhető, hogy egy *X::cannot_copy* kivétel után néhány elem „titokza-

tosan" megkettőződik (lásd §E.8[11]), másik megvalósítást alkalmazva pedig a vektor végén lévő elemek tűnhetnek el, mert csak így lehet biztosítani, hogy a tároló érvényes állapotban maradjon és ne szerepeljenek benne érvénytelen elemek.

Sajnos az erős biztosítás megvalósítása a *vector* osztály *insert()* függvénye esetében lehetetlen, ha megengedjük, hogy az elemek másoló konstruktora kivételt váltson ki. Ha egy vektorban teljesen meg akarnánk védeni magunkat az elemek áthelyezése közben keletkező kivételektől, a költségek elviselhetetlenül megnőnének az egyszerű, alapbiztosítást nyújtó megoldáshoz képest.

Sajnos nem ritkák az olyan elemtípusok, melyek másoló konstruktora kivételt eredményezhet. Már a standard könyvtárban is találhatunk példát: a *vector<string>*, a *vector< vector<double> >* és a *map<string, int>* is ilyen.

A *list* és a *vector* tároló ugyanolyan biztosítást ad az *insert()* egyelemű és többelemű változatához, mert azok megvalósítási módja azonos. A *map* viszont erős biztosítást ad az egyelemű beszúráshoz, míg a többeleműhöz csak alapbiztosítást. Az egyelemű *insert()* a *map* esetében könnyen elkészíthető erős biztosítással, a többelemű változat egyetlen logikus megvalósítási módja azonban a *map* esetében az, hogy az új elemeket egymás után szúrjuk be, és ehhez már nagyon nehéz lenne erős garanciákat adni. A gondot itt az jelenti, hogy nincs egyszerű visszalépési lehetőség (nem tudunk korábbi sikeres beszúrásokat visszavonni), ha valamelyik elem beszúrása nem sikerül.

Ha olyan többelemű beszúró műveletre van szükségünk, amely erős biztosítást ad, azaz vagy minden elemet hibátlanul beilleszt, vagy egyáltalán nem változtatja meg a tárolót, legegyszerűbben úgy valósíthatjuk meg, hogy egy teljesen új tárolót készítünk, majd ennek sikeres létrehozása után egy *swap()* műveletet alkalmazunk:

```
template<class C, class Iter>
void safe_insert(C& c, typename C::const_iterator i, Iter begin, Iter end)
{
    C tmp(c.begin(),i);                         // az elöl levő elemek másolása ideiglenes változóba
    copy(begin,end,inserter(tmp,tmp.end()));    // új elemek másolása
    copy(i,c.end(),inserter(tmp,tmp.end()));    // a záró elemek másolása
    swap(c,tmp);
}
```

Szokás szerint, ez a függvény is hibásan viselkedhet, ha az elemek destruktora kivételt vált ki, ha viszont az elemek másoló konstruktora okoz hibát, a paraméterben megadott tároló változatlan marad.

E.4.3. A swap()

A másoló konstruktorokhoz és értékadásokhoz hasonlóan a *swap()* eljárások is nagyon fontos szerepet játszanak sok szabványos algoritmusban és közvetlenül is gyakran használják a felhasználók. A *sort()* és a *stable_sort()* például általában a *swap()* segítségével rendezi át az elemeket. Tehát ha a *swap()* kivételt vált ki, miközben a tárolóban szereplő értékeket cserélgeti, akkor a tároló elemei a csere helyett vagy változatlanok maradnak, vagy megkettőződnek.

Vizsgáljuk meg a standard könyvtár *swap()* függvényének alábbi, egyszerű megvalósítását (§18.6.8):

```
template<class T> void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Erre teljesül, hogy a *swap()* csak akkor eredményezhet kivételt, ha azt az elemek másoló konstruktora vagy másoló értékadása váltja ki.

Az asszociatív tárolóktól eltekintve a szabványos tárolók biztosítják, hogy a *swap()* függvény ne váltson ki kivételeket. A tárolókban általában úgy is meg tudjuk valósítani a *swap()* függvényt, hogy csak az adatszerkezeteket cseréljük fel, melyek mutatóként szolgálnak a tényleges elemekhez (§13.5, §17.1.3). Mivel így magukat az elemeket nem kell mozgatnunk, azok konstruktorára vagy értékadó műveletére nincs szükségünk, tehát azok nem kapnak lehetőséget kivétel kiváltására. Ezenkívül a szabvány biztosítja, hogy a könyvtár *swap()* függvénye nem tesz érvénytelenné egyetlen hivatkozást, mutatót és bejárót sem azok közül, melyek a felcserélt tárolók elemeire hivatkoznak. Ennek következtében kivételek egyetlen ponton léphetnek fel: az asszociatív tárolók összehasonlító objektumaiban, melyeket az adatszerkezet leírójának részeként kell másolnunk. Tehát az egyetlen kivétel, amit a szabványos tárolók *swap()* eljárása eredményezhet, az összehasonlító objektum másoló konstruktorából vagy értékadó műveletéből származik (§17.1.4.1). Szerencsére az összehasonlító objektumoknak általában annyira egyszerű másoló műveleteik vannak, hogy nincs lehetőségük kivétel kiváltására.

A felhasználói *swap()* függvények viszonylag egyszerűen nyújthatnak ugyanilyen biztosításokat, ha gondolunk rá, hogy „mutatókkal” ábrázolt adatok esetében elegendő csak a mutatókat felcserélnünk, ahelyett, hogy lassan és precízen lemásolnánk a mutatók által kijelölt tényleges adatokat (§13.5, §16.3.9, §17.1.3).

E.4.4. A kezdeti értékadás és a bejárók

Az elemek számára való memóriafoglalás és a memóriaterületek kezdeti értékadása alapvető része minden tárolónak (§E.3). Ebből következik, hogy a fel nem töltött (előkészítetlen) memóriaterületen objektumot létrehozó szabványos eljárások – az *uninitialized_fill()*, az *uninitialized_fill_n()* és az *uninitialized_copy()* (§19.4.4) – semmiképpen sem hagyhatnak létrehozott objektumokat a memóriában, ha kivételt váltanak ki. Ezek az algoritmusok erős biztosítást valósítanak meg (§E.2), amihez gyakran kell elemeket törölni, tehát az a követelmény, miszerint a destruktoroknak tilos kivételt kiváltaniuk, elengedhetetlen ezeknél a függvényeknél is (lásd §E.8[14]). Ezenkívül azoknak a bejáróknak is megfelelően kell viselkedniük, melyeket paraméterként adunk át ezeknek az eljárásoknak. Tehát érvényes bejáróknak kell lenniük, érvényes sorozatokra kell hivatkozniuk, és a bejáró műveleteknek (például a ++, a != vagy a * operátornak) nem szabad kivételt kiváltaniuk, ha érvényes bejárókra alkalmazzuk azokat.

A bejárók (iterátorok) olyan objektumok, melyeket a szabványos algoritmusok és a szabványos tárolók műveletei szabadon lemásolhatnak, tehát ezek másoló konstruktora és másoló értékadása nem eredményezhet kivételt. A szabvány garantálja, hogy a szabványos tárolók által visszaadott bejárók másoló konstruktora és másoló értékadása nem vált ki kivételt, így a *vector<T>::begin()* által visszaadott bejárót például nyugodtan lemásolhatjuk, nem kell kivételtől tartanunk.

Figyeljünk rá, hogy a bejárókra alkalmazott ++ vagy -- művelet eredményezhet kivételt. Például egy *istreambuf_iterator* (§19.2.6) egy bemenethibát (logikusan) egy kivétel kiváltásával jelezhet, egy tartományellenőrzött bejáró pedig teljesen szabályosan jelezheti kivétellel azt, hogy megpróbáltunk kilépni a megengedett tartományból (§19.3). Akkor azonban nem eredményezhetnek kivételt, ha a bejárót úgy irányítjuk át egy sorozat egyik eleméről a másikra, hogy közben a ++ vagy a -- egyetlen szabályát sem sértjük meg. Tehát az *uninitialized_fill()*, az *uninitialized_fill_n()* és az *uninitialized_copy()* feltételezi, hogy a bejárókra alkalmazott ++ és -- művelet nem okoz kivételt. Ha ezt mégis megteszik, a szabvány megfogalmazása szerint ezek nem is igazán bejárók, vagy az általuk megadott „sorozat" nem értelmezhető sorozatként. Most is igaz, hogy a szabvány nem képes megvédeni a felhasználót a saját maga által okozott nem meghatározható viselkedéstől (§E.2).

E.4.5. Hivatkozások elemekre

Ha elemre hivatkozó mutatót, referenciát vagy bejárót adunk át egy eljárásnak, az tönkreteheti a listát azzal, hogy az adott elemet érvénytelenné teszi:

```
void f(const X& x)
{
    list<X> lst;
    lst.push_back(x);
    list<X>::iterator i = lst.begin();
    *i = x;          // x listába másolása
    // ...
}
```

Ha az *x* változóban érvénytelen érték szerepel, a *list* destruktora nem képes hibátlanul megsemmisíteni az *lst* objektumot:

```
struct X {
    int* p;

    X() { p = new int; }
    ~X() { delete p; }
    // ...
};

void malicious()
{
    X x;
    x.p = reinterpret_cast<int*>(7);        // hibás x
    f(x);                                   // időzített bomba
}
```

Az *f()* végrehajtásának befejeztével meghívódik a *list<X>* destruktora, amely viszont meghívja az *X* destruktorát egy érvénytelen értékre. Ha megpróbáljuk a *delete p* parancsot végrehajtani egy olyan *p* értékre, amely nem *0* és nem is létező *X* típusú értékre mutat, az eredmény nem meghatározható lesz és akár a rendszer azonnali összeomlását okozhatja. Egy másik lehetőség, hogy a memória érvénytelen állapotba kerül, ami sokkal később, a program olyan részében okoz „megmagyarázhatatlan" hibákat, amely teljesen független a tényleges problémától.

Ez a hibalehetőség nem gátolja meg a programozókat abban, hogy referenciákat és bejárókat használjanak a tárolók elemeinek kezelésére, hiszen mindenképpen ez az egyik legegyszerűbb és leghatékonyabb módszer az ilyen feladatok elvégzéséhez. Mindenesetre érdemes különösen elővigyázatosnak lennünk a tárolók elemeire való hivatkozásokkal kapcsolatban. Ha egy tároló épsége veszélybe kerülhet, érdemes a kevésbé gyakorlott felhasználók számára biztonságosabb, ellenőrzött változatokat is készítenünk, például megadhatunk egy olyan eljárást, amely ellenőrzi, hogy az új elem érvényes-e, mielőtt beszúrja azt a „fontos" tárolóba. Természetesen ilyen ellenőrzéseket csak akkor végezhetünk, ha pontosan ismerjük a tárolóban tárolt elemek típusát.

Általában, ha egy tároló valamelyik eleme érvénytelenné válik, a tárolóra alkalmazott minden további művelet hibákat eredményezhet. Ez nem csak a tárolók sajátja: bármely objektum, amely valamilyen szempontból hibás állapotba kerül, a későbbiekben bármikor okozhat problémákat.

E.4.6. Predikátumok

Számos szabványos algoritmus és tároló használ olyan predikátumokat, melyeket a felhasználók adhatnak meg. Az asszociatív tárolók esetében ezek különösen fontos szerepet töltenek be: az elemek keresése és beszúrása is ezen alapul.

A szabványos tárolók műveletei által használt predikátumok is okozhatnak kivételeket, és ha ez bekövetkezik, a standard könyvtár műveletei legalább alapbiztosítást nyújtanak, de sok esetben (például az egyelemű *insert()* műveletnél) erős biztosítás áll rendelkezésünkre (§E.4.1). Ha egy tárolóján végzett művelet közben egy predikátum kivételt vált ki, elképzelhető, hogy az ott tárolt elemek nem pontosan azok lesznek, amelyeket szeretnénk, de mindenképpen érvényes elemek. Például ha az == okoz kivételt a *list::unique()* (§17.2.2.3) művelet végrehajtása közben, nem várhatjuk el, hogy minden értékismétlődés eltűnjön. A felhasználó mindössze annyit feltételezhet, hogy a listában szereplő értékek érvényesek maradnak (lásd §E.5.3).

Szerencsére a predikátumok ritkán csinálnak olyasmit, ami kivételt eredményezhet. Ennek ellenére a felhasználói <, ==, és != predikátumokat figyelembe kell vennünk, amikor kivételbiztosságról beszélünk.

Az asszociatív tárolók összehasonlító objektumairól a *swap()* művelet végrehajtása során másolat készül (§E.4.3), ezért érdemes biztosítanunk, hogy azon predikátumok másoló műveletei, melyeket felhasználhatunk összehasonlító objektumokként, ne válthassanak ki kivételt.

E.5. A standard könyvtár további részei

A kivételbiztosság legfontosabb célja, hogy fenntartsuk az objektumok épségét és következetességét, azaz az önálló objektumok alap-invariánsa mindig igaz maradjon és az egymással kapcsolatban álló objektumok se sérüljenek. A standard könyvtár szemszögéből nézve

a kivételbiztosság fenntartása a tárolók esetében a legbonyolultabb. Ha a kivételbiztosságra összpontosítunk, a standard könyvtár többi része nem túl érdekes, de a kivételbiztosság szempontjából a beépített tömb is egy tároló, melyet felelőtlen műveletekkel könnyen tönkretehetünk.

A standard könyvtár függvényei általában csak olyan kivételeket válthatnak ki, melyeket meghatároznak vagy amelyeket az általuk meghívott felhasználói műveletek eredményezhetnek. Emellett azok az eljárások, melyek (közvetve vagy közvetlenül) memóriát foglalnak le, a memória elfogyását kivétellel jelezhetik (általában az *std::bad_alloc* kivétellel).

E.5.1. Karakterláncok

A *string* objektumokon végzett műveletek sokféle kivételt okozhatnak, a *basic_string* viszont karaktereit a *char_traits* (§20.2) osztály által biztosított függvényekkel kezeli és ezeknek nem szabad kivételt okozniuk. A standard könyvtár *char_traits* objektumai nem váltanak ki kivételeket, és ha egy felhasználói *char_traits* valamelyik eljárása eredményez ilyet, azért a standard könyvtár semmilyen felelősséget nem vállal. Különösen fontos, hogy a *basic_string* osztályban elemként (karakterként) használt típus nem rendelkezhet felhasználói másoló konstruktorral és értékadással, mert így nagyon sok kivétel-lehetőségtől szabadulunk meg.

A *basic_string* nagyon hasonlít a szabványos tárolókra (§17.5, §20.3), elemei valójában egy egyszerű sorozatot alkotnak, melyet a *basic_string<Ch,Tr,A>::iterator* vagy a *basic_string<Ch,Tr,A>::const_iterator* objektumokkal érhetünk el. Ennek következtében a *string* alapbiztosítást (§E.2) ad és az *erase()*, az *insert()*, a *push_back()* és a *swap()* (§E.4.1) függvény garanciái a *basic_string* osztály esetében is érvényesek. A *basic_string<Ch,Tr,A>::push_back()* például erős biztosítást nyújt.

E.5.2. Adatfolyamok

Ha egy adatfolyamot megfelelően állítunk be, annak függvényei az állapotváltozásokat kivételekkel jelzik (§21.3.6). Ezek jelentése pontosan meghatározott és nem okoznak kivételbiztossági problémákat. Ha egy felhasználói *operator<<()* vagy *operator>>()* eljárás okoz kivételt, az úgy jelenhet meg a programozó számára, mintha azt az *iostream* könyvtár okozta volna. Ennek ellenére ezek a kivételek nem hatnak az adatfolyam állapotára (§21.3.3). Az adatfolyam későbbi műveletei esetleg nem találják meg az általuk várt adatokat – mert egy korábbi művelet kivételt váltott ki a szabályos befejeződés helyett –, de ma-

ga az adatfolyam nem válik érvénytelenné. Szokás szerint az I/O problémák után szükség lehet a *clear()* függvény meghívására, mielőtt további írást vagy olvasást kezdeményeznénk (§21.3.3, §21.3.5).

A *basic_string* osztályhoz hasonlóan az *iostream* is egy *char_traits* objektumra hivatkozik a karakterkezelés megvalósításához (§20.2.1, §E.5.1), tehát feltételezheti, hogy a karaktereken végzett műveletek nem okoznak kivételt, illetve semmilyen biztosítást nem kell adnia, ha a felhasználó megsérti ezt a kikötést.

Ahhoz, hogy a standard könyvtár kellően hatékony optimalizálást alkalmazhasson, feltételezzük, hogy a *locale* (§D.2) és a *facet* (§D.3) objektumok sem okozhatnak kivételt. Ha mégis így működnek, akkor az azokat használó adatfolyamok érvénytelenné válhatnak. Ennek ellenére a leggyakoribb ilyen kivétel – az *std::bad_cast* a *use_facet* (§D.3.1) függvényben – csak olyan, felhasználó által írt programrészletekben fordul elő, melyek függetlenek a szabványos adatfolyamoktól, így a legrosszabb esetben is csak a kiírás félbeszakadását vagy hibás beolvasást eredményez, az adatfolyam (legyen az akár *istream*, akár *ostream*) érvényes marad.

E.5.3. Algoritmusok

Eltekintve az *uninitialized_copy()*, az *uninitialized_fill()* és az *uninitialized_fill_n()* függvénytől (§E.4.4) a standard könyvtár az algoritmusokhoz alapbiztosítást (§E.2) ad. Ez azt jelenti, hogy ha a felhasználó által megadott objektumok a követelményeknek megfelelően viselkednek, az algoritmusok fenntartják a standard könyvtár invariánsait és elkerülik az erőforrás-lyukakat. A nem meghatározott viselkedés elkerülése érdekében a felhasználói műveleteknek mindig érvényes állapotban kell hagyniuk paramétereiket és a destruktoroknak nem szabad kivételeket kiváltaniuk.

Az algoritmusok maguk nem okoznak kivételeket, ehelyett visszatérési értékükön keresztül jelzik a problémákat. A kereső algoritmusok például többnyire a sorozat végét adják vissza annak jelzésére, hogy nem találták meg a keresett elemet (§18.2). Tehát a szabványos algoritmusokban keletkező kivételek valójában mindig egy felhasználói eljárásból származnak. Ez azt jelenti, hogy a kivétel vagy az egyik elemen végzett művelet – predikátum (§18.4), értékadás vagy *swap()* – közben jött létre, vagy egy memóriafoglaló (§19.4) okozta.

Ha egy ilyen művelet kivételt okoz, az algoritmusok azonnal befejezik működésüket és az algoritmust elindító függvény feladata lesz, hogy a kivételt kezelje. Néhány algoritmus esetében előfordulhat, hogy a kivétel akkor következik be, amikor a tároló állapota a felhasználó szempontjából elfogadhatatlan. Néhány rendező eljárás például az elemeket ideigle-

nesen egy átmeneti tárba másolja és később innen teszi azokat vissza az eredeti tárolóba. Egy ilyen *sort()* eljárás esetleg sikeresen kimásolja az elemeket a tárolóból (azt tervezve, hogy hamarosan a megfelelő sorrendben írja azokat vissza), helyesen végzi el a törlést is, de ezután azonnal kivétel következik be. A felhasználó szempontjából a tároló teljesen megsemmisül, ennek ellenére minden elem érvényes állapotban van, tehát az alapbiztosítás megvalósítása egyszerű feladat.

Gondoljunk rá, hogy a szabványos algoritmusok a sorozatokat bejárókon keresztül érik el, sohasem közvetlenül a tárolókon dolgoznak, hanem azok elemein. A tény, hogy ezek az algoritmusok soha nem közvetlenül vesznek fel elemeket egy tárolóba vagy törölnek elemeket onnan, leegyszerűsíti annak vizsgálatát, hogy egy kivételnek milyen következményei lehetnek. Ha egy adatszerkezetet csak konstans bejárókon, mutatókon vagy referenciákon (például *const Rec**) keresztül érhetünk el, általában nagyon egyszerűen ellenőrizhetjük, hogy a kivételek művelnek-e valamilyen veszélyes dolgot.

E.5.4. A valarray és a complex

A számkezelő függvények sem okoznak kifejezetten kivételeket (22. fejezet), de a *valarray* osztálynak memóriát kell foglalnia, így használatakor előfordulhat *std::bad_alloc* kivétel. Ezenkívül a *valarray* és a *complex* kaphat olyan elemtípust is (skalárokat), amely kivételeket válthat ki. Szokás szerint a szabvány alapbiztosítást (§E.2) nyújt, de a kivételek által megszakadt számítások eredményéről semmit sem feltételezhetünk.

A *basic_string* osztályhoz hasonlóan (§E.5.1) a *valarray* és a *complex* is feltételezheti, hogy a sablonparaméterében megadott típus nem rendelkezik felhasználói másoló műveletekkel, tehát egyszerűen, bájtonként másolható. A standard könyvtár numerikus típusainak többsége a sebességre optimalizált, így feltételezi, hogy elemtípusai nem okoznak kivételeket.

E.5.5. A C standard könyvtára

A standard könyvtár kivétel-meghatározás nélküli műveletei az adott C++-változattól függően válthatnak ki kivételeket, a C standard könyvtárának függvényeinél azonban biztosak lehetünk abban, hogy csak akkor okoznak kivételeket, ha a nekik paraméterként átadott eljárások kivételt okoznak, hiszen végeredményben ezeket a függvényeket C programok is használják és a C-ben nincsenek kivételek. Egy szép megvalósítás a szabványos C függvényeket üres *kivétel-meghatározással* adhatja meg (*throw()*), ezzel lehetőséget adhat a fordítónak jobb kód előállítására.

Az olyan függvények, mint a *qsort()* vagy a *bsearch()*, egy függvényre hivatkozó mutatót vesznek át paraméterként, így okozhatnak kivételt, ha paraméterük képes erre. Az alapbiztosítás (§E.2) ezekre a függvényekre is kiterjed.

E.6. Javaslatok a könyvtár felhasználói számára

A standard könyvtár vizsgálatakor a kivételbiztosságra úgy tekinthetünk, mint egy problémamentesítő eszközre, amely sok mindentől megvéd minket, ha nem okozunk saját magunknak kellemetlenségeket. A könyvtár mindaddig helyesen fog működni, amíg a felhasználói eljárások teljesítik az alapkövetelményeket (§E.2). A szabványos tárolók műveletei által kiváltott kivételek többnyire nem okoznak memória-elszivárgást és a tárolót érvényes állapotban hagyják. Tehát a könyvtár használóinak a legfontosabb kérdés a következő: hogyan határozzuk meg saját típusainkat ahhoz, hogy elkerüljük a kiszámíthatatlan viselkedést és a memória-lyukak keletkezését?

Az alapszabályok a következők:

1. Amikor egy objektumot frissítünk, soha ne módosítsuk az eredeti ábrázolást addig, amíg az új értéket teljesen létre nem hoztuk és nem biztosítottuk, hogy kivétel veszélye nélkül le tudjuk cserélni az értéket. Példaképpen nézzük meg a *vector::operator=()*, a *safe_assign()* vagy a *vector::push_back()* függvény megvalósítását az §E.3 pontban.
2. Mielőtt kivételt váltunk ki, szabadítsunk fel minden olyan lefoglalt erőforrást, amelyet nem kötöttünk (más) objektumhoz.
   2a A „kezdeti értékadás az erőforrás megszerzésével” módszer (§14.4) és a nyelv szabályai, melyek szerint a részben létrehozott objektumok olyan mértékben törlődnek, amennyire létrejöttek (§14.4.1), nagyban elősegítik ezt a célt. Példaképpen nézzük meg a *leak()* függvényt. (§E.2).
   2b Az *uninitialized_copy()* és testvérei automatikus erőforrás-felszabadítást tesznek lehetővé, ha egy objektumhalmaz létrehozása nem sikerül (§E.4.4).
3. Mielőtt kivételt váltunk ki, ellenőrizzük, hogy minden operandus érvényes állapotban van-e, azaz minden objektumot olyan állapotban kell hagynunk, hogy az később szabályosan elérhető és törölhető legyen anélkül, hogy nem meghatározható eredményeket kapnánk és a destruktornak kivételt kellene kiváltania. Példaképpen a *vector* értékadását említhetjük (§E.3.2).

3a A konstruktorok abban is eltérnek az átlagos eljárásoktól, hogy ha ezekben keletkezik kivétel, nem jön létre objektum, amelyet később törölnünk kellene. Ebből következik, hogy ha egy konstruktorban kell kivételt kiváltanunk, akkor nem kell invariánst helyreállítanunk, de minden erőforrást fel kell szabadítanunk, amit a konstruktor megszakadása előtt lefoglaltunk.
3b A destruktorok abban különböznek a többi művelettől, hogy ha itt kivétel keletkezik, szinte biztosan elrontunk valamilyen invariánst és akár a *terminate()* függvény azonnali meghívását is előidézhetjük.

A gyakorlatban ezeket a szabályokat meglepően nehéz betartani. Ennek legfőbb oka az, hogy a kivételek gyakran ott következnek be, ahol egyáltalán nem várjuk azokat. Egy jó példa erre az *std::bad_alloc*. Bármely függvény okozhatja ezt a kivételt, amely közvetve vagy közvetlenül használja a *new* operátort vagy egy *allocator* objektumot memória lefoglalásához. Bizonyos programokban ezt a hibát elkerülhetjük, ha nem igénylünk a lehetségesnél több memóriát, az olyan programok esetében azonban, amelyek elég sokáig futnak vagy jelentős mennyiségű adatot kell feldolgozniuk, fel kell készülnünk a legkülönbözőbb hibákra az erőforrás-foglalásokkal kapcsolatban. Ez azt jelenti, hogy feltételeznünk kell, hogy minden függvény képes bármely kivétel kiváltására, amíg mást nem bizonyítottunk rájuk.

A meglepetések elkerülésének egyik módja az, hogy csak olyan elemekből építünk tárolókat, melyek nem használnak kivételeket (például mutatókból vagy egyszerű, konkrét típusokból) vagy láncolt tárolókat (például *list*) használunk, melyek erős biztosítást nyújtanak (§E.4). A másik, ellentétes megközelítés, hogy elsősorban az olyan műveletekre számítunk, melyek erős biztosítást nyújtanak (például a *push_back()*). Ezek vagy sikeresen befejeződnek, vagy egyáltalán nincs hatásuk (§E.2), de önmagukban nem elegendőek az erőforrás-lyukak elkerülésére és csak rendezetlen, pesszimista hibakezelést és helyreállítást tesznek lehetővé. A *vector<T*>* például típusbiztos, ha a *T* típuson végzett műveletek nem okoznak kivételeket, de ha kivétel következik be a *vector* objektumban és nem gondoskodunk valahol a mutatott objektumok törléséről, azonnal memória-lyukak keletkeznek. Ebből következik, hogy be kell vezetnünk egy *Handle* osztályt, amely mindig elvégzi a szükséges felszabadításokat (§25.7), és az egyszerű *vector<T*>* helyett a *vector< Handle<T> >* szerkezetet kell használnunk. Ez a megoldás az egész programot rugalmasabbá teszi.

Amikor új programot készítünk, lehetőségünk van arra, hogy átgondoltabb megközelítést találjunk és biztosítsuk, hogy erőforrásainkat olyan osztályokkal ábrázoljuk, melyek invariánsa alapbiztosítást nyújt (§E.2). Egy ilyen rendszerben lehetőség nyílik arra, hogy kiválasszuk a létfontosságú objektumokat és ezek műveleteihez visszagörgetési módszereket alkalmazzunk (azaz erős biztosítást adhatunk – néhány egyedi feltétel mellett).

A legtöbb program tartalmaz olyan adatszerkezeteket és programrészeket, melyeket a kivételbiztosságra nem gondolva írtak meg. Ha szükség van rá, ezek a részek egy kivételbiztos keretbe ágyazhatók. Az egyik lehetőség, hogy biztosítjuk, hogy kivételek ne következzenek be (ez történt a C standard könyvtárával, §E.5.5), a másik megoldás pedig az, hogy felületosztályokat használunk, melyekben a kivételek viselkedése és az erőforrások kezelése pontosan meghatározható.

Amikor olyan új típusokat tervezünk, amelyek kivételbiztos környezetben futnak majd, külön figyelmet kell szentelnünk azoknak az eljárásoknak, melyeket a standard könyvtár használni fog: a konstruktoroknak, a destruktoroknak, az értékadásoknak, összehasonlításoknak, *swap* függvényeknek, a predikátumként használt függvényeknek és a bejárókat kezelő eljárásoknak. Ezt legkönnyebben úgy valósíthatjuk meg, hogy egy jó osztályinvariánst határozunk meg, amelyet minden konstruktor könnyedén biztosíthat. Néha úgy kell megterveznünk az osztályinvariánst, hogy az objektumoknak legyen egy olyan állapota, melyben egyszerűen törölhetők, ha egy művelet „kellemetlen" helyen ütközik hibába. Ideális esetben ez az állapot nem egy mesterségesen megadott érték, amit csak a kivételkezelés miatt kellett bevezetni, hanem az osztály természetéből következő állapot (§E.3.5).

Amikor kivételbiztossággal foglalkozunk, a fő hangsúlyt az objektumok érvényes állapotainak (invariánsainak) meghatározására és az erőforrások megfelelő felszabadítására kell helyeznünk. Ezért nagyon fontos, hogy az erőforrásokat közvetlenül osztályokkal ábrázoljuk. A *vector_base* (§E.3.2) ennek egyszerű példája. Az ilyen erőforrás-osztályok konstruktora alacsonyszintű erőforrásokat foglal le (például egy memóriatartományt a *vector_base* esetében), és invariánsokat állít be (például a mutatókat a megfelelő helyekre állítja a *vector_base* osztályban). Ezen osztályok destruktora egyszerűen felszabadítja a lefoglalt erőforrást. A részleges létrehozás szabályai (§14.4.1) és a „kezdeti értékadás az erőforrás lefoglalásával" módszer (§14.4) alkalmazása lehetővé teszi, hogy az erőforrásokat így kezeljük.

Egy jól megírt konstruktor minden objektum esetében beállítja a megfelelő invariánst (§24.3.7.1), tehát a konstruktor olyan értéket ad az objektumnak, amely lehetővé teszi, hogy a további műveleteket egyszerűen meg tudjuk írni és sikeresen végre tudjuk hajtani. Ebből következik, hogy a konstruktoroknak gyakran kell erőforrást lefoglalniuk. Ha ezt nem tudják elvégezni, kivételt válthatnak ki, így az objektum létrehozása előtt foglalkozhatunk a jelentkező problémákkal. Ezt a megközelítést a nyelv és a standard könyvtár közvetlenül támogatja (§E.3.5).

Az a követelmény, hogy az erőforrásokat fel kell szabadítanunk és az operandusokat érvényes állapotban kell hagynunk a kivétel kiváltása előtt, azt jelenti, hogy a kivételkezelés terheit megosztjuk a kivételt kiváltó függvény, a hívási láncban levő függvények és a kivételt ténylegesen kezelő eljárás között. Egy kivétel kiváltása nem azt a hibakezelési stílust jelen-

ti, hogy „hagyjuk az egészet valaki másra”. Minden függvénynek, amely kivételt vált ki vagy ad tovább, kötelessége felszabadítani azokat az erőforrásokat, melyek hatáskörébe tartoznak, operandusait pedig megfelelő értékre kell állítania. Ha az eljárások ezt a feladatot nem képesek végrehajtani, a kivételkezelő nemigen tehet mást, minthogy megpróbálja „szépen” befejezni a program működését.

E.7. Tanácsok

[1] Legyünk tisztában azzal, milyen szintű kivételbiztosságra van szükségünk. §E.2.
[2] A kivételbiztosságnak egy teljes körű hibatűrési stratégia részének kell lennie. §E.2.
[3] Az alapbiztosítást minden osztályhoz érdemes megvalósítani, azaz az invariánsokat mindig tartsuk meg és az erőforrás-lyukakat mindig kerüljük el. §E.2, §E.3.2, §E.4.
[4] Ahol lehetőség és szükség van rá, valósítsunk meg erős biztosítást, azaz egy művelet vagy sikeresen hajtódjon végre, vagy minden operandusát hagyja változatlanul. §E.2, §E.3.
[5] Destruktorokban ne fordulhasson elő kivétel. §E.2, §E.3.2, §E.4.
[6] Ne váltson ki kivételt egy érvényes sorozatban mozgó bejáró. §E.4.1, §E.4.4.
[7] A kivételbiztosság foglalja magában az önálló műveletek alapos vizsgálatát. §E.3.
[8] A sablon osztályokat úgy tervezzük meg, hogy azok „átlátszóak” legyenek a kivételek számára. §E.3.1.
[9] Az *init()* függvény helyett használjunk konstruktort az erőforrások lefoglalásához. §E.3.5.
[10] Adjunk meg invariánst minden osztályhoz, hogy ezzel pontosan meghatározzuk érvényes állapotaikat. §E.2, §E.6.
[11] Győződjünk meg róla, hogy objektumaink mindig érvényes állapotba állíthatók anélkül, hogy kivételektől kellene tartanunk. §E.3.2, §E.6.
[12] Az invariánsok mindig legyenek egyszerűek. §E.3.5.
[13] Kivétel kiváltása előtt minden objektumot állítsunk érvényes állapotba. §E.2, §E.6.
[14] Kerüljük el az erőforrás-lyukakat. §E.2, §E.3.1, §E.6.
[15] Az erőforrásokat közvetlenül ábrázoljuk. §E.3.2, §E.6.
[16] Gondoljunk rá, hogy a *swap()* függvény gyakran használható az elemek másolása helyett. §E.3.3.

[17] Ha lehetőség van rá, a *try* blokkok használata helyett a műveletek sorrendjének jó megválasztásával kezeljük a problémákat. §E.3.4.
[18] Ne töröljük a „régi” információkat addig, amíg a helyettesítő adatok nem válnak biztonságosan elérhetővé. §E.3.3, §E.6.
[19] Használjuk a „kezdeti értékadás az erőforrás megszerzésével” módszert. §E.3, §E.3.2, §E.6.
[20] Vizsgáljuk meg, hogy asszociatív tárolóinkban az összehasonlító műveletek másolhatók-e. §E.3.3.
[21] Keressük meg a létfontosságú adatszerkezeteket és ezekhez adjunk meg olyan műveleteket, melyek erős biztosítást adnak. §E.6.

E.8. Gyakorlatok

1. (*1) Soroljuk fel az összes kivételt, amely előfordulhat az §E.1 pont *f()* függvényében.
2. (*1) Válaszoljunk az §E.1 pontban, a példa után szereplő kérdésekre.
3. (*1) Készítsünk egy *Tester* osztályt, amely időnként a legalapvetőbb műveletekben okoz kivételt, például a másoló konstruktorban. A *Tester* osztály segítségével próbáljuk ki saját standard könyvtárunk tárolóit.
4. (*1) Keressük meg a hibát az §E.3.1 pontban szereplő *vector* konstruktorának rendezetlen változatában és írjunk programot, amely tönkreteszi az osztályt. Ajánlás: először írjuk meg a *vector* destruktorát.
5. (*2) Készítsünk egyszerű listát, amely alapbiztosítást nyújt. Állapítsuk meg nagyon pontosan, milyen követelményeket kell a felhasználónak teljesítenie a biztosítás megvalósításához.
6. (*3) Készítsünk egyszerű listát, amely erős biztosítást nyújt. Alaposan ellenőrizzük az osztály működését. Indokoljuk meg, miért tartjuk ezt a megoldást biztonságosabbnak.
7. (*2.5)Írjuk újra a §11.12 *String* osztályát úgy, hogy ugyanolyan biztonságos legyen, mint a szabványos tárolók.
8. (*2) Hasonlítsuk össze a *vector* osztályban meghatározott értékadás és a *safe_assign()* függvény különböző változatait a futási idő szempontjából. (§E.3.3)
9. (*1.5) Másoljunk le egy memóriafoglalót az értékadó operátor használata nélkül (hiszen az *operator=()* megvalósításához erre van szükségünk az §E.3.3 pontban).

10. (*2) Írjunk a *vector* osztályhoz alapbiztosítással egy egyelemű és egy többelemű *erase()*, illetve *insert()* függvényt.§E.3.2.
11. (*2) Írjunk a *vector* osztályhoz erős biztosítással egy egyelemű és egy többelemű *erase()*, illetve *insert()* függvényt (§E.3.2). Hasonlítsuk össze ezen függvények költségét és bonyolultságát az előző feladatban szereplő függvényekével.
12. (*2) Készítsünk egy *safe_insert()* függvényt (§E.4.2), amely egy létező *vector* objektumba szúr be elemet (nem pedig egy ideiglenes változót másol le). Milyen kikötéseket kell tennünk a műveletekre?
13. (*2.5) Hasonlítsuk össze méret, bonyolultság és hatékonyság szempontjából a 12. és a 13. feladatban szereplő *safe_insert()* függvényt az §E.4.2 pontban bemutatott *safe_insert()* függvénnyel.
14. (*2.5) Írjunk egy jobb (gyorsabb és egyszerűbb) *safe_insert()* függvényt, kifejezetten asszociatív tárolókhoz. Használjuk a *traits* eljárást egy olyan *safe_insert()* megvalósításához, amely automatikusan kiválasztja az adott tárolóhoz optimális megvalósítást. Ajánlás: §19.2.3.
15. (*2.5) Próbáljuk megírni az *uninitialized_fill()* függvényt (§19.4.4, §E.3.1) úgy, hogy az megfelelően kezelje a kivételeket kiváltó destruktorokat is. Lehetséges ez? Ha igen, milyen áron? Ha nem, miért nem?
16. (*2.5) Keressünk egy tárolót egy olyan könyvtárban, amely nem tartozik a szabványhoz. Nézzük át dokumentációját és állapítsuk meg, milyen kivételbiztossági lehetőségek állnak rendelkezésünkre. Végezzünk néhány tesztet, hogy megállapítsuk, mennyire rugalmas a tároló a memóriafoglalásból vagy a felhasználó által megadott programrészekből származó kivételekkel szemben. Hasonlítsuk össze a tapasztaltakat a standard könyvtár megfelelő tárolójának szolgáltatásaival.
17. (*3) Próbáljuk optimalizálni az §E.3 pontban szereplő *vector* osztályt a kivételek lehetőségének figyelmen kívül hagyásával. Például töröljünk minden *try* blokkot. Hasonlítsuk össze az így kapott változat hatékonyságát a standard könyvtár *vector* osztályának hatékonyságával. Hasonlítsuk össze a két változatot méret és bonyolultság szempontjából is.
18. (*1) Adjunk meg invariánst a *vector* osztály (§E.3) számára úgy, hogy megengedjük, illetve megtiltjuk a *v==0* esetet (§E.3.5).
19. (*2.5) Nézzük végig egy *vector* osztály megvalósításának forráskódját. Milyen biztosítás áll rendelkezésünkre az értékadásban, a többelemű *insert()* utasításban és a *resize()* függvényben?
20. (*3) Írjuk meg a *hash_map* (§17.6) olyan változatát, amely ugyanolyan biztonságos, mint a szabványos tárolók.