SCOTT MEYERS

Hatékony C++

50 jó tanács programjaink
és programterveink javítására

Tartalom

Előszó 8

Köszönetnyilvánítás 12

Bevezetés 18

Átállás C-ről C++-ra 30

1. jó tanács: `#define` helyett használjunk inkább `const`-ot és `inline`-t 32
2. jó tanács: Az `<stdio.h>` helyett használjuk inkább az `<iostream>`-et 35
3. jó tanács: A `malloc` és a `free` helyett használjuk a `new`-t és a `delete`-et 37
4. jó tanács: Használjunk C++ stílusú megjegyzéseket 39

Memóriakezelés 40

5. jó tanács: Az egymásnak megfelelő `new`-t és `delete`-et mindig használjuk azonos formában 42
6. jó tanács: Destruktorokban használjunk `delete`-et a pointer tagokra 43
7. jó tanács: Készüljünk fel azokra az esetekre, amikor elfogy a memória 44
8. jó tanács: Ragaszkodjunk a konvenciókhoz az `operator new` és az `operator delete` megírásakor 52
9. jó tanács: A `new` „normál" formáját sose takarjuk el 55
10. jó tanács: Ha írunk `operator new`-t, írjunk hozzá `operator delete`-et is 57

Konstruktorok, destruktorok és értékadó operátorok 66

11. jó tanács: Deklaráljunk másoló konstruktort és értékadó operátort olyan osztályokban, melyek dinamikusan allokált memóriát használnak 68
12. jó tanács: Konstruktorokban értékadás helyett válasszuk inkább az inicializálást 71

13. jó tanács: Az inicializáló listában az adattagokat a deklarációjuk sorrendjében soroljuk fel 75
14. jó tanács: A bázisosztályok destruktora mindig legyen virtuális 76
15. jó tanács: Az `operator=` egy `*this`-re hivatkozó referenciával térjen vissza 81
16. jó tanács: Az `operator=`-ben minden adattagnak adjunk értéket 85
17. jó tanács: Ellenőrizzük az önértékadást az `operator=`-ben 88

Osztályok és függvények: tervezés és deklaráció 94

18. jó tanács: Törekedjünk teljes és minimális osztályfelületek kialakítására 97
19. jó tanács: Tegyünk különbséget tagfüggvények, nemtagfüggvények és barátfüggvények között 101
20. jó tanács: Kerüljük az adattagokat nyilvános felületen 105
21. jó tanács: Amikor csak lehet, használjunk `const`-ot 107
22. jó tanács: Érték szerinti átadás helyett használjunk inkább referencia szerintit 113
23. jó tanács: Ne próbáljunk meg referenciát visszaadni akkor, amikor objektumot kell 117
24. jó tanács: Válasszunk körültekintően a függvénytúlterhelés és az alapértelmezett paraméterezés között 121
25. jó tanács: Kerüljük a túlterhelést pointereken és numerikus típusokon 125
26. jó tanács: Óvakodjunk a potenciális többértelműségtől 128
27. jó tanács: Explicit tiltsuk meg azoknak az implicit függvényeknek a használatát, amelyekre nem tartunk igényt 131
28. jó tanács: Particionáljuk a globális névteret 132

Osztályok és függvények: implementáció 138

29. jó tanács: Kerüljük a belső adatokra hivatkozó „leírók" visszaadását 140
30. jó tanács: Kerüljük az olyan tagfüggvényeket, amelyek náluk szigorúbb láthatóságú tagokra hivatkozó nem `const` pointereket vagy referenciákat adnak vissza 144
31. jó tanács: Soha ne térjünk vissza lokális objektumra, vagy egy dereferenciált pointerre hivatkozó referenciával, amely pointert a függvényen belül `new`-val inicializáltunk 147
32. jó tanács: Késleltessük a változók definiálását, amíg csak lehet 150
33. jó tanács: Használjuk az `inline`-t megfontoltan 152
34. jó tanács: Minimalizáljuk a fájlok közötti fordítási függőséget 158

Öröklődés és objektumorientált tervezés 168

35. jó tanács: Bizonyosodjunk meg arról, hogy a publikus öröklődés az „azegy" (isa) relációt fejezi ki 170
36. jó tanács: Tegyünk különbséget a felület öröklése és az implementáció öröklése között 176

37. jó tanács: Soha ne definiáljunk át egy örökölt nemvirtuális függvényt 183
38. jó tanács: Soha ne definiáljunk át egy örökölt alapértelmezett paraméterértéket 185
39. jó tanács: Az öröklődési hierarchiában kerüljük a lefelé irányuló konverziót 188
40. jó tanács: Modellezzük a „vanegy" vagy a „keresztül implementált" kapcsolatokat rétegekkel 195
41. jó tanács: Tegyünk különbséget az öröklés és a sablonok között 198
42. jó tanács: Használjuk a privát öröklődést megfontoltan 202
43. jó tanács: Használjuk a többszörös öröklődést megfontoltan 207
44. jó tanács: Azt mondjuk, amit gondolunk, és értsük, amit mondunk 221

EGYEBEK 224

45. jó tanács: Tudjunk róla, milyen függvényeket ír és hív meg titokban a C++ 226
46. jó tanács: Részesítsük előnyben a fordítási és linkelési idejű hibákat a futási idejűekkel szemben 230
47. jó tanács: Biztosítsuk, hogy a nemlokális statikus objektumok inicializálódjanak felhasználásuk előtt 233
48. jó tanács: Figyeljünk oda a fordító figyelmeztetéseire 236
49. jó tanács: Ismerjük meg a szabvány könyvtárat 237
50. jó tanács: Tökéletesítsük C++-tudásunkat 244

UTÓSZÓ 249

INDEX 251

MINI SZÓTÁR 268

Előszó

Nancynek,
aki nélkül semmit sem lenne érdemes csinálni.

Tudás és szépség ritkán járnak együtt.
Petronius Arbiter *Satyricon, XCIV*

Profi programozók C++ oktatása során szerzett tapasztalataim eredményeként született ez a könyv. Úgy vettem észre, hogy a legtöbb hallgató egy hét intenzív oktatás után otthon érzi magát a nyelv alapvető szerkezetei között, mégsem bízik igazán abban, hogy ezeket a szerkezeteket hatékonyan tudja kombinálni. Ezért fogtam hozzá olyan rövid, pontos, könnyen megjegyezhető irányelvek megalkotásához, amelyek a hatékony C++ szoftverfejlesztést hivatottak segíteni. Összegeztem azokat a dolgokat, melyeket tapasztalt C++ programozók szinte mindig alkalmaznak, illetve amelyeket majdnem mindig elkerülnek.

Kezdetben olyan szabályok érdekeltek, amelyeket egy `lint`-szerű programmal ellenőrizni lehet. E cél érdekében olyan eszközök fejlesztésének kutatásába kezdtem, amelyek a felhasználó által definiált feltételek megsértését keresik a C++ forráskódban.[1] Sajnos a kutatás abbamaradt, mielőtt működő prototípust sikerült volna készíteni. Szerencsére ma már kereskedelmi forgalomban is kaphatók C++ kódot ellenőrző termékek.

Habár eredetileg olyan programozási szabályok foglalkoztattak, amelyek automatikusan ellenőrizhetők, hamar rájöttem ennek a megközelítésnek a korlátaira. A legtöbb olyan irányelv, amivel egy jó C++ programozó tisztában van, vagy túl bonyolult ahhoz, hogy szabályt lehessen alkotni belőle, vagy túl sok olyan fontos kivétellel jár, ami miatt nem lehet programba foglalni. Így jutottam arra az elhatározásra, hogy egy számítógépes programnál kevésbé precíz, de egy átlagos C++ könyvnél mégis sokkal összefogottabb, célirányosabb megoldást keresek. Ennek eredményét tartja most kezében az olvasó: egy könyvet 50 olyan jó tanáccsal, amely segít tovább tökéletesíteni C++ programjainkat és a programtervezés folyamatát.

Ez a könyv arra nézve ad javaslatokat, hogy mit miért tegyünk, illetve, hogy mit miért kerüljünk. Alapvetően persze az indoklások fontosabbak, mint maguk a jó tanácsok, de sokkal egyszerűbb javaslatokat megjegyezni, mint tankönyv ízű indoklásokat kívülről megtanulni.

A legtöbb C++ könyvtől eltérően az itt közreadott mű nem az egyes nyelvi elemek, hanem egy-egy jó tanács köré épül, tehát nem a konstruktorokat tárgyalja az egyik he-

[1] A kutatásról található egy áttekintés a *Hatékony C++ (Effective C++)* angol nyelvű internetes oldalán, a következő címen: `http://www.awl.com/cp/ec++.html`.

lyen, a virtuális függvényeket egy másikon vagy az öröklődést egy harmadikon. Az egyes témák kifejtése a címükben szereplő irányelvhez igazodik, ezért az egyes nyelvi elemek jellegzetességeit leíró részek több különböző helyen is fellelhetők.

Előnye ennek a megközelítésnek, hogy jobban tükrözi azoknak a szoftverrendszereknek az összetettségét, melyek megírásához a C++-t gyakran választják. Az ilyen rendszerek írásakor a nyelvi elemek megértése önmagában nem elég. Tapasztalt C++ fejlesztők tudják például, hogy az inline függvények és a virtuális destruktorok megértéséből *nem* következik szükségszerűen az inline virtuális destruktorok megértése. Az ilyen harcedzett fejlesztők tisztában vannak azzal, hogy C++-ban a programozás hatékonyságának szempontjából kétségtelenül a nyelvi elemek közötti *kölcsönhatás* megértése a legfontosabb. A könyv szerkezete ezt az alapvető igazságot tükrözi.

Ennek a felépítésnek hátránya viszont az, hogy az egy-egy nyelvi elemmel kapcsolatos mondandómat az olvasónak több helyről kell összeszednie. Hogy ez a megközelítés ne okozzon annyi kényelmetlenséget, a szöveget bőségesen elláttam utalásokkal, és részletes index is található a könyv végén.

Ennek a második kiadásnak a készítésekor az elsőnél mindenképpen jobb változattal akartam előállni, de aggodalmaim is voltak emiatt. Programozók tízezrei szerették meg az első kiadást, és nem akartam megváltoztatni a könyv egyik olyan tulajdonságát sem, amelyik számukra vonzó volt. A könyv első kiadásának megírása óta eltelt hat év alatt azonban változott a C++, változott a C++ könyvtár (lásd a 49. jó tanácsot), változott az én felfogásom is a C++-ról, de a C++ elfogadott használata is megváltozott. Sok minden változott, én pedig fontosnak tartottam, hogy a *Hatékony C++* szakmai anyagának átdolgozása mindezt tükrözze. Minden tőlem telhetőt megtettem, hogy oldalról oldalra haladva aktualizáljam az új kiadást, de a könyv és a szoftver kísértetiesen hasonlít egymásra abban, hogy eljön az idő, amikor a helyi javítgatások már nem elegendőek, és az egyetlen megoldás a teljes átírás. Ez a könyv ennek az átírásnak az eredménye: *Hatékony C++* 2.0 verzió.

Az első kiadás ismerőinek érdekes lehet, hogy az összes jó tanácsot átdolgoztam. A könyv átfogó szerkezetét ez azonban nem befolyásolja, mert abban nagyon kevés a változás. Az eredeti 50 fejezetből 48-at megtartottam, habár néhány fejezetcím szövegezésén változtattam (azon felül, hogy átdolgoztam az azt kísérő tárgyalást). A 32. és a 49. jó tanácsot kivettem, ezek helyére teljesen új anyag került, habár a 32. jó tanácsból sok információ átkerült az átdolgozott 1. jó tanácsba. A 41. és 42. jó tanácsot felcseréltem, mert ebben a sorrendben az átdolgozás miatt érthetőbbek. Végül, megfordítottam az öröklődést jelző nyilaim irányát is. Most már a majdnem mindenütt használatos konvenciót követik, azaz a leszármazott osztálytól mutatnak a bázisosztályra. Ugyanehhez a szabályhoz igazodtam az 1996-os *More Effective C++* c. könyvemben, melyről egy rövid áttekintés található e kötet 250–252. oldalán.

Az itt összegyűjtött irányelvek listája messze nem teljes, de jóval nehezebb jó szabályokat meghatározni – olyanokat, amelyek majdnem minden területen alkalmazhatók, majdnem mindig –, mint amilyennek ez látszik. Lehet, hogy az olvasó ismer más irányelveket is, vagy más módszereket, melyek segítségével hatékonyan lehet C++-ban programozni. Ha így van, örömmel hallanék ezekről.

Persze az is lehet, hogy az olvasó úgy gondolja, a könyv bizonyos fejezetei nem tekinthetők egyetemes érvényűnek; hogy van jobb módszer egy-egy példafeladat megoldására; vagy egyes szakmai fejtegetések hiányosak, félrevezetők vagy nem elég világosak. Bátorítom az olvasót, hogy tájékoztasson ezekről is.

Donald Knuth már régóta jutalmat ajánl fel azoknak, akik figyelmeztetik a könyveiben előforduló hibákra. A tökéletes könyvre való törekvés már alapjában dicséretes dolog, de tekintettel a piacot elárasztó, hibáktól hemzsegő C++ könyvekre, nagyon erős bennem a késztetés arra, hogy Knuth példáját kövessem. Ezért a jövőbeli nyomtatásokban minden felém jelzett hibáért cserébe – legyen az szakmai, nyelvtani, sajtóhiba vagy bármi egyéb – örömmel feltüntetem a köszönetnyilvánításban annak a nevét, aki elsőként hívja fel a figyelmemet egy hibára.

Kérem, küldje megjegyzéseit, kritikáját, a javasolt irányelveket és – hát, igen! – a hibák leírását a következő címre:

Scott Meyers
c/o Publisher, Corporate and Professional Publishing
Addison Wesley Longman, Inc.
1 Jacob Way
Reading, MA 01867
U. S. A.

Levél elektronikusan is küldhető az `ec++@awl.com` címre.

Listát vezetek a könyv első nyomtatása óta történt változtatásokról, beleértve a hibajavításokat, magyarázatokat, szakmai szempontú frissítéseket. Ez a lista elérhető az Effective C++ weboldalán, a `http://www.awl.com/cp/ec++.html` címen. Ha valaki szeretne egy ilyen listát, de nincs internetelérése, küldje el igényét a fenti címek valamelyikére, és gondoskodom arról, hogy eljusson hozzá a lista.

Scott Douglas Meyers

Stafford, Oregon
1997 júliusa

Bevezetés

Egy programozási nyelv alapjainak elsajátítása, valamint az, hogy *hatékony* programokat tervezzünk és implementáljunk ezen a nyelven, két egymástól teljesen különböző dolog. Ez különösen igaz a C++-ra, amely a hatókör *(scope)* és a kifejezés lehetőségeinek tekintetében egyedülállóan nagy választékkal rendelkezik. Egy teljes körű szolgáltatást nyújtó konvencionális programozási nyelvre (C) épül, ugyanakkor támogatja az objektumorientált szolgáltatások széles körét, csakúgy, mint a sablonokat *(templates)* és a kivételkezelést.

Megfelelően használva, a C++-szal öröm dolgozni. Sokféle – mind objektumorientált, mind hagyományos – programterv kifejezésére és hatékony implementálására alkalmas. Új adattípusokat definiálhatunk, amelyeket szinte meg sem lehet különböztetni beépített hasonmásaiktól, mégis lényegesen rugalmasabbak. Egy megfontoltan kiválasztott és gondosan elkészített osztálykészlet – amely automatikusan kezeli a memóriagazdálkodást, az objektumra álnéven *(alias)* történő hivatkozást, az inicializálást és felszabadítást, a típuskonverziót és az összes többi olyan veszedelmet, amely egy programozóra leselkedik – könnyűvé, ötletessé, hatékonnyá és majdnem hibamentessé teheti az alkalmazásfejlesztést. Nem nehéz hatékony C++ programokat írni, *ha* tudjuk, hogyan kell.

A fegyelmezetlenül használt C++ olyan kódot eredményez, amely érthetetlen, használhatatlan, nem lehet karbantartani, nem fejleszthető tovább, azaz egész egyszerűen rossz.

Érdemes felfedeznünk a C++ lehetséges buktatóit, hogy aztán megtanulhassuk elkerülni azokat. Ez a könyv ezzel a céllal készült. Feltételezem, hogy az olvasó ismeri a C++-t mint *nyelvet* és van már némi gyakorlata a használatában. Könyvem a nyelv *hatékony* használatához készített kalauz, mely azzal a céllal készült, hogy a szoftverünk érthető, karbantartható, bővíthető legyen és lehetőleg az elvárásainknak megfelelően működjön.

A tanácsaim két nagy kategóriába sorolhatók: általános tervezési stratégiákra és a nyelv egyes elemeinek gyakorlati alkalmazására.

A tervezés tárgyalása arra összpontosít, hogy a C++ adta megvalósítási lehetőségek közül hogyan válasszunk. Mit válasszunk, öröklődést vagy sablonokat? Sablonokat vagy generikus pointereket? Privát vagy publikus öröklődést? Privát öröklődést vagy beágyazást *(layering)*? Függvénytúlterhelést *(function overloading)* vagy paraméter alapértelmezést? Virtuális vagy nemvirtuális függvényeket? Érték szerinti vagy referencia szerinti pa-

raméterátadást? Fontos, hogy ezeket a döntéseket rögtön az elején helyesen hozzuk meg, mert előfordulhat, hogy a helytelen választás csak a fejlesztés későbbi szakaszában derül ki, amikor a helyesbítés már nehéz, időigényes, demoralizáló és drága.

Még ha pontosan tudjuk is, mit akarunk, akkor sem biztos, hogy olyan egyszerű elérni, hogy minden zökkenőmentesen menjen. Mi az értékadó operátor *(assignment operator)* helyes visszatérési típusa? Hogyan kell az `operator new`-nak működni, ha nem talál elég memóriát? Mikor legyen a destruktor virtuális? Hogyan kell megírni egy taginicializáló listát? Végig kell küzdenünk magunkat ezeken a részleteken, különben a programunk nagy valószínűséggel meglepetésszerű, sőt megtévesztő viselkedést fog produkálni. Sőt, lehet, hogy a hibás működés ki sem derül azonnal, a kódban a rejtett hibáknak menedéket nyújtó kódkísértet pedig átsiklik a minőségellenőrzésen: a hibák időzített bombaként ketyegnek tovább, amíg fel nem robbannak.

Ez egy olyan könyv, amely nem csak akkor érthető, ha az elejétől olvassuk el a végéig. 50 jó tanácsból áll, amelyek többé-kevésbé megállják a helyüket önmagukban is. Gyakran hivatkoznak a fejezetek egymásra, ezért úgy is lehet olvasni a könyvet, hogy egy minket érdeklő fejezettel kezdjük és követjük a hivatkozásokat.

A jó tanácsokat témakörök szerint csoportosítottam, így ha egy konkrét terület iránt érdeklődünk, mint például a memóriakezelés vagy az objektumorientált tervezés, fogjunk az idevágó rész olvasásába, és azt vagy egyben olvassuk végig, vagy innen kezdjük el követni a hivatkozásokat. Ki fog azonban derülni, hogy a könyv teljes anyaga elég alapvető a hatékony C++ programozáshoz, ezért, így vagy úgy, minden mindennel összefügg.

Ez nem egy C++ kézikönyv, nem lehet belőle teljesen kezdőként a nyelvet megtanulni. Lelkesen elmagyarázok például mindent egy saját `operator new` verzió megírásával kapcsolatban (lásd a 7–10. jó tanácsot), de feltételezem, hogy az olvasó más forrásból is meg tudja tanulni, hogy ennek a függvénynek `void*`-ot kell visszaadnia és az első paramétere `size_t` típusú kell, hogy legyen. Számos olyan C++-ról szóló alapkönyv létezik, amelyben ez az információ megtalálható.

Ez a könyv a C++ programozásnak azokra a vonásaira tér ki, amelyeket általában felszínesen tárgyalnak, ha egyáltalán említik őket. Vannak könyvek, amelyek a nyelv alkotórészeit írják le. Jelen mű arról szól, hogyan készíthetünk ezekből az alkotórészekből hatékony programokat. Vannak könyvek, amelyekből megtudhatjuk, hogyan tehetjük a programjainkat fordíthatóvá. Ebből a könyvből az is kiderül, hogyan kerülhetjük el azokat a problémákat, amelyeket a fordítóprogram elhallgat előlünk.

A legtöbb nyelvhez hasonlóan a C++-nak is programozóról programozóra szálló, gazdag hagyományvilága van, azaz léteznek szájhagyomány útján terjedő nyelvi tradíciói is. Ezúton próbálkozom meg azzal, hogy lejegyezzem és hozzáférhetőbb formába öntsem ezt a felgyülemlett bölcsességet.

A könyv anyaga ugyanakkor a szabályokhoz igazodó, *hordozható* C++ korlátain belül marad. Csak az ISO/ANSI nyelvi szabványnak megfelelő elemekkel dolgozik. Mivel a hordozhatóság kulcskérdésként szerepel ebben a könyvben, az, aki implementációfüggő tákolmányokat keres, itt nem fogja megtalálni.

Sajna, a szabványban leírt C++ néha különbözik attól, amit a fordítóprogramunk nagyon barátságos helyi szállítója favorizál. Ezért, amikor olyan területekre tévedek, ahol a viszonylag új nyelvi lehetőségekre haszonnal támaszkodhatunk, akkor arra is kitérek, hogyan készíthetünk hatékony szoftvert ezek hiányában. Butaság lenne figyelmen kívül hagynunk azt, amit a jövő már bizonyosan magában hordoz, ugyanakkor nem állhatunk

le azért, mert a legkorszerűbb, legpompásabb, mindentudóként véglegesített C++ fordítók még nincsenek a gépünkre telepítve. Azokkal az eszközökkel kell dolgoznunk, amelyek rendelkezésünkre állnak, ez a könyv pedig ebben segít.

Nem véletlenül írtam *fordítókat*, többes számban. Az egyes fordítóprogramok a szabvány más-más megközelítését valósítják meg, ezért arra bátorítom az olvasót, hogy legalább két fordítóprogrammal fejlesszen. Ha így tesz, véletlenül sem lesz kiszolgáltatva egy adott szállító kizárólagosan alkalmazott nyelvi kiterjesztéseinek, vagy annak, hogy a szállító a szabványt félreértelmezte. Ez a módszer abban is segít, hogy kellő távolságot tartsunk a fordítóprogramok legmodernebb technológiai fejlesztéseitől, azaz az olyan nyelvi elemektől, melyeket csak egy szállító támogat. Ezeket a nyelvi elemeket legtöbbször rosszul implementálják (hibásak vagy lassúak, gyakran mindkettő), és bevezetésükkor a C++-t használók közösségének még nincs elég tapasztalata ahhoz, hogy tanácsot adhasson a helyes használatra vonatkozóan. Az úttörő munka izgalmas lehet, de ha megbízható kódot akarunk készíteni, az a legcélravezetőbb, ha másra hagyjuk a bozótvágást.

Ebben a kiadványban *nem* találja meg senki a C++ evangéliumát, a tökéletes C++ szoftverhez vezető „egyetlen helyes ösvényt". Az 50 jó tanács útmutatásainak segítségével javíthatunk programterveinken, elkerülhetjük a gyakran felmerülő problémákat, nagyobb hatékonyságot érhetünk el, de egyik jó tanács sem egyetemes érvényű. A szoftvertervezés és -implementálás összetett feladat, melynek során a hardver, az operációs rendszer és az alkalmazás korlátai állandóan színesítik a munkát, így leginkább annyit tehetek, hogy *irányelveket* adok jobb programok írásához.

Ha mindig követjük az irányelveket, valószínűleg nem esünk áldozatul a C++ leggyakoribb csapdáinak, de az irányelvek alól – természetükből adódóan – vannak kivételek. Ezért kapcsolódik minden jó tanácshoz magyarázat is. A magyarázatok a könyv legfontosabb részei. Csak akkor tudunk megalapozottan dönteni arról, hogy egy-egy jó tanács alkalmazható-e az általunk fejlesztett szoftver és a munkánkat behatároló egyedi kötöttségek esetében, ha megértjük a jó tanácsok mögött húzódó okfejtést.

Leginkább arra érdemes használni ezt a könyvet, hogy bepillantást nyerjünk a C++ működésébe, abba, miért úgy működik, és hogyan fordíthatjuk e működést hasznunkra. Helytelen dolog a jó tanácsokat gondolkodás nélkül alkalmazni, mégis, csak abban az esetben tanácsos eltérni az irányelvektől, ha jó okunk van rá.

Efféle könyvben nincs értelme leragadni a terminológiánál, ehhez a sporthoz a nyelvművelők jobban értenek. Van azonban egy alap C++ szókincs, amellyel tisztában kell lennünk. Az alábbiakban felsorolt szakszavak elég gyakran felbukkannak ahhoz, hogy megállapodjunk abban, mit is értünk rajtuk.

A *deklaráció* megadja a fordítóprogramnak egy objektum, függvény, osztály vagy sablon nevét és típusát, de nem tér ki minden részletre. Ezek itt deklarációk:

```
extern int x;                  // Objektumdeklaráció.
int numDigits(int number);     // Függvénydeklaráció.
class Clock;                   // Osztálydeklaráció.
template<class T>
class SmartPointer;            // Sablondeklaráció.
```

A *definíció* ugyanakkor a részleteket adja meg a fordítóprogramnak. Egy objektum esetében a fordító a definícióban foglal memóriát az objektum számára. Egy függvény

vagy függvénysablon esetében a definíció biztosítja a kód törzsét. Egy osztály vagy egy osztálysablon számára a definíció tartalmazza az osztály vagy a sablon tagjait:

```
int x;                          // Objektumdefiníció.

int numDigits(int number)       // Függvénydefiníció.
{                               // (Ez a függvény a paraméterében
   int digitsSoFar = 1;         // lévő szám számjegyeinek számát
                                // adja vissza).
   if (number < 0) {
      number = -number;
      ++digitsSoFar;
   }
   while (number /= 10) ++digitsSoFar;
   return digitsSoFar;
}
class Clock {                   // Osztálydefiníció.
public:
   Clock();
   ~Clock();
   int hour() const;
   int minute() const;
   int second() const;
   ...
};

template<class T>
class SmartPointer {            // Sablondefiníció.
public:
   SmartPointer(T *p = 0);
   ~SmartPointer();
   T * operator->() const;
   T& operator*() const;
   ...
};
```

Ezzel el is jutottunk a konstruktorokig. Az *alapértelmezett konstruktort (default constructor)* paraméterek nélkül lehet meghívni. Az ilyen konstruktornak vagy nincs paramétere, vagy minden paraméterének alapértelmezett értéke van. Általában akkor van szükség alapértelmezett konstruktorra, ha objektumokból tömböket szeretnénk definiálni:

```
class A {
public:
   A();                         // Alapértelmezett konstruktor.
};
A arrayA[10];                   // 10 konstruktorhívás.
```

```
class B {
public:
   B(int x = 0);            // Alapértelmezett konstruktor.
};
B arrayB[10];               // 10 konstruktorhívás,
                            // mindig 0 paraméterrel.
class C {
public:
   C(int x);                // Nem alapértelmezett konstruktor.
};
C arrayC[10];               // Hiba!
```

A fordító visszautasíthatja a tömbképzést objektumokból, ha az osztály alapértelmezett konstruktorának alapértelmezett paraméterértékei vannak. Néhány fordító például nem fogadja el az `arrayB` fenti definícióját, pedig erre a C++ szabvány áldását adja. Előfordulhatnak hasonló ellentmondások a C++ szabvány szerinti leírása és a nyelv egy adott fordítóprogram általi megvalósítása között is. Nem ismerek olyan fordítóprogramot, amelynek ne lennének ehhez hasonló hiányosságai. Amíg a fordítóprogramok szállítói nem érik utol a szabványt, rugalmasnak kell lennünk. Vigasztaljon minket a bizonyosság, hogy egy napon, a nem túl távoli jövőben, a szabványban leírt C++ azonos lesz azzal a nyelvvel, amelyet a fordítóprogramok elfogadnak.

Ha objektumokból olyan tömböt akarunk definiálni, amelyre nincs alapértelmezett konstruktor, a szokásos trükk az, hogy pointerekből álló tömböt definiálunk helyette. Ezután a *pointereknek* külön-külön adunk kezdeti értéket a `new`-val:

```
C *ptrArray[10];               // Nem hívódik meg konstruktor
ptrArray[0] = new C(22);       // Lefoglal és létrehoz 1 C
                               // objektumot.
ptrArray[1] = new C(4);        // Dettó.
...
```

Visszatérve a terminológia frontjára, a *másoló konstruktorral (copy constructor)* egy adott objektumnak egy másik, ugyanolyan típusú objektummal adunk kezdőértéket:

```
class String {
public:
   String();                   // Alapértelmezett konstruktor.
   String(const String& rhs);  // Másoló konstruktor.
   ...

private:
   char *data;
};
String s1;          // Az alapértelmezett konstruktort hívja meg.
String s2(s1);      // A másoló konstruktort hívja meg.
String s3 = s2;     // A másoló konstruktort hívja meg.
```

A másoló konstruktornak valószínűleg az a legfontosabb felhasználási területe, hogy meghatározza, mit jelentsen egy objektum érték szerinti átadása és visszaadása. Példaként tekintsük a következő (nem hatékony) módját annak, miként írhatunk két `String` objektumot összefűző függvényt:

```
const String operator+(String s1, String s2)
{
   String temp;
   delete [] temp.data;
   temp.data =
      new char[strlen(s1.data) + strlen(s2.data) + 1];
   strcpy(temp.data, s1.data);
   strcat(temp.data, s2.data);
   return temp;
}

String a("Hello");
String b(" world");
String c = a + b;            // c = String(„Hello world")
```

Ez az `operator+` két `String` objektumot fogad és egy `String` objektumot ad vissza mint eredményt. Mindkét paraméter és az eredmény is érték szerint adódik át, ezért meghívódik a másoló konstruktor, hogy `s1`-et `a`-val inicializálja, aztán meghívódik még egyszer, hogy `s2`-t `b`-vel inicializálja, majd pedig azért, hogy `c`-t temp-pel inicializálja. A másoló konstruktor ennél akár többször is meghívódhat, ha a fordító úgy dönt, hogy közbenső ideiglenes objektumokat hoz létre, ami megengedett. A lényeg: az érték szerinti átadás azt *jelenti*, hogy meghívjuk a másoló konstruktort.

Valójában persze nem így implementáljuk az `operator+`-t a `String` osztályban. A `const String` objektum visszaadása helyes, (lásd a 21. és 23. jó tanácsot) de a két paramétert referencia szerint kéne átadni (lásd a 22. jó tanácsot).

Ha meg bírjuk állni, egyáltalán ne írjunk `operator+`-t a `String`-ekbe, és ezt azért majdnem mindig meg lehet állni. Azért van ez így, mert a szabvány C++ könyvtár (lásd a 49. jó tanácsot) tartalmaz egy olyan sztring típust (nagy ravaszul `string` néven), és a `string` objektumok számára egy olyan `operator+`-t is, amely majdnem ugyanazt teszi, mint amit az `operator+` fentebb. Ebben a könyvben használom a `String` objektum és a `string` objektum elnevezést is, de különbözőképpen. (Vegyük észre, hogy az első név nagybetűs, a második nem.) Ha csak egy generikus sztringre van szükségem, és számomra nem lényeges, hogyan valósul meg, a `string` típust fogom használni, amely a szabvány C++ könyvtár része. Az olvasónak is ajánlatos ezt tennie. Vannak alkalmak, amikor viszont a C++ működéséről magyarázok el valamit, és implementációs kódrészletet kell példaként felhoznom. Ilyen esetekben használom a (nem szabvány) `String` osztályt. Ha programozás során sztring objektumra van szükségünk, a szabvány `string` típust kell használnunk. Elmúltak már azok az idők, amikor beavatási rítusnak számított saját sztring osztályt írni. Mégis fontos megértenünk a `string` és a hozzá hasonló osztályok fejlesztésekor felmerülő problémákat. A `String` erre a célra pont megfelel, de csak erre a célra. Ami a nyers `char*` alapú sztringeket illeti, csak akkor hasz-

náljuk ezeket a kövületeket, ha *nagyon* jó okunk van rá. A jól megvalósított `string` típusok ma már lényegében minden tekintetben felülmúlják a `char*`-okat, beleértve a hatékonyságot is (lásd a 49. jó tanácsot).

Az *inicializálás* és az *értékadás (assignment)* a következő két szakkifejezés, amivel meg kell birkóznunk. Egy objektum inicializálása akkor történik, amikor legelőször értéket kap. Konstruktorral rendelkező struktúrák vagy osztályokból álló objektumok esetén az inicializálást *mindig* a meghívott konstruktor végzi. Teljesen más a helyzet az objektumok értékadásával, mivel ez akkor történik, amikor a már inicializált objektum új értéket kap:

```
string s1;                    // Inicializálás.
string s2("Hello");           // Inicializálás.
string s3 = s2;               // Inicializálás.
s1 = s3;                      // Értékadás.
```

Tisztán a működés szempontjából az inicializálás és az értékadás között az a különbség, hogy az előbbit egy konstruktor végzi, az utóbbit pedig az `operator=`. Azaz, a két folyamat két különböző függvényhívást jelent.

A két függvény megkülönböztetése nem véletlen, mert mind a kettőnek más miatt kell aggódnia. A konstruktoroknak általában ellenőrizni kell paramétereik érvényességét, míg az értékadó operátorok biztosra vehetik, hogy a paraméterük helyes (mivel az már létrejött). Az értékadás céljának viszont lehetnek már lefoglalt erőforrásai, ellentétben egy olyan objektummal, amely konstruktorral jön létre. Ezeket az erőforrásokat fel kell szabadítani, mielőtt újakat foglalhatunk. Az egyik ilyen erőforrás leggyakrabban a memória. Mielőtt az értékadó operátor memóriát foglalhatna az új érték számára, fel kell szabadítania azt a memóriát, amely a régi értéknek lett foglalva.

Íme, így valósítható meg a `String` osztály konstruktora és értékadó operátora:

```
// egy lehetséges String konstruktor
String::String(const char *value)
{
   if (value) {          // Ha a „value" pointer nem nulla.
      data = new char[strlen(value) + 1];
      strcpy(data,value);
   }
   else {                // Kezelje a nulla „value" pointert[11].
      data = new char[1];
      *data = '\0';      // Hozzáadjuk a záró nulla karaktert.
   }
}
```

[11] A `String`-em konstruktora, amely `const char*` paramétert vesz fel, kezeli azt az esetet, amikor nullpointert kap, de a szabvány `string` típus esetében nem követelmény ez a tolerancia. Az a próbálkozás, hogy nullpointerből `string`-et hozzunk létre, definiálatlan eredményhez vezet. Ellenben üres `char*` alapú sztringből, azaz ""-ból biztonsággal hozhatunk létre `string` objektumot. (Valójában `char`-okból álló tömb első elemére mutat. – A ford.)

```
// egy lehetséges String értékadó operátor
String& String::operator=(const String& rhs)
{
   if (this == &rhs)
      return *this;            // Lásd a 17. jó tanácsot
   delete [] data;             // Felszabadítjuk a régi memóriát.
   data =                      // Lefoglaljuk az új memóriát.
      new char[strlen(rhs.data) + 1];
   strcpy(data, rhs.data);
   return *this;               // Lásd a 15. jó tanácsot.
}
```

Vegyük észre, hogy a konstruktornak ellenőriznie kell a paraméter érvényességét, dolgoznia kell azon, hogy a `data` tag megfelelően inicializált legyen, azaz nullával terminált `char*`-ra mutasson . Ezzel ellentétben az értékadó operátor biztosra veszi, hogy a paramétere szabályos, és inkább a patologikus esetek után kutat, mint például az önmagának adott értékadás (lásd a 17. jó tanácsot). Arra is koncentrál, hogy a régi memória fel legyen szabadítva, mielőtt az újat lefoglalná. A két függvény eltérései az objektum inicializálása és az objektum értékadása közötti különbséget jól jelképezik. Ja, és ha a `[]` jelölés a `delete`-nél újdonság az olvasónak, az 5. jó tanács eloszlat minden zűrzavart.

Az utolsó szakszó, amely tárgyalást igényel a *felhasználó (client)*. A felhasználó az a programozó, aki használja a kódot, amelyet írunk. Amikor felhasználókról írok ebben a könyvben, ezen azokat az embereket értem, akik megnézik a kódunkat, és megpróbálják kideríteni, hogy mit csinál; elolvassák az osztály definícióinkat, megpróbálják eldönteni, hogy örököltessenek-e az osztályainkból; megvizsgálják a tervezés során hozott döntéseinket, remélve, hogy betekintést nyerhetnek ezek logikájába.

A felhasználókra sokan talán nem is szoktak gondolni, én viszont sokáig fogom győzködni olvasóimat arról, hogy amennyire csak lehet, könnyítsük meg a felhasználók dolgát. Végtére is, mi is felhasználói vagyunk olyan szoftvereknek, amelyeket mások fejlesztenek. Nem azt szeretnénk mi is, ha ezek a fejlesztők könnyebbé tennék számunkra az életet? Ráadásul egy szép napon abban a kényelmetlen helyzetben találhatjuk magukat, hogy a *saját* kódunkkal kell dolgoznunk, saját magunk felhasználóivá válunk!

Használok két olyan nyelvi elemet ebben a könyvben, amely lehet, hogy még ismeretlen. Mindkettő viszonylag új kiegészítése a C++-nak. Az első a `bool` típus, amelynek a `true` *(igaz)* és a `false` *(hamis)* kulcsszó a lehetséges értéke. A beépített összehasonlító operátorok (azaz a <, >, == stb.) ma már ezt a típust adják vissza, és ezt teszteli az `if`, `for`, `while` és `do` utasítások feltételvizsgáló része. Ha a fordítónk nem támogatja a `bool`-t, közelítésként egyszerűen használjunk egy típusdefiníciót a `bool`-ra és konstans objektumokat a `true`-ra és a `false`-ra:

```
typedef int bool;
const bool false = 0;
const bool true  = 1;
```

Ez a C és a C++ hagyományos szemantikájával teljesen összhangban van. Nem változik az ilyen közelítést használó program működése, ha `bool`-t támogató fordítóra kerül át.

A másik elem valójában négy elem, a négy, konverzióhoz *(casting)* használt nyelvi elem: a `static_cast`, a `const_cast`, a `dynamic_cast` és a `reinterpret_cast`. A hagyományos C stílusú konverziók így néznek ki:

```
(típus) kifejezés                  // A kifejezést típus
                                   // típusúra konvertálja.
```

Az új típusú konverziók pedig így:

```
static_cast<típus> (kifejezés)     // A kifejezést típus
                                   // típusúra konvertálja.
const_cast<típus> (kifejezés)
dynamic_cast<típus> (kifejezés)
reinterpret_cast<típus> (kifejezés)
```

Ezek a konvertálásra használt elemek más-más célt szolgálnak:

- a `const_cast` célját tekintve megszünteti az objektumok és a pointerek konstansságát; a 21. jó tanácsban tárgyalom ezt a témát
- a `dynamic_cast`-tal „biztonságos bázisirányú konverziót *(downcasting)*" lehet végezni; ezt a 31. jó tanácsban vizsgálom
- a `reinterpret_cast`-ot implementáció-függő eredményt adó konverziókra tervezték, pl. függvény pointer típusok között. (Elég ritkán van szükség a `reinterpret_cast`-ra, én ebben a könyvben egyáltalán nem használom.)
- a `static_cast` afféle mindenes. Ott alkalmazzuk, ahol a többi nem használható. Jelentésében ez áll legközelebb a hagyományos C stílusú konverzióhoz.

A hagyományos konverziók még mindig szabályosak, de az újakat részesítsük inkább előnyben. Sokkal könnyebben azonosíthatók a kódban (akár emberekről, akár pl. a `grep`-ről van szó), és a célirányosabban meghatározott konvertáló elemekkel a fordító már meg tudja állapítani a felhasználói hibákat. Csak a `const_cast` használható például arra, hogy megszüntesse a konstansságot. Ha egy objektum vagy egy pointer konstans jellegét az újak közül egy másik konvertáló elemmel próbáljuk megszüntetni, a kifejezés nem fordul le.

Az új típusú konverziókról további információt vagy egy korszerű C++ tankönyvben keressünk, vagy lapozzuk fel a *More Effective C++* c. könyvem 2. fejezetét. (A *More Effective C++*-ról a 250–252. oldalon található egy rövid áttekintés.)

Jelen könyvem példakódjaiban megpróbáltam beszédes neveket adni az objektumoknak, osztályoknak, függvényeknek stb. Az elnevezések kiválasztásánál sok könyvben azt a régi bölcsességet tartják szem előtt, hogy a rövidség a lelke minden okos beszédnek, de én sziporkázás helyett inkább világosan akartam kifejezni magam. Szakítottam a programozási nyelvekről szóló könyvek hagyományaival és kerültem a titokzatos elnevezések használatát. Néhol azonban mégsem tudtam ellenállni a kísértésnek, és használom két kedvenc paraméternevemet. A jelentésük lehet, hogy nem lesz elsőre világos, főleg, ha az olvasónak nem kellett még rabláncra fűzve fordítóprogramot írnia.

Az `lhs` és az `rhs` névről van szó, melyek sorrendben a left-hand side *(bal oldali)* és right-hand side *(jobb oldali)* angol kifejezések rövidítései. Bináris operációkat megvalósító függvényekben használom őket mint paraméterneveket, főleg az `operator==` esetében, és olyan operátorokban, mint az aritmetikus `operator*`. Ha például az `a` és `b` objektum racionális számokat ábrázol, és a racionális számokat a nem tag `operator*` függvény segítségével lehet összeszorozni, az

```
a * b
```

kifejezés ekvivalens az

```
operator*(a, b)
```

függvényhívással. A 23. jó tanácsból kiderül majd, hogy én az `operator*`-ot így deklarálom:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Mint látható `a`, a bal oldali operandusz `lhs`-ként ismert a függvény belsejében, a jobb oldali pedig `rhs`-ként.

Azt is eldöntöttem, hogy a pointerek neveit a következő szabály szerint rövidítem: a `T` típusra mutató pointert gyakran hívom `pt`-nek, amely rövidítés a „pointer T-re" kifejezésből ered. Íme néhány példa:

```
string *ps;                    // ps = ptr string-re
class Airplane;
Airplane *pa;                  // pa = ptr Airplane-re
class BankAccount;
BankAccount *pba;              // pba = ptr BankAccount-ra
```

Hasonló szabály szerint nevezem el a referenciákat, azaz az `rs` lehet egy referencia `string`-re, az `ra` pedig egy referencia `Airplane`-re.

Alkalmanként, amikor tagfüggvényekről *(member functions)* van szó, használom az `mf` elnevezést.

Elég valószínűtlen, hogy előfordul, de okozhat keveredést a C programozási nyelv említése a könyvben. Én mindenhol az ISO/ANSI által szentesített C változatra gondolok, nem a régebbi, kevésbé erősen típusos „klasszikus" C-re.

Átállás C-ről C++-ra

Mindenkinek időbe telik hozzászokni a C++-hoz, de a C programozásban megőszült fejlesztők számára ez a folyamat különösen elbizonytalanító lehet. Mivel a C lényegében a C++ egy részhalmaza, a régi C trükkök továbbra is működnek, többségük azonban nem jelent többé helyénvaló megoldást. Nekünk, C++ programozóknak egy pointerre mutató pointer például kicsit mókásnak tűnik. Csodálkozunk, hogy miért nem egy pointerre mutató *referenciát* használt helyette a programozó.

A C egy aránylag egyszerű nyelv. Valójában csak makrókat, pointereket, struktúrákat, tömböket és függvényeket tartalmaz. A problémától függetlenül a megoldás mindig makrókra, pointerekre, struktúrákra, tömbökre és függvényekre vezethető vissza. Nem így a C++ esetében. A makrók, pointerek, struktúrák, tömbök és függvények persze nem hiányoznak belőle, de kiegészülnek még a privát és védett *(protected)* tagokkal, a függvénytúlterheléssel *(function overloading)*, az alapértelmezett *(default)* paraméterekkel, a konstruktorokkal és destruktorokkal, a felhasználó által definiált operátorokkal, az inline függvényekkel, a referenciákkal, a barátfüggvényekkel és osztályokkal *(friends)*, a sablonokkal *(templates)*, a kivételekkel, a névterekkel *(namespaces)* és sok minden mással. A C++ több teret enged a tervezésnek, mint a C, egyszerűen sokkal több lehetőséget vehetünk figyelembe.

Sok C programozó elbizonytalanodik, ha ilyen sok választási lehetőséggel kerül szembe, és inkább ragaszkodik ahhoz, amit megszokott. Legtöbbször ez nem is baj, de némelyik C szokás ellentétes a C++ szellemével. Ezekkel nemes egyszerűséggel fel *kell* hagyni.

1. JÓ TANÁCS: `#define` HELYETT HASZNÁLJUNK INKÁBB `const`-OT ÉS `inline`-T

Ennek a tételnek találóbb címe lehetne az, hogy „A preprocesszor helyett használjuk inkább a fordítót", mert a `#define`-t általában nem kezelik a nyelv részeként. Ez az egyik baj vele. Ha valami ilyet írunk:

```
#define ASPECT_RATIO 1.653
```

akkor az `ASPECT_RATIO` szimbolikus nevet lehet, hogy a fordító sohasem látja: a preprocesszor eltűntetheti, mielőtt a forráskód a fordítóhoz kerülne. Emiatt az `ASPECT_RATIO` név be sem kerül a szimbólum táblába. Ez megtévesztő lehet, ha a fordító olyan hibát jelez, amely összefügg a konstans használatával, mivel a hibaüzenet az `1.653`-ra fog hivatkozni és nem az `ASPECT_RATIO`-ra. Ha az `ASPECT_RATIO` egy olyan fejállományban *(header file)* lenne definiálva, amelyet nem mi írtunk, akkor halvány fogalmunk sem lenne róla, hogy honnan jön az `1.653`, és elmenne egy csomó időnk a keresésével. Ez a probléma szimbolikus *debugger*-ekben is felbukkanhat, és megint csak azért, mert a programozás közben használt név lehet, hogy nincs benne a szimbólumtáblában.

Van erre a sajnálatos esetre egy egyszerű és frappáns megoldás. A preprocesszor makrója helyett definiáljunk egy konstanst:

```
const double ASPECT_RATIO = 1.653;
```

Ez a megközelítés csodaszerként hat, habár van két speciális eset, amely említést érdemel.

Elsőként az, hogy konstans pointerek definiálása során a dolgok kicsit bonyolultabbá válnak. Mivel a konstans definíciók többnyire fejállományokba kerülnek – ahol sok forrásfájl fogja használni őket –, fontos, hogy maga a *pointer* is `const`-ként legyen deklarálva, ahogyan legtöbbször az az adat is, amelyre a pointer mutat. Ha például konstans `char*` típusú sztringet akarunk definiálni egy fejállományban, akkor a `const` szót kétszer kell leírnunk:

```
const char* const authorName = "Scott Meyers";
```

A `const` jelentéseinek és felhasználási lehetőségeinek tárgyalása, különös tekintettel a pointerekkel összefüggő használat esetére, a 21. jó tanácsban található.

Második helyen az érdemel említést, hogy gyakran bizonyul hasznos dolognak osztályspecifikus konstanst definiálni, ami egy kicsit más taktikát igényel. Egy konstans hatókörét *(scope)* úgy tudjuk egy osztályra szűkíteni, ha az osztály tagjává tesszük, és úgy tudjuk biztosítani, hogy legfeljebb egy példánya létezzen, ha *statikus* taggá tesszük:

```
class GamePlayer {
private:
```

```
    static const int NUM_TURNS = 5;   // A konstans deklarálása.
    int scores[NUM_TURNS];            // A konstans használata.
    ...
};
```

Ennek csak egy szépséghibája van, mégpedig az, hogy ami fent látható, az a `NUM_TURNS` *deklarációja*, és nem a definíciója. Egy implementációs fájlban továbbra is definiálni kell a statikus osztálytagokat:

```
const int GamePlayer::NUM_TURNS;    // Kötelező definíció;
                                    // ez az osztályt
                                    // implementáló
                                    // fájlba kerül.
```

Mégsem kell emiatt éjszakánként álmatlanul forgolódnunk. Ha el is felejtjük a definíciót, a linkernek emlékeztetnie kell rá.

Régebbi fordítók lehet, hogy nem fogadják el ezt a szintaxist, mert régebben még tilos volt egy statikus osztálytagnak a deklarálás pillanatában kezdőértéket adni. Továbbá az osztályon belüli inicializálás csak integrális típusok (pl. `int`, `bool`, `char` stb.) és csak konstansok esetében volt engedélyezve. Azokban az esetekben, amikor a fenti szintaxis nem használható, a kezdőértéket a definíciónál kell megadni:

```
class EngineeringConstants {        // Ez az osztály
private:                            // fejállományába kerül.
    static const double FUDGE_FACTOR;
    ...

};
// ez az osztályt implementáló fájlba kerül
const double EngineeringConstants::FUDGE_FACTOR = 1.35;
```

Legtöbbször ennyivel elboldogulunk. Az egyetlen kivétel ez alól, amikor egy osztálykonstans értékére az osztály fordítása közben van szükség, például a `GamePlayer::scores` tömb fenti deklarációja esetében (amikor is a fordító ragaszkodik a tömb méretének ismeretéhez a fordítás pillanatában). Létezik egy elfogadott módja annak, hogyan cselezzük ki azokat a fordítókat, amelyek – helytelenül – nem engedélyezik a kezdőérték osztályon belüli megadását integrális osztálykonstansok esetén. Ezt mi, Amerikában szeretetteljesen csak „enum hack"-nek hívjuk. Ez az eljárás kihasználja azt a tényt, hogy `int`-ek helyett használhatók az enumerációs típusok értékei, tehát a `GamePlayer`-t így is definiálhattuk volna:

```
class GamePlayer {
private:
    enum { NUM_TURNS = 5 };     // Az „enum hack": a NUM_TURNS-
                                // ből az 5-nek szimbolikus
                                // nevet készít.
```

```
    int scores[NUM_TURNS];      // Rendben, működik.
    ...
};
```

Csak akkor van szükségünk az „enum hack”-re, ha olyan fordítóprogramot használunk, amely elsősorban történelmi jelentőségre tarthat számot (azaz 1995 előtt íródott). Mégsem árt megismerkednünk vele, mert gyakran előfordul a régi, egyszerűbb időkből származó kódokban.

Visszatérve a preprocesszorhoz, a `#define` direktívát gyakran használják (helytelenül) arra is, hogy olyan makrókat implementáljanak vele, amelyek nagyon hasonlítanak a függvényekre, de megtakarítják a függvényhívás többletköltségét. Ennek iskolapéldája két érték maximumának a kiszámítása:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

Ez a kis mutatvány olyan sok hátránnyal jár, hogy rágondolni is rossz. Jobban járunk, ha az autópályán csúcsforgalomban focizunk!

Valahányszor ilyen makrót írunk, nem szabad elmulasztanunk zárójelbe tenni a paramétereket a makró törzsének megírásakor, mert különben bajba kerülünk, ha valaki egy kifejezéssel hívja meg a makrót. Ha ezt nem rontjuk el, nézzük, milyen hátborzongató dolgok történhetnek még akkor is:

```
int a = 5, b = 0;

max(++a, b);                // Az „a” kétszer növekszik.
max(++a, b+10);             // Az „a” egyszer növekszik.
```

Hogy itt mi történik az `a`-val a `max`-on belül, attól függ, mivel hasonlítjuk össze!

Szerencsére nem kell megelégednünk ezzel az ostoba megoldással. Egy inline függvénnyel is megvalósíthatjuk egy makró hatékonyságát és azt a kiszámíthatóságot és típusbiztonságot, amelyet egy szokásos függvény nyújt (lásd a 33. jó tanácsot):

```
inline int max(int a, int b) { return a > b ? a : b; }
```

Ez így nem teljesen azonos a fenti makróval, mert a `max`-nak ez a verziója csak `int`-ekkel hívható meg, de egy sablon nagyon jól megoldja a problémát :

```
template<class T>
inline const T& max(const T& a, const T& b)
{ return a > b ? a : b; }
```

Ez a sablon függvények egész családját hozza létre, melyek mindegyike két, azonos típusra konvertálható objektumot vár és a két objektum közül a nagyobbikra (annak konstans verziójára) ad vissza egy referenciát. Mivel nem tudjuk, mi lesz a `T` típus, a paramétert és a visszatérési értéket a hatékonyság érdekében referencia szerint adjuk át (lásd a 22. jó tanácsot).

Egyébiránt, mielőtt sablont írnánk egy olyan gyakran felhasználható függvényhez, mint amilyen például a max, nézzük meg a szabvány könyvtárat (lásd a 49. jó tanácsot), hátha megtaláljuk benne. A max esetében kellemesen fogunk csalódni, nyugodtan üldögélhetünk mások babérjain, mert a max része a szabvány C++ könyvtárnak.

Mivel a const-ok és az inline-ok rendelkezésünkre állnak, nincs már olyan nagy szükségünk a preprocesszorra, de azért még nem felejthetjük el teljesen. Messze van még az a nap, amikor elhagyhatjuk a #include-ot, és az #ifdef/#ifndef is még sokáig fontos szerepet fog játszani a fordítás vezérlésében. Még korai lenne nyugdíjazni a preprocesszort, de mindenképpen vegyük tervbe az egyre gyakoribb és egyre hosszabb szabadságolását.

2. JÓ TANÁCS: AZ <stdio.h> HELYETT HASZNÁLJUK INKÁBB AZ <iostream>-ET

Igen, hordozható. Igen, hatékony. Igen, már tudjuk, hogyan kell használni. Igen, igen, igen. Habár mélyen tiszteljük őket, tény, hogy a scanf-nek, printf-nek és fajtájának hasznára válna egy kis fejlesztés. Főleg, mert nem típusbiztosak és nem bővíthetők. Mivel a típusbiztonság és a bővíthetőség a C++-életmód alapkövei, érdemes már most beletörődnünk fontosságukba. Mindemellett a printf/scanf függvénycsalád – a FORTRAN-hoz hasonlóan – külön kezeli a beolvasandó és kiírandó változókat az írást és olvasást szabályozó formázási információktól. Ideje érzékeny búcsút vennünk az 1950-es évektől.

Nem meglepő, hogy a printf/scanf ezen gyengeségei az operator>> és az operator<< erősségei.

```
int i;
Rational r;                    // r egy racionális szám.
...
cin >> i >> r;
cout << i << r;
```

Ahhoz, hogy ez a kód forduljon, léteznie kell olyan operator>> és operator<< függvénynek, amely képes kezelni Rational típusú objektumokat. Ha ezek a függvények hiányoznak, hibajelzést kapunk. (Az int-ek esetében ezek a szabvány részét képezik). A fordító feladata kitalálni, hogy a különböző változó típusok kezeléséhez melyik operátorverziót kell meghívni, ezért nekünk nem kell azzal foglalkoznunk, hogy az első kiírandó vagy beolvasandó változó int, a második pedig Rational legyen.

Sőt, a beolvasandó és a kiírandó objektumok ugyanazzal a szintaktikával kerülnek átadásra, emiatt nem kell ostoba szabályokra figyelnünk, mint a scanf esetében, ahol, ha nem pointerről van szó, akkor címet kell képezni, de ha pointerről van szó, akkor *nem szabad* címet képezni. Hagyjuk a C++ fordítókra ezeket a részleteket. Nekik úgysincs jobb dolguk, nekünk meg *van*. Végül, vegyük észre, hogy a beépített típusok (pl. int) beolvasása és kiírása ugyanúgy történik, mint a felhasználó által definiált típusoké (pl. Rational). *Ezt* a scanf-fel és a printf-fel nem tudnánk megtenni!

Íme, egy racionális számokat ábrázoló osztály kimeneti rutinjának egy lehetséges megvalósítása:

```
class Rational {
public:
   Rational(int numerator = 0, int denominator = 1);
   ...
private:
   int n, d;                    // Számláló és nevező
friend ostream& operator<<(ostream& s, const Rational& r);
};

ostream& operator<<(ostream& s, const Rational& r)
{
   s << r.n << '/' << r.d;
   return s;
}
```

Ez az `operator<<` változat szemléltet néhány olyan apró (de fontos) részletet, amelyet a könyv más részében fogunk tárgyalni. Például, az `operator<<` nem tagfüggvény (a 19. jó tanács megmagyarázza, hogy miért), a kimeneti `Rational` objektum pedig `const`-ra való referenciaként és nem objektumként kerül átadásra az `operator<<`-nak (lásd a 22. jó tanácsot). A megfelelő bemeneti függvényt, az `operator>>`-t hasonlóképpen lehetne deklarálni és megvalósítani.

Vonakodva ugyan, de el kell ismernem, hogy bizonyos esetekben mégis célszerű lehet a régi, bevált módszereket alkalmazni. Először is, néhány iostream művelet implementációja nem olyan hatékony, mint a nekik megfelelő C-s „stream" művelet, tehát lehetséges (bár nem valószínű), hogy van olyan alkalmazásunk, ahol ez komoly különbséget jelent. Ne felejtsük el azonban, hogy ez a megállapítás nem *általában* szól az iostream-ekről, hanem csak néhány iostream implementációra vonatkozik. Másodszor, az iostream könyvtár a szabványosítási folyamat során alapjaiban megváltozott (lásd a 49. jó tanácsot), emiatt azokban az alkalmazásokban, ahol fontos a maximális hordozhatóság, kiderülhet, hogy a szállítók egymástól eltérő módon közelítenek a szabvány előírásaihoz. Végül, mivel az iostream könyvtár osztályainak van konstruktora, az `<stdio.h>` függvényeinek pedig nincs, kivételes – a statikus objektumok inicializálási sorrendjét érintő (lásd a 47. jó tanácsot) – esetekben előfordulhat, hogy a szabvány C könyvtár hasznosabb, egyszerűen azért, mert bármikor büntetlenül meghívható.

Az iostream könyvtár függvényei és osztályai által nyújtott típusbiztonság és bővíthetőség sokkal hasznosabb, mint azt elsőre gondolnánk, tehát ne dobjuk félre őket csak azért, mert az `<stdio.h>`-hoz vagyunk szokva. Az emlékeink az átállás után is megmaradnak.

Egyébként a jó tanács címében nincs elírás, valóban `<iostream>`-et akartam írni és nem `<iostream.h>`-t. A szó szoros értelmében véve nincs is olyan, hogy `<iostream.h>`: a szabványosítási bizottság törölte az `<iostream>` kedvéért, amikor lerövidítette az olyan fejállományok nevét, amelyek nem feleltek meg a C szabványnak. Ennek okaira a 49. jó tanács tér ki, azt azonban mindenképp tudnunk kell, hogy ha a for-

dítónk támogatja (és valószínűleg támogatja) mind az <iostream>-et, mind az <iostream.h>-t, akkor a fejállományok egy kicsit eltérnek egymástól. Ha például az #include <iostream>-et használjuk, akkor az iostream könyvtár elemeit kapjuk az std névtérbe *(namespace)* zárva (lásd a 28. jó tanácsot), míg az #include <iostream.h>-val globális névtérbe ágyazva kapjuk meg ugyanezeket az elemeket. A globális névtérbe ágyazott nevek használata névütközést *(name conflict)* okozhat, pontosan olyan névütközést, amelynek kezelésére a névtereket kitalálták. Emellett az <iostream> beírásához kevesebbet kell gépelni, mint a <iostream.h>-hoz. Sokak számára már ez is elég ok az <iostream> használatára.

3. JÓ TANÁCS: A malloc ÉS A free HELYETT HASZNÁLJUK A new-T ÉS A delete-ET

A malloc-kal és a free-vel (valamint változataikkal) nagyon egyszerű a probléma: fogalmuk sincs a konstruktorokról és a destruktorokról.

Vegyük a következő két lehetőséget egy tíz string objektumot tartalmazó tömb lefoglalására, az egyiket a malloc, a másikat a new alkalmazásával:

```
string *stringArray1 =
   static_cast<string*>(malloc(10 * sizeof(string)));

string *stringArray2 = new string[10];
```

A stringArray1 tíz string objektum számára elegendő memóriára mutat, de egy objektum sem jön létre ezen a memóriaterületen. Ráadásul zavaros nyelvi csavarok nélkül lehetőség sincs az objektumok inicializálására. A stringArray1 nem sok hasznunkra van. Ezzel ellentétben a stringArray2 tíz tökéletesen elkészített string objektumra mutat, amelyek mindegyike biztonságosan használható bármilyen string műveletben.

Tételezzük fel, hogy mégis sikerült valamilyen mágikus módszerrel a stringArray1 tömb objektumait inicializálni. A programunkban később a következőket szeretnénk tenni:

```
free(stringArray1);

delete [] stringArray2;       // Az 5. jó tanács elmagyarázza,
                              // hogy miért kell a „[]”.
```

A free meghívása felszabadítja ugyan azt a memóriát, amelyre a stringArray1 mutat, a benne lévő string objektumokra azonban nem hívódik meg egy destruktor sem. Ha a string objektumok maguk is foglalnának memóriát, amire hajlamosak, akkor az általuk lefoglalt összes memória elveszne. A másik megoldást tekintve viszont, ha a stringArray2-re meghívjuk a delete-et, akkor a delete – mielőtt a memóriát felszabadítaná – meghívja a tömbben lévő összes objektum destruktorát.

Nyilvánvaló, hogy a new és a delete alkalmazása a jobb választás, mert ezek úgy használják a konstruktorokat és a destruktorokat, ahogy azt kell.

A new és delete keverése a malloc-kal és free-vel nem igazán jó ötlet. Ha a free-t olyan pointerrel próbáljuk meghívni, amely a new-tól származik, vagy ha a delete-et olyan pointerrel, amely a malloc-tól, akkor az eredmény definiálatlan lesz, és mindannyian tudjuk, hogy az effajta bizonytalanság mit jelent: azt, hogy a fejlesztés ideje alatt minden működik, teszteléskor is, de a legfontosabb ügyfelünk szeme láttára elszáll a program.

A new/delete és a malloc/free közti összeférhetetlenség érdekes következményekkel járhat. A <string.h>-ban gyakran megtalálható strdup függvény például char* alapú sztringet fogad és annak másolatát adja vissza:

```
char * strdup(const char *ps);    // Visszaadja egy másolatát
                                  // annak, amire a ps mutat.
```

Vannak helyek, ahol a C és a C++ ugyanazt az strdup verziót használja, így a függvényen belül lefoglalt memória a malloc-tól származik. Ilyen esetekben az strdup meghívásakor a kevésbé ügyes C++ programozók figyelmét könnyen elkerülheti az, hogy az strdup-tól visszakapott pointert free-vel fel kell szabadítani. De várjunk csak! Bizonyos helyek úgy próbálják elkerülni ezt a hibaforrást, hogy C++-ban újraírják az strdup-ot úgy, hogy a függvény belsejében ez az újraírt változat a new-t hívja meg, amivel a későbbiekben a delete használatára kényszerítik a függvény hívóját. Nem nehéz elképzelnünk, milyen rémálomba illő hordozhatósági problémákat okozhat mindez akkor, ha a kódot olyan helyek között akarjuk mozgatni, amelyek az strdup különböző formáit használják.

Azért a C++ programozók is legalább annyira érdekeltek egy kód újrafelhasználásában, mint a C programozók, és az is nyilvánvaló tény, hogy sok olyan malloc-ot és free-t tartalmazó kódon alapuló C könyvtár létezik, amelyet érdemes újra felhasználni. Ha élünk az ilyen könyvtárak adta lehetőségekkel, akkor annak a felelőssége is minket terhel, hogy a könyvtár által malloc-kal lefoglalt memóriát free-vel szabadítsuk fel és/vagy malloc-kal foglaljuk le azt a memóriát, amelyet maga a könyvtár free-vel fog felszabadítani. Ez így rendben van. Egészen addig nincs is semmi baj a malloc és a free hívásával egy C++ programban, amíg garantáljuk, hogy a malloc-tól származó pointerek egy free-ben találkoznak újra teremtőjükkel, a new-ból érkező pointerek pedig eljutnak a delete-ükig. A problémák akkor kezdődnek, amikor felületessé válunk, és keverni kezdjük a new-t a free-vel vagy a malloc-ot a delete-tel. Ilyenkor csak a sorsot hívjuk ki magunk ellen.

Figyelembe véve, hogy a malloc-nak és a free-nek nincs tudomása a konstruktorokról és a destruktorokról, továbbá, hogy a malloc/free és a new/delete páros összekeverése legalább annyira heves következményekkel járhat, mint egy egyetemista szittyóbuli, a legjobban akkor járunk, ha – amennyire ez csak lehetséges – ragaszkodunk egy kizárólagosan new-ból és delete-ből álló diétához.

4. JÓ TANÁCS: HASZNÁLJUNK C++ STÍLUSÚ MEGJEGYZÉSEKET

A megjegyzések régi jó C szintaktikája C++-ban is működik, de a C++ újkeletű sorkommentjének van néhány határozott előnye. Vegyük a következő szituációt:

```
if ( a > b ) {
    // int temp = a;            // a és b felcserélése
    // a = b;
    // b = temp;
}
```

Ez a kódblokk valamilyen okból megjegyzésbe került, de a szoftvertervezés csodás példájaként már az a programozó is, aki eredetileg magát a kódot írta, fűzött hozzá egy megjegyzést, így dokumentálva, mi történik. Mivel a fenti esetben a blokk a C++ stílusában került kommentbe, a beágyazott megjegyzés egyáltalán nem okoz gondot, míg ha mindenki a C stílusát követve készített volna kommentet, abból komoly baj kerekedett volna:

```
if ( a > b ) {
    /* int temp = a;            /* a és b felcserélése */
       a = b;
       b = temp;
    */
}
```

Vegyük észre, hogy a beágyazott megjegyzés akaratlanul is félbevágja a kódblokkból létrehozott megjegyzést.

A C stílusú blokk-kommenteknek azért még mindig megvan a maguk helye. Felbecsülhetetlen értékűek például olyan fejállományokban, amelyeket a C és a C++ fordítóprogramok is feldolgoznak. Általában mégis jobban járunk, ha lehetőség szerint inkább a C++ stílusú sorkommentezést használjuk.

Említésre méltó azonban, hogy az olyan maradi preprocesszorok, melyeket kifejezetten csak a C nyelvhez készítettek, nem tudnak megküzdeni a C++ stílusában írt megjegyzésekkel, így például az alábbi kód nem feltétlenül működik az elvárások szerint:

```
#define LIGHT_SPEED 3e8         // m/sec (vákuumban)
```

Ha egy C++-t nem ismerő preprocesszort használunk, akkor a sorvégi megjegyzés a *makró részévé* válik! Természetesen, amint azt az 1. jó tanácsban megtárgyaltuk, amúgy sem lenne szabad a preprocesszort konstansok definiálására használni.

Memóriakezelés

A C++-ban felmerülő, memóriakezeléssel kapcsolatos aggodalmak két csoportra oszthatók: a helyes, illetve a hatékony memóriakezelésre. Egy jó programozó tudja, hogy a problémák megoldásának ez a helyes sorrendje, mert egy szédítően gyors és megdöbbentően kicsi program használhatatlan, ha nem úgy működik, ahogyan kellene. A legtöbb programozó számára a helyes memóriakezelés egyenértékű a memóriafoglaló és -felszabadító rutinok helyes meghívásával. A hatékony memóriakezelés viszont gyakran jelenti a memóriafoglaló és -felszabadító rutinok testreszabását, amelyeknél még inkább fontos, hogy helyesen menjenek a dolgok.

Helyesség szempontjából nézve a C++ megörökölte a C egyik legnagyobb nyűgét, a memóriaszivárgás *(memory leak)* lehetőségét. Még a virtuális memória is – legyen bármilyen ragyogó találmány –, véges, és nem is áll mindenkinek eleve rendelkezésére.

C-ben akkor történik memóriaszivárgás, ha a `malloc`-kal lefoglalt memóriát soha nem szabadítjuk fel `free`-vel. C++-ban a szereplők neve `new` és `delete`, de a történet nagyjából ugyanaz. A destruktorok jelenléte azért mégis javít a helyzeten, mivel a destruktorok gyűjtőhelyei azoknak a `delete`-eknek, amelyeket felszámolása pillanatában minden objektumnak meg kell hívnia. Használatuk ugyanakkor több figyelmet igényel, mert a `new` implicit módon hívja meg a konstruktorokat, a `delete` pedig implicit módon a destruktorokat. Tovább bonyolítja a helyzetet, hogy egy osztályon belül és kívül egyaránt definiálhatunk saját `operator new` és `operator delete` verziót. Ez megnöveli a hibalehetőségek számát. Ha megfogadjuk az itt következő jó tanácsokat, elkerülhetjük a leggyakoribb hibákat.

5. jó tanács: Az egymásnak megfelelő new-t és delete-et mindig használjuk azonos formában

Mi a baj az alábbi kóddal?

```
string *stringArray = new string[100];
...
delete stringArray;
```

Minden jónak tűnik – a new használata a delete használatával párosul –, de valami mégis nagyon el van rontva: a program viselkedése definiálatlan. A 100 string objektum közül, amelyekre a stringArray mutat, legalább 99 aligha semmisül meg helyesen, mert a destruktoruk valószínűleg sohasem hívódik meg.

A new használatakor két dolog történik. Az első, a memória lefoglalása az operator new függvénnyel, amiről a 7–10. jó tanácsban van még mondanivalóm. A második, az egy vagy több konstruktor meghívása a lefoglalt memóriaterületen. A delete használatakor is két esemény történik: egy vagy több destruktor kerül meghívásra a lefoglalt memóriaterületen, majd a memória felszabadul (az operator delete függvénnyel – lásd a 8. jó tanácsot). A delete szempontjából az a legfontosabb kérdés, hogy *hány* objektum van a törlendő memóriában. Az erre adott válasz határozza meg, hogy hány destruktort kell meghívni.

A nagy kérdés valójában ennél egyszerűbb: a törlendő pointer objektumra vagy objektumtömbre mutat? A választ a delete csak úgy tudhatja meg, ha megmondjuk neki. Ha nem használunk szögletes zárójelet a delete meghíváskor, akkor a delete arra számít, hogy egyetlen objektumra mutatunk, máskülönben azt gondolja, hogy egy tömbre:

```
string *stringPtr1 = new string;
string *stringPtr2 = new string[100];
...
delete stringPtr1;              // Egy objektum törlése.
delete [] stringPtr2;           // Egy objektumtömb törlése.
```

Mi történne, ha a [] formát használnánk a stringPtr1-re? Az eredmény definiálatlan lenne. Mi történne, ha nem a [] formát használnánk a stringPtr2-re? Nos, ennek az eredménye is definiálatlan lenne. Sőt, a beépített adattípusok, pl. int-ek esetében is definiálatlan az eredmény, még úgy is, hogy ezeknek a típusoknak nincs destruktoruk. A szabály tehát egyszerű: ha []-et használunk a new hívásakor, akkor []-et kell használni a delete hívásakor is. Ha nem használunk []-et a new hívásakor, akkor ne használjunk []-et a delete hívásakor sem.

Különösen fontos ezt a szabályt akkor szem előtt tartanunk, amikor olyan osztályt írunk, amelynek pointer adattagja és több, mint egy konstruktora van, mert a pointer adattag inicializálására minden konstruktorban a new-nak *ugyanazt a formáját* kell

használni. Ha nem ezt tennénk, akkor honnét tudnánk, hogy a `delete` melyik formáját kell majd a destruktorunkban meghívni? Az eset további vizsgálatával a 11. jó tanács foglalkozik.

Ez a szabály a `typedef`-et kedvelők számára is fontos, mert azt jelenti, hogy a `typedef` szerzőjének dokumentálnia kell, hogy a `delete` melyik formáját kell használni a `typedef` típusú objektumok `new`-val történő létrehozásakor. Vegyük például a következő `typedef`-et:

```
typedef string AddressLines[4];  // Egy személy címe
                                 // 4 sorból áll, melyek
                                 // mindegyike egy string.
```

Mivel az `AddressLines` egy tömb, ennek a `new`-nak a hívását:

```
string *pal = new AddressLines;  // Vegyük észre, hogy
                                 // a „new AddressLines"
                                 // éppúgy string*-ot
                                 // ad vissza, mint azt a
                                 // „new string[4]" tenné.
```

a `delete` *tömb* formájú változatával kell párosítani:

```
delete pal;                      // Definiálatlan!
delete [] pal;                   // Ez helyes.
```

Ha el akarjuk kerülni az ilyen kavarodást, legjobb, ha tömb típusok esetén tartózkodunk a `typedef` használatától. Márpedig ez nem nehéz feladat, mert a szabvány C++ könyvtár (lásd a 49. jó tanácsot) tartalmaz `string` és `vector` sablont, ami közel nullára csökkenti a beépített tömbök használatának szükségességét. Ebben az esetben például, az `AddressLines`-t `string`-ek `vector`-aként is definiálhattuk volna, tehát az `AddressLines` lehetne `vector<string>` típusú is.

6. jó tanács: Destruktorokban használjunk `delete`-et a pointer tagokra

A dinamikus memóriafoglalást végző osztályok legtöbbször `new`-val foglalnak memóriát a konstruktor(ok)ban, amit később a destruktorban `delete`-tel szabadítanak fel. Ezt az osztály megírásának pillanatában könnyű helyesen elvégezni, feltéve természetesen, hogy nem felejtjük el a `delete`-et *minden* olyan tagra alkalmazni, amely valamelyik konstruktorban memóriát kaphatott.

A helyzet azonban bonyolultabbá válik az osztályok karbantartása és továbbfejlesztése során, mert a módosításokat végző programozó nem feltétlenül azonos azzal, aki az osztályt eredetileg megírta. Ilyenkor már könnyű elfelejteni, hogy egy pointer tag hozzáadásakor szinte mindig el kell végeznünk a következőket:

- Inicializáljuk a pointert minden konstruktorban. Ha egy konstruktorban nem rendelünk memóriát egy pointerhez, akkor a pointert nullával kell inicializálni (azaz nullpointert kell létrehozni).
- Szabadítsuk fel a meglévő memóriát, és az új memóriát az értékadó operátorban adjuk értékül. (Lásd a 17. jó tanácsot is.)
- Töröljük a pointert a destruktorban.

Ha egy pointernek egy konstruktorban elfelejtünk kezdőértéket adni, vagy ha elfelejtjük kezelni egy értékadó operátoron belül, akkor a baj aránylag hamar kiderül, így a gyakorlatban ezek az esetek nem nehezítik meg az életünket. Ha azonban a destruktorban elfelejtünk törölni egy pointert, az már nem produkál nyilvánvaló külső tüneteket. Ehelyett rejtélyes memóriaszivárgásként *(memory leak),* lassan növekvő rákos daganatként jelentkezik, amely végül felfalja a címteret és a program korai halálát okozza. Mivel ez a probléma általában nem hívja fel magára a figyelmet, fontos, hogy ne feledkezzünk meg róla, amikor új pointer tagot adunk egy osztályhoz.

Jut eszembe, jegyezzük meg, hogy egy nullpointer törlése mindig biztonságos (ugyanis nem csinál semmit). Emiatt, ha a konstruktorainkat, értékadó operátorainkat és a többi tagfüggvényünket úgy írjuk meg, hogy az osztály minden pointer tagja mindig érvényes memóriaterületre mutasson vagy nullpointer legyen, akkor a destruktorban bátran `delete`-elhetünk, tekintet nélkül arra, hogy használtuk-e valaha is a `new`-t a kérdéses pointerre.

Nincs értelme fanatikusan ragaszkodni a jelen jó tanácshoz. Nyilván nem fogjuk olyan pointerre alkalmazni a `delete`-et, amely nem a `new`-tól kapott kezdőértéket. Egy eredetileg nekünk átadott pointert meg főleg nem fogunk szinte *soha* kitörölni. Más szóval, az osztályunk destruktorában általában csak akkor tanácsos a `delete`-et alkalmazni, ha olyan osztálytagjaink vannak, amelyek eredendően a `new`-t használták.

7. jó tanács: Készüljünk fel azokra az esetekre, amikor elfogy a memória

Ha az `operator new` nem tudja lefoglalni a kért mennyiségű memóriát, kivételt dob. (Volt idő, amikor nullát adott vissza, és némelyik régebbi fordító még mindig ezt teszi. Fordítónkat, ha akarjuk, újra rávehetjük erre, de a jó tanács végén még visszatérek a témára.) Lelkünk mélyén tudjuk, hogy a memóriahiányhoz kapcsolódó kivételek kezelése erkölcsi kötelességünk, de annak is nyomasztó tudatában vagyunk, hogy ez nagyon macerás. Emiatt van rá esély, hogy néha-néha elfelejtjük kezelni ezt az esetet. Ami akár azt is jelentheti, hogy mindig. Pedig folyamatosan lelkiismereti kérdést kellene csinálnunk belőle, mert mi van, ha a `new` tényleg kivételt dob?

Az olvasónak talán az jut eszébe, hogy ésszerű megoldás lehet erre a problémára, ha visszatérünk a nehéz gyermekkorhoz, azaz, ha a preprocesszort használjuk. C-ben például általánosan elterjedt megoldásnak számít típusfüggetlen makró definiálása a memória lefoglalására, majd annak ellenőrzése, hogy a memóriafoglalás sikeres volt-e. C++-ban ez a makró így nézhetne ki:

```
#define NEW(PTR, TYPE)                          \
   try { (PTR) = new TYPE; }                    \
   catch (std::bad_alloc&) { assert(0); }
```

(Állj! Mi ez az std::bad_alloc? – kérdezheti az olvasó. A bad_alloc annak a kivételnek a típusa, amelyet az operator new dob, ha nem tud kielégíteni egy memóriafoglalási igényt, az std pedig a névtér neve (lásd a 28. jó tanácsot), amelyben a bad_alloc-ot definiálták. Jó – folytatja az olvasó –, de mi ez az assert? Nos, ha megnézzük az <assert.h> nevű szabvány C fejállományban (vagy a névterekre is felkészített C++-os megfelelőjében, a <cassert>-ben – lásd a 49. jó tanácsot), akkor azt találjuk, hogy az assert egy makró. Ez a makró ellenőrzi, hogy a neki átadott kifejezés értéke nulla-e, és ha az, akkor hibaüzenetet küld és meghívja az abort-ot. Jó, ezt csak akkor teszi meg, ha az NDEBUG nevű szabvány makró nincs definiálva, tehát debug üzemmódban. A felhasználói verzióban, vagyis mikor az NDEBUG definiálva van, az assert semmit sem tesz, egy void kifejezésre fejtődik ki. Így feltételezéseinket – az assert-eket – csak debug üzemmódban ellenőrizzük.)

Ennek a NEW makrónak tipikus hibája az, hogy assert-et használ egy olyan feltétel ellenőrzésére, amely a felhasználói verzióban is felléphet (elvégre a memória bármikor elfogyhat). De van egy másik, speciálisan a C++-szal összefüggő hátránya is: nem veszi számításba a new használatának kismillió lehetőségét. Egy T típusú új objektum létrehozásának három elterjedt szintaktikai formája létezik, és mind a három dobhat kivételt, amelyekkel foglalkozni kell:

```
new T;
new T(a konstruktor paraméterei);
new T[méret];
```

Ezzel azért elnagyoltuk a problémát, hiszen a felhasználók definiálhatnak saját (túlterhelt *(overloaded)*) operator new verziókat is, tehát minden program tetszőleges számú, különböző szintaktikai formájú operator new-t használhat.

Akkor hogyan boldogulhatunk? Ha hajlandóak vagyunk egy nagyon egyszerű hibakezelési stratégiával beérni, akkor intézhetjük úgy, hogy egy általunk megadott hibakezelő függvény hívódjon meg kielégíthetetlen memóriaigény esetén. Ez a stratégia egy megszokott eljáráson alapul: abban az esetben, ha az operator new nem tud teljesíteni egy memóriaigényt, akkor meghív egy felhasználó által megadható hibakezelő függvényt, mielőtt kivételt dobna. Ezt a függvényt gyakran hívják new-handler-nek. (Az operator new valójában ennél egy kicsit bonyolultabban működik. A részleteket a 8. jó tanácsban fejtem ki.)

A memóriahiányt kezelő függvényt az ügyfél a set_new_handler hívásával állíthatja be, amely a <new> fejállományban többé-kevésbé így néz ki:

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

Látható, hogy a new_handler egy olyan függvényre mutató pointer típust definiál, amelynek sem paramétere, sem visszatérési értéke nincs. A set_new_handler pedig olyan függvény, amely new_handler-t kap és ad vissza.

A `set_new_handler` paramétere arra a függvényre mutató pointer, amelyet az `operator new`-nak kell meghívni, ha nem képes a kért memóriát lefoglalni. A `set_new_handler` visszatérési értéke arra a függvényre mutató pointer, amely a `set_new_handler` meghívása előtt volt érvényben ugyanezzel a céllal.

A `set_new_handler`-t így használjuk:

```
// ezt a függvényt kell meghívni, ha az operator new nem tud
// elég memóriát foglalni
void noMoreMemory()
{
    cerr << "Nincs elég memória a kérés teljesítéséhez\n";
    abort();
}
int main()
{
    set_new_handler(noMoreMemory);
    int *pBigDataArray = new int[100000000];
    ...
}
```

Ha az `operator new` nem tud 100 000 000 egészértéknek helyet foglalni, ami nagyon valószínű, akkor meghívja a `noMoreMemory`-t, és a program egy hibaüzenet kiírása után leáll. Ez valamivel jobb megoldás a program megszakítására, mint egy egyszerű core dump. (Egyébként, gondoljuk meg, mi történik, ha a hibaüzenet `cerr`-re írásakor dinamikusan kell memóriát foglalni...)

Ha az `operator new` nem tud egy memóriaigényt kielégíteni, akkor nemcsak egyszer hívja meg a new-handler függvényt, hanem *többször,* egészen addig, amíg elegendő memóriát nem talál. A 8. jó tanácsban található olyan kód, amelyik ilyen ismétlődő hívásokat idéz elő, de ennek a leegyszerűsített példának az alapján is levonhatjuk azt a következtetést, hogy egy jól megtervezett new-handler függvénynek a következők egyikét kell tennie:

- **Tegyen elérhetővé több memóriát.** Ezzel az `operator new` következő memóriafoglalási kísérlete sikeres lehet. Stratégiánkat úgy is megvalósíthatjuk például, hogy a program indulásakor lefoglalunk egy nagy memóriablokkot, amelyet a new-handler első hívásakor felszabadítunk. Az ilyen memória-felszabadítás során a felhasználó gyakran kap figyelmeztető üzenetet, hogy kevés a memória, és a további igények valószínűleg csak akkor elégíthetők ki, ha valahogy memóriát szabadítunk fel.
- **Állítson be másik new-handler-t.** Ha az aktuális new-handler nem tud több memóriát felszabadítani, akkor elképzelhető, hogy ismer egy másik, találékonyabb new-handler-t. Ha így van, akkor az aktuális new-handler beállíthatja maga helyett a másik new-handler-t (a `set_new_handler` meghívásával). Amikor az `operator new` legközelebb meghívja a new-handler függvényt, akkor már az utoljára beállított függvényt kapja. (További variáció erre a témára: a new-handler meg tudja változtatni saját működését úgy, hogy a rákövetkező hívás alkalmával már máshogy viselkedjen. Ezt azzal

lehet például elérni, ha a new-handlert olyan statikus, vagy globális adatok megváltoztatására tesszük képessé, amelyek befolyásolják a new-handler viselkedését.)

- **Iktassa ki a new-handler-t**, azaz a `set_new_handler`-nek a nullpointert adja át. Beállított new-handler nélkül az `operator new` egy `std::bad_alloc` típusú kivételt dob sikertelen memóriafoglalás esetén.
- **Dobjon kivételt**, amelynek típusa `std::bad_alloc` vagy egy abból származtatott típus legyen. Az `operator new` az ilyen kivételeket nem kapja el, így azok oda kerülnek, ahonnét a memóriakérés származik. (Az `operator new` kivétel-specifikációját megsérti egy más típusú kivétel. Az alapértelmezett működés ebben az esetben az `abort` hívása, tehát, ha a new-handler kivételt dob, akkor biztosítanunk kell, hogy ez a kivétel az `std::bad_alloc` hierarchiából származzon.)
- **Ne térjen vissza**. Tipikus megoldás az `abort` vagy az `exit` meghívása, melyek mindegyike megtalálható a szabvány C könyvtárban (és így a szabvány C++ könyvtárban is – lásd a 49. jó tanácsot).

Ezek a választási lehetőségek jelentős rugalmasságot biztosítanak a new-handler függvények megvalósításához.

Elképzelhető, hogy többféleképpen szeretnénk kezelni a memóriafoglalási hibákat, attól függően, hogy a lefoglalandó objektum melyik osztályhoz tartozik:

```
class X {
public:
   static void outOfMemory();
   ...
};
class Y {
public:
   static void outOfMemory();
   ...
};
X* p1 = new X;                 // Ha a foglalás sikertelen,
                               // meghívja az X::outOfMemory-t.
Y* p2 = new Y;                 // Ha a foglalás sikertelen,
                               // meghívja az Y::outOfMemory-t.
```

A C++ nem támogatja az osztályspecifikus new-handler-eket, de nincs is rá szükség. Ezt a működést mi magunk is meg tudjuk valósítani. Csak azt kell elérnünk, hogy minden osztály szolgáltasson saját `set_new_handler` és saját `operator new` verziót. Az osztály `set_new_handler`-ével a felhasználó *(client)* saját new-handler-t tud meghatározni az osztályhoz (épp úgy, ahogy a szabvány `set_new_handler`-rel a globális new-handler-t tudja megadni). Az osztály `operator new`-ja pedig arról gondoskodik, hogy a globális helyett az osztályspecifikus new-handler működjön akkor, amikor az osztály objektumai számára memóriát foglalunk.

Tekintsünk egy `X` osztályt, amelyben kezelni szeretnénk a memóriafoglalási hibákat. Nyomon kell követnünk, hogy melyik függvényt kell meghívni, ha az `operator new`

nem tud elég memóriát lefoglalni egy X típusú objektum számára. Ezért deklarálunk egy statikus new_handler típusú tagot, amely az osztály new-handler függvényére mutat. Ez az X osztály így fog kinézni:

```
class X {
public:
   static new_handler set_new_handler(new_handler p);
   static void * operator new(size_t size);

private:
   static new_handler currentHandler;
};
```

Statikus osztálytagokat az osztály definícióján kívül kell definiálni. Mivel szeretnénk kihasználni a statikus objektumok alapértelmezett kezdőértékadását, a 0-ra állítást, az X::currentHandler-t inicializálás nélkül definiáljuk:

```
new_handler X::currentHandler;    // alapértelmezés szerint
                                  // 0-ra (azaz nullára)
                                  // állítja a currentHandler-t
```

Az X osztály set_new_handler függvénye a neki átadott pointert eltárolja, a hívás előtt tárolt pointert pedig visszaadja. A set_new_handler szabvány változata is pontosan ezt teszi:

```
new_handler X::set_new_handler(new_handler p)
{
   new_handler oldHandler = currentHandler;
   currentHandler = p;
   return oldHandler;
}
```

Végül, az X osztály operator new-ja a következőket fogja tenni:

1. Meghívja a szabvány set_new_handler-t az X hibakezelő függvényével. Ezzel globális new-handler-nek állítja be X new-handler-ét. Vegyük észre, hogy az alábbi kódban a "::" notáció használatával explicit hivatkozunk az std hatókörre *(scope)* (ahol a szabvány set_new_handler tartózkodik).
2. Meghívja a globális operator new-t és lefoglalja a kért memóriát. Ha az első próbálkozás sikertelen, akkor a globális operator new meghívja az X new-handler-ét, mert ezt a függvényt épp most állítottuk be globális new-handler-nek. Ha a globális operator new végül mégsem képes módot találni a memória lefoglalására, akkor egy std::bad_alloc típusú kivételt dob, amit az X operator new-ja kap el. Az X operator new-ja ezután visszaállítja az eredeti globális new-handler-t, és a kivétel továbbadásával visszatér.

3. Ha a globális `operator new`-nak sikerül elegendő memóriát foglalni az X típusú objektum számára, akkor az X `operator new`-ja újra meghívja a szabvány `set_new_handler`-t, hogy visszaállítsa a globális hibakezelő függvényt arra, ami az eredetileg volt. Ezután a lefoglalt memóriaterület pointerét adja vissza. Mindezt így fejezzük ki C++-ban:

```
void * X::operator new(size_t size)
{
   new_handler globalHandler =                       // X handler-ének
      std::set_new_handler(currentHandler);          // installálása.
   void *memory;
   try {                                             // Foglalási
      memory = ::operator new(size);                 // kísérlet.
   }
   catch (std::bad_alloc&) {                         // A handler
      std::set_new_handler(globalHandler);           // visszaállítása;
      throw;                                         // Kivétel
   }                                                 // továbbadása;
   std::set_new_handler(globalHandler);              // A handler
                                                     // visszaállítása.
   return memory;
}
```

Az X osztály felhasználói így élhetnek a new-handler adta lehetőségekkel:

```
void noMoreMemory();          // Annak a függvénynek a
                              // deklarációja,amelyet az X
                              // objektumok sikertelen memória-
                              // foglalása esetén kell meghívni.

X::set_new_handler(noMoreMemory);
                              // A noMoreMemory-t az X
                              // new-t kezelő függvényének
                              // állítja be.

X *px1 = new X;               // Ha a memóriafoglalás sikertelen,
                              // akkor a noMoreMemory-t hívja meg.

string *ps = new string;      // Ha a memóriafoglalás
                              // sikertelen, akkor a globális
                              // new-handler függvényt hívja
                              // meg (ha van ilyen).

X::set_new_handler(0);        // A semmire állítja az
                              // X saját new-t kezelő
                              // függvényét (azaz, nullára).
```

```
X *px2 = new X;                // Ha a memóriafoglalás
                               // sikertelen, azonnal kivételt
                               // dob (az X osztálynak nincs
                               // new-handler függvénye).
```

Az olvasó itt azt a megfigyelést teheti, hogy ez a kód osztálytól függetlenül bárhol megvalósítja ezt a feladatot, tehát joggal gondolhatja, hogy máshol is újra beilleszthető. Amint azt a 41. jó tanács elmagyarázza, öröklődéssel és sablonokkal is készíthető többször felhasználható kód. Jelen esetünkben azonban a kettőt együtt kell alkalmazni a kívánt eredmény eléréséhez.

Egy kevert stílusú *(mixin-style)* bázisosztályt kell készítenünk csupán, azaz egy olyan bázisosztályt, amely megengedi a származtatott osztályoknak, hogy egy bizonyos képességet örököljenek. Itt ez az osztály saját new-handler-ének beállíthatóságát jelenti. Aztán a bázisosztályt átírjuk sablonná. A megoldás bázisosztály része teszi lehetővé azt, hogy a származtatott osztályok örökölhessék a számukra fontos `set_new_handler` és az `operator new` függvényeket. A sablonrész pedig azt biztosítja, hogy minden leszármazott osztály egyedi `currentHandler` adattaggal rendelkezzen. Az eredmény egy kicsit bonyolultnak tűnhet, de a kód megnyugtatóan ismerős lesz. Az egyetlen különbség gyakorlatilag az, hogy most már bármelyik osztály újra fel tudja használni a kódot:

```
template<class T>              // Kevert stílusú bázisosztály
class NewHandlerSupport {      // az osztályspecifikus
public:                        // set_new_handler támogatására.

   static new_handler set_new_handler(new_handler p);
   static void * operator new(size_t size);

private:
   static new_handler currentHandler;
};
template<class T>
new_handler NewHandlerSupport<T>::set_new_handler(new_handler p)
{
   new_handler oldHandler = currentHandler;
   currentHandler = p;
   return oldHandler;
}
template<class T>
void * NewHandlerSupport<T>::operator new(size_t size)
{
   new_handler globalHandler =
      std::set_new_handler(currentHandler);
   void *memory;
   try {
      memory = ::operator new(size);
   }
```

```
    catch (std::bad_alloc&) {
        std::set_new_handler(globalHandler);
        throw;
    }
    std::set_new_handler(globalHandler);
    return memory;
}
// minden currentHandler-t 0-ra állít
template<class T>
new_handler NewHandlerSupport<T>::currentHandler;
```

Ezzel az osztálysablonnal könnyű `set_new_handler` támogatást adni egy `X` osztályhoz. `X` egyszerűen a `NewHandlerSupport<X>`-ből származik:

```
class X: public NewHandlerSupport<X> {
                                    // Vegyük észre az öröklődést
                                    // a kevert stílusú bázisosztály
                                    // sablonból.

    ...                             // Mint előbb, csak nincs deklarálva
};                                  // set_new_handler vagy operator new.
```

`X` felhasználói nem fogják észrevenni a színfalak mögött zajló eseményeket, a régi kódjuk ugyanúgy működni fog. És ez így jó, mert a felhasználók figyelmetlenségére biztonsággal építhetünk.

A `set_new_handler` használatával egyszerűen és kényelmesen tudjuk kezelni a memóriahiány lehetséges eseteit. Sokkal vonzóbb megoldás, mint minden egyes `new`-t `try` blokkba foglalni. Ráadásul a `NewHandlerSupport`-hoz hasonló sablonokkal egyszerűen adhatunk osztályspecifikus new-handler-t bármelyik olyan osztályhoz, amelyiknek szüksége van rá. A kevert stílusú öröklődés témája azonban elkerülhetetlenül a többszörös öröklődéshez vezet. Mielőtt elindulnánk ezen a csúszós lejtőn, mindenképpen olvassuk el a 43. jó tanácsot.

A C++ 1993-ig nulla visszatérési értéket követelt meg az `operator new`-tól, amikor az nem tudott egy memóriaigényt kielégíteni. Az `operator new` jelenleg egy `std::bad_alloc` típusú kivételt dob, de sok C++ kód azelőtt született, hogy a fordítók támogatták volna az átdolgozott specifikációt. A C++ szabványosítási bizottsága nem akart lemondani a bevett, nulla értéket ellenőrző kódbázisról, ezért létrehozta az `operator new` (és az `operator new[]` – lásd a 8. jó tanácsot) alternatív formáját, amely továbbra is a hagyományos, hiba esetén a 0 értéket visszaadó viselkedést kínálja fel. Ezt a formát „nothrow" formának is hívják, mert sohasem dob kivételt, és olyan `nothrow` nevű objektumot használ a `new` hívásánál, amely a `<new>` szabvány fejállományban van definiálva:

```
class Widget { ... };
Widget *pw1 = new Widget;           // std::bad_alloc-ot dob, ha
                                    // a foglalás sikertelen.
```

```
if (pw1 == 0) ...                    // Ez a feltétel sosem teljesül.

Widget *pw2 =
   new (nothrow) Widget;             // 0-t ad vissza, ha a foglalás
                                     // sikertelen.
if (pw2 == 0) ...                    // Ez a feltétel teljesülhet.
```

Fontos felkészülnünk a sikertelen memóriafoglalás kezelésére, akár „normál" (azaz kivételt dobó) new-t, akár „nothrow" new-t használunk. Ezt legegyszerűbben a set_new_handler-rel oldhatjuk meg, mert mindkét new formával működik.

8. Jó tanács: Ragaszkodjunk a konvenciókhoz az operator new és az operator delete megírásakor

Ha az operator new megírásának terhét magunkra vesszük (a 10. jó tanács kifejti, miért akarnánk ilyet tenni), fontos, hogy a függvényünk/függvényeink viselkedése összhangban legyen az alapértelmezett operator new viselkedésével. A gyakorlatban ez azt jelenti, hogy megfelelő visszatérési értéket adunk vissza, meghívjuk a hibakezelő függvényt, ha nincs elegendő memória (lásd a 7. jó tanácsot), és felkészülünk arra, hogy nulla méretű memóriát kérnek tőlünk. Figyelnünk kell arra is, hogy a new „normál" formáját véletlenül se takarjuk el, de ez a 9. jó tanács témája.

A visszatérési érték kezelése egyszerű. Ha le tudjuk foglalni a kért memóriát, akkor egyszerűen visszaadunk rá egy pointert, ha nem, akkor követjük a 7. jó tanácsban leírt szabályt, és egy std::bad_alloc típusú kivételt dobunk.

A dolog azért nem ilyen egyszerű, mert az operator new nemcsak egyszer próbál meg memóriát foglalni, hanem *minden sikertelen kísérlet után meghívja a hibakezelő* függvényt, mert feltételezi, hogy a hibakezelő tud memóriát felszabadítani. Kivételt csak akkor dob, ha a hibakezelő függvény pointere nullpointer.

Ezen kívül a C++ szabvány megköveteli, hogy az operator new érvényes pointert adjon vissza akkor is, ha nulla bájtot kérnek tőle. (Hisszük, vagy sem, ennek a furcsa viselkedésnek a beépítése a nyelv más részein éppenséggel leegyszerűsíti a dolgokat.)

Ilyen feltételekkel egy nem tag operator new pszeudokódja így néz ki:

```
void * operator new(size_t size) // A saját op. new-nk további
{                                // paramétereket is felvehet.
   if (size == 0) {              // 0 bájtos kérések-
      size = 1;                  // 1 bájtos kérésként való
   }                             // kezelése.

   while (true) {
      megpróbálunk size mennyiségű bájtot lefoglalni;
      if (foglalás sikeres volt)
         return (a memóriára mutató pointer);
```

```
        // a foglalás sikertelen volt; derítsük ki, hogy mi az
        // aktuális hibakezelő függvény (lásd a 7. jó tanácsot)
        new_handler globalHandler = set_new_handler(0);
        set_new_handler(globalHandler);
        if (globalHandler) (*globalHandler)();
        else throw std::bad_alloc();
    }
}
```

A nulla bájt hosszúságú kérések egy bájt hosszúságú kérésként való kezelése olcsó trükknek tűnhet, de egyszerű, szabályos és működik. Amúgy is, hány nulla bájt hosszúságú kérésre számíthatunk?

Ferde szemmel nézhetünk a pszeudokódra ott is, ahol a hibakezelő függvény pointerét nullára állítjuk, majd hirtelen visszaállítjuk arra, ami eredetileg volt. Sajnos a hibakezelőfüggvény pointerét nem lehet közvetlenül elérni, mindenképpen meg kell hívni a `set_new_handler`-t ahhoz, hogy megtudjuk, mi az. Kicsit durva megoldás, de hatékony.

A 7. jó tanácsban megállapítottuk, hogy az `operator new` végtelen ciklust tartalmaz. A fenti kód egyértelműen mutatja ezt a ciklust. A `while (true)` végtelenebb már nem is lehetne. A ciklusból csak úgy lehet kilépni, ha sikerül memóriát foglalni, vagy ha a new-handler függvény végrehajt egyet a 7. jó tanácsban ismertetett lehetőségek közül: több memóriát tesz elérhetővé, egy másik new-handler-t állít be, kiiktatja a new-handler-t, egy `std::bad_alloc` típusú, vagy abból származtatott kivételt dob, vagy nem tér vissza. Remélem, most már világos, hogy a new-handler-nek miért kell megtenni a fentiek egyikét. Ha ezt elmulasztja, akkor az `operator new`-ban lévő ciklus soha nem ér véget.

Sokan nem veszik észre az `operator new`-val kapcsolatban azt, hogy a származtatott osztályok öröklik. Ez érdekes bonyodalmakhoz vezethet. Vegyük észre, hogy az `operator new` fenti pszeudokódjában a függvény `size` mennyiségű bájtot próbál lefoglalni (hacsak a `size` nem 0), ami teljesen logikus, hiszen ezt a paramétert adjuk át a függvénynek. A legtöbb osztályspecifikus `operator new`-t azonban – beleértve azt is, amelyet a 10. jó tanácsban találunk –, egy *bizonyos* osztályra tervezik, nem egy tetszőleges osztályra, vagy annak valamelyik származtatott osztályára. Ez azt jelenti, hogy ha vesszük egy `X` osztály `operator new`-ját, akkor ennek a függvénynek a viselkedése szinte mindig, gondosan `sizeof(X)` méretű objektumokra lesz hangolva, se nem nagyobbakra, se nem kisebbekre. Az öröklődés miatt azonban elképzelhető, hogy valamelyik bázisosztály `operator new`-ja azért hívódik meg, hogy egy leszármazott osztály objektumának memóriát foglaljon:

```
class Base {
public:
    static void * operator new(size_t size);
    ...
};
class Derived: public Base      // A Derived nem deklarál
{ ... };                        // operator new-t.
Derived *p = new Derived;       // A Base::operator new-t
                                // hívja meg!
```

Ha a Base osztályspecifikus operator new-ját nem készítették fel arra, hogy ilyenkor is boldoguljon – erre pedig elég nagy az esély –, legkönnyebben úgy fogja mégis kezelni a helyzetet, ha megszabadul a „rossz" méretű memóriakéréstől és továbbadja a szabvány operator new-nak, így:

```
void * Base::operator new(size_t size)
{
   if (size != sizeof(Base))          // ha a méret „rossz", ak-
      return ::operator new(size);    // kor a szabvány operator
                                      // new kezelje a kérést.

   ...                                // Máskülönben kezeljük
                                      // a kérést itt.
}
```

– Álljon meg a menet! – hallom az olvasó hangját. – Elfelejtettük ellenőrizni azt a „patologikus-de-mégis-elképzelhető" esetet, amikor a size 0! – Éppenséggel nem felejtettük el, és kérem, ne kötőjelekben kiabáljon! Az ellenőrzés ott van, ugyanis bele van foglalva a size és a sizeof(Base) összehasonlításába. A C++ szabvány útjai kifürkészhetetlenek, az egyik ezek közül kimondja, hogy a szabadon álló osztályok mérete nem nulla. A sizeof(Base) definíció szerint sohasem lehet nulla (még akkor sem, ha nincs tagja), tehát ha a size nulla, akkor a kérés a ::operator new-hoz kerül. A kérést most már ennek a függvénynek kell megfelelően kezelni.

Ha a tömbök memóriafoglalási kéréseit osztályonként szeretnénk kezelni, akkor alkalmaznunk kell az operator new tömbök kezelésére készült testvérét, az operator new[]-t. (Ezt a függvényt gyakran nevezik „tömb new"-nek, mert az „operator new[]"-t nehéz kiejteni.) Ha úgy döntünk, hogy operator new[]-t írunk, akkor ne felejtsük el, hogy csupán nyers memóriát foglalunk, és még nem tudunk semmit sem kezdeni a tömbben lévő, még nem létező objektumokkal. Gyakorlatilag még azt sem tudjuk kiszámolni, hogy hány objektum lesz a tömbben, mivel nem ismerjük az egyes objektumok méretét. Végül is, az öröklődés miatt egy bázisosztály operator new[]-ját is meg lehet hívni, hogy a származtatott osztály objektumaiból készítendő tömb számára memóriát foglaljon. A származtatott osztályok objektumai pedig többnyire nagyobbak, mint a bázisosztály objektumai. Következésképp a Base::operator new[]-n belül nem mondhatjuk azt, hogy a tömbbe kerülő objektumok mérete sizeof(Base), tehát abban sem lehetünk biztosak, hogy a tömb objektumainak száma: *(kért bájtok)*/sizeof(Base).

Ennyit az operator new (és az operator new[]) írásakor követendő konvenciókról. Az operator delete (és tömb párja az operator delete[]) esetében a dolgok egyszerűbbek. Nagyjából csak azt kell megjegyeznünk, hogy a C++-ban garantáltan biztonságos egy nullpointer törlése, úgyhogy érdemes tiszteletben tartanunk ezt a garanciát. Íme egy nem tag operator delete pszeudokódja:

```
void operator delete(void *rawMemory)
{
   if (rawMemory == 0) return;     // Nullpointer törlése esetén
                                   // ne tegyen semmit.
```

```
    a rawMemory által mutatott memóriaterület felszabadítása;
    return;
}
```

A függvény tagi változata is egyszerű, eltekintve attól, hogy ellenőriznünk kell annak a méretét, amit törlünk. Ha feltételezzük, hogy a „rossz" méretű kéréseket az osztályunk `operator new`-ja továbbadja a `::operator new`-nak, akkor a „rossz" méretű törlési kéréseket is tovább kell adnunk a `::operator delete`-nek:

```
class Base {                        // Mint előbb, de most deklarálva
public:                             // van az operator delete.
    static void * operator new(size_t size);
    static void operator delete(void *rawMemory, size_t size);
    ...
};
void Base::operator delete(void *rawMemory, size_t size)
{
    if (rawMemory == 0) return;  // Ellenőrzés nullpointerre
    if (size != sizeof(Base)) {  // „rossz" méret esetén.
        ::operator delete(rawMemory);
                                 // A szabvány operator delete-
        return;                  // tel kezeltetjük a kérést.
    }
    a rawMemory által mutatott memóriaterület felszabadítása;
    return;
}
```

Az `operator new`-val és az `operator delete`-tel kapcsolatos konvenciók tehát nem túl bonyolultak, mégis fontos betartanunk őket. Ha a memóriafoglaló rutinjaink támogatják a new-handler függvényeket, és helyesen kezelik a nulla méretű kéréseket, már majdnem készen is vagyunk. Ha a felszabadító rutinjaink megbirkóznak a nullpointerekkel, akkor már szinte semmi tennivaló nem marad. Támogassuk az öröklődést a függvények tagi változatában, és *presto!* – kész is vagyunk.

9. jó tanács: A `new` „normál" formáját sose takarjuk el

Ha belső hatókörben *(inner scope)* deklarálunk egy nevet, akkor ezzel eltakarjuk a külső hatókörökben *(outer scopes)* lévő, ezzel megegyező nevet. Tehát ha globális és osztály hatókörben is van egy `f` nevű függvény, akkor a tagfüggvény eltakarja a globális függvényt:

```
void f();                           // Globális függvény.
class X {
```

```
public:
   void f();                    // Tagfüggvény.
};
X x;
f();                            // A globális f-et hívja meg.
x.f();                          // Az X::f-et hívja meg.
```

Ez nem meglepő és általában nem okoz kavarodást, mert a globális és tagfüggvényeket többnyire eltérő szintaktikai formában hívjuk meg. De ha egy olyan `operator new`-t adunk ehhez az osztályhoz, amely további paramétereket is fogad, akkor az eredmény láttán valószínűleg el fog kerekedni a szemünk:

```
class X {
public:
   void f();
   // egy operator new, amely lehetővé teszi
   // new-handler függvény megadását
   static void * operator new(size_t size, new_handler p);
};
void specialErrorHandler();        // A definíció máshol van.
X *px1 =
   new (specialErrorHandler) X;    // Az X::operator new-t
                                   // hívja meg.
X *px2 = new X;                    // Hiba!
```

Akaratlanul is megakadályoztuk a `new` normál formájához való hozzáférést azzal, hogy az osztályon belül egy „operator new" nevű függvényt deklaráltunk. Ennek magyarázata az 50. jó tanácsban található. Minket a probléma megelőzése most jobban érdekel.

Erre egy megoldás az, ha olyan osztályspecifikus `operator new`-t írunk, amely támogatja a „normál" hívást. Ha ugyanazt teszi, mint a globális változat, akkor inline függvényként hatékonyan és elegánsan foglalható osztályba:

```
class X {
public:
   void f();

   static void * operator new(size_t size, new_handler p);
   static void * operator new(size_t size)
   { return ::operator new(size); }
};
X *px1 =
   new (specialErrorHandler) X;    // Meghívja az X::operator
                                   // new(size_t, new_handler)-t.
X* px2 = new X;                    // Az X::operator
                                   // new(size_t)-t hívja meg.
```

Alternatív megoldásként minden további, az `operator new`-nak átadott paraméternek adhatunk alapértelmezett paraméterértéket (lásd a 24. jó tanácsot):

```
class X {
public:
   void f();
   static
      void * operator new(size_t size,  // Vegyük észre, hogy p
                  new_handler p = 0);   // értéke alapértelmezett.

};
X *px1 = new (specialErrorHandler) X;  // Jó.
X* px2 = new X;                        // Szintén jó.
```

Bármelyik megoldást választjuk, ha később úgy döntünk, hogy a „normál" `new` viselkedését is egyedi igényre szabjuk, akkor csak át kell írni a függvényt, és ezután a hívók már automatikusan a testre szabott viselkedést kapják az újralinkeléskor.

10. jó tanács: Ha írunk `operator new`-t, írjunk hozzá `operator delete`-et is

Térjünk vissza az alapokhoz egy pillanatra. Először is, miért is akarna valaki saját `operator new`-t vagy `operator delete`-et írni?

A válasz többnyire a hatékonyság. Az `operator new` és az `operator delete` alapértelmezett változata tökéletesen megfelel általános célú felhasználásra, de pontosabban körvonalazott összefüggésekben rugalmassága miatt magától értetődően lehetőséget teremt a teljesítmény javítására. Ez olyan alkalmazásokra igaz különösen, amelyek dinamikusan foglalnak le sok kis méretű objektumot.

Példaként vegyünk egy repülőgépeket leíró osztályt, ahol az `Airplane` osztály csak a repülőgépeket aktuálisan reprezentáló objektumokra mutató pointert tartalmaz (a 34. jó tanácsban ismertetett technika):

```
class AirplaneRep { ... };     // Egy Airplane objektum
                               // reprezentációja.
class Airplane {
public:
   ...
private:
   AirplaneRep *rep;           // Pointer a reprezentációra.
};
```

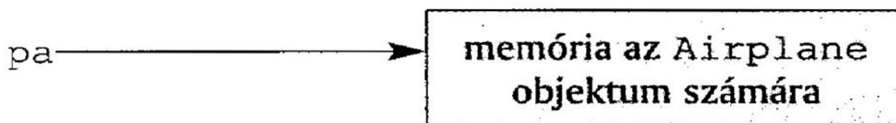
Egy `Airplane` objektum nem túl nagy, egyetlenegy pointert tartalmaz. (Amint azt a 14. jó tanács kifejti, implicit módon tartalmazhat még egy második pointert is, akkor, ha az `Airplane` osztály deklarál virtuális függvényeket.) Ha az `operator new` hívá-

sával foglalunk le egy Airplane objektumot, akkor feltehetően több memóriát kapunk, mint amennyi egy pointer (vagy kettő) tárolásához szükséges. Ennek a látszólag különös viselkedésnek az oka az operator new és az operator delete közötti kommunikáció szükségességében keresendő.

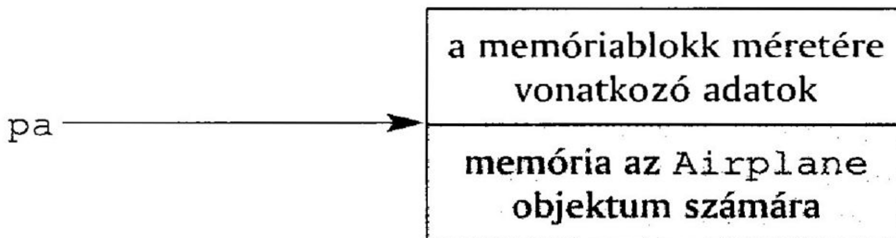
Az operator new alapértelmezett változatát általános célú memóriafoglalásra találták ki, ezért bármilyen méretű memóriablokk lefoglalására alkalmasnak kell lennie. Hasonlóképpen, az operator delete alapértelmezett változatának késznek kell lennie arra, hogy az operator new által lefoglalt, bármilyen méretű memóriablokkot felszabadítson. Ahhoz, hogy az operator delete tudja, mennyi memóriát kell felszabadítania, valami módon ki kell derítenie, hogy eredetileg mennyi memóriát foglalt az operator new. Az operator new általában úgy közli az operator delete-tel a lefoglalt memória méretét, hogy a visszaadott memória elé a lefoglalt blokk méretére vonatkozó többletinformációt fűz. Tehát, amikor azt mondjuk, hogy

```
Airplane *pa = new Airplane;
```

nem szükségszerűen kapunk vissza ilyen kinézetű memóriablokkot:



Az esetek többségében inkább ehhez hasonló memóriablokkot kapunk helyette:



Kis objektumok – mint például az Airplane osztály objektumai – esetében ez a többletnyilvántartás még a kétszeresénél is többre emelheti a dinamikusan lefoglalt objektumok memóriaigényét (különösen, ha az osztály nem tartalmaz virtuális függvényeket).

Ha olyan környezetben fejlesztünk szoftvert, ahol a memória értékes, nem biztos, hogy az ilyen pazarló memóriafoglalást megengedhetjük magunknak. Ha saját operator new-t írunk az Airplane osztályhoz, akkor kihasználhatjuk azt a tényt, hogy minden Airplane objektum ugyanolyan méretű, így nincs szükség arra, hogy nyilvántartást vezessünk minden egyes lefoglalt memóriablokkra.

Az osztályspecifikus operator new megvalósításának egy lehetséges módja az, hogy nagy méretű, nyers memóriablokkokat kérünk az alapértelmezett operator new-tól. Olyan blokkokat, amelyek elég nagyok nagyszámú Airplane objektum tárolásához. Az egyes Airplane objektumok számára szükséges memóriadarabokat ezekből a nagy blokkokból hasítjuk ki. Az éppen nem használt memóriadarabokat egy láncolt listába szervezzük – a szabad listába *(free list)* –, amely az Airplane-hez a jövőben használható memóriadarabokat tartalmazza. Úgy tűnhet, hogy ez minden objektum esetében egy next mező többletköltségével jár (a lista fenntartására), de nem. A rep

mezőhöz használt hely – amelyre csak Airplane objektumokként használt memóriadarabok esetében van szükség – lesz a next pointer tárolási helye is egyben (mivel erre a pointerre csak olyan memóriadaraboknál van szükség, amelyek nem Airplane objektumokként vannak használatban). Ezt a munkamegosztást a szokásos módon hajtjuk végre: egy union-t használunk.

Tervünk megvalósításához úgy kell módosítani az Airplane definícióját, hogy az támogassa a testreszabott memóriakezelést. Ezt így tehetjük meg:

```
class Airplane {              // Módosított osztály: már támogatja
public:                       // a testreszabott memóriakezelést.
   static void * operator new(size_t size);
   ...
private:
   union {
      AirplaneRep *rep;       // A használatban lévő objektumoknak.
      Airplane *next;         // A szabad listás objektumoknak.
   };
   // ez az osztályspecifikus konstans határozza meg (lásd az
   // 1. jó tanácsot), hogy hány Airplane objektum fér egy nagy
   // memóriablokkba; az alábbiakban található az inicializálása
   static const int BLOCK_SIZE;

   static Airplane *headOfFreeList;
};
```

Itt hozzáadtuk az operator new deklarációit: egy olyan union-t, amellyel a rep és a next mező ugyanazt a memóriát foglalja el; egy osztályspecifikus konstanst, amely meghatározza, hogy a lefoglalt memóriablokkok milyen méretűek legyenek; és egy statikus pointert, hogy a szabad lista fejét nyomon tudjuk követni. Fontos, hogy statikus tagot használjunk erre az utóbbi feladatra, hiszen nincs szabad lista minden egyes Airplane *objektum*hoz, hanem csak egy az egész *osztály*hoz.

A következő teendő az új operator new elkészítése:

```
void * Airplane::operator new(size_t size)
{
   // a „rosszul" méretezett kéréseket továbbítjuk a
   // ::operator new()-nak, a részleteket lásd a 8. jó tanács-
   ban
   if (size != sizeof(Airplane))
      return ::operator new(size);
   Airplane *p =                 // p most már a szabad lista
      headOfFreeList;            // fejére mutató pointer.
   // ha p érvényes, akkor a listafejet egyszerűen
   // a szabad lista következő elemére állítjuk
   if (p)
      headOfFreeList = p->next;
```

```
    else {
        // A szabad lista üres. BLOCK_SIZE méretű Airplane
        // objektumok tárolására alkalmas nagyságú memóriablokk
        // lefoglalása.
        Airplane *newBlock =
            static_cast<Airplane*>(::operator new(BLOCK_SIZE *
                                                  sizeof(Airplane)));
        // Egy új szabad lista készítése a memóriadarabok.
        // Egybefűzésével; a nulladik elemet kihagyjuk, mert
        // ezt adjuk vissza az operator new meghívójának.
        for (int i = 1; i < BLOCK_SIZE-1; ++i)
            newBlock[i].next = &newBlock[i+1];
        // A láncolt listát lezárjuk egy nullpointerrel.
        newBlock[BLOCK_SIZE-1].next = 0;
        // p-t a lista elejére állítjuk, a headOfFreeList-et
        // pedig a következő memóriadarabra.
        p = newBlock;
        headOfFreeList = &newBlock[1];
    }
    return p;
}
```

Ha elolvastuk a 8. jó tanácsot, akkor tudjuk, hogy ha az `operator new` nem tud teljesíteni egy memóriakérést, akkor egy sor rituális tevékenységet kell végrehajtania a new-handler függvényekkel és kivételekkel. Ennek semmi jele a fentiekben, ugyanis ez az `operator new` az összes általa kezelt memóriát a `::operator new`-tól kapja. Ami azt jelenti, hogy ennek az `operator new`-nak a végrehajtása csak akkor lehet sikertelen, ha a `::operator new`-é is az. De ha a `::operator new` sikertelen, akkor el kell játszania a new-handler rituálét (ami feltehetően egy kivétel dobásába torkollik), tehát nincs szükség arra, hogy az `Airplane operator new`-ja is végigcsinálja ugyanezt. Más szavakkal, a new-handler viselkedés ott van, csak nem látszik, mert el van rejtve a `::operator new` belsejében.

Ez után az `operator new` után már csak az `Airplane` statikus adattagjainak kötelező definíciója maradt hátra:

```
Airplane *Airplane::headOfFreeList;      // Ezek a definíciók
                                         // implementációs
const int Airplane::BLOCK_SIZE = 512;    // állományba kerülnek,
                                         // nem fejállományba.
```

Nincs szükség a `headOfFreeList` pointer explicit kinullázására, mert a statikus adattagok kezdőértéke alapértelmezés szerint 0. Természetesen a `BLOCK_SIZE`-nak adott érték határozza meg, hogy mekkora memóriadarabokat kapunk a `::operator new`-tól.

A `operator new`-nak ez a változata szépen működik. Nemcsak az igaz rá, hogy sokkal kevesebb memóriát használ az `Airplane` objektumok esetében, mint az alapértelmezett `operator new`, de valószínűleg gyorsabb is, akár *két nagyságrenddel* is.

Ez nem meglepő. Végül is az `operator new` általános változatának boldogulnia kell különböző méretű memóriakérésekkel, aggódnia kell a külső és belső töredezettség *(internal and external fragmentation)* miatt stb., míg a mi `operator new`-nknak csak néhány pointert kell kezelnie egy láncolt listában. Könnyű gyorsnak lenni, ha nem kell rugalmasnak is lenni egyszerre.

Na végre eljutottunk oda, hogy az `operator delete`-ről beszéljünk. Emlékszünk még az `operator delete`-re? Ez a jó tanács az `operator delete`-ről szól. Jelen formájában az `Airplane` osztályunk deklarálja ugyan az `operator new`-t, de nem deklarálja az `operator delete`-et. Most gondoljuk végig, mi történik, ha egy felhasználó teljesen magától értetődően a következőket írja:

```
Airplane *pa = new Airplane;    // Az Airplane::operator new-t
                                // hívja meg.
...
delete pa;                      // A ::operator delete-et
                                // hívja meg.
```

Ha a kód olvasása közben hegyezzük a fülünket, akkor hallhatjuk, amint egy repülőgép (Airplane) lezuhan és kiég, a mindezt előre sejtő programozók sírása és jajveszékelése közepette. A probléma az, hogy az `operator new` – amit az `Airplane`-ben definiáltunk – egy olyan memóriára mutató pointert ad vissza, amely *nem tartalmaz fejléc információt,* az (alapértelmezett, globális) `operator delete` viszont azt feltételezi, hogy a neki átadott memória tartalmaz fejléc információt! A katasztrófa elkerülhetetlen.

Ez a példa jól illusztrálja azt az általános szabályt, hogy az `operator new`-t és az `operator delete`-et egymással harmóniában kell megírni úgy, hogy ugyanazokkal a feltételezésekkel éljenek. Ha saját memóriafoglaló rutint szándékozunk írni, mindig írjunk saját felszabadító rutint is.

Íme, a megoldás az `Airplane` osztály esetében:

```
class Airplane {                // Mint előbb, csak az operator
public:                         // delete is deklarált.
   ...
   static void operator delete(void *deadObject,size_t size);
};
// az operator delete egy memóriadarabot kap, amely,
// ha megfelelő méretű, egyszerűen a memóriadarabok
// szabad listájának az elejére kerül
void Airplane::operator delete(void *deadObject,size_t size)

{
   if (deadObject == 0) return;       // Lásd a 8. jó tanácsot.
   if (size != sizeof(Airplane)) {    // Lásd a 8. jó tanácsot.
      ::operator delete(deadObject);
      return;
   }
```

```
    Airplane *carcass =
       static_cast<Airplane*>(deadObject);
    carcass->next = headOfFreeList;
    headOfFreeList = carcass;
}
```

Mivel elővigyázatosak voltunk az `operator new`-ban, és a „rosszul" méretezett kéréseket továbbadtuk a globális `operator new`-nak (lásd a 8. jó tanácsot), hasonló gondossággal eljárva kell biztosítanunk, hogy ezeket a „helytelen méretű" objektumokat az `operator delete` globális változata kezelje. Ha nem így teszünk, akkor pontosan abba a problémába ütközünk, amelyet olyan kemény munkával épp elkerülni akartunk: szemantikai eltérést okozunk a `new` és a `delete` között.

Érdekes módon, ha a törlendő objektum virtuális destruktor nélküli bázisosztályból származna, akkor az a `size_t` érték, amelyet a C++ az `operator delete`-nek átad, *rossz* lehetne. Ez már önmagában is elegendő ok arra, hogy a bázisosztályaink mindig tartalmazzanak virtuális destruktort, de a 14. jó tanács egy másik, bizonyíthatóan jobb indokra is rávilágít. Most elég annyit megjegyeznünk, hogy az `operator delete` függvények nem biztos, hogy helyesen működnek, ha a bázisosztályokból kihagyjuk a virtuális destruktorokat.

Ez mind szép és jó, de az olvasó összeráncolt szemöldökét látva tudom, hogy igazából a memóriaszivárgás *(memory leak)* miatt aggódik. Mindazzal a szoftverfejlesztési tapasztalattal, amelyet le tud tenni az asztalra, biztosan nem kerüli el a figyelmét az a tény, hogy az `Airplane operator new`-ja a `::operator new` hívásával foglal le nagy méretű memóriablokkokat, de ezeket az `Airplane operator delete`-je nem szabadítja fel.[12] *Memóriaszivárgás! Memóriaszivárgás!* Szinte hallom a fejekben megszólaló vészcsengőt.

Tessék jól figyelni: *nincs memóriaszivárgás.*

Memóriaszivárgás akkor keletkezik, ha lefoglalunk egy memóriablokkot, de később minden erre mutató pointer elveszik. Szemétgyűjtés és más nyelven kívüli eljárások hiányában az ilyen memória nem szerezhető vissza. De ebben a megoldásban nincs memóriaszivárgás, mert sohasem fordul elő, hogy minden memóriapointer elveszik. Minden nagy méretű memóriablokkot először `Airplane` nagyságú darabokra szedünk, majd ezek a darabok egy szabad listába kerülnek. Ha a felhasználók meghívják az `Airplane::operator new`-t, akkor ezeket a darabokat levesszük a szabad listáról, és az ezekre mutató pointereket adjuk vissza a felhasználóknak. Az `operator delete` hívásakor ezek a darabok visszakerülnek a szabad listára. Ha ezt a módszert alkalmazzuk, akkor minden memóriadarab vagy `Airplane` objektumként kerül használatba – amikor is a felhasználó felelőssége, hogy ne szivárogtassa ki a memóriáját – vagy pedig rajta van a szabad listán –, amikor is mindig létezik egy memóriára mutató pointer. Nincs memóriaszivárgás.

Akárhogy is, de a `::operator new` által lefoglalt memóriát az `Airplane::operator delete` sohasem szabadítja fel. Kell, hogy legyen erre *valamilyen* elnevezés. Van is. Ezt hívják memória *pool*-nak. Titulálja ezt szemantikai ügyeskedésnek, aki akarja,

[12] Ebben teljesen biztos vagyok, mivel az első kiadásból kihagytam ennek a problémának a tárgyalását, és ezt a mulasztásomat azóta sok olvasó a szemembe hányta. Nincs is annál jobb érzés, mint amikor az embernek több ezer korrektor hívja fel a figyelmét esendőségére.

de lényeges különbség van a memóriaszivárgás és a memória pool között. A memóriaszivárgás korlátok nélkül növekedhet, még akkor is, ha a felhasználók jól viselkednek, de a memória pool sohasem nő nagyobbra, mint a felhasználók által maximálisan lefoglalt memória mérete.

Nem lenne nehéz az Airplane memóriakezelési rutinjait úgy módosítani, hogy a `::operator new` által lefoglalt memória automatikusan felszabaduljon, ha már nincs rá szükség, de két okunk is lehet arra, hogy ezt ne tegyük meg.

Az első ok azzal a személyes indíttatással hozható összefüggésbe, hogy a memóriakezelést testreszabjuk. E mögött az a megfontolás áll leggyakrabban, hogy úgy érezzük, az alapértelmezett `operator new` és `operator delete` túl sok memóriát használ, vagy túl lassú (vagy mindkettő). Ha ez az ok, akkor minden egyes további bájt és utasítás, amelyet ezeknek a nagy memóriablokkoknak a nyomon követésére és felszabadítására használunk, tovább csökkenti a teljesítményt: a szoftverünk több memóriát használ és lassabban fut, mintha a pool-stratégiát alkalmaztuk volna. Olyan alkalmazásoknál és könyvtáraknál, ahol a teljesítmény különösen értékes, és ahol a pool mérete várhatóan értelmes keretek között marad, a pool-megközelítés adhatja a legjobb megoldást.

A második ok a patologikus viselkedéssel kapcsolatos. Tételezzük fel, hogy az `Airplane` memóriakezelő rutinjait úgy módosítjuk, hogy az `Airplane operator delete`-je felszabadít minden olyan nagy memóriablokkot, amely nem tartalmaz aktív objektumot. Most vegyük a következő programot:

```
int main()
{
   Airplane *pa = new Airplane;     // Első foglalás: nagy méretű
                                    // blokk lefoglalása, szabad
                                    // lista készítése stb.
   delete pa;                       // A blokk most üres;
                                    // felszabadítjuk.
   pa = new Airplane;               // Hmm, újabb blokkfoglalás,
                                    // szabad lista stb.
   delete pa;                       // Oké, a blokk újra üres;
                                    // felszabadítjuk.
   ...                              // Most már biztosan értjük...
   return 0;
}
```

Ez a csúf kis program még annál is lassabban fut, mint ahogyan az alapértelmezett `operator new`-val és `operator delete`-tel futna, és több memóriát is használ, a függvények pool alapú változatáról nem is beszélve!

Természetesen van gyógyír erre a betegségre, de minél többet kódolunk a szokatlan és speciális esetek kezelésére, annál közelebb kerülünk az alapértelmezett memóriakezelő függvények újraírásához, de akkor meg mit nyertünk? A memória pool nem válasz minden memóriakezelési kérdésre, de a legtöbb problémára jó.

Igazából olyan sok esetben jó, hogy biztosan nem hagy minket nyugodni a gondolat, hogy több osztály esetében is jó lenne újra alkalmazni. Biztosan van rá valami mód – gondolhatjuk magunkban –, hogy úgy csomagoljunk egy meghatározott méretű memóriafog-

lalót, hogy könnyű legyen újra felhasználni. Van is, de ez a jó tanács már olyan hosszúra nyúlt, hogy a részletek kidolgozásának elrettentő feladatát az olvasóra hagyom.

Ehelyett mutatok viszont egy minimális osztályfelületet (lásd a 18. jó tanácsot) egy `Pool` osztályhoz, amelyben minden `Pool` típusú objektum memóriafoglalást végez a `Pool` konstruktorában meghatározott méretű objektumok számára:

```
class Pool {
public:
   Pool(size_t n);              // Memóriakezelőt készítünk
                                // n méretű objektumok számára.
   void * alloc(size_t n) ;     // Egy objektumnak elegendő
                                // memóriát foglalunk; tartsuk be
                                // a 8. jó tanácsban található
                                // operator new konvenciókat.
   void free(void *p, size_t n);
                                // A p pointerrel hivatkozott
                                // memóriát visszarakjuk a
                                // pool-ba; tartsuk be a
                                // 8. jó tanácsban olvasható
                                // operator delete konvenciókat.
   ~Pool();                     // A pool-ban lévő összes
                                // memóriát felszabadítjuk.
};
```

Ez az osztály lehetővé teszi, hogy `Pool` objektumok létrejöjjenek, memóriát foglaljanak és memóriát felszabadítsanak, illetve, hogy megsemmisüljenek. Ha törlünk egy `Pool` objektumot, akkor az felszabadítja az összes általa lefoglalt memóriát. Ez azt jelenti, hogy most már van mód annak a memóriaszivárgásra hasonlító viselkedésnek az elkerülésére, amelyet az `Airplane` függvényeinél láttunk. De ez azt is jelenti egyben, hogy ha egy `Pool` destruktorát túl korán hívjuk meg (mielőtt még a memóriáját használó összes objektumot töröltük volna), néhány objektum alól kiránthatjuk a memóriát, mielőtt azok befejeznék a memória használatát. Ha azt mondjuk, hogy az ebből eredő viselkedés definiálatlan, akkor nagylelkűek vagyunk.

Ezzel a `Pool` osztállyal még egy Java programozó is minden további nélkül tud testreszabott memóriakezelő függvényeket adni az `Airplane`-hez:

```
class Airplane {
public:
   ...                          // Szokásos Airplane függvények.
   static void * operator new(size_t size);
   static void operator delete(void *p, size_t size);
private:
   AirplaneRep *rep;            // Pointer a reprezentációra.

   static Pool memPool;         // Az Airplane-ek memória pool-ja;
};                              // kapcsolódó megjegyzés lent.
```

```
inline void * Airplane::operator new(size_t size)
{ return memPool.alloc(size); }
inline void Airplane::operator delete(void *p, size_t size)
{ memPool.free(p, size); }
// Új pool-t készítünk az Airplane objektumoknak; ez az
// osztály implementációs állományába kerül.
Pool Airplane::memPool(sizeof(Airplane));
```

Ez sokkal tisztább megoldás, mint amit korábban láttunk. Az `Airplane` osztály ugyanis nincs telezsúfolva az `Airplane`-hez nem kötődő részletekkel. Eltűnt például a `union`, a szabad lista feje, vagy a nyers memóriablokkok nagyságát meghatározó konstans. Mindez a `Pool`-ban van elrejtve, és valójában ott is a helye. A memóriakezelés apró részletei miatt aggódjon a `Pool` szerzője. A mi feladatunk annyi, hogy az `Airplane` objektum helyes viselkedéséről gondoskodjunk.

Mellékesen megjegyzem, hogy az `Airplane::memPool`-t a C++ szabvány az elbűvölő „nemlokális statikus objektum" névvel illeti. Nekünk kell gondoskodnunk arról, hogy az ilyen objektumokat véletlenül se lehessen addig használni, amíg nem kapnak kezdőértéket. A 47. jó tanács részletezi, hogy mit, hogyan és miért tegyünk.

Persze nagyon érdekes látni, hogy a memóriakezelő rutinok testreszabása hogyan növeli egy program teljesítményét, és érdemes azt is kitalálni, hogyan lehet ilyen rutinokat olyan osztályba foglalni, mint pl. a `Pool`, de azért ne kalandozzunk el nagyon a tárgytól. A lényeg az, hogy az `operator new`-nak és az `operator delete`-nek együtt kell működnie, tehát ha írunk egy `operator new`-t, mindig írjunk egy `operator delete`-et is hozzá.

KONSTRUKTOROK, DESTRUKTOROK ÉS ÉRTÉKADÓ OPERÁTOROK

Amikor osztályokat írunk, majdnem mindig lesz bennük egy vagy több konstruktor, egy destruktor és egy értékadó operátor *(assignment operator)*. Nem csoda! Ezek számunkra mind nagyon alapvető függvények: meghatározzák, milyen műveleteket kell elvégezni az új objektumok létrehozásakor, és biztosítják az objektumok inicializálását, megszabadítanak a felesleges objektumoktól, gondosan kitakarítva utánuk, illetve új értéket adnak egy objektumnak. Ha ezekben a függvényekben hibát ejtünk, akkor annak igencsak kellemetlen és messzire ható következményei lesznek az összes többi osztályban. Ezért nagyon fontos, hogy ezeket a függvényeket jól írjuk meg. Ebben a fejezetben útmutatásokat próbálok adni arra nézve, hogyan írjunk olyan függvényeket, amelyek egy jól megformált osztály gerincét alkotják.

11. JÓ TANÁCS: DEKLARÁLJUNK MÁSOLÓ KONSTRUKTORT ÉS ÉRTÉKADÓ OPERÁTORT OLYAN OSZTÁLYOKBAN, MELYEK DINAMIKUSAN ALLOKÁLT MEMÓRIÁT HASZNÁLNAK

Tekintsük az alábbi, `String`-eket reprezentáló osztályt:

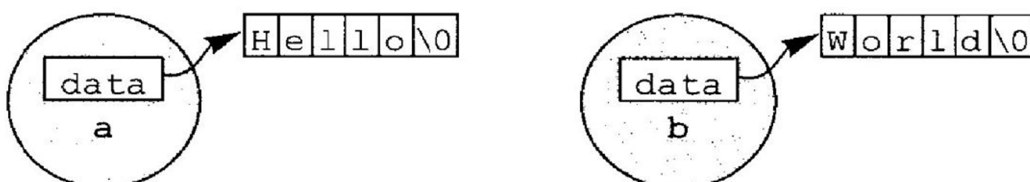
```
// egy szegényesen megtervezett String osztály
class String {
public:
   String(const char *value);
   ~String();
   ...                  // Nincs másoló konstr. vagy operator=.
private:
   char *data;
};
String::String(const char *value)
{
   if (value) {
      data = new char[strlen(value) + 1];
      strcpy(data, value);
   }
   else {
      data = new char[1];
      *data = '\0';
   }
}
inline String::~String() { delete [] data; }
```

Vegyük észre, hogy ebben az osztályban nincs deklarálva sem értékadó operátor, sem másoló konstruktor *(copy constructor)*. Mint látni fogjuk, ennek sajnálatos következményei lesznek.

Ha a következő objektumokat definiáljuk:

```
String a("Hello");
String b("World");
```

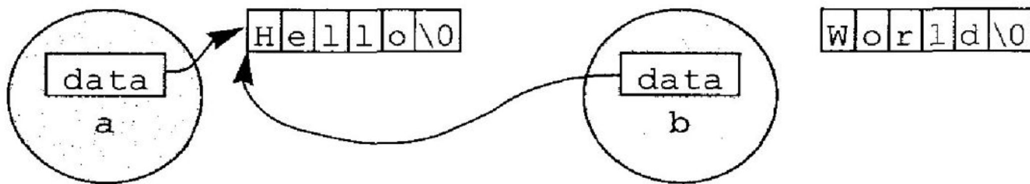
az alábbi ábrán látható helyzet áll elő:



Az a objektumban van egy pointer, amely a „Hello" karaktersztringet tartalmazó memóriaterületre mutat. Ettől függetlenül létezik a b objektum, amelyben egy pointer a „World" sztringre mutat. Ha most végrehajtunk egy értékadást:

```
b = a;
```

akkor, mivel nem hívható meg olyan `operator=`, melyet a felhasználó definiált volna, a C++ létrehozza és meghívja helyette az alapértelmezett értékadó operátort (lásd a 45. jó tanácsot). Ez az alapértelmezett értékadó operátor tagonkénti értékadást hajt végre az a tagjaiból a b megfelelő tagjaira, ami pointerek (`a.data` és `b.data`) esetén csak egy bitenkénti másolás *(bitwise copy)*. Az értékadás eredménye tehát a következő:



Ennek két szépséghibája is van. Egyrészt a b által korábban hivatkozott memória nem lett felszabadítva, örökre elveszett. Ez a memóriaszivárgás *(memory leak)* klasszikus esete. Másrészt mind a-ban, mind b-ben van most egy olyan pointer, amelyik ugyanarra a karaktersztringre mutat. Amikor az egyik hatókörön *(scope)* kívülre kerül, a destruktor felszabadítja a másik által még mindig hivatkozott memóriát. Például:

```
String a("Hello");          // a definíciója és létrehozása.
{                           // Új hatókör nyitása.
String b("World");          // b definíciója és létrehozása.
   ...
   b = a;                   // Az alapértelmezett op= végrehajtása,
                            // b memóriája elveszik.
}                           // Hatókör bezárása, b destruktorának
                            // meghívása.
String c = a;               // c.data definiálatlan!
                            // a.data már fel lett szabadítva.
```

A fenti példa utolsó utasítása a másoló konstruktor meghívása, amely szintén nincs definiálva az osztályban. Ezért aztán a C++ az értékadó operátorhoz hasonló módon generál egyet (ismét lásd a 45. jó tanácsot), ugyanazzal az eljárással: maguknak a pointereknek a bitenkénti másolásával. Ez hasonló problémához vezet, de már a memóriaszivárgás veszélye nélkül, hiszen az éppen inicializált objektum még nem mutathatott semmilyen lefoglalt memóriaterületre. A fenti kód példájánál maradva, `c.data`-nak `a.data` értékével történő inicializálásakor nincs memóriaszivárgás, mert `c.data` még nem mutat sehova. Miután azonban c-t a-val inicializáljuk, mind `c.data`, mind `a.data` ugyanarra a területre mutat, így ez a terület kétszer is felszabadul: egyszer c megszüntetésekor, másodszor pedig a megszüntetésekor.

A másoló konstruktor esete mégis más, mint az értékadó operátoré, méghozzá azért, mert az érték szerinti paraméterátadásra is oda kell figyelni. Ugyan a 22. jó tanács-

ból az derül ki, hogy csak ritkán ajánlatos objektumokat érték szerint átadnunk, most mégis tekintsük az alábbi kódrészletet:

```
void doNothing(String localString) {}
String s = "The Truth Is Out There";
doNothing(s);
```

Ez így teljesen ártalmatlannak látszik, de mivel a `localString` érték szerint adódik át, az `s`-ből csak a(z) (alapértelmezett) másoló konstruktor inicializálhatja. Ennélfogva a `localString` tartalmazza az `s`-en belüli pointer másolatát. Amikor a `doNothing` végrehajtása befejeződik, a `localString` hatókörön kívülre kerül, és meghívódik a destruktora. A végeredmény most már ismerős: `s` egy olyan memóriaterületre mutató pointert tartalmaz, amely területet a `localString` már felszabadított.

Mellesleg a `delete` viselkedése egy már törölt pointer esetén definiálatlan, ezért aztán, még ha `s`-t nem is használjuk többet, könnyen adódhat baj akkor, amikor hatókörön kívülre kerül.

Az ehhez hasonló, álnevesítő pointerekből *(pointer aliasing)* fakadó problémákat úgy oldhatjuk meg, hogy amennyiben vannak pointerek az osztályunkban, megírjuk a másoló konstruktor és az értékadó operátor saját változatát. Ezekben a függvényekben vagy lemásoljuk a mutatott adatstruktúrákat és így mindegyik objektumnak saját példánya lesz belőlük, vagy pedig valamilyen referenciaszámláló eljárást implementálunk, és így tartjuk nyilván, hogy éppen hány objektum hivatkozik egy adott adatstruktúrára. A referenciaszámláló megközelítés bonyolultabb, és többletmunkával jár a konstruktorok és destruktorok belsejében, néhány alkalmazásban azonban (korántsem mindben!) komoly memóriahasználat-csökkenést és sebességnövekedést eredményezhet.

Bizonyos osztályok esetében sokkal több problémát okoz a másoló konstruktor és az értékadó operátor megírása, mint amennyit nyerhetünk vele. Főleg akkor, ha jó okunk van azt feltételezni, hogy a felhasználók nem fognak másolást vagy értékadást végrehajtani. A fenti példákból az a következtetés vonható le, hogy a megfelelő tagfüggvények hiánya rossz tervezés eredménye, de mit tegyünk akkor, ha a megírásuk mégsem praktikus? A megoldás egyszerű: kövessük ennek a fejezetnek a tanácsát. *Deklaráljuk* ezeket a függvényeket (mint látni fogjuk az osztály `private` részében), de véletlenül se definiáljuk (azaz ne implementáljuk) őket! Így a felhasználók nem tudják meghívni, a fordítók pedig nem fogják létrehozni őket. Ennek a klassz kis trükknek a részleteit a 27. jó tanács tárgyalja.

Van még egy fontos megjegyzésem a fejezetben használt `String` osztályhoz. A konstruktor törzsében odafigyeltem arra, hogy a `new` művelet `[]`-es változatát használjam mindkét hívás esetében, noha az egyik helyen csak szingli objektumra volt szükségem. Amint azt az 5. jó tanács leírja, alapvető fontosságú, hogy ugyanabban a formában használjuk a `new`-t és a `delete`-et egymáshoz kapcsolódó alkalmazásuk esetén, ezért figyeltem oda a `new` következetes használatára. Ezt érdemes megjegyezni. *Mindig* győződjünk meg arról, hogy akkor és csak akkor használjuk a `delete` művelet `[]`-es változatát, ha a megfelelő `new`-nak is a `[]`-es változatát használtuk.

12. jó tanács: Konstruktorokban értékadás helyett válasszuk inkább az inicializálást

Tekintsünk egy osztályokat generáló sablont, amely lehetővé teszi név hozzárendelését valamilyen `T` típusra mutató pointerhez:

```
template<class T>
class NamedPtr {
public:
   NamedPtr(const string& initName, T *initPtr);
   ...
private:
   string name;
   T *ptr;
};
```

(Annak fényében, hogy az álnevesítés *(aliasing)* felmerülhet a pointer tagokkal rendelkező objektumok értékadása *(assignment)* és másolva létrehozása során (lásd a 11. jó tanácsot), érdemes meggondolni, nem a `NamedPtr`-nek kellene-e implemetálnia ezeket a függvényeket. Súgok: annak kellene! (Lásd a 27. jó tanácsot.))

Amikor a `NamedPtr` konstruktort írjuk, a paraméterek értékét át kell mozgatnunk a megfelelő adattagokba. Ezt kétféleképpen is megtehetjük. Az első lehetőség a taginicializáló lista használata:

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr)
: name(initName), ptr(initPtr)
{}
```

A második lehetőség értékadások végrehajtása a konstruktor törzsében:

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr)
{
   name = initName;
   ptr = initPtr;
}
```

Fontos különbségek vannak a két megközelítés között.

Tisztán gyakorlati szempontból nézve, vannak olyan esetek, amikor az inicializáló listát *kell* használni. Különösen `const`- és referenciatagok esetén, mivel azok *csak* inicializálhatók, értékadáson keresztül nem kaphatnak értéket. Ezért ha úgy döntenénk, hogy egy `NamedPtr<T>` objektum sohase változtathassa meg se a nevét, se a pointerét, akkor a 21. jó tanácsot követve ezeket a tagokat `const`-nak deklarálnánk:

```
template<class T>
class NamedPtr {
public:
   NamedPtr(const string& initName, T *initPtr);
   ...
private:
   const string name;
   T * const ptr;
};
```

Ez az osztálydefiníció *megköveteli* a taginicializáló lista használatát, mert a `const` tagok csak inicializálhatók, értéket értékadáson keresztül nem kaphatnak.

Egészen eltérő viselkedést eredményezne, ha úgy döntenénk, hogy a `NamedPtr<T>` objektum inkább egy már létező névre hivatkozó *referenciát* tartalmazzon. De még ekkor is érvényes az, hogy a referenciát inicializálnunk kell a konstruktorok taginicializáló listáiban. Természetesen a két lehetőséget vegyesen is alkalmazhatjuk, olyan `NamedPtr<T>` objektumokat vezetve be, amelyeknek csak olvasási hozzáférésük van azokhoz a nevekhez, amelyek az osztályon kívül változtathatók:

```
template<class T>
class NamedPtr {
public:
   NamedPtr(const string& initName, T *initPtr);
   ...
private:
   const string& name;          // Inicializáló listában kell
                                // inicializálni.
   T * const ptr;               // Inicializáló listában kell
                                // inicializálni.
};
```

Az eredeti osztálysablon azonban sem `const`-, sem referenciatagokat nem tartalmaz. Ám még ebben az esetben is célszerűbb taginicializáló listát használni a konstruktor belsejében végrehajtott értékadások helyett. Ez alkalommal az ok a hatékonyság. Amikor taginicializáló listával dolgozunk, csak egyetlen `string` tagfüggvény hívódik meg. Ha értékadást alkalmazunk a konstruktoron belül, akkor kettő. Hogy megértsük, miért, gondoljuk végig, mi történik, amikor egy `NamedPtr<T>` objektumot deklarálunk.

Az objektumok létrehozása két fázisban történik:

1. Adattagok inicializálása. (Lásd még a 13. jó tanácsot.)
2. A meghívott konstruktor törzsének a végrehajtása.

(Bázisosztályokkal rendelkező objektumok esetén a bázisosztály(ok) taginicializálása és konstruktortörzsének végrehajtása megelőzi ugyanezeknek a végrehajtását a származtatott osztályokban.)

A `NamedPtr` osztályok esetén ez azt jelenti, hogy a `name` nevű `string` objektum konstruktora *mindig* meghívódik, mielőtt a `NamedPtr` konstruktor törzsébe jutnánk. Az egyetlen kérdés tehát: melyik `string` konstruktor fog meghívódni?

Ez a `NamedPtr` osztályok taginicializáló listájától függ. Ha nem adunk meg inicializáló paramétert a `name`-hez, akkor az alapértelmezett `string` konstruktor hívódik meg. Amikor később a `NamedPtr` konstruktorok belsejében értéket adunk a `name`-nek, akkor az `operator=`-t hívjuk meg a `name`-re. Ez összesen két `string` tagfüggvény meghívását jelenti: az alapértelmezett konstruktorét és az értékadásét.

Másrészt, ha taginicializáló listán keresztül adjuk meg, hogy a `name`-et az `initName`-mel kell inicializálni, akkor a `name` a másoló konstruktoron keresztül kap értéket, ami csak egy függvényhívásba kerül.

Még az egyszerű `string` típus esetében is jelentős lehet egy szükségtelen függvényhívás költsége. Az egyre nagyobbá és összetettebbé váló osztályok konstruktorai is egyre bonyolultabbak lesznek, így az objektumok létrehozásának költsége is nő. Ha kialakítjuk magunkban azt a szokást, hogy ahol csak lehet, a taginicializáló listát használjuk, akkor nemcsak egy, a `const`- és referenciatagokkal szembeni elvárásnak felelünk meg, hanem az adattagok hatékony inicializálásának az esélyét is növeljük.

Más szóval, a taginicializáló listán keresztüli inicializálás *mindig* szabályos, és mindig *legalább* annyira hatékony, mint a konstruktor törzsén belüli értékadások, sőt, gyakran hatékonyabb. Ezen felül leegyszerűsíti az osztály karbantartását, mert ha később úgy változtatjuk meg az adattag típusát, hogy az *megköveteli* a taginicializáló lista használatát, akkor semmit sem kell a konstruktorban megváltoztatnunk.

Van azonban egy olyan eset, amikor ésszerűbb lehet az osztály adattagjainak inicializálása helyett értékadást alkalmazni, méghozzá akkor, ha nagyszámú *beépített típusú adattag*unk van az osztályban, és ugyanúgy akarjuk őket inicializálni minden konstruktorban. Íme egy osztály, amely megfelel ezeknek a kritériumoknak:

```
class ManyDataMbrs {
public:
    // alapértelmezett konstruktor
    ManyDataMbrs();
    // másoló konstruktor
    ManyDataMbrs(const ManyDataMbrs& x);

private:
    int a, b, c, d, e, f, g, h;
    double i, j, k, l, m;
};
```

Tegyük fel, hogy az összes int-et 1-re, és az összes double-t 0-ra akarjuk inicializálni, még a másoló konstruktorban is. A taginicializáló listát használva ezt kell írnunk:

```
ManyDataMbrs::ManyDataMbrs()
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }
```

```
ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }
```

Ez nemcsak, hogy lélekölő munka, hanem rövid távon sok hibalehetőséget rejt magában, hosszú távon pedig nehéz karbantartani.

Hasznot húzhatunk azonban abból a tényből, hogy a beépített típusok (nem `const`, nemreferencia) objektumainak inicializálása és értékadása között nincs műveleti különbség. Így aztán bátran lecserélhetjük a tagonkénti inicializáló listákat egy olyan függvény meghívására, amely elvégzi a közös inicializáló műveleteket:

```
class ManyDataMbrs {
public:
   // alapértelmezett konstruktor
   ManyDataMbrs();
   // másoló konstruktor
   ManyDataMbrs(const ManyDataMbrs& x);
private:
   int a, b, c, d, e, f, g, h;
   double i, j, k, l, m;
   void init();                // Adattagok inicializálására
                               // használatos.
};
void ManyDataMbrs::init()
{
   a = b = c = d = e = f = g = h = 1;
   i = j = k = l = m = 0;
}
ManyDataMbrs::ManyDataMbrs()
{
   init();
   ...
}
ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
{
   init();
   ...
}
```

Mivel az inicializáló művelet az osztály implementációjához tartozik, természetesen kellő óvatossággal járunk el, és a `private` részbe tesszük, ugye?

Jegyezzük meg jól, hogy a `static` osztálytagokat *soha* nem szabad az osztály konstruktorában inicializálni. A statikus tagok programfutásonként csak egyszer inicializálódnak, ezért aztán nincs értelme minden egyes alkalommal a „kezdeti értékadás"-sal próbálkozni, amikor az osztály típusának megfelelő objektumot hozunk létre. Ez enyhén

szólva nem lenne hatékony: miért fizessünk egy objektum „kezdeti értékadás"-áért többször? És mivel a statikus adattagok inicializálása nagyon eltér a nem statikusakétól, egy teljes jó tanácsot – a 47.-et – szenteltem a témának.

13. jó tanács: Az inicializáló listában az adattagokat a deklarációjuk sorrendjében soroljuk fel

Elvetemült Pascal- és Ada-programozók gyakran sóvárognak az után, hogy olyan tömböket definiálhassanak, amelyeknek tetszőleges indexhatára lehet, pl. 10–20-ig, 0–10-ig helyett. A régóta C-ben programozók kitartanak amellett, hogy mindenki, aki valamit is ad magára, 0-tól kezdi a számozást, de a `begin`/`end`-hívő tömegeket is elég egyszerű megbékíteni. Csak definiálnunk kell egy saját `Array` osztálysablont:

```
template<class T>
class Array {
public:
   Array(int lowBound, int highBound);
   ...
private:
   vector<T> data;              // A tömb adatait egy vector
                                // objektumban tároljuk,
                                // a vektorsablonról a 49. jó
                                // tanácsban található info.
   size_t size;                 // A tömb elemeinek száma.
   int lBound, hBound;          // Alsó határ, felső határ.
};
template<class T>
Array<T>::Array(int lowBound, int highBound)
: size(highBound - lowBound + 1),
   lBound(lowBound), hBound(highBound),
   data(size)
{}
```

Egy igazán strapabíróra tervezett konstruktor elvégezne pár ellenőrzést a paraméterek értelmességére vonatkozólag, így biztosítva, hogy a `highBound` értéke legalább akkora legyen, mint a `lowBound`-é, de itt egy sokkal komiszabb hiba is van: még tökéletes indexhatár-értékek mellett sem lehet tudni, hogy a `data` hány elemet tartalmaz.

„Ez meg hogy lehet?" – hallom az olvasó kiáltását. „Hiszen inicializáltam a `size` tagot, még mielőtt átadtam volna a `vector` konstruktorának!" Sajnos ez nem igaz, csak megpróbálta. A játékszabály az, hogy az adattagok az *osztálybeli deklarációjuk sorrendjében* kerülnek inicializálásra, azaz a taginicializáló listában meghatározott sorrend egyáltalán nem számít. Így az `Array` sablonunkból generált osztályokban először mindig a `data` inicializálódik, majd azt követi a `size`, a `lBound`, és a `hBound`. Mindig.

Ez így nyakatekertnek tűnhet, de van benne logika. Tekintsük az alábbi forgatókönyvet:

```
class Wacko {
public:
   Wacko(const char *s): a(s), b(0) {}
   Wacko(const Wacko& rhs): b(rhs.a), a(0) {}

private:
   string a, b;
};
Wacko w1 = "Hello world!";
Wacko w2 = w1;
```

Ha az adattagok az inicializáló listabeli előfordulásuk sorrendjében lennének inicializálva, akkor a `w1` és `w2` adattagjai különböző sorrendben jönnének létre. Emlékezzünk vissza, hogy egy objektum adattagjainak destruktorai mindig a konstruktoraik sorrendjével ellentétes sorrendben hívódnak meg. Így, ha a fentieket megengednénk, akkor a fordítóknak *minden objektumra* el kellene tárolniuk a tagok inicializálási sorrendjét, csak azért, hogy a destruktorok a megfelelő sorrendben hívódjanak meg. Ez bizony elég drága vállalkozás volna. Azzal, hogy a létrehozás és megszüntetés sorrendje egy adott típus minden objektumára ugyanaz, és az inicializáló listabeli tagsorrendet nem vesszük figyelembe, megtakarítjuk ezt a plusz költséget.

Ha igazán pontosak akarunk lenni, csak a nemstatikus adattagok inicializálódnak a szabály szerint. A statikus adattagok úgy viselkednek, mint a globális és a névtér *(namespace)* objektumok, és így csak egyszer inicializálódnak (a részleteket lásd a 47. jó tanácsban). Továbbá a bázisosztályok adattagjai a leszármazott osztályok adattagjai előtt inicializálódnak, ezért ha öröklődést használunk, célszerű a bázisosztályok inicializálóit a taginicializáló listák legelején felsorolni. (Ha többszörös öröklődéssel *(multiple inheritance)* dolgozunk, a bázisosztályok abban a sorrendben inicializálódnak, amilyen sorrendben származtatunk belőlük; a taginicializáló listákban meghatározott sorrend megint csak nem számít. Többszörös öröklődés alkalmazásakor azonban sokkal fontosabb dolgok miatt is aggódnunk kell. Ha az olvasó mégsem aggódik, a 43. jó tanács szívesen kisegíti a többszörös öröklődés aggasztó tulajdonságai tekintetében.)

A lényeg a következő: ha szeretnénk megérteni, hogy pontosan mi is történik az objektumaink inicializálásakor, akkor az adattagokat az inicializáló listában mindig az osztálybeli deklarációjuk sorrendjében soroljuk fel.

14. JÓ TANÁCS: A BÁZISOSZTÁLYOK DESTRUKTORA MINDIG LEGYEN VIRTUÁLIS

Néha fontos lehet egy osztálynak számon tartania, hogy hány ilyen típusú objektum létezik. Magától adódó megoldás egy statikus osztálytag bevezetése az objektumok számlálására. Az adattagot 0-val inicializáljuk, az osztály konstruktoraiban eggyel megnöveljük, a destruktorban pedig eggyel lecsökkentjük az értékét.

Képzeljünk el egy katonai alkalmazást, amelyben az ellenséges célpontokat reprezentáló osztály valahogy így néz ki:

```
class EnemyTarget {
public:
   EnemyTarget() { ++numTargets; }
   EnemyTarget(const EnemyTarget&) { ++numTargets; }
   ~EnemyTarget() { --numTargets; }

   static size_t numberOfTargets()
   { return numTargets; }
   virtual bool destroy();      // Visszaadja, hogy sikeres
                                // volt-e az EnemyTarget
                                // objektum elpusztítása.
private:
   static size_t numTargets;    // Objektumszámláló.
};

// az osztály statikus tagjait az osztályon kívül definiáljuk;
// alapértelmezés szerint 0-ra inicializálódik
size_t EnemyTarget::numTargets;
```

Nem valószínű, hogy ezzel az osztállyal honvédelmi célokat szolgáló állami megrendelést nyernénk, de jelenlegi szándékainknak pont megfelel, és a mi igényeink lényegesen alacsonyabbak, mint a Honvédelmi Minisztériumé. Legalábbis reméljük.

Vegyünk egy konkrét ellenséges célpontot, egy ellenséges harckocsit, amelyet – magától értetődő módon (lásd a 35. jó tanácsot) – az `EnemyTarget`-ből publikusan származtatott osztályként modellezünk. Mivel az összes ellenséges célpont számán kívül az összes ellenséges harckocsi száma is érdekel bennünket, ugyanazt a trükköt alkalmazzuk a leszármazott osztályban, mint amit a bázisosztályban alkalmaztunk:

```
class EnemyTank: public EnemyTarget {
public:
   EnemyTank() { ++numTanks; }
   EnemyTank(const EnemyTank& rhs)
   : EnemyTarget(rhs)
   { ++numTanks; }
   ~EnemyTank() { --numTanks; }
   static size_t numberOfTanks()
   { return numTanks; }
   virtual bool destroy();

private:
   static size_t numTanks;      // Tank objektumok számlálója.
};
size_t EnemyTank::numTargets; // A számláló inicializálása.
```

Végül tegyük fel, hogy valahol az alkalmazásunkban dinamikusan létrehozunk egy `EnemyTank` objektumot `new`-val, amelytől aztán később `delete`-tel szabadulunk meg.

```
EnemyTarget *targetPtr = new EnemyTank;
...
delete targetPtr;
```

Eddig minden teljesen kósernek tűnik. Mindkét osztály visszacsinálja a destruktorában azt, amit a konstuktorában tett, és persze az alkalmazásunkkal sincs semmi gond, hiszen miután végeztünk azzal az objektummal, amelyet korábban a `new`-val varázsoltunk elő, meghívtuk rá a `delete`-et. Mindazonáltal van itt egy nagy baj. A programunk viselkedése *definiálatlan*, nem tudhatjuk, *mi* fog történni.

A C++ nyelv szabványa szokatlanul tisztán fogalmaz ebben a kérdésben. Ha egy leszármazott osztályhoz tartozó objektumot egy bázisosztály pointerén keresztül próbálunk meg felszabadítani, és a bázisosztály destruktora *nemvirtuális* (mint az `EnemyTarget`-é), akkor az eredmény definiálatlan. Ez azt jelenti, hogy a fordítóprogram olyan kódot generálhat, amellyel azt tehet, amit csak akar: újraformázhatja a lemezünket, sokatmondó leveleket küldhet a főnökünknek, forráskódot faxolhat el a versenytársainknak, bármit. (Futási időben gyakran az történik, hogy a leszármazott osztály destruktora sosem hívódik meg. Ebben a példában ez azt jelentené, hogy az `EnemyTank`-ek számlálása nem módosulna a `targetPtr` törlésekor. Így az ellenséges tankok számát rosszul adnánk meg, ami bizony elég nyugtalanító kilátás a pontos harctéri információktól függő katonáknak.)

Ezt a problémát könnyen elkerülhetjük, ha az `EnemyTarget` destruktort virtuálissá tesszük. A *virtuálisnak* deklarált destruktor olyan pontosan meghatározott viselkedést biztosít, amelynek során az történik, amit akarunk: mind az `EnemyTank`, mind az `EnemyTarget` destruktora meghívódik, mielőtt az objektumot tartalmazó memóriaterület felszabadulna.

Az `EnemyTarget` osztály most már tartalmaz virtuális függvényt, miként ez a bázisosztályoktól általában elvárható. A virtuális függvények célja nem más, mint hogy lehetővé tegyék a viselkedés testreszabását a származtatott osztályokban (lásd a 36. jó tanácsot), ezért aztán majdnem minden bázisosztálynak van virtuális függvénye.

Ha egy osztály *nem* tartalmaz virtuális függvényt, akkor ez gyakran annak a jele, hogy az osztályt nem bázisosztálynak szánták. Ha egy osztályt nem úgy terveztek, hogy bázisosztályként funkcionáljon, akkor a destruktor virtuálissá tétele általában nem jó ötlet. Gondoljuk végig az alábbi, az ARM (Annotated Reference Manual – *Jegyzetekkel ellátott kézikönyv*)-ben tárgyalt eseten alapuló példát (lásd az 50. jó tanácsot):

```
// 2D pontokat reprezentáló osztály
class Point {
public:
   Point(short int xCoord, short int yCoord);
   ~Point();
private:
   short int x, y;
};
```

Ha egy `short int` 16 bitet foglal el, akkor egy `Point` objektum elfér egy 32 bites regiszterben. Továbbá egy Point objektum, 32 bites egységként, más nyelveken (mint például C vagy FORTRAN) írt függvényeknek is átadható. Ha viszont a `Point` destruktorát virtuálissá tesszük, megváltozik a helyzet.

A virtuális függvények implementációja megköveteli, hogy az egyes objektumokban legyen némi plusz információ, amely futási időben felhasználható annak meghatározására, mely virtuális függvényeket kell meghívni az adott objektumra. A legtöbb fordítóprogramban ez az extra információ egy `vptr` (virtuális tábla pointer *virtual table pointer*) nevű pointer formájában van jelen. A `vptr` egy `vtbl` (virtuális tábla) nevű, függvénypointereket tartalmazó tömbre mutat. Minden olyan osztályhoz, amelyben van virtuális függvény, tartozik egy `vtbl` tömb. Amikor egy virtuális függvény meghívódik egy objektumra, akkor az aktuálisan meghívott függvényt úgy lehet meghatározni, hogy követjük az objektum `vptr`-ét a `vtbl`-be, és a `vtbl`-ben megkeressük a megfelelő függvénypointert.

A virtuális függvények implementációjának részletei nem fontosak. Az viszont fontos, hogy ha a `Point` osztálynak van virtuális függvénye, akkor e típus objektumainak a mérete értelemszerűen *megkétszereződik*: két 16 bites `short`-ról két 16 bites `short`-ra, meg egy 32 bites `vptr`-re! A `Point` objektumok már nem fognak beleférni egy 32-bites regiszterbe. Ezen kívül a C++-beli `Point` objektumok sem úgy néznek már ki, mint egy más nyelven (mondjuk C-ben) definiált, ugyanilyen struktúra, mert az idegennyelven megírt párjukban nem lesz `vptr`. Ennek eredményeképp a Point objektumok már nem adhatók át más nyelveken írt függvényeknek, és nem vehetők át azoktól, hacsak explicit nem pótoljuk a `vptr`-t, ami meg persze már implementációs részlet, és így nem hordozható.

A lényeg tehát az, hogy ugyanolyan rossz eljárás az összes destruktort ok nélkül virtuálisnak deklarálni, mint az, ha soha egyetlen destruktort sem deklarálunk virtuálisnak. Valójában nagyon sokan így összegzik a helyzetet: akkor és csak akkor deklaráljuk a destruktort egy osztályban virtuálisnak, ha az osztály tartalmaz legalább egy virtuális függvényt.

Ez egy jó szabály, és az esetek többségében működik is. De sajnos a nemvirtuális destruktor problémája virtuális függvények hiányában is okozhat nekünk fejfájást. A 13. jó tanács például egy osztálysablont vizsgál olyan tömbök implementálására, amelyeknek a felhasználó definiálhatja a határait. Tegyük fel, úgy döntünk, hogy írunk egy sablont azokhoz a származtatott osztályokhoz, amelyek nevesített tömböket *(named arrays)* reprezentálnak (azaz olyan osztályokhoz, amelyekben minden tömbnek neve van):

```
template<class T>              // Bázisosztály sablonja
class Array {                  // (a 13. jó tanácsból).
public:
   Array(int lowBound, int highBound);
   ~Array();
private:
   vector<T> data;
   size_t size;
   int lBound, hBound;
};
```

```
template<class T>
class NamedArray: public Array<T> {
public:
   NamedArray(int lowBound, int highBound, const string& name);
   ...
private:
   string arrayName;
};
```

Ha egy alkalmazás bármely pontján egy `NamedArray`-re mutató pointert valahogy `Array`-re mutatóvá konvertálunk, majd a `delete`-et használjuk az `Array` pointerre, azonnal a definiálatlan viselkedés területén találjuk magunkat:

```
NamedArray<int> *pna =
   new NamedArray<int>(10, 20, "Közelgő katasztrófa");
Array<int> *pa;
...
pa = pna;              // NamedArray<int>* -> Array<int>*
...
delete pa;             // Definiálatlan! (az X akták főcímzenéje
                       // jön ide); a gyakorlatban a
                       // pa->arrayName gyakran elszivárog,
                       // mert a *pa NamedArray része
                       // sosem lesz felszabadítva.
```

Ez a helyzet sokkal gyakrabban fordul elő, mintsem gondolnánk. Egyáltalán nem ritka ugyanis, hogy egy már létező, bizonyos feladatok elvégzésére alkalmas osztályból – ebben az esetben az `Array`-ből – egy olyan osztályt akarunk származtatni, amelyik pontosan ugyanazt tudja, sőt többre is képes. A `NamedArray` semmilyen szempontból sem definiálja át az `Array` viselkedését (változtatás nélkül örökli az összes függvényét), csak a képességei körét bővíti. A nemvirtuális destruktor problémája azonban így is megmarad.

Végezetül érdemes megemlíteni, hogy hasznos lehet bizonyos osztályokban tisztán virtuális destruktort *(pure virtual destructor)* deklarálni. Biztos emlékszünk, hogy a tisztán virtuális függvények *absztrakt* osztályokat eredményeznek: olyan osztályokat, amelyek nem példányosíthatók (értsd: ilyen típusú objektumokat nem lehet létrehozni). Néha viszont az a helyzet, hogy van egy osztályunk, amely szeretnénk, ha virtuális lenne, de épp nincs egyetlen tisztán virtuális függvényünk sem. Mit tegyünk ilyenkor? Nos, mivel egy absztrakt osztályt mindig bázisosztálynak szánunk, és mivel egy bázisosztálynak virtuális destruktora kell, hogy legyen, és mivel egy tisztán virtuális függvény absztrakt osztályt eredményez, a megoldás egyszerű: az absztraktnak szánt osztályban deklaráljunk egy tisztán virtuális destruktort.

Íme egy példa:

```
class AWOV {                  // AWOV = "Abstract w/o Virtuals"
                              // (absztrakt virtuálisok nélkül).
public:
```

```
    virtual ~AWOV() = 0;          // A tisztán virtuális destruktor
                                  // deklarációja.
};
```

Ez az osztály tartalmaz egy tisztán virtuális függvényt, tehát absztrakt. Másrészt virtuális destruktora van, így aztán nem kell rettegnünk a destruktorprobléma miatt. Van azonban a dologban egy csavar: meg kell adnunk a tisztán virtuális destruktor *definíció*ját:

```
AWOV::~AWOV() {}                  // A tisztán virtuális
                                  // destruktor definíciója.
```

Erre a definícióra azért van szükségünk, mert virtuális destruktorok esetében először a legalsó leszármazott osztály destruktora hívódik meg, majd sorra az egyes bázisosztályok destruktorai. Ez azt jelenti, hogy a fordítók generálni fognak egy `~AWOV` hívást akkor is, ha az osztály absztrakt, így aztán a függvény törzsét el kell készítenünk. Ha ezt nem tesszük meg, akkor a linker hiányzó szimbólumra fog panaszkodni, és akkor úgyis vissza kell mennünk, hogy létrehozzunk egyet.

Ebben a függvényben azt csinálhatunk, amit csak akarunk, de mint azt a fenti példában is láttuk, nem szokatlan, hogy nincs semmi tennivaló. Ha így van, akkor valószínűleg kísértésbe esünk, hogy az üres függvényhívás költségét megtakarítsuk, és a destruktort inline-nak deklaráljuk. Ez teljesen ésszerű stratégia, de van benne egy csavar, amit nem árt ismerni.

Mivel a destruktorunk virtuális, a címét be kell tenni az osztály `vtbl`-ébe. De mivel az inline függvények nem szoktak önálló függvényként létezni (ugyanis ez az `inline` jelentése!), intézkednünk kell, hogy címet szerezzünk nekik. A 33. jó tanács elmeséli az egész történetet, de a lényeg a következő: ha egy virtuális destruktort `inline`-nak deklarálunk, akkor meghívásakor valószínüleg megtakarítjuk a függvényhívás költségét, de a fordítóprogramnak így is létre kell hoznia egy soron kívüli másolatot a függvényből valahova.

15. jó tanács: Az `operator=` egy `*this`-re hivatkozó referenciával térjen vissza

Bjarne Stroustrupnak, a C++ tervezőjének, sok problémát kellett leküzdenie ahhoz, hogy a felhasználói típusok viselkedése a lehető legjobban hasonlítson a beépített típusokéra. Ezért terhelhetők túl *(overload)* az operátorok, ezért írhatunk típuskonverziós *(type conversion)* függvényeket, ezért határozhatjuk meg, hogy mi történjen értékadáskor *(assignment)* és másoláskor *(copy construction)* stb. Az ő fáradozásai után a legkevesebb, amit tehetünk, hogy folytatjuk e munkát.

Ez el is vezet minket az értékadáshoz. A beépített típusok esetén az értékadásokat láncba fűzhetjük, valahogy így:

```
int w, x, y, z;
w = x = y = z = 0;
```

Ezért aztán a felhasználói típusokra is meg kellene engedni az értékadások láncolását:

```
string w, x, y, z;          // A string egy „felhasználói"
                            // típus a szabvány C++
                            // könyvtárban (lásd a 49. jó tanácsot).

w = x = y = z = "Hello";
```

Amilyen a sors, az értékadó operátor jobbasszociatív, azaz az értékadáslánc szintaktikailag így értelmezendő:

```
w = (x = (y = (z = "Hello")));
```

Érdemes ugyanezt leírni a vele ekvivalens függvényformában is. Hacsak nem vagyunk titokban LISP programozók, e példa láttán hálásak leszünk az infix operátorok definiálásának lehetőségéért:

```
w.operator=(x.operator=(y.operator=(z.operator=("Hello"))));
```

Ez a forma nagyon szemléletes, mert hangsúlyozza, hogy a `w.operator=`, az `x.operator=` és az `y.operator=` paramétere az előző `operator=` hívás visszatérési értéke. Ezért aztán az `operator=` visszatérési típusának olyannak kell lenni, hogy azt maga a függvény bemeneti paraméterként is elfogadja. Egy C osztályban az alapértelmezett `operator=` szignatúrája a következő (lásd a 45. jó tanácsot):

```
C& C::operator=(const C&);
```

Majdnem mindig követendő ez a konvenció, vagyis az, hogy az `operator=` mind bemeneti paraméterként, mind visszatérési értékként egy osztálybeli objektumra hivatkozó referenciát használjon. Néha azonban túlterhelhetjük az `operator=`-t azért, hogy különböző paramétertípusokat is felvehessen. A standard `string` típus például két különböző értékadó operátorról is gondoskodik:

```
string&                          // Egy string értékül adása
operator=(const string& rhs);    // egy string-nek.

string&                          // Egy char* értékül adása
operator=(const char *rhs);      // egy string-nek.
```

Vegyük észre azonban, hogy még túlterhelés esetén is igaz az, hogy a visszatérési érték az osztály egy objektumára hivatkozó referencia.

Kezdő C++ programozók gyakran elkövetik azt a hibát, hogy az `operator=`-vel `void`-ot adatnak vissza. Ez a döntés elég ésszerűnek tűnik egészen addig, amíg rá nem jövünk, hogy ez megakadályozza az értékadásláncok használatát. Ezért ilyet ne tegyünk!

Egy másik gyakori hiba az, ha az `operator=`-vel egy `const` objektumra hivatkozó referenciát adatunk vissza, mint itt:

```
class Widget {
public:
    ...                                             // Vegyük észre
    const Widget& operator=(const Widget& rhs);     // a const
    ...                                             // visszatérési
};                                                  // típust.
```

Általában az a cél, hogy a felhasználók ne követhessenek el az alábbihoz hasonló ostobaságokat:

```
Widget w1, w2, w3;
...
(w1 = w2) = w3;          // w2 értékül adása w1-nek, majd w3
                         // értékül adása az eredménynek! (Ha a
                         // Widget operator=-jének const értéket
                         // adunk vissza, akkor ez nem fordul le.)
```

Ez azért mégsem akkora ostobaság, hogy beépített típusokra ne lenne megengedett:

```
int i1, i2, i3;
...
(i1 = i2) = i3;                // Szabályos! i2-t adja értékül
                               // i1-nek, majd i3-mat i1-nek!
```

Ennek a megoldásnak a gyakorlati felhasználására még nem láttam példát, de ha ez így jó az `int`-eknek, akkor nekem és az osztályaimnak is jó lesz. És szerintem az olvasónak és osztályainak is jó kell, hogy legyen. Miért is vezetnénk be indokolatlanul olyat, ami összeférhetetlen a beépített típusokra vonatkozó szabályokkal?

Az alapértelmezett szignatúrájú értékadó operátorban két nyilvánvaló jelölt is van a visszaadandó objektum szerepére. Az egyik az értékadás bal oldalán álló objektum (erre mutat a `this` pointer), a másik pedig a jobb oldalán szereplő (az az objektum, amelyik a paraméterlistában szerepel). Melyik a jó? Íme a lehetőségek egy `String` osztály esetén (ehhez az osztályhoz feltétlenül írunk értékadó operátort a 11. jó tanács alapján):

```
String& String::operator=(const String& rhs)
{
...
return *this;             // Visszatérés a bal oldali objektumra
                          // hivatkozó referenciával.
}
String& String::operator=(const String& rhs)
{
...
return rhs;               // Visszatérés a jobb oldali
                          // objektumra hivatkozó referenciával.
}
```

Első ránézésre az egyik tizenkilenc, a másik egy híján húsz, de fontos különbségek vannak a kettő között.

Először is, az rhs-t visszaadó változat nem fordul le, mert az rhs egy const String-re hivatkozó referencia, az operator= viszont egy String-re hivatkozó referenciát ad vissza. A fordítók végeláthatatlan szenvedést fognak okozni nekünk, ha nem const referenciát próbálunk meg visszaadni akkor, amikor maga az objektum const. Persze ezt könnyű kikerülni, elegendő újra deklarálnunk az operator=-t az alábbi módon:

```
String& String::operator=(String& rhs) { ... }
```

Sajnos így meg a felhasználó kódja nem fordul le! Nézzük csak meg újra az eredeti értékadáslánc utolsó darabját:

```
x = "Hello";                    // Ugyanaz, mint x.op=(„Hello”);
```

Mivel az értékadás jobb oldali paramétere nem a megfelelő típusú – char tömb és nem String –, a fordítónak egy ideiglenes String objektumot kellene létrehoznia (a String konstruktorán keresztül) ahhoz, hogy a hívás sikeres legyen. Ez azt jelenti, hogy az alábbival nagyjából egyező kódot kellene generálnia:

```
const String temp("Hello");     // Ideiglenes objektum létrehozása
x = temp;                       // az objektum átadása op=-nek.
```

A fordítók készségesen el is készítik ezt az ideiglenes objektumot (hacsak a hozzá tartozó konstruktor nem explicit – lásd a 19. jó tanácsot), de vegyük észre, hogy az ideiglenes objektum const. Ez azért fontos, mert így véletlenül sem tudunk egy ideiglenes objektumot olyan függvénynek átadni, amely változtatja a paraméterét. Ha ez megengedett lenne, akkor a programozók csodálkozva tapasztalnák, hogy csak a fordító által generált ideiglenes objektum változik meg, az a paraméter viszont, amelyet a hívás helyén adnak át, *nem*. (Ez már bizonyított tény, mert a C++ korai változatai megengedték az ilyen ideiglenes objektumok generálását, átadását és módosítását, az eredmény pedig egy rakás csodálkozó programozó lett.)

Most már láthatjuk, hogy a fenti felhasználói kód nem fordul le, ha a String osztály operator= műveletét úgy deklaráljuk, hogy egy nem const String-re hivatkozó referenciát kapjon paraméterként. Teljesen szabályellenes egy const objektumot olyan függvénynek átadni, amely az illeszkedő paramétert nem deklarálja const-nak. Egyszerűen a const-helyesség alapján kell eljárnunk.

Így aztán abban a boldog helyzetben találjuk magunkat, hogy nincs más választásunk, mindig úgy kell definiálnunk az értékadó operátorainkat, hogy azok a bal oldali paraméterükre hivatkozó referenciát adjanak vissza, azaz a *this-t. Bármilyen más megoldással megakadályoznánk az értékadások láncolását, vagy a hívások helyén az implicit típuskonverziót, vagy mindkettőt.

16. JÓ TANÁCS: AZ operator=-BEN MINDEN ADATTAGNAK ADJUNK ÉRTÉKET

A 45. jó tanácsból kiderül, hogy a C++ elkészíti számunkra az értékadó operátort *(assignment operator)*, ha mi nem deklarálunk magunknak egyet. A 11. jó tanács pedig elmagyarázza, hogy ez az értékadó operátor általában miért nem tetszik nekünk. Az olvasó lehet, hogy elgondolkozik azon, hogyan lehetne mindkettőből a jót megtartani, vagyis hagyni a C++-t, hogy generáljon egy alapértelmezett értékadó operátort, és csak a nekünk nem tetsző részeket felülírni. Nincs ilyen szerencsénk. Ha az értékadási folyamat bármelyik részének a vezérlését kezünkbe akarjuk venni, akkor az egészet nekünk kell csinálnunk.

A gyakorlatban ez azt jelenti, hogy az objektumunk *minden* adattagjának értéket kell adnunk, amikor az értékadó operátorunkat (operátorainkat) megírjuk:

```
template<class T>          // Sablon olyan osztályokhoz, melyek
class NamedPtr {           // neveket és pointereket rendelnek
                           // egymáshoz (a 12. jó tanácsból).
public:
   NamedPtr(const string& initName, T *initPtr);
   NamedPtr& operator=(const NamedPtr& rhs);
private:
   string name;
   T *ptr;
};
template<class T>
NamedPtr<T>& NamedPtr<T>::operator=(const NamedPtr<T>& rhs)
{
   if (this == &rhs)
   return *this;              // Lásd a 17. jó tanácsot.
   // értékadás az összes adattagnak
   name = rhs.name;           // Értékadás a name-nek.
   *ptr = *rhs.ptr;           // ptr-nél a mutatott
                              // objektumokat adjuk értékül,
                              // nem magát a pointert
   return *this;              // lásd a 15. jó tanácsot.
}
```

Elég egyszerű ezt a szabályt az emlékezetünkbe idézni, amikor az osztályt létrehozzuk, de ugyanilyen fontos, hogy az értékadó operátor(oka)t akkor is frissítsük, amikor új adattagokat adunk az osztályhoz. Például, ha úgy döntünk, hogy a `NamedPtr` sablon tartalmazza a név utolsó megváltoztatásának időpecsétjét, akkor egy új adattagot kell bevezetnünk, és emiatt mind a konstruktor(oka)t, mind az értékadó operátor(oka)t frissítenünk kell. Az osztály módosítgatása és az új tagfüggvények bevezetése körüli sürgés-forgásban nagyon könnyű megfeledkezni az ilyen frissítésekről és társairól.

Az igazi móka akkor kezdődik, amikor az öröklődés is csatlakozik a társasághoz, ugyanis a leszármazott osztály értékadó operátorának vagy operátorainak a bázisosztály adattagjainak értékadását is kezelni kell. Tekintsük az alábbi példát:

```
class Base {
public:
   Base(int initialValue = 0): x(initialValue) {}
private:
   int x;
};
class Derived: public Base {
public:
   Derived(int initialValue)
   : Base(initialValue), y(initialValue) {}
   Derived& operator=(const Derived& rhs);
private:
   int y;
};
```

A `Derived` értékadó operátorát logikusan így írnánk:

```
// hibás értékadó operátor
Derived& Derived::operator=(const Derived& rhs)
{
   if (this == &rhs) return *this;  // Lásd a 17. jó tanácsot.
   y = rhs.y;                       // Értékadás a Derived
                                    // egyetlen adattagjának.

   return *this;                    // lásd a 15. jó tanácsot
}
```

Pechünkre, ez így rossz, mert a `Derived` objektum `Base` részében lévő `x` adattagra ez az értékadó operátor nincs hatással. Tekintsük például ezt a kódrészletet:

```
void assignmentTester()
{
   Derived d1(0);              // d1.x = 0, d1.y = 0
   Derived d2(1);              // d2.x = 1, d2.y = 1
   d1 = d2;                    // d1.x = 0, d1.y = 1!
}
```

Figyeljük meg, hogy `d1`-nek a `Base` részét nem változtatja meg az értékadás.

A probléma egyszerű megoldása az lenne, hogy ha `x`-nek a `Derived::operator=`-ben adnánk értéket. Sajna, ez nem szabályos, mert `x` a `Base`-nek privát tagja. Ehelyett a `Derived` értékadó operátorának belsejéből explicit kell értéket adnunk a `Derived` objektum `Base` részének.

Ezt ekképp tehetjük meg:

```
// helyes értékadó operátor
Derived& Derived::operator=(const Derived& rhs)
{
   if (this == &rhs) return *this;
   Base::operator=(rhs);        // this->Base::operator= hívása.
   y = rhs.y;
   return *this;
}
```

Itt egyszerűen explicit meghívjuk a `Base::operator=`-t. Ez a hívás, mint minden olyan tagfüggvényhívás, amely más tagfüggvényből történik, a `*this`-t használja implicit bal oldali objektumként. Az eredmény az lesz, hogy a `Base::operator=` a `*this`-nek a `Base` részén végzi el a munkát, és mi pont ezt akarjuk elérni.

Sajnos, néhány fordító (helytelenül) nem engedi meg a bázisosztály értékadó operátorának ilyen meghívását, ha azt az értékadó operátort a fordító állította elő (lásd a 45. jó tanácsot). Az ilyen engedetlen fordítókat meg tudjuk békíteni, ha a `Derived::operator=` függvényt így implementáljuk:

```
Derived& Derived::operator=(const Derived& rhs)
{
   if (this == &rhs) return *this;
   static_cast<Base&>(*this) = rhs;     // operator= meghívása a
                                        // *this Base részére.
   y = rhs.y;
   return *this;
}
```

Ez a szörnyűség a `*this`-t először Base-re hivatkozó referenciává konvertálja *(cast)*, majd a konverzió eredményének ad értéket. Ez a `Derived` objektum `Base` részén végez csak értékadást. Azért legyünk óvatosak! Nagyon fontos, hogy egy `Base` objektumra vonatkozó *referenciává* konvertáljuk, ne pedig magává a `Base` objektummá. Ha ugyanis `*this`-t `Base` objektummá konvertáljuk, akkor a `Base` osztály másoló konstruktorát *(copy consrtuctor)* is elkerülhetetlenül meghívjuk, és ez az újonnan létrehozott objektum lesz az értékadás célja, a `*this` viszont változatlan marad. Mi pedig nem ezt akarjuk.

Függetlenül attól, hogy melyik megközelítést alkalmazzuk, miután értéket adtunk a `Derived` objektum `Base` részének, áttérhetünk a `Derived` osztály értékadó operátorára, értéket adva az összes `Derived`-beli adattagnak.

A származtatott osztályok másoló konstruktorának implementálásakor gyakran felmerül egy ehhez hasonló, ugyancsak az öröklődéssel kapcsolatos probléma. Nézzük az alábbi, az imént vizsgált kód analógiájára készített másoló konstruktoros példát:

```
class Base {
public:
   Base(int initialValue = 0): x(initialValue) {}
```

```
    Base(const Base& rhs): x(rhs.x) {}
private:
    int x;
};
class Derived: public Base {
public:
    Derived(int initialValue)
    : Base(initialValue), y(initialValue) {}

    Derived(const Derived& rhs)     // Hibás másoló
    : y(rhs.y) {}                   // konstruktor.

private:
    int y;
};
```

A `Derived` osztály jól példázza a C++ világában az egyik legkomiszabb programhibát: amikor egy `Derived` objektum másoló konstruktorral jön létre, a bázisosztály rész nem másolódik le. Természetesen az ilyen `Derived` objektum `Base` része is konstruktorral jön létre, de a `Base` *alapértelmezett* konstruktorával. Az `x` adattag 0-ra (az alapértelmezett konstruktor paraméterének alapértelmezett értékére) lesz inicializálva, függetlenül a másolt objektum `x` adattagjának értékétől!

A probléma megoldásához a `Derived` osztály másoló konstruktorának biztosítania kell, hogy a `Base` alapértelmezett konstruktora helyett a másoló konstruktora hívódjon meg. Ezt könnyű megtenni. Csak meg kell adnunk egy inicializáló értéket a `Base` számára a `Derived` másoló konstruktorának taginicializáló listájában:

```
class Derived: public Base {
public:
    Derived(const Derived& rhs): Base(rhs), y(rhs.y) {}
    ...
};
```

Ezek után, ha a felhasználó egy `Derived`-ot már meglévő, ugyanilyen típusú objektum másolásával hoz létre, akkor a `Base` részt is lemásolja.

17. JÓ TANÁCS: ELLENŐRIZZÜK AZ ÖNÉRTÉKADÁST AZ `operator=`-BEN

Önértékadás *(assignment to self)* akkor fordulhat elő, ha valami ilyesmit teszünk:

```
class X { ... };
X a;
a = a;                       // a értékadása önmagának.
```

Ez elég idétlen dolognak tűnhet, de teljesen szabályos, úgyhogy egy pillanatig se kételkedjünk abban, hogy a programozók ezt meg is fogják tenni. Ami még fontosabb, az önértékadás az alábbi, szelídebbnek tűnő formában is megjelenhet:

```
a = b;
```

Ha `b` csak `a` egy másik neve (például egy `a`-ra inicializált referencia), akkor ez is egy önértékadás, csak nem látszik rajta. Ez az álnevesítés *(aliasing)* egy példája: kettő vagy több nevünk van ugyanarra az alapobjektumra. Ahogy azt ennek a fejezetnek a végén látni fogjuk, az álnevek gonoszabbnál gonoszabb álruhába bújhatnak, úgyhogy ezt függvényíráskor mindig figyelembe kell vennünk.

Két oka is van annak, miért kell az értékadó operátorokban különösen nagy gondot fordítanunk arra, hogy egy lehetséges álnevesítés ne fogjon ki rajtunk. A kisebbik ok a hatékonyság. Ha kiszűrjuk az önértékadást az értékadó operátorunk elején, akkor egyből visszatérhetünk, és így sok olyan munkát takaríthatunk meg, amelyet egyébként az értékadás implementálása során el kellene végeznünk. A 16. jó tanács például felhívja a figyelmet arra, hogy egy származtatott osztály jól megírt értékadó operátorának minden egyes bázisosztályhoz meg kell hívnia egy értékadó operátort. És mivel ezek a bázisosztályok szintén lehetnek származtatottak, sok függvényhívást megspórolhatunk, ha átugorjuk a származtatott osztály(ok) értékadó operátorának törzsét.

Az önértékadás ellenőrzésének másik, fontosabb oka a helyes programozás biztosítása. Emlékezzünk vissza, hogy egy értékadó operátornak először rendszerint fel kell szabadítania az objektumhoz rendelt erőforrásokat (azaz meg kell szabadulnia annak régi értékétől), és csak utána foglalhat az új értéknek megfelelően új erőforrásokat. Önértékadás során az erőforrásoknak ez a felszabadítása katasztrofális lehet, mert a régi erőforrásokra szükség lehet az újak létrehozásakor.

Vizsgáljuk meg közelebbről az alábbi `String` objektumok értékadását. Ezeknél az értékadó operátor nem ellenőrzi az önértékadást:

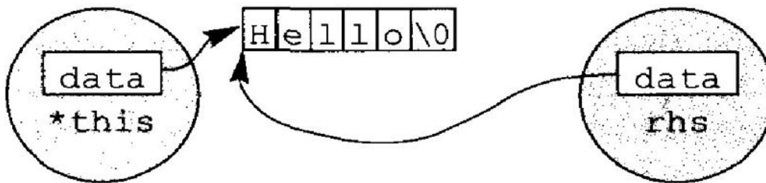
```
class String {
public:
   String(const char *value); // A függvény definíciója
                              // a 11. jó tanácsban található.
   ~String();                 // A függvény definíciója
                              // a 11. jó tanácsban található.
   ...
   String& operator=(const String& rhs);
private:
   char *data;
};
// Értékadó operátor, amely kihagyja az
// önértékadás ellenőrzését.
String& String::operator=(const String& rhs)
{
   delete [] data;         // Régi memóriaterület felszabadítása.
   // Új memóriaterület foglalása, és az rhs értékének odamásolása
```

```
    data = new char[strlen(rhs.data) + 1];
    strcpy(data, rhs.data);
    return *this;                  // Lásd a 15. jó tanácsot.
}
```

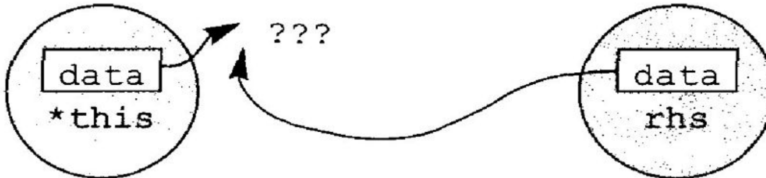
Gondoljuk végig mi történik ebben az esetben:

```
String a = "Hello";
a = a;                             // Ugyanaz, mint a.operator=(a).
```

Az értékadó operátoron belül a `*this` és az `rhs` különböző objektumnak látszanak, de ebben az esetben csak különböző nevei ugyanannak az objektumnak. Ezt az alábbi ábra szemlélteti:



Az értékadó operátor első dolga az, hogy meghívja a `delete`-et a `data`-ra, ami az alábbi helyzetet eredményezi:



Így, amikor az értékadó operátor megpróbálja meghívni az `strlen`-t az `rhs.data`-ra, az eredmény definiálatlan. Méghozzá azért, mert az `rhs.data` törlődött, amikor `data` törölve lett, hiszen `data`, `this->data`, és `rhs.data` mind ugyanaz a pointer! Innét kezdve a dolgok már csak rosszabbodhatnak.

Mostanra már tudjuk, hogy a helyzet kulcsa az önértékadás vizsgálata, majd az azonnali visszatérés, ha ilyen értékadás nyomaira bukkanunk. Sajnálatos módon sokkal könnyebb egy ilyen ellenőrzésről beszélni, mint megírni, ugyanis egyből azzal kell szembesülnünk, mit is jelent az, hogy két objektum „ugyanaz".

Az előttünk álló feladatot az *„objektum identitásának problémája"* néven szokták emlegetni, és elég ismert téma objektumorientált körökben. Ennek a könyvnek nem célja az objektumidentitás megvitatása, de talán érdemes megemlítenünk a probléma két alapvető megközelítését.

Az egyik megközelítés szerint két objektum akkor egyezik meg (akkor ugyanaz az identitása), ha egyenlő az értékük. E szerint például két `String` objektumot akkor tekintenénk azonosnak, ha ugyanazt a karaktersorozatot reprezentálnák:

```
String a = "Hello";
String b = "World";
String c = "Hello";
```

Itt a-nak és c-nek ugyanaz az értéke, így őket megegyezőnek tekintjük, míg b mindkettőjüktől különbözik. Ha az identitásnak ezt a definícióját használnánk a String osztályunkban, akkor az értékadó operátor valahogy így nézne ki:

```
String& String::operator=(const String& rhs)
{
   if (strcmp(data, rhs.data) == 0) return *this;
   ...
}
```

Az értékek egyenlőségét gyakran az operator== határozza meg, ezért egy olyan C osztályban, ahol az objektum-identitást az értékek egyenlősége definiálja, az értékadó operátor így írható:

```
C& C::operator=(const C& rhs)
{
   // önértékadás ellenőrzése
   if (*this == rhs)          // Felteszi, hogy op== létezik.
      return *this;
   ...
}
```

Figyeljük meg, hogy ez a függvény *objektumokat* hasonlít össze (az operator==-n keresztül), és nem pointereket. Ha az értékek egyenlősége határozza meg az objektumok azonosságát, akkor nem érdekes, hogy két objektum ugyanazt a memóriaterületet foglalja-e el, csak az általuk reprezentált értékek számítanak.

A másik lehetőség az, ha egy objektum identitását a memóriabeli címével azonosítjuk. Az egyezőség ezen definíciója szerint két objektum akkor és csak akkor egyezik meg, ha ugyanazon a memóriacímen találhatók. Ez a definíció sokkal gyakoribb a C++ programokban, ugyanis egyszerűen implementálható és gyorsan végrehajtható, ami nem mindig mondható el akkor, ha az objektumok identitását az értékek alapján vizsgáljuk. Ha a címek egyezőségét vizsgáljuk, egy szokásos értékadó operátor így néz ki:

```
C& C::operator=(const C& rhs)
{
   // Önértékadás ellenőrzése.
   if (this == &rhs) return *this;
   ...
}
```

Ez nagyon sok programban elegendő is.

Ha mégis kifinomultabb eljárásra van szükség két objektum egyezőségének meghatározására, akkor azt saját magunknak kell implementálnunk. Az egyik leggyakoribb megközelítés egy olyan tagfüggvény bevezetése, amely valamilyen objektumazonosítót ad vissza:

```
class C {
public:
   ObjectID identity() const;     // Lásd még a 36. jó tanácsot.
   ...
};
```

Ha adott két pointer, a és b, akkor az általuk mutatott objektumokat akkor és csak akkor tekintjük azonosnak, ha `a->identity() == b->identity()`. Természetesen mi vagyunk felelősek azért, hogy az `ObjectID` számára elkészüljön az `operator==`.

Az álnevek és az objektumok identitásának problémája persze nem csak az `operator=` keretein belül fordulhat elő, de ez egy olyan függvény, ahol különösen nagy valószínűséggel botlunk beléjük. Ahol referenciák és pointerek fordulnak elő, bármely két, egymással összeegyeztethető típusú objektum neve vonatkozhat ugyanarra az objektumra. Íme néhány további eset, amelyben az álnevek Medúza-szerű arcot mutatnak:

```
class Base {
   void mf1(Base& rb);           // rb és *this ugyanaz is lehet.
   ...
};
void f1(Base& rb1,Base& rb2); // rb1 és rb2 ugyanaz is lehet.

class Derived: public Base {
   void mf2(Base& rb);           // rb és *this ugyanaz is lehet.
   ...
};
int f2(Derived& rd, Base& rb); // rd és rb ugyanaz is lehet.
```

Ezekben a példákban most épp referenciák szerepelnek, de pointerekkel is végigjátszható ugyanez.

Ahogy láttuk, az álnevek sokféle álruhát ölthetnek, ezért aztán nem szabad megfeledkeznünk róluk abban reménykedve, hogy nem futunk össze velük. Lehet, hogy ezt az *olvasó* megteheti, de a legtöbbünk nem. Könnyen lehet, hogy ez most képzavar lesz, de ez nekem tipikus este annak, amikor egy csepp gondosság a súlyával egyenértékű aranyat ér. Ha olyan függvényt írunk, ahol az álnevek jelenléte elképzelhető, a kód írásakor ezt a lehetőséget mindenképpen figyelembe *kell* vennünk.

Osztályok és függvények: Tervezés és deklaráció

Ha programunkban új osztályt deklarálunk, akkor egy új típus is létrejön, azaz egy osztály tervezése egy *típus* tervezését is jelenti. Az olvasónak valószínűleg nincs sok gyakorlata típusok tervezésében, mert a legtöbb nyelv egyáltalán nem ad lehetőséget ilyen irányú gyakorlat megszerzésére. C++-ban azonban ez alapvető fontosságú, egyrészt, mert bármikor tervezhetünk típust, másrészt, mert egy osztály deklarálásával úgyis ezt tesszük, akár akarjuk, akár nem.

Nehéz jó osztályt tervezni, mert jó típust is nehéz tervezni. Egy jó típusnak magától értetődő a szintakszisa, ötletes a szemantikai szerkesztése és akár több feladat megoldására is hatékonyan alkalmazható. C++-ban egy elhibázott osztálydefiníció lehetetlenné teheti az előbbi célok elérését. A tagfüggvények *(member functions)* definíciója – és persze a deklarációja is – meghatározza egy osztály tagfüggvényeinek a használhatóságát is.

Hogyan tervezhetünk hatékony osztályokat? Először is meg kell értenünk, milyen problémákkal állunk szemben. Szinte minden osztály esetében számolnunk kell az alábbi kérdésekkel. A rájuk adott válaszok gyakran korlátozzák lehetőségeinket a tervezésben.

- **Hogyan hozzunk létre és semmisítsünk meg objektumokat?** Ez erősen befolyásolja konstruktoraink és destruktorunk tervezését, valamint a saját `operator new`, `operator new[]`, `operator delete`, és `operator delete[]` verziónkat, ha írunk egyáltalán ilyeneket.
- **Miben különbözik az objektuminicializálás az objektum-értékadástól** ***(object assignment)*****?** A válasz meghatározza a konstruktorok és az értékadó operátorok viselkedését és a köztük lévő különbségeket.
- **Mit jelent az új típusú objektumok érték szerinti átadása?** Jegyezzük meg jól, hogy a másoló konstruktor *(copy constructor)* határozza meg, mit jelentsen az objektumok érték szerinti átadása.
- **Milyen korlátozások vonatkoznak az új típus érvényes értékeire?** Ezek a korlátozások határozzák meg, milyen hibaellenőrzést kell elvégeznünk a tagfüggvények belsejében, különösen a konstruktorok és az értékadó operátorok esetében. Befolyásolhatják a függvények által dobott kivételeket és a függvényeink kivételspecifikációját is, ha használjuk egyáltalán ezt a lehetőséget.

- **Beleillik-e az új típus egy öröklődési gráfba?** Ha meglévő osztályoktól származtatunk, azok felépítéséhez kell igazodnunk, különösen ahhoz, hogy az örökölt függvények virtuálisak-e vagy sem. Ha meg akarjuk engedni, hogy a mi osztályunktól örökölhessenek más osztályok, azt kell eldöntenünk, mely függvényeinket deklaráljuk virtuálisnak.
- **Milyen típuskonverziókat engedjünk meg?** Ha meg akarjuk engedni egy A objektum *implicit* konverzióját egy B objektumra, akkor írjunk vagy egy típuskonverziós függvényt *(type conversion function)* az A osztályban, vagy egy olyan nem `explicit` konstruktort a B osztályban, amelyet egy paraméterrel meg lehet hívni. Ha csak az *explicit* konverziókat akarjuk megengedni, írjunk hozzájuk függvényeket, de ne írjunk se típuskonverziós operátorokat, se egyparaméteres nem `explicit` konstruktorokat.
- **Milyen operátoroknak és függvényeknek van értelme az új típus esetében?** A kérdésre adott válasz határozza meg, hogy milyen függvényeket fogunk deklarálni az osztályfelületen.
- **Milyen szabvány operátorokat és függvényeket tiltsunk le?** Ezeket `private`-ként kell deklarálnunk.
- **Kinek legyen hozzáférése az új típus tagjaihoz?** Ez a kérdésfelvetés segít eldönteni, hogy mely tagokat tegyük nyilvánossá, melyeket védetté és melyeket privátttá. Abban is segít, hogy mely osztályok és/vagy függvények legyenek barátok *(friends)*, valamint, hogy van-e értelme beágyazni az egyik osztályt a másikba.
- **Mennyire általános az új típus?** Előfordulhat, hogy nem is egy új típust definiálunk. Lehet, hogy típusok egy egész *családját* hozzuk létre. Ha így van, ne egy új osztályt definiáljunk, hanem inkább egy új osztálysablont.

Nem könnyű válaszolni ezekre a kérdésekre, ezért nem is olyan egyszerű hatékony osztályokat definiálni C++-ban. Ha mégis sikerül, a felhasználó által definiált osztályok olyan típusokat eredményeznek, amelyek szinte megkülönböztethetetlenek a beépített típusoktól, ez pedig megéri a fáradozást.

Egy-egy fenti kérdés részletes tárgyalása már maga kitenne egy önálló könyvet, ezért az itt következő iránymutatás nem terjed ki mindenre. Megvilágítja viszont a tervezés legfontosabb szempontjait, figyelmeztet a leggyakoribb hibákra, és megoldást ad az osztályok tervezésekor felmerülő leggyakoribb problémákra. A tanácsok nagy része éppúgy alkalmazható nemtagfüggvényekre *(non-member functions)*, mint tagfüggvényekre, így a globális és a névtérben helyet foglaló *(namespace-resident)* függvények tervezését és deklarálását is tárgyalni fogom ebben a részben.

18. JÓ TANÁCS: TÖREKEDJÜNK TELJES ÉS MINIMÁLIS OSZTÁLYFELÜLETEK KIALAKÍTÁSÁRA

Az osztály felhasználói felülete *(client interface)* az osztályt használó programozó számára hozzáférhető felület. Általában csak függvények vannak ezen a felületen, mert az adattagok felhasználói felületen való szerepeltetésének számos hátulütője van (lásd a 20. jó tanácsot).

Azt kigondolni, hogy milyen függvények kerüljenek egy osztályfelületbe, őrületbe kergetheti az embert. Két teljesen különböző elvárásnak szeretnénk egyszerre megfelelni. Egyrészt szeretnénk olyan osztályt készíteni, amely könnyen érthető és könnyen implementálható, a használata pedig pofonegyszerű. Ez minél kevesebb tagfüggvény *(member function)* felvételét sejteti, amelyek mindegyike jól elkülöníthető feladatot lát el. Másrészt szeretnénk, ha osztályunkat kényelmesen és minél több feladatra lehetne használni. Ez sokszor jelenti olyan új függvények felvételét, amelyek támogatják a gyakori feladatok elvégzését. Hogyan döntsük el, melyik függvény kerüljön az osztályba és melyik ne?

Próbáljuk meg a következőt: törekedjünk *teljes* és *minimális* osztályfelület létrehozására.

Egy *teljes* felület megengedi a felhasználónak, hogy az ésszerűség határain belül bármit megtegyen. Ez azt jelenti, hogy a felhasználó bármilyen ésszerű feladatra tud találni egy ésszerű megoldást, még ha az nem is olyan kényelmes, mint szeretné. Ezzel szemben egy *minimális* felület a lehető legkevesebb függvényt tartalmazza, a felület függvényeinek működésében pedig nincs átfedés. Ha teljes, ugyanakkor minimális felületet nyújtunk, a felhasználók bármit meg tudnak valósítani, az osztályfelület mégsem lesz bonyolultabb a kelleténél.

Elég egyértelmű, miért kívánatos a teljes felület, de miért kell, hogy minimális legyen? Miért ne adjunk meg a felhasználóknak mindent, amit csak kérnek? Miért ne bővíthetnénk a felhasználhatóság körét egészen addig, míg mindenki elégedett nem lesz?

A morális vonatkozástól eltekintve – egyáltalán *helyes* dolog-e elkényeztetni felhasználóinkat? – gyakorlati szempontból is határozottan hátrányos a függvényekkel teletömött osztályfelület. Először is, minél több függvény van benne, annál nehezebben érthető egy potenciális felhasználó számára. Minél nehezebben érthető, annál kevésbé élvezetes elsajátítani, mire alkalmazható. Egy tíz függvényből álló osztályt a legtöbben kezelhetőnek tartanak, de egy száztagúval már sok programozót ki lehetne kergetni a világból. Míg arra törekszünk, hogy újabb és újabb függvények hozzáadásával az osztályunkat egyre vonzóbbá tegyük, könnyen úgy járhatunk, hogy teljesen elvesszük a felhasználók kedvét az osztály használatának megismerésétől.

Egy nagy felület zűrzavart is okozhat. Tegyük fel, hogy létrehozunk egy osztályt, amely a kognitív funkciókat támogatja egy mesterséges intelligencia-alkalmazásban. Az egyik tagfüggvényét elnevezzük `gondolkodás`-nak, de később kiderül, hogy néhányan `fontolgatás`-nak, mások inkább `töprengés`-nek hívnák. Hogy mindenkinek a kedvében járjunk, mind a három függvényt rendelkezésre bocsátjuk, noha mindegyik pontosan ugyanazt tudja. Képzeljük magunkat annak a potenciális felhasználónak a helyzetébe, aki ki akar ezen igazodni! Szembe találja magát három különböző függvénnyel, amelyek állítólag ugyanúgy működnek. Ez biztosan így van? Nincs esetleg valami egészen apró különbség a három között, talán hatékonyságban, megbízhatóságban vagy az alkalmazhatóság terén? Ha nincs, miért van belőlük három? Ez a potenciális felhasználó nem a rugalmasságunkat fogja díjazni, hanem inkább azon fogja törni a fejét, hogy mi az ördögre gondoltunk – mit fontolgattunk, min töprengtünk.

A nagy osztályfelület másik hátránya a karbantartásnál jelentkezik. Sokkal bonyolultabb egy nagy, sok függvényből álló osztály bővítése és karbantartása, mint egy kicsié. Sokkal nehezebben tudjuk elkerülni a kódismétléseket – nem is beszélve az ezzel járó hibaismétlésekről! – és nehezebben tudunk következetesek maradni a felület egészét tekintve. Ezen kívül nehezebb dokumentálni is.

Végül pedig, a hosszú osztálydefiníciók hosszú fejállományokat *(header files)* eredményeznek. Mivel fordításkor a fejállományokat minden alkalommal be kell olvasni (lásd a 34. jó tanácsot), a kelleténél hosszabb osztálydefiníciók nagy valószínűséggel megbosszulják magukat, ha az egy projektre jutó összes fordítási időt nézzük.

Egy szó, mint száz, ha fölöslegesen adunk függvényeket egy felülethez, mindenképp fizetnünk kell érte. Alaposan fontoljuk meg, hogy nem fizetünk-e túl nagy árat a komplexitás, érthetőség, karbantarthatóság és fordítási sebesség terén az új függvénnyel járó kényelemért cserébe: ugyanis csak kényelmi szempontokat szolgálhat egy új függvény hozzáadása a már teljes felülethez. Annak sincs azonban értelme, hogy túlságosan zsugoriak legyünk. Gyakran tényleg indokolt többet nyújtani egy minimális függvénykészletnél. Ha egy rendszeresen végzett feladat sokkal hatékonyabban kivitelezhető tagfüggvényként, akkor jogosan vesszük fel a felületbe. Ha egy új tagfüggvény lényegesen megkönnyíti az osztály használatát, ez is elég ok a beépítésre. Az is nyomós érv, ha egy új tagfüggvény hozzáadásával felhasználói hibákat előzhetünk meg.

Nézzünk egy konkrét példát. Tekintsünk egy sablont olyan osztályokra, amelyek a felhasználó által megadott felső és alsó határokkal rendelkező tömböket hoznak létre, a határok opcionális ellenőrzésével. Egy ilyen tömbsablon kezdetei láthatók az alábbiakban:

```
template<class T>
class Array {
public:
   enum BoundsCheckingStatus {NO_CHECK_BOUNDS = 0,
                              CHECK_BOUNDS    = 1};
   Array(int lowBound, int highBound,
         BoundsCheckingStatus check = NO_CHECK_BOUNDS);
   Array(const Array& rhs);
   ~Array();
   Array& operator=(const Array& rhs);
private:
   int lBound, hBound;        // Alsó határ, felső határ.
   vector<T> data;            // A tömb tartalma; a vector-ról
                              // a 49. jó tanácsban található info.
   BoundsCheckingStatus checkingBounds;
};
```

Az eddig definiált tagfüggvényeket szinte gondolkodás (fontolgatás, töprengés) nélkül meg lehet írni. Van egy konstruktorunk, amelynek segítségével a felhasználó megadhatja az egyes tömbök határait, van egy másoló konstruktorunk, egy értékadó operátorunk és egy destruktorunk. Ebben az esetben a destruktort nemvirtuálisnak deklaráltuk, ami azt is jelenti, hogy ezt az osztályt nem ajánlatos bázisosztályként használni (lásd a 14. jó tanácsot).

Az értékadó operátor fenti deklarációja tulajdonképpen nem annyira egyértelmű, mint amilyennek elsőre látszik. Miután a C++ beépített tömbjei nem engedik meg az értékadást, azt gondolhatnánk, hogy ezt nekünk is le kell tiltanunk a saját `Array` objektumaink esetében (lásd a 27. jó tanácsot). A tömbre nagyon hasonlító `vector` sablon viszont – amely a szabvány könyvtár eleme (lásd a 49. jó tanácsot) – megengedi az

értékadást a `vector` objektumok között. Példánkban a `vector`-t vesszük irányadónak, és ahogyan a következőkben látni fogjuk, ez a döntés az osztály felületének más részeire is hatással lesz.

Öreg C guruk megijednének egy ilyen felülettől. Hol van ebben a támogatás egy meghatározott méretű tömb deklarálására? Elég könnyű lenne egy új konstruktort beilleszteni,

```
Array(int size,
      BoundsCheckingStatus check = NO_CHECK_BOUNDS);
```

de egy minimális felületben ennek már nincs helye, mert a felső és alsó határ megadására szolgáló konstruktort is használhatjuk helyette. Ennek ellenére bölcs politikai húzás lenne a vén tengerimedvék kedvére tenni, mondjuk „az alapnyelv következetes követése" címszó alatt.

Milyen függvényekre lesz még szükségünk? Egy teljes felülethez mindenképpen hozzátartozik a tömbök indexelhetősége.

```
// Visszatérési elem íráshoz/olvasáshoz.
T& operator[](int index);
// Visszatérési elem csak olvasáshoz.
const T& operator[](int index) const;
```

Ha ugyanazt a függvényt kétszer, egyik esetben `const`-ként, a másikban nem `const`-ként deklaráljuk, támogatást nyújtunk mind a `const`, mind pedig a nem `const` `Array` objektumoknak. A visszatérési típusokban jelentős különbség lesz. Ennek magyarázata a 21. jó tanácsban található.

Jelen állás szerint az `Array` sablon támogatja a létrehozást, a megsemmisítést, az érték szerinti átadást, az értékadást és az indexelést, ami azt az érzést keltheti, hogy teljes felületről van szó. De nézzük meg közelebbről. Tegyük fel, hogy egy felhasználó ciklust akar futtatni egy egészekből álló tömbön, kiíratva minden elemét, valahogy így:

```
Array<int> a(10, 20);          // „a" határai 10 és 20
...
for (int i = "a" alsó határa; i <= "a" felső határa; ++i)
   cout << "a[" << i << "] = " << a[i] << '\n';
```

Hogyan fogja a felhasználó az `a` tömb határait megkapni? Attól függ, mi történik az `Array` objektumok értékadásakor, vagyis az `Array::operator=` belsejében. Ha például az értékadás megváltoztathatja egy `Array` objektum határait, olyan tagfüggvényeket kell rendelkezésre bocsátani, amelyek az éppen aktuális határokat adják vissza, mert a felhasználók nem fogják *a priori* tudni, hogy a program bármely adott pontján mik a határok. Ha a fenti példában az `a` tömb definiálása és a ciklusban való felhasználása közötti időben értékadás történne, a felhasználó biztosan nem tudná meghatározni `a` aktuális határait.

Ugyanakkor, ha egy `Array` objektum határait nem lehet értékadás közben megváltoztatni, akkor azok a definiáláskor rögzítve lettek. Ebben az esetben a felhasználó egy kis odafigyeléssel nyomon tudná követni a határokat. A kényelem kedvéért fel lehetne kínálni

az aktuális értékhatárt visszaadó függvényeket, de ezek egy igazán minimális felülethez már nem tartoznak hozzá.

Továbbgondolva azt az esetet, amikor az értékadás módosíthatja egy objektum határait, az értékhatár-függvényeket a következőképpen deklarálhatjuk:

```
int lowBound() const;
int highBound() const;
```

Mivel ezek a függvények nem módosítják azt az objektumot, amelyre meghívják őket, és mi, ahol csak lehet, a `const` használatát részesítjük előnyben (lásd a 21. jó tanácsot), mindkettőt `const` tagfüggvénynek deklaráljuk. Ezekkel a függvényekkel a fenti ciklus így festene:

```
for (int i = a.lowBound(); i <= a.highBound(); ++i)
   cout <<  "a[" << i << "] = "<< a[i] << '\n';
```

Talán szükségtelen megemlíteni, de a fenti `T` típusú objektumokból álló tömbre írt ciklus működéséhez definiálni kell egy `operator<<` függvényt a `T` típusú objektumokhoz. (Ez így nem teljesen pontos, mert `T`-hez, vagy egy olyan típushoz, kell léteznie egy `operator<<`-nak, amellyé `T` implicit módon konvertálható. A lényeg így is érthető.)

Biztosan van olyan programtervező, aki most amellett érvelne, hogy az `Array` osztályban olyan függvényre is szükség lenne, amely egy `Array` objektum elemeinek a számát adja vissza. Nincs igazán szükség ilyen függvényre, mert az elemek száma egyszerűen `highBound()-lowBound()+1`. Az ötlet mégsem olyan rossz, figyelembe véve, hogy milyen gyakran számoljuk el ezt az eredményt eggyel.

Érdemes lehet felvenni az osztályba input és output feladatokat végző függvényeket és különböző relációs operátorokat (pl. `<`, `>`, `= =` stb.) is. Ezek a függvények azonban már nem részei egy minimális felületnek, mert mindegyik megvalósítható az `operator[]` hívását tartalmazó ciklusokkal.

Ha már az `operator<<`, `operator>>` függvényekről és a relációs operátorokról van szó, a 19. jó tanács témája, hogy miért implementálják őket tagfüggvény helyett sokszor inkább nemtag-barátfüggvényként *(non-member friend functions)*. Ez lévén a helyzet, ne feledjük, hogy gyakorlati szempontból a barátfüggvények is hozzátartoznak egy osztály felületéhez. Ez azt jelenti, hogy a barátfüggvények is számítanak, ha egy osztály teljességét és minimális voltát tekintjük

19. jó tanács: Tegyünk különbséget tagfüggvények, nemtagfüggvények és barátfüggvények között

A tag- és nemtagfüggvények *(member and non-member functions)* között az a legnagyobb különbség, hogy a tagfüggvények lehetnek virtuálisak, a nemtagfüggvények viszont nem. Ezért, ha van egy függvényünk, amelynek dinamikusan kötöttnek *(dynamically bound)* kell lennie (lásd a 38. jó tanácsot), olyan virtuális függvényt kell használnunk, amely

tagja valamilyen osztálynak. Ennyire egyszerű. Ha viszont a függvényünknek nem muszáj virtuálisnak lennie, zavarosabb vizekre kell eveznünk.

Tekintsünk az alábbiakban egy olyan osztályt, amely a racionális számokat ábrázolja:

```
class Rational {
public:
   Rational(int numerator = 0, int denominator = 1);
   int numerator() const;
   int denominator() const;
private:
   ...
};
```

Ez az osztály most még semmire sem használható. (A 18. jó tanács fogalmaival élve: a felület minimális, de *messze* nem teljes.) Azt tudjuk, hogy szeretnénk támogatni az olyan aritmetikai műveleteket, mint az összeadás, kivonás, szorzás stb. Abban viszont nem vagyunk biztosak, hogy tagfüggvénnyel, nemtagfüggvénnyel vagy nemtag-barát-függvénnyel *(a non-member friend function)* valósítsuk-e meg ezt a célt.

Ha ilyen kétségek merülnek fel bennünk, legyünk objektumorientáltak! Azt tudjuk például, hogy a racionális számok szorzása a `Rational` osztályhoz kapcsolódik. Próbáljuk meg összekötni a műveletet az osztállyal úgy, hogy tagfüggvényt írunk hozzá:

```
class Rational {
public:
   ...
   const Rational operator*(const Rational& rhs) const;
};
```

(Ha az olvasó nem érti pontosan, miért így deklaráltuk ezt a függvényt, azaz miért érték szerinti `const`-tal tér vissza, amikor referencia szerinti `const` paramétert vesz át, lapozzon a 21., 22. és 23. jó tanácshoz.)

Most már nagyon könnyen megy a racionális számok szorzása:

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);

Rational result = oneHalf * oneEighth;     // Rendben.
result = result * oneEighth;               // Rendben.
```

De nem vagyunk elégedettek. Szeretnénk támogatni az olyan vegyes műveleteket is, ahol a `Rational` szorozható mondjuk int-tel. Ha ezzel próbálkozunk, félmegoldást kapunk:

```
result = oneHalf * 2;                      // Rendben.
result = 2 * oneHalf;                      // Hiba!
```

Ez rossz jel. Ugye emlékszünk még arra, hogy a szorzás kommutatív?

A probléma láthatóvá válik, ha az utolsó két példát átírjuk a nekik megfelelő függvényalakba:

```
result = oneHalf.operator*(2);             // Rendben.

result = 2.operator*(oneHalf);             // Hiba!
```

A `oneHalf` objektum egy olyan osztály példánya, amely tartalmaz egy `operator*`-ot, így a fordítók ezt a függvényt meghívják. Ezzel ellentétben a 2-höz mint egész számhoz nem társítható osztály, így `operator*` tagfüggvény sem. A fordítóprogramok olyan nem tag – vagyis látható névtérben *(visible namespace)* lévő, vagy globális – `operator*` után is kutatni fognak, amely a következőképpen hívható meg:

```
result = operator*(2, oneHalf);            // Hiba!
```

Sikertelen lesz a keresés, mivel nem létezik olyan nem tag `operator*`, amely elfogadna egy `int`-et és egy `Rational`-t is.

Nézzük meg újból azt a függvényhívást, amelyik sikerült. Látjuk, hogy a második paramétere a 2 egész szám, noha a `Rational::operator*` egy Rational objektumot vesz át paraméterként. Mi folyik itt? Miért működik a 2 az egyik helyen, és miért nem működik a másikon?

Azért, mert implicit típuskonverzió *(implicit type conversion)* megy végbe. A fordítók tudják, hogy egy `int`-et adunk át, annak ellenére, hogy a függvény `Rational`-t vár. De azt is tudják, hogy elő tudnak varázsolni egy megfelelő `Rational`-t, ha az általunk megadott `int`-tel meghívják a `Rational` konstruktorát, és ezt meg is teszik. Más szóval úgy kezelik a hívást, mintha az többé-kevésbé így lenne megírva:

```
const Rational temp(2);        // A 2-ből létrehozunk egy
                               // ideiglenes Rational objektumot.

result = oneHalf * temp;       // Ugyanaz, mint a
                               // oneHalf.operator*(temp);
```

Természetesen a fordítók csak akkor járnak el így, ha nem `explicit` konstruktorokról van szó, mert `explicit` konstruktorokkal nem lehet implicit konverziót végezni – ez az `explicit` jelentése. Ha a `Rational` definíciója így nézne ki:

```
class Rational {
public:
   explicit Rational(int numerator = 0,       // Ez a konstr
                     int denominator = 1);    // most explicit.
   ...
   const Rational operator*(const Rational& rhs) const;

   ...
};
```

egyik utasítás sem fordulna le az alábbi kettőből:

```
result = oneHalf * 2;           // Hiba!
result = 2 * oneHalf;           // Hiba!
```

Ez aligha minősülne a vegyes típusú aritmetikai műveletek támogatásának, de legalább a két utasítás viselkedése összhangban lenne.

Az általunk vizsgált `Rational` osztályt úgy tervezték, hogy megengedje a beépített típusok implicit konverzióját `Rational`-ekké – a `Rational` konstruktorát ezért nem deklaráljuk `explicit`-nek. Mivel ez a helyzet, a fordítópogramok végre fogják hajtani azt az implicit konverziót, amely a result első értékadásának lefordulásához kell. A fordítónk annyira király, hogy elvégzi ezt az implicit típuskonverziót, ha kell, *minden* függvényhívás *minden* paraméterére. Illetve csak a *paraméterlistában szereplő* paraméterekre, arra az objektumra viszont, amelyre a tagfüggvény meghívódik, azaz a tagfüggvényen belül a `*this`-nek megfelelő objektumra *soha*. Mindebből következően ez a hívás működik:

```
result = oneHalf.operator*(2);    // int -> Rational
                                  // konverziót hajt végre.
```

ez viszont nem:

```
result = 2.operator*(oneHalf);    // Nem hajt végre
                                  // int -> Rational konverziót.
```

Az első példa olyan paraméterrel dolgozik, amelyik benne van a függvénydeklarációban, a második viszont nem.

Mivel továbbra is a vegyes aritmetikai műveletek támogatása a célunk, mostanra talán már világos, mit kell tennünk: az `operator*`-ot nemtagfüggvényként kell megvalósítanunk, így a fordítók az *összes* paraméteren el tudják végezni az implicit típuskonverziót:

```
class Rational {
...                             // Nem tartalmaz operator*-ot.
};
// Ezt globálisan, vagy egy névtéren belül kell deklarálnunk
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
   return Rational(lhs.numerator() * rhs.numerator(),
                   lhs.denominator() * rhs.denominator());
}
Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2;         // Rendben.
result = 2 * oneFourth;         // Hurrá! Működik!
```

Minden jó, ha a vége jó, már csak egy nyugtalanító kérdés maradt: ne legyen-e az `operator*` a Rational osztály barátfüggvénye?

A válasz ebben az esetben nemleges, mert az `operator*`-ot implementálhatjuk kizárólag az osztály nyilvános felületére *(public interface)* támaszkodva. A fenti kód egyike a lehetséges megvalósításoknak. Kerüljük el a barátfüggvényeket, amikor csak tudjuk, mert – akárcsak a való életben – a barátok gyakran több bajt okoznak, mint amennyit érnek. Mindazonáltal nem szokatlan, hogy a nem tag, de koncepcionálisan az osztályfelülethez tartozó függvényeknek hozzá kell férniük az osztály nem nyilvános *(non-public)* tagjaihoz.

Térjünk vissza egy példa erejéig a `String` osztályhoz, amely ebben a könyvben már eddig is becsületesen húzta az igát. Ha `String` objektumok írásához és olvasásához megpróbáljuk túlterhelni *(overload)* az `operator>>`-t és az `operator<<`-t, hamar rájövünk, hogy jobb, ha ezek nem tagfüggvények. Ha azok lennének, akkor függvényhíváskor a `String` objektumot a bal oldalra kellene helyezni:

```
// Olyan osztály, amelyik az operator>>-t és az operator<<-t
// helytelenül tagfüggvénynek deklarálja.
class String {
public:
   String(const char *value);
   ...
   istream& operator>>(istream& input);
   ostream& operator<<(ostream& output);
private:
   char *data;
};
String s;
s >> cin;                  // Megengedett, de konvencióellenes.
s << cout;                 // Dettó.
```

Ez mindenkit összezavarna, ezért jobb, ha ezek a függvények nem lesznek tagfüggvények. Vegyük észre, hogy ez az eset különbözik a fent tárgyalttól. Itt az a cél, hogy minél magától értetődőbb legyen a függvényhívás szintakszisa, míg korábban az implicit típuskonverziókkal foglalkoztunk.

Ha mi terveznénk ezeket a függvényeket, valami ilyesmivel állhatnánk elő:

```
istream& operator>>(istream& input, String& string)
{
   delete [] string.data;
   olvassuk be inputból valahová a memóriába és a string.data
   mutasson rá
   return input;
}
ostream& operator<<(ostream& output, const String& string)
{
   return output << string.data;
}
```

Vegyük észre, hogy mindkét függvénynek hozzá kell férnie a `String` osztály `da-ta` mezőjéhez, amely privát mező. Az is egyértelmű viszont, hogy nemtagfüggvényként kell megvalósítanunk őket. Sarokba szorultunk és nem marad más választásunk, mint hogy azokat a nemtagfüggvényeket, amelyeknek hozzáférés kell egy osztály nem nyilvános tagjaihoz, az osztály barátfüggvényeivé tegyük. E fejezet tanulságait az alábbiakban foglaltam össze. Legyen `f` az a függvény, amelyet szabályosan akarunk deklarálni és `C` az az osztály, amelyhez `f` koncepcionálisan tartozik:

- **A virtuális függvényeknek tagoknak kell lenniük.** Ha `f`-nek virtuálisnak kell lennie, legyen a `C` tagfüggvénye.
- **Az `operator>>` és az `operator<<` sohasem tag.** Akár `operator<<`, akár `operator>>` az `f`, legyen nemtagfüggvény. Ha ezen kívül `f`-nek el kell érnie `C` nem nyilvános tagjait is, akkor `f` legyen a `C` barátfüggvénye.
- **Bal szélső paraméteren csak nemtagfüggvényeknél történik típuskonverzió.** Ha `f` bal szélső paraméterén típuskonverziókat kell végrehajtani, legyen `f` nemtagfüggvény. Ha ezen kívül `f`-nek `C` nem nyilvános tagjaihoz is kell hozzáférés, legyen `f` a `C` barátfüggvénye.
- **Minden más legyen tagfüggvény.** Ha a fentiek egyike sem vonatkozik esetünkre, legyen `f` a `C` tagfüggvénye.

20. jó tanács: Kerüljük az adattagokat nyilvános felületen

Először vessünk egy pillantást erre a témára a következetesség szempontjából. Ha a nyilvános felületben *(public interface)* csak függvények vannak, az osztályunk felhasználóinak nem kell azon törniük a fejüket, hogy tegyenek-e zárójelet, amikor el akarják érni az osztály egy tagját. Automatikusan kiteszik, mert csak függvények vannak. Sok fejtörést megspórolhatunk ezzel egy élet során.

Az olvasó nem vevő erre a magyarázatra? És ha azt mondom, hogy a függvények használata sokkal precízebb ellenőrzést tesz lehetővé az adattagok *(data members)* elérése felett? Ha egy adattagot nyilvánossá teszünk, mindenkinek lesz írási és olvasási jogosultsága felette. Ha viszont az adattag értékének lekéréséhez és beállításához függvényeket használunk, akkor kialakíthatunk csak olvasási jogosultságot, írási-olvasási jogosultságot vagy meg is tagadhatjuk a hozzáférést. Sőt, ha akarunk, akár csak írási jogosultságot is létrehozhatunk:

```
class AccessLevels {
public:
   int getReadOnly() const { return ReadOnly; }
   void setReadWrite(int value) { readWrite = value; }
   int getReadWrite() const { return readWrite; }

   void setWriteOnly(int value) { writeOnly = value; }
```

```
  private:
    int noAccess;                   // Ez az int nem hozzáférhető.
    int readOnly;                   // Ez az int csak olvasható.
    int readWrite;                  // Ez az int írható és olvasható.
    int writeOnly;                  // Ez az int csak írható.
  };
```

Még mindig nem elég meggyőző? Akkor elő a nagyágyúval: a függvényabsztrakcióval. Ha függvénnyel valósítjuk meg az adattaghoz való hozzáférést, akkor az adattagot később egy számítással is helyettesíthetjük, és ezt az osztály használói észre sem fogják venni.

Tegyük fel, hogy olyan alkalmazást írunk, amelyben egy automatikus berendezés az elhaladó autók sebességét figyeli. Minden egyes elhaladó autó sebességét kiszámolja, és ezt az értéket beleteszi az addig összegyűjtött összes sebességadat tárolójába:

```
  class SpeedDataCollection {
  public:
    void addValue(int speed);           // Új adatérték hozzáadása.
    double averageSoFar() const;        // A sebességátlag
                                        // visszaadása.
  };
```

Nézzük, hogyan valósítható meg az `averageSoFar` tagfüggvény *(member function)*. Egy lehetséges megoldás az, ha az osztályban olyan adattagot szerepeltetünk, amely az addig összegyűjtött sebességadatok átlagát folyamatosan követi. Híváskor az `average-SoFar` ennek az adattagnak az értékét adja vissza. Egy másik megoldás az lehet, ha az `averageSoFar` híváskor számolja ki az új értéket, amit mondjuk a tároló minden egyes adatértékének vizsgálatával ér el.

Az első megközelítés – a folyamatosan változó átlag követése – minden egyes `Speed-DataCollection` objektumot megnövel, mert területet kell foglalni a változó átlagot tartalmazó adattagnak. Az `averageSoFar`-t viszont nagyon hatékonyan meg lehet írni olyan inline függvényként (lásd a 33. jó tanácsot), amely az adattag értékét adja vissza. Az egyes hívások során kért átlag kiszámítása le fogja lassítani az `averageSoFar-t`, a `SpeedDataCollection` objektumok viszont kisebbek lesznek.

Ki tudná megmondani melyik a jobb? Egy olyan gépen, ahol kevés a memória, és egy olyan alkalmazásban, ahol az átlagok csak ritkán kellenek, az alkalmanként számolt átlag a jobb megoldás. Olyan alkalmazásban, ahol gyakran van szükség az átlagértékekre, lényeges a gyorsaság, a memória pedig nem gond, a folyamatosan változó átlag követése a kedvezőbb. Fontos tudnunk viszont, hogy tagfüggvénnyel *bármelyik* megoldást választhatjuk az átlag elérésére. A tagfüggvény használata be fog válni, mert olyan rugalmasságot biztosít, amely nem állna rendelkezésünkre, ha a folyamatosan számolt átlagot adattagként vennénk fel a nyilvános felületbe.

Az egészből annyi a tanulság, hogy csak bajt hozunk magunkra azzal, ha nyilvános felületen adattagokat szerepeltetünk. Jobb, ha biztonsági játékosok maradunk és elrejtjük az adattagokat a függvényabsztrakció védőbástyája mögé. Ha ezt már *most* megteszi a kedves olvasó, ráadásként kapja tőlünk a következetességet és a részletesen kidolgozott hozzáférési beállításokat, grátisz!

21. jó tanács: Amikor csak lehet, használjunk `const`-ot

A `const`-ban az a csodálatos, hogy szemantikai korlátozás előírását teszi lehetővé, mégpedig azt, hogy egy bizonyos objektumot *ne* lehessen módosítani. Ezt a korlátozást a fordítóprogramok érvényesítik. A `const` segítségével a fordítóprogramokkal és a programozókkal is közölni tudjuk, ha egy értéknek változatlannak kell maradnia. Fontos, hogy ezt minden alkalommal egyértelműen értésre adjuk, mert így a fordítóprogramot saját szolgálatunkba állítva biztosíthatjuk a korlátozás betartását.

A `const` kulcsszó figyelemre méltóan sokoldalú. Osztályon kívül használható globális vagy névtér konstansok *(global* vagy *namespace constants)* (lásd az 1. és a 47. jó tanácsot) és statikus – fájlra vagy blokkra nézve lokális – objektumok esetén, osztályon belül pedig statikus és nemstatikus adattagokra (lásd a 12. jó tanácsot).

Pointerek esetében meghatározhatjuk, hogy maga a pointer legyen-e `const`, vagy az adat, amelyre mutat, mindkettő, vagy egyik sem:

```
char *p                   = "Hello";     // Nem const pointer,
                                         // nem const adat.
const char *p             = "Hello";     // Nem const pointer,
                                         // const adat.
char * const p            = "Hello";     // const pointer,
                                         // nem const adat.
const char * const p = "Hello";          // const pointer,
                                         // const adat.
```

Ez a szintakszis nem annyira önkényes, mint amilyennek látszik. Húzzunk képzeletben egy függőleges vonalat a pointer deklarációjában szereplő csillagon keresztül. Ha a `const` szó a vonaltól balra esik, akkor az konstans, *amire mutatunk,* ha jobbra, akkor *maga a pointer* konstans, ha mindkét oldalon szerepel, akkor mindketten konstansok.

Ha az konstans, amire mutatunk, akkor vannak programozók, akik a típusnév elé teszik a `const` szócskát. Mások ilyenkor a típusnév után, de még a csillag elé írják. Ennek eredményeképpen a következő függvények ugyanolyan típusú paramétert vesznek fel:

```
class Widget { ... };
void f1(const Widget *pw);     // f1 egy konstans Widget objek-
                               // tumra mutató pointert vesz fel

void f2(Widget const *pw);     // f2 is
```

Mivel forráskódban mindkét verzió előfordulhat, érdemes megbarátkozni velük.

A `const` leghatékonyabb alkalmazásai közül jó néhány a függvénydeklarációkra épül.

[13] A C++ szabvány szerint a `"Hello"` típusa `const char[]`, amely típust majdnem mindig `const char*`-ként kezelnek. Ezért azt várnánk, hogy ha egy `char*` változót egy olyan karakterliterállal inicializálunk, mint a `"Hello"`, megsértjük a `const`-ra vonatkozó szabályokat. C-ben azonban annyira elterjedt ez a gyakolat, hogy a szabvány különleges felmentést ad az előírások alól az ehhez hasonló inicializációkra. Ennek ellenére kerüljük őket, mert sokan helytelenítik az effajta alkalmazásukat.

Függvénydeklaráción belül a `const` vonatkozhat a függvény visszatérési értékére, egyedi paraméterekre, tagfüggvény *(member function)* esetén pedig a függvény egészére.

Gyakran csökkenti a felhasználói hibák előfordulásának valószínűségét, ha a függvényünkkel konstans értéket adatunk vissza, anélkül, hogy ez akár a hatékonyság, akár a biztonság rovására menne. Sőt, ahogy azt a 29. jó tanács majd megmutatja, ha a `const`-ot visszatérési értékre vonatkoztatjuk, még *javíthatunk* is egy egyébként problémás függvény biztonsági és hatékonysági jellemzőin.

Vegyük például az `operator*` függvénynek azt a deklarációját valós számokra, amelyet a 19. jó tanácsban már bemutattunk:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Ez sok programozónak nem tetszik első ránézésre. Miért kellene az `operator*`-nak `const` objektumot adnia eredményül? Azért, mert ha nem azt adna, akkor a felhasználók ehhez hasonló rémtetteket követhetnének el:

```
Rational a, b, c;
...
(a * b) = c;                  // Értéket ad a*b eredményének!
```

Nem tudom, miért akarna egy programozó értéket adni két szám eredményének, de azt tudom, hogy ez simán szabályellenes lenne, ha `a`, `b` és `c` beépített típus lenne. A felhasználók által jól definiált típusok egyik ismérve az, hogy viselkedésük összhangban van a beépített típusokkal, indokolatlanul nem térnek el attól. Márpedig értékadást megengedni, amikor két szám eredményéről van szó, igencsak indokolatlannak tűnik számomra. Elkerülhetjük ezt a hibát, ha az `operator*` visszatérési értékét `const`-ként deklaráljuk, ezért Ez A Helyes Lépés.

A `const` paraméterekről nem lehet sok újat mondani – ugyanúgy viselkednek, mint a lokális `const` objektumok. A `const` típusú tagfüggvények működése viszont már egy teljesen más történet.

A `const` tagfüggvények célja természetesen az, hogy megszabják, mely tagfüggvények alkalmazhatók `const` objektumokra. Sokan átsiklanak afelett, hogy azok a tagfüggvények, amelyek *csak* `const` mivoltuk miatt különböznek a többitől, túlterhelhetők, noha a C++-nak ez fontos vonása. Nézzük meg a `String` osztályt még egyszer:

```
class String {
public:
   ...
   // operator[] nem const objektumoknak.
   char& operator[](int position)
   { return data[position]; }
   // operator[] const objektumoknak.
   const char& operator[](int position) const
   { return data[position]; }
```

```
   private:
      char *data;
   };
   String s1 = "Hello";
   cout << s1[0];              // Nem const String::operator[]
                               // hívása.
   const String s2 = "World";
   cout << s2[0];              // const String::operator[] hívása.
```

Ha az `operator[]`-t túlterheljük *(overload)*, és az egyes verzióknak más-más visszatérési értéket adunk, akkor a `const` és a nem `const` `String`-ek kezelését elkülöníthetjük egymástól:

```
   String s = "Hello";         // Nem const String objektum.
   cout << s[0];               // Helyes - nem const String
                               // olvasása.
   s[0] = 'x';                 // Helyes - nem const String írása
   const String cs = "World";  // const String objektum.
   cout << cs[0];              // Helyes - egy const String
                               // olvasása.
   cs[0] = 'x';                // Hiba! - egy const String írása.
```

Egyébként az itt felmerülő hibának csak a meghívott `operator[]` *visszatérési értékéhez* van köze, maguk az `operator[]` hívások rendben vannak. A hiba egy `const char&` számára történő értékadási kísérletből ered, mert ez a `const` verziójú `operator[]` visszatérési értéke.

Vegyük észre azt is, hogy a nem `const` `operator[]` visszatérési értékének egy `char`-ra való *referenciának* kell lennie, a `char` önmagában nem elég. Ha az `operator[]` egyszerűen csak egy `char`-t adna vissza, az ehhez hasonló kifejezések nem fordulnának le:

```
   s[0] = 'x';
```

Ez azért van így, mert a szabályok szerint nem szabad megváltoztatni egy olyan függvény visszatérési értékét, amely beépített típust ad vissza. Még ha ez megengedett is lenne, mivel a C++ érték szerint adja vissza az objektumokat (lásd a 22. jó tanácsot), az `s.data[0]` helyett annak csak egy *másolata* módosulna, és mi nem ezt akarjuk elérni.

Tegyünk egy kis filozófiai kitérőt. Mit is jelent pontosan egy tagfüggvény számára `const`-nak lenni? Két uralkodó elképzelés létezik: a bitenkénti konstansság *(bitwise constness)* és a fogalmi konstansság *(conceptual constness)*.

A bitenkénti `const` nézet tábora úgy hiszi, hogy egy tagfüggvény akkor és csak akkor `const`, ha nem változtatja meg az objektum egyetlen adattagját sem (kivéve azokat, amelyek statikusak), azaz, ha nem változtatja meg az objektum belsejében lévő bitek egyikét sem. A bitenkénti konstansság szépsége abban rejlik, hogy könnyű kinyomozni a szabálysértést: a fordítók csak megkeresik, az adattagoknál hol történik értékadás. A bitenkénti konstansság valójában a C++ konstansságra vonatkozó definíciója,

így egy `const` tagfüggvény nem változtathatja meg egy adattagját sem annak az objektumnak, amelyre meghívják.

Sajnos sok olyan tagfüggvény átmegy a bitenkénti vizsgán, amely nem igazán viselkedik úgy, mintha `const` lenne. Gyakran például azok a tagfüggvények, amelyek megváltoztatják azt, amire egy pointer *mutat*, nem viselkednek `const`-ként. De ha csak a *pointer* van az objektumban, a függvény bitenkénti `const` lesz és a fordítóprogramok nem fognak reklamálni. Ez lehet, hogy nem olyan viselkedést fog produkálni, amilyet ösztönösen várnánk:

```
class String {
public:
   // A konstruktor eléri, hogy a data a value által mutatott
   // terület másolatára mutasson.
   String(const char *value);
   ...
   operator char *() const { return data; }
private:
   char *data;
};
const String s = "Hello";          // A konstans objektum
                                   // deklarálása.

char *nasty = s;                   // Meghívja az
                                   // op char*() const-ot.

*nasty = 'M';                      // Módosítja az s.data[0]-t.

cout << s;                         // Kiírja, hogy „Hello”.
```

Valami biztos nem stimmel, mert miközben létrehozunk egy meghatározott értékkel rendelkező konstans objektumot és csak `const` tagfüggvénnyel hívjuk meg, meg tudjuk változtatni az értékét! (Ennek a példának a részletesebb tárgyalását lásd a 29. jó tanácsban.)

A fentiek a fogalmi konstansság gondolatához vezetnek. E filozófia követői azzal érvelnek, hogy egy `const` tagfüggvény változtathat ugyan néhány bitet abban az objektumban, amelyre meghívják, ezeket a változásokat azonban a felhasználók nem tudják felderíteni. Vegyük például azt az esetet, amikor a `String` osztályunk tárolni szeretné az objektum hosszát minden olyan alkalommal, amikor ezt a kérést kapja:

```
class String {
public:
   // A konstruktor eléri, hogy a data a value által mutatott
   // terület másolatára mutasson.
   String(const char *value): lenghtIsValid(false) { ... }
   ...
   size_t length() const;
```

```
private:
   char *data;
   size_t dataLength;            // A string legutoljára
                                 // számított hossza.

   bool lengthIsValid;           // A hossz érvényességének
                                 // ellenőrzése.
};
size_t String::length() const
{
   if (!lengthIsValid) {
      dataLength = strlen(data);     // Hiba!
      lengthIsValid = true           // Hiba!
   }
   return dataLength;
}
```

A `length` ilyenfajta alkalmazása egyáltalán nem a bitenkénti konstansságot jelzi – a `dataLength` és a `LengthIsValid` is módosítható –, ennek ellenére úgy tűnik, mintha a `const String` objektumokra mégis ez lenne érvényes. A fordítóprogramok, ahogyan azt látni fogjuk, udvariasan tiltakoznak és kitartanak a bitenkénti konstansság mellett. Mi a teendő?

A megoldás egyszerű: használjuk ki azt a `const` esetében létező szabad mozgásteret, amelyet a nagyon is előrelátó C++ szabványügyi bizottság pont ilyen esetekre talált ki. Ez a mozgástér a `mutable` kulcsszó formájában ölt testet. Ha a `mutable`-t nemstatikus adattagokra alkalmazzuk, akkor felmenti azokat a bitenkénti konstansság kényszere alól:

```
class String {
public:
   ...                           // Ua., mint fent.
private:
   char *data;
   mutable size_t dataLength;       // Ezek az adattagok most
                                    // már mutable-ek; bárhol.

   mutable bool lengthIsValid;      // Módosíthatók, még const
};                                  // függvények belsejében is.
size_t String::length() const
{
   if (!lengthIsValid) {
      dataLength = strlen(data);     // Most már ez is helyes
      lengthIsValid = true;          // és ez is helyes.
   }
   return dataLength;
}
```

A `mutable` csodálatos megoldás a „nem éppen a bitenkénti konstansságra gondoltam" problémára, de mivel a szabványosítási folyamat során viszonylag későn vették fel a C++-ba, előfordulhat, hogy a fordítónk még nem támogatja. Ha ez a helyzet, akkor alá kell szállnunk a C++ sötét bugyraiba, ahol az élet nem sokat ér, és a `const` használatával olykor fel kell hagynunk.

Egy C osztályba tartozó tagfüggvény belsejében a `this` pointer úgy viselkedik, mintha a következőképpen deklarálták volna:

```
C * const this;             // Nem const tagfüggvények számára.
const C * const this;       // const tagfüggvények számára.
```

Ebben a helyzetben ahhoz, hogy a problémás `String::length` (vagyis az, amit `mutable`-lel hozhatnánk rendbe, ha támogatná a fordító) érvényes legyen mind `const`, mind pedig nem `const` objektumokra, meg kell változtatnunk a `this` típusát `const C * const`-ról `C * const`-ra. Ezt közvetlenül nem lehet elvégezni, de meg lehet cinkelni az egészet úgy, hogy inicializálunk egy olyan lokális pointert, amely ugyanarra az objektumra mutat, mint a `this`. Ezután már el tudjuk érni a módosítani kívánt tagokat a lokális pointeren keresztül.

```
size_t String::length() const
{
   // Készítsük el a this egy olyan lokális változatát,
   // amely nem egy const-ra mutató pointer.
   String * const localThis =
      const_cast<String * const>(this);

   if (!lenghtIsValid) {
      localThis->dataLength = strlen(data);
      localThis->lengthIsValid = true;
   }
   return dataLength;
}
```

Nem valami szép, az igaz, de egy programozónak meg kell tennie, ha meg kell tennie. Persze csak akkor, ha garantáltan működik, már pedig a jó öreg „konstalanító" trükk működésére néha nincs garanacia. Főleg akkor jár definiálatlan következményekkel a „konstalanítás", ha az objektum, amelyre a `this` mutat eredendően `const`, azaz `const`-nak lett deklarálva a definiáláskor. Ha valamelyik tagfüggvényünkből ki akarjuk iktatni a konstansságot, jobban tesszük, ha előbb megnézzük, hogy az objektum, amelyen keresztül ezt megtesszük, eredetileg nem `const`-ként volt-e definiálva.

Van még egy eset, amikor hasznos és biztonságos lehet a konstansság kiiktatása. Nevezetesen, amikor egy `const` objektumot akarunk átadni egy olyan függvénynek, amely nem `const` paramétert vár és *tudjuk, hogy a paraméter a függvényen belül nem fog módosulni.* Ez a második feltétel fontos, mert mindig biztonsággal változtathatjuk meg egy olyan objektum konstansságát, amelyet csak olvasni fognak – írni nem, még akkor is, ha az objektumot eredetileg `const`-nak definiálták.

Néhány könyvtár például híres arról, hogy helytelenül deklarálja az `strlen` függvényt, méghozzá a következőképpen:

```
size_t strlen(char *s);
```

Természetesen az `strlen` nem fogja módosítani azt, amire az `s` mutat – legalábbis az az `strlen`, amelyiken én nőttem fel, biztosan nem. A fenti deklaráció miatt viszont lehetetlen lenne meghívni `const char *` típusú pointerekre. A probléma megoldásához nyugodtan iktassuk ki az ilyen pointerek konstansságát, amikor átadjuk őket az `strlen`-nek:

```
const char *klingonGreeting = "nuqneH";     // A „nuqneH”
                                            // „Hello”-t jelent
                                            // klingonul.
size_t length =
   strlen(const_cast<char*>(klingonGreeting));
```

Ne bízzuk azért el magunkat, mert ennek a megoldásnak a működésében csak akkor lehetünk biztosak, ha a hívott függvény, esetünkben az `strlen`, nem próbálja meg módosítani azt, amire a paramétere mutat.

22. JÓ TANÁCS: ÉRTÉK SZERINTI ÁTADÁS HELYETT HASZNÁLJUNK INKÁBB REFERENCIA SZERINTIT

A C-ben minden érték szerint adódik át. A C++ azzal adózik ennek az örökségnek, hogy az érték szerinti átadást *(pass-by-value)* alapértelmezettnek veszi. Hacsak nem rendelkezünk másként, a függvényparaméterek a tényleges paraméterek *másolataival* inicializálódnak, és a függvényhívók a függvény visszatérési értékének *másolatát* kapják vissza.

Ahogyan e könyv bevezetésében már kiemeltem, egy objektum érték szerinti átadásának jelentését az objektum osztályának másoló konstruktora *(copy constructor)* határozza meg. Emiatt az érték szerinti átadás igen drága műveletté válhat. Példának tekintsük az alábbi (elég mesterkélt) osztályhierarchiát:

```
class Person {
public:
   Person();                 // A paramétereket az egy-
                             // szerűség kedvéért elhagytam.
   ~Person();
   ...

private:
   string name, address;
};
```

```
class Student: public Person {
public:
   Student();                  // A paramétereket az egyszerűség
                               // kedvéért elhagytam.
   ~Student();
   ...
private:
   string schoolName, schoolAddress;
};
```

Most nézzünk meg egy olyan egyszerű függvényt, mint a `returnStudent`, amely egy `Student` paramétert vesz át (érték szerint) és azonnal visszaadja azt (szintén érték szerint), valamint ennek a függvénynek a hívását:

```
Student returnStudent(Student s) { return s; }
Student plato;                 // Platón Szókratész
                               // tanítványa volt.
returnStudent(plato);          // A returnStudent meghívása.
```

Mi történik ez alatt az ártalmatlannak tűnő függvényhívás alatt?

A magyarázat egyszerű. Meghívódik a `Student` másoló konstruktor, hogy inicializálja `s`-t `plato`-val. Ez után újra meghívódik a `Student` másoló konstruktor, hogy inicializálja a függvény által visszaadott objektumot `s`-sel. Ezután a destruktor hívódik meg `s`-re. Végül a destruktor a `returnStudent` által visszaadott objektumra hívódik meg. Így ennek a semmittevő függvénynek a költsége a `Student` másoló konstruktor és a `Student` destruktor kétszeri meghívása.

De várjunk csak! Van itt még valami. A `Student` objektumon belül van két `string` objektum, tehát a `Student` létrehozásakor mindig létrejön két `string` objektum is. A `Student` objektum örököl is egy `Person` objektumtól, így a `Student` minden egyes létrehozásakor létre kell hoznunk egy `Person` objektumot is. Egy `Person`-on belül két további `string` objektum van, így minden `Person` létrehozása maga után vonja két újabb `string` létrehozását. Végeredményben egy `Student` objektum érték szerinti átadása egy `Student` másoló konstruktor, egy `Person` másoló konstruktor és négy `string` másoló konstruktor meghívásához vezet. Amikor a `Student` objektum másolata megsemmisül, minden konstruktorhíváshoz párosul egy destruktorhívás, így egy `Student` objektum érték szerinti átadása összesen hat konstruktor- és hat destruktorhívásba kerül. Mivel a `returnStudent` függvény az érték szerinti átadást kétszer használja (egyszer a paraméterhez és egyszer a visszatérési értékhez), ennek a függvénynek a meghívása mindent egybevetve *tizenkét* konstruktor és *tizenkét* destruktor meghívásába kerül!

Hogy korrektek legyünk a C++ fordítóprogramok szerzőihez világszerte, tudnunk kell, hogy ez a legrosszabb eshetőség. A fordítóprogramok az ehhez hasonló másoló konstruktor hívások némelyikét elhagyhatják (a C++ szabvány – lásd az 50. jó tanácsot – pontosan leírja azokat a feltételeket, amelyek esetén lehetővé válik az efféle varázslat). Némelyik fordítóprogram ezt a jogosítványt optimalizálásra használja. Amíg azonban az ilyen optimalizálások nem válnak általánossá, oda kell figyelnünk arra, hogy az objektumok érték szerinti átadása mibe kerül.

Nagyon nagy árat is fizethetünk érte, amit jobb elkerülni. Ezért a dolgokat ne érték, hanem inkább referencia szerint adjuk át:

```
const Student& returnStudent(const Student& s)
{ return s; }
```

Ez sokkal hatékonyabb: sem konstruktor, sem destruktor nem hívódik meg, mert nem keletkezik új objektum.

A referencia szerinti paraméterátadásnak *(pass-by-reference)* más előnye is van. Fel sem merül a szeletelődés *(slicing)* problémája. Ha egy származtatott osztály objektumát bázisosztály objektumként adjuk át, minden olyan egyéni jellemzője, amely miatt származtatott objektumként működik, leszeletelődik róla, és ott állunk egy egyszerű bázisosztály objektummal. Szinte soha nem ezt akarjuk. Tegyük fel például, hogy egy grafikus ablakrendszert megvalósító osztálykészleten dolgozunk:

```
class Window {
public:
   string name() const;              // Adja vissza az ablak nevét.
   virtual void display() const;  // Rajzolja meg az ablakot
                                     // és annak tartalmát.
};
class WindowWithScrollBars: public Window {
public:
   virtual void display() const;
};
```

Minden `Window` objektumnak van neve, amelyet megkaphatunk a `name` függvénnyel, és mindegyik ablak megjeleníthető a `display` függvény segítségével. Mivel a `display` virtuális, ebből az következik, hogy a bázisosztály egyszerű `Window` objektumainak megjelenítése nagy valószínűséggel különbözni fog a `WindowWithScrollBars` objektumok költséges és különleges megjelenésétől (lásd a 36. és 37. jó tanácsot).

Tegyük fel, hogy szeretnénk írni egy függvényt, amely először kiírja az ablak nevét, aztán megjeleníti az ablakot. Nézzük, hogyan *ne* írjuk meg ezt a függvényt:

```
// Egy függvény, amely szeletelődési problémában szenved.
void printNameAndDisplay(Window w)
{
   cout << w.name();
   w.display();
}
```

Gondoljuk végig, mi történik, amikor meghívjuk egy `WindowWithScrollBars` objektummal:

```
WindowWithScrollBars wwsb;
printNameAndDisplay(wwsb);
```

Létrejön a w paraméter – érték szerinti átadással, ugye emlékszünk? – mint *Window* objektum, és az összes olyan egyénre szabott információ, amely a wwsb-t WindowWithScrollBars objektumként működtette, leszeletelődik. A printNameAndDisplay belsejében a w mindig a Window osztály objektumaként fog viselkedni – mert valóban a Window osztály objektuma –, tekintet nélkül annak az objektumnak a típusára, amely átadódik a függvénynek. A display hívása a printNameAndDisplay-en belül mindig egy Window::display hívás lesz, és sohasem egy WindowWithScrollBars::display.

A szeletelődési probléma megoldódik, ha a w-t referencia szerint adjuk át:

```
// Egy függvény, amely nem szenved szeletelődési problémában.
void printNameAndDisplay(const Window& w)
{
   cout << w.name();
   w.display();
}
```

A w most már annak megfelelően fog viselkedni, amilyen ablakot éppen kap. Arra fektettük a hangsúlyt, hogy a függvény w-t ne módosítsa, noha érték szerint adódik át, tehát megfogadtuk a 21. jó tanácsot és w-t gondosan const-nak deklaráltuk. Milyen jól tettük!

A referencia szerinti átadás csodálatos dolog, de sajátos bonyodalmakhoz vezethet, amelyek közül a leghírhedtebb az álnevesítés *(aliasing)* (a 17. jó tanácsban tárgyaltuk). Azt is fontos felismerni, hogy néha egyszerűen nem lehet a dolgokat referencia szerint átadni (lásd a 23. jó tanácsot). Végül pedig tagadhatatlan tény, hogy a referenciák majdnem mindig pointerként valósulnak meg, ezért a referencia szerinti átadás igazából pointer átadását jelenti. Épp ezért kis objektumok esetén, amilyen pl. egy int, tulajdonképpen hatékonyabb is az érték szerinti átadás.

23. jó tanács: Ne próbáljunk meg referenciát visszaadni akkor, amikor objektumot kell

Azt mondják, Albert Einstein egyszer a következő tanácsot adta: oldjunk meg mindent annyira egyszerűen, amennyire csak lehet, de ne egyszerűbben. A C++-analógia az lehetne, hogy oldjunk meg mindent annyira hatékonyan, amennyire csak lehet, de ne hatékonyabban.

Ha a programozók egyszer megértik, hogy objektumok esetében hogyan befolyásolja a hatékonyságot az érték szerinti átadás *(pass-by-value)* (lásd a 22. jó tanácsot), felcsapnak keresztes lovagnak, és felesküsznek arra, hogy az érték szerinti átadásban rejlő gonoszt gyökerestül kiirtják, bárhol is rejtőzzék. Elvakultan tűzik zászlajukra a referencia szerinti átadás *(pass-by-reference)* erkölcsi tisztaságát, a küzdelem során azonban egy végzetes hibát mindig elkövetnek. Átadnak nem létező objektumokra hivatkozó referenciát. Ez pedig nem jó dolog.

Vegyünk egy racionális számokat ábrázoló osztályt, benne egy barátfüggvénnyel *(friend function)* (lásd a 19. jó tanácsot) két racionális szám összeszorzására:

```
class Rational {
public:
   Rational(int numerator = 0, int denominator = 1);
   ...
private:
   int n, d;                              // Számláló és nevező
friend
   const Rational                         // Lásd a 21. jó tanácsot
      operator*(const Rational& lhs,      // hogy a visszatérési
                const Rational& rhs);     // érték miért const.
};
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
   return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Világos, hogy ez az `operator*` verzió a belőle származó objektumot érték szerint adja vissza. Elhanyagolnánk hivatásunkból adódó kötelességeinket, ha nem aggódnánk, mibe kerül nekünk ennek az objektumnak a létrehozása és megsemmisítése. Az is biztos, hogy sóherek vagyunk, és nem akarunk fizetni egy ilyen átmeneti objektumért, ha nem muszáj. A kérdés tehát az, hogy kell-e fizetnünk érte.

Nos, a válasz nem, ha egy referenciát tudunk visszaadni helyette. De ne felejtsük el, hogy a referencia csak egy *név*, valamilyen *létező* objektum neve. Valahányszor egy referencia deklarációjával találkozunk, azonnal kérdezzük meg magunktól, hogy minek lehet egy másik neve, mert biztosan más elnevezése *valaminek*. Ha a függvénynek referenciát kell visszaadnia az `operator*` esetében, akkor ennek egy olyan referenciának kell lennie, amely egy már létező, és a két összeszorzásra váró objektum eredményét is tartalmazó `Rational` objektumra vonatkozik.

Természetesen nem vehetjük biztosra, hogy az `operator*` meghívását megelőzően már létezik egy ilyen objektum. Azaz, ha

```
Rational a(1, 2);             // a = 1/2
Rational b(3, 5);             // b = 3/5
Rational c = a * b;           // c-nek 3/10-nek kellene lenni.
```

akkor indokolatlan azt gondolnunk, hogy már létezik egy racionális szám, amelynek az értéke három tized. Nem létezik, ezért ha az `operator*`-nak referenciát kell visszaadnia egy ilyen számra, akkor saját magának kell létrehoznia ezt a számobjektumot.

Egy függvény csak kétféleképpen hozhat létre új objektumot: veremben *(stack)* vagy halomban *(heap)*. Veremben egy lokális változó definiálásával történik a létrehozás. Ezt a stratégiát használva a következőképpen próbálkozhatunk `operator*`-unk megírásával:

```
// A függvény megírásának első rossz megközelítése
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
   Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
   return result;
}
```

Ezt kapásból ejthetjük is, mert éppen a konstruktorhívást akartuk elkerülni, a `result`-ot viszont ugyanúgy konstruktorral kell létrehozni, mint bármely más objektumot. Ráadásul van nagyobb baj is ezzel a függvénnyel, méghozzá az, hogy egy lokális objektumra vonatkozó referenciát ad vissza. Ez a hiba részletesebben a 31. jó tanács témája.

Nem marad más hátra, mint hogy halomban hozzunk létre egy objektumot és ezután egy rá irányuló referenciát adjunk vissza. Halom alapú objektumok a `new`-val jönnek létre. Ebben az esetben az `operator*` így nézhetne ki:

```
// A függvény megírásának második helytelen megközelítése.
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
   Rational *result =
      new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
   return *result;
}
```

A konstruktorhívásnak *még mindig* van költsége, mert a `new` által lefoglalt memória egy megfelelő konstruktor hívásával inicializálódik (lásd az 5. jó tanácsot), de felmerül itt egy másik probléma is: ki fogja végrehajtani a `delete`-et arra az objektumra, amelyet a `new`-val varázsoltunk elő?

Itt igazából garantált a memóriaszivárgás *(memory leak)*. Még ha az `operator*` hívóit rá is lehetne venni arra, hogy átvegyék a függvény eredményének címét és végrehajtsanak rajta egy `delete`-et (ez szinte lehetetlen – a 31. jó tanácsból kiderül, hogyan kellene ehhez a kódnak kinéznie), a bonyolult kifejezések név nélküli átmeneti egységeket eredményeznének, amelyekhez a programozók sohasem tudnának hozzáférni. Például, a

```
Rational w, x, y, z;
w = x * y * z;
```

kódrészletben mindkét `operator*` hívás átmeneti egységeket hoz létre, amelyeket a programozó sohasem fog látni és így törölni sem lesz képes. (Lásd a 31. jó tanácsot.)

De az is lehet, hogy az olvasó okosabbnak tartja magát az átlag medvénél – vagyis az átlag programozónál. Észreveszi, hogy akár a verem, akár a halom oldaláról közelítünk, sajnos minden egyes `operator*` által visszaadott eredményhez meg kell hívni egy konstruktort. Talán arra is emlékszik még, hogy a célunk eredetileg az ilyen konstruktorhívások elkerülése volt. Aztán arra gondol, hogy mégis tud egy módszert, amellyel, egy kivételével az összes konstruktorhívás elkerülhető. Talán pont a következő megol-

dás jut eszébe, amely azon alapul, hogy az `operator*` a függvény *belsejében* definiált *statikus* `Rational` objektumra történő referenciát ad vissza:

```
// A függvény megírásának harmadik helytelen megközelítése.
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
   static Rational result;     // Erre a statikus objektumra
                               // ad vissza referenciát.
   valahogy szorozzuk össze lhs-t és rhs-t, a keletkező értéket
   pedig tegyük result-ba;
   return result;
}
```

Ez ígéretesnek tűnik, bár ha megpróbálunk C++ forráskódot írni a fenti dőltbetűs pszeudokódra, kiderül, hogy `Rational` konstruktor hívása nélkül lényegében lehetetlen helyes értéket adni a `result`-nak, pedig pont az ilyen hívások kiiktatása lenne a játék lényege. Tegyük fel, hogy ez mégis sikerül valahogy, mert lehetünk bármilyen okosak, ennek az eleve rossz csillagok világán fakadott ötletnek a végiggondolását úgysem spórolhatjuk meg.

Hogy miért, arra nézzük a következő teljesen elfogadható felhasználói kódot:

```
bool operator==(const Rational& lhs,    // Egy operator==
                const Rational& rhs);   // a Rational-ekre.
Rational a, b, c, d;
...
if ((a * b) == (c * d)) {
   ha az eredmények megegyeznek, tegye azt, amit kell;
} else {
   ha nem egyeznek meg, akkor annak megfelelően cselekedjen;
}
```

Most jön a feladvány: az `((a*b) == (c*d))` kifejezés értéke mindig `true` lesz `a`, `b`, `c`, és `d` értékétől függetlenül!

Legkönnyebben úgy érthetjük meg ezt a nyugtalanító viselkedést, ha az egyenlőségre vonatkozó vizsgálatot átírjuk az ezzel ekvivalens függvényformára:

```
if (operator==(operator*(a, b), operator*(c, d)))
```

Vegyük észre, hogy amikor meghívjuk az `operator==`-ot, már *két* aktív `operator*` hívásunk lesz, amelyek mindegyike egy referenciát ad vissza az `operator*` belsejében lévő statikus `Rational` objektumra. Így azt kérjük az `operator==`-től, hogy hasonlítsa össze az `operator*` belsejében lévő statikus `Rational` objektumot az `operator*` belsejében lévő statikus `Rational` objektummal. Nagyon meglepő lenne, ha nem egyeznének. Akár egyszer is.

Ha szerencsém van, ennyivel meggyőztem már az olvasót arról, hogy olyan függvényből, mint az `operator*` időpocsékolás referenciát visszaadni, de nem vagyok

azért olyan naiv, hogy csak a szerencsében bízzam. Néhányan olvasóim közül – akik most biztosan magukra ismernek – ebben a pillanatban azt gondolják: „Hát, ha *egy* statikus elem nem elég, akkor lehet, hogy egy statikus *tömb* kell a mutatványhoz..."

Álljunk le! Könyörgöm! Hát nem szenvedtünk eddig eleget?

Nem bírom rászánni magam, hogy kódrészlettel tiszteljem meg ezt az elgondolást, azt viszont tudom vázolni, miért kellene már a miatt is mélyen *szégyenkeznünk*, ha egyáltalán foglalkozunk a gondolattal. Először is, meg kell választanunk *n*-t, a tömb méretét. Ha *n* túl kicsi, elfogyhat a függvények visszatérési értékeit tároló hely, és ebben az esetben semmivel sem jutunk előbbre, mint az épp előbb kegyvesztetté vált „egy `static` elem" taktikával. Ha viszont *n* túl nagy, akkor rontjuk a program teljesítményét, mert a tömb *összes* objektuma létrejön a függvény első felhívásakor. Ez nekünk *n* darab konstruktorba és *n* darab destruktorba kerül, még akkor is, ha csak egyszer hívjuk meg a kérdéses függvényt. Ha az optimalizálás olyan folyamat, amely egy szoftver teljesítményét javítja, akkor az ilyen ténykedést „pesszimalizálás"-nak kellene hívni. Végül gondoljuk végig, hogyan tennénk a tömb objektumaiba azokat az értékeket, amelyekre szükségünk van, és mibe kerülne mindez? Az értékek objektumok közötti átadásának legközvetlenebb módja az értékadás, de mibe kerül egy értékadás? Általában körülbelül ugyanannyiba, mint egy destruktorhívás (ami megsemmisíti a régi értéket) plusz egy konstruktorhívás (ami átmásolja az új értéket). De hát pont az a célunk, hogy megspóroljuk a konstruktor- és destruktorhívás költségét! Nézzünk szembe azzal, hogy ez a megközelítés nem működik.

Nem, mert ha olyan függvényt akarunk írni, amelynek egy új objektumot kell visszaadnia, akkor járunk a helyes úton, ha a függvénnyel egy új objektumot adatunk vissza. A `Rational operator*`-ára nézve ez vagy a következő kódot jelenti (amelyet először a 117. oldalon láttunk), vagy valami ezzel lényegében megegyezőt:

```
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Persze akár magunkra is vállalhatjuk az `operator*` visszatérési értéke létrehozásának és megsemmisítésének költségét, de ez csak hosszú távon kifizetődő a helyes működésért cserébe. Még az is előfordulhat, hogy a rettegett számla soha nem érkezik meg. Mint minden programozási nyelv, a C++ is megengedi a fordítóprogramok íróinak, hogy optimalizálással javítsák a generált kód teljesítményét, sőt, esetenként az `operator*` visszatérési értéke biztonsággal ki is küszöbölhető. Ha a fordítóprogramok élnek ezzel a lehetőséggel (és a mostaniak gyakran élnek), a program továbbra is úgy fog viselkedni, ahogy kell, csak gyorsabban, mint várnánk.

Összefoglalva: amikor döntenünk kell, hogy referenciát vagy objektumot adjunk e vissza, nekünk a helyesen működő megoldást kell választanunk. Számolgassák a fordítóprogramok készítői azt, hogy ez a választás mikor kerül a lehető legkevesebbe.

24. jó tanács: Válasszunk körültekintően a függvénytúlterhelés és az alapértelmezett paraméterezés között

A függvénytúlterhelést *(function overloading)* és az alapértelmezett paraméterezést *(parameter defaulting)* azért keverik össze olyan gyakran, mert mindkettő megengedi egyetlen függvénynév többféleképpen történő meghívását:

```
void f();                         // f túlterhelt.
void f(int x);

f();                              // Meghívja f()-t.
f(10);                            // Meghívja f(int)-et.

void g(int x = 0);                // g-nek van egy alapértelmezett
                                  // paraméter értéke.

g();                              // Meghívja g(0)-t.
g(10);                            // Meghívja g(10)-et.
```

Mikor melyiket használjuk?

A válasz két másik kérdéstől függ. Először is: van-e alapértelmezettként használható érték? Másodszor: hány algoritmust akarunk használni? Általában igaz az, hogy ha tudunk megfelelő alapértelmezett értéket választani és csak egy algoritmust akarunk alkalmazni, akkor használjuk az alapértelmezett paraméterezést (lásd a 38. jó tanácsot is). Minden más esetben a függvénytúlterhelést.

Az alábbi függvény kiszámítja legfeljebb öt `int` maximumát. Ez a függvény – vegyünk mély levegőt és legyünk erősek – az `std::numeric_limits<int>::min()`-t használja alapértelmezett paraméterként. Ehhez mindjárt lesz még hozzáfűznivalóm, de először álljon itt a kód:

```
int max(int a,
        int b = std::numeric_limits<int>::min(),
        int c = std::numeric_limits<int>::min(),
        int d = std::numeric_limits<int>::min(),
        int e = std::numeric_limits<int>::min())

{
  int temp = a > b ? a : b;
  temp = temp > c ? temp : c;
  temp = temp > d ? temp : d;
  return temp > e ? temp : e;
}
```

Nyugodjunk meg. Az `std::numeric_limits<int>::min()` a C++ szabvány könyvtárnak megfelelő új divatú leírása annak, amit a C az `INT_MIN` makrón keresztül fogalmaz meg a `<limits.h>`-ban, azaz ez egy `int` legkisebb lehetséges értéke, akármilyen fordítóval történjen is a C++ forráskódunk feldolgozása. Igaz, ez már kicsit messze van attól a tömörségtől, amiről a C híres, de a kettőspontokban és a többi szintaktikai förmedványben van rendszer.

Tegyük fel, hogy olyan függvénysablont szeretnénk írni, amely paraméterként bármilyen beépített numerikus típust *(numeric type)* felvehet, és szeretnénk a sablonból generált függvényekkel kiíratni azt a minimum értéket, amelyet a paraméterek példányosítási típusa *(instantiation type)* ábrázolhat. A sablon valahogy így nézne ki:

```
template<class T>
void printMinimumValue()
{
   cout << az a minimum érték, amelyet T ábrázolhat;
}
```

Ezt a függvényt nehéz megírni, ha csak a `<limits.h>`-val és a `<float.h>`-val dolgozhatunk. Nem tudjuk, hogy `T` pontosan mi, így azt sem tudhatjuk, hogy `INT_MIN`-t vagy `DBL_MIN`-t írassunk ki, vagy egyáltalán most mi legyen.

Ezeknek a nehézségeknek a kikerülésére a szabvány C++ könyvtár (lásd a 49. jó tanácsot) definiál egy osztálysablont a `<limits>` fejállományban `numeric_limits` néven, amely maga is definiál statikus tagfüggvényeket *(static member functions)*. Minden egyes függvény információt ad vissza arról a típusról, amely a sablont példányosítja. Ennek megfelelően a `numeric_limits<int>`-ben lévő függvények az `int` típusról adnak tájékoztatást, a `numeric_limits<double>`-beliek a `double` típusról, és így tovább. A `min` is a `numeric_limits` függvényei között szerepel. A példányosításra szolgáló típus legkisebb ábrázolható értékét adja vissza, tehát a `numeric_limits<int>::min()` a legkisebb ábrázolható egész értéket küldi vissza.

Mivel adott a `numeric_limits` (amely, mint majdnem minden a szabvány könyvtárban, az `std` névtérben van (lásd a 28. jó tanácsot) – maga a `numeric_limits` a `<limits>` fejállományban található), a `printMinimumValue` megírása ennél könnyebben már nem is mehetne:

```
template<class T>
void printMinimumValue()
{
   cout << std::numeric_limits<T>::min();
}
```

A típusfüggő konstansok meghatározásának ez a `numeric_limits`-re épülő megközelítése költségesnek tűnhet, pedig nem az. Azért nem, mert a forráskód tekervényessége nem tükröződik az eredményül kapott objektumkódban. Valójában a `numeric_limits`-ben lévő függvényekre irányuló hívások egyáltalán nem generálnak utasításokat. Hogy megértsük, ez miképp lehetséges, tekintsük a következő példát, amely a `numeric_limits<int>::min` egy magától értetődő megvalósítása:

```
#include <limits.h>
namespace std {
   inline int numeric_limits<int>::min() throw ()
   { return INT_MIN; }
}
```

Mivel ez a függvény deklarációja szerint inline, a rá irányuló hívásokat a törzsével kellene helyettesíteni (lásd a 33. jó tanácsot). Ez pont az `INT_MIN`, amely az implementáció során definiált valamelyik konstanshoz tartozó egyszerű `#define`. Habár a jó tanács elején szereplő `max` függvény első ránézésre minden egyes alapértelmezett paraméterértékre végrehajt egy függvényhívást, valójában csak okosan hivatkozik egy típusfüggő konstansra, jelen esetben az `INT_MIN` értékére. A C++ szabvány könyvtár bővelkedik az ehhez hasonlóan hatásos leleményekben. A 49. jó tanácsot tényleg érdemes elolvasni.

Visszatérve a `max` függvényre, döntő fontosságú azt észrevennünk, hogy a `max` mindig ugyanazzal a (nem igazán hatékony) algoritmussal számítja ki saját értékét – tekintet nélkül a hívó által átadott paraméterek számára. A függvényen belül sehol nem kíséreljük meg kideríteni, mely paraméterek „igaziak" és melyek alapértelmezettek. Ehelyett egy olyan alapértelmezett értéket választunk ki, amely valószínűleg nem befolyásolja az általunk használt algoritmus számításainak érvényességét. Ettől lesz az alapértelmezett paraméterértékek használata életképes megoldás.

Sok függvényhez nincs értelme alapértelmezett értéket rendelni. Tegyük fel, például, hogy írni akarunk egy függvényt, amely legfeljebb öt `int` elem átlagát számolja ki. Ebben az esetben nem használhatunk alapértelmezett értékeket, mert a függvény eredménye az átadott paraméterek számától függ: ha 3 paramétert adtunk át, az összeget 3-mal osztjuk, ha 5-öt, akkor 5-tel. Továbbá nincs olyan alapértelmezettként használható „mágikus szám", amellyel azt jelezhetnénk, hogy a paramétert nem a felhasználó adta meg, mert a szóba jöhető `int`-ek mind érvényes értékei a paramétereknek. Ebben az esetben nincs más hátra, túlterhelt függvényeket *kell* használni:

```
double avg(int a);
double avg(int a, int b);
double avg(int a, int b, int c);
double avg(int a, int b, int c, int d);
double avg(int a, int b, int c, int d, int e);
```

A másik olyan eset, amikor muszáj túlterhelt függvényeket használni, akkor áll elő, amikor egy meghatározott feladat végrehajtásánál a megadott inputoktól függ, hogy milyen algoritmust fogunk használni. Általában ez a helyzet a konstruktorokkal, hiszen egy alapértelmezett konstruktor a semmiből hoz létre egy objektumot, míg a másoló konstruktor egy létező objektumból:

```
// Természetes számokat ábrázoló osztály.
class Natural {
public:
   Natural(int initValue);
   Natural(const Natural& rhs);
```

```
private:
    unsigned int value;
    void init(int initValue);
    void error(const string& msg);
};
inline
void Natural::init(int initValue) { value = initValue; }
Natural::Natural(int initValue)
{
    if (initValue > 0) init(initValue);
    else error("Illegal initial value");
}
inline Natural::Natural(const Natural& x)
{ init(x.value); }
```

Az `int`-et átvevő konstruktornak hibaellenőrzést kell végeznie, a másoló konstruktornak viszont nem, így két különböző függvényre van szükség. Ez túlterhelést jelent. Ne feledjük azonban, hogy az új objektumnak mindkét függvénytől kell kezdeti értéket kapnia. Ez kódismétléshez vezethet a két konstruktorban, amire megoldás, ha egy olyan privát `init` tagfüggvényt írunk, amelyik a két konstruktor kódjának közös részét tartalmazza. Ez az eljárás – vagyis az olyan túlterhelt függvények használata, amelyek meghívnak egy közös függvényt munkájuk egy részének elvégzésére – méltó arra, hogy megjegyezzük, mert gyakran hasznunkra van (lásd, pl. a 12. jó tanácsot).

25. JÓ TANÁCS: KERÜLJÜK A TÚLTERHELÉST POINTEREKEN ÉS NUMERIKUS TÍPUSOKON

A nap triviális kérdése: mi a nulla?
Konkrétabban, mi fog itt történni?

```
void f(int x);
void f(string *ps);

f(0);                        // Az f(int)-et, vagy az
                             // f(string*)-ot hívja meg?
```

A válasz az, hogy a `0` egy `int` – egy egész literál konstans *(a literal integer constant)*, hogy precízek legyünk –, így mindig az `f(int)` kerül meghívásra. Pont ezzel van a baj, mert az ember nem mindig ezt akarja. A C++ világában kivételes esettel állunk szemben: van egy hívás, amelyről az emberek azt gondolják, hogy nem kellene egyértelműnek lennie, a fordítóprogramok viszont nem így gondolják.

Jó lenne egy szimbolikus név használatával – ami lehetne pl. `NULL` a nullpointerek esetében – lábujjhegyen elosonni a probléma mellett, de ez sokkal nehezebb, mint esetleg képzeljük.

Elsőre talán arra hajlanánk, hogy deklaráljunk egy `NULL` nevű konstanst, de a konstansoknak van típusa. A `NULL` milyen típusú lehetne? Kompatibilisnek kellene lennie a pointerek összes típusával, de ezt a feltételt csak a `void*` típus elégíti ki, `void*` pointereket viszont nem lehet átadni típusos pointereknek explicit konverzió *(explicit cast)* nélkül. Ez nem csak, hogy csúnya, de első ránézésre nem is igazán jobb, mint az eredeti felállás:

```
void * const NULL = 0;                 // A NULL egy lehetséges
                                       // definíciója.

f(0);                                  // Még az f(int)-et hívja meg.
f(static_cast<string*>(NULL));         // Az f(string*)-ot hívja meg.
f(static_cast<string*>(0));            // Az f(string*)-ot hívja meg.
```

Jobban meggondolva a dolgot, a `NULL`-nak `void*` konstansként való használata mégis jobbnak tűnik egy árnyalatnyival annál, mint amit az elején néztünk, mert ha a nullpointerek jelzésére csak a `NULL`-t használjuk, elkerülhetjük a félreértéseket:

```
f(0);                                  // Meghívja az f(int)-et
f(NULL);                               // hiba! - típus össze-
                                       // férhetetlenség.
f(static_cast<string*>(NULL));         // Rendben, az f(string*)-ot
                                       // hívja meg.
```

Most legalább lecseréltünk egy futási idejű hibát (a „rossz" `f` hívását 0-val) egy fordítási idejű hibára (annak megkísérlésére, hogy a `void*`-ot `string*` paraméterként adjuk át). Ez javít valamit a dolgokon (lásd a 46. jó tanácsot), de a konverzió *(cast)* még mindig nem kielégítő.

Ha szégyentől égő arccal visszasunnyogunk az előfordítóhoz, az sem mutatja meg a kiutat, mert nyilvánvaló alternatívának a

```
#define NULL 0
```

és a

```
#define NULL ((void*) 0)
```

tűnik. Az első megoldás éppen a `0` literál, amely alapjában véve egy egész konstans (ez volt a problémánk eredetileg, ha visszaemlékszünk), míg a második lehetőség ahhoz a problémához kanyarodik vissza, amikor típusos pointereknek `void*` pointereket akartunk átadni.

Ha bemagoltuk a típuskonverziókra vonatkozó szabályokat, esetleg tudjuk, hogy a C++-ban a `long int` konverzió `int`-re se nem jobb, se nem rosszabb, mint a `long int 0` nullpointerre konvertálása. Ezt kihasználva be is vezethetjük az `int`/pointer kérdéskörbe a többértelműséget, amelyről az olvasó valószínűleg amúgy is azt gondolta, hogy eleve benne van:

```
#define NULL 0L                    // A NULL most egy long int.
void f(int x);

void f(string *p);
f(NULL);                           // Hiba! - többértelmű.
```

Ez nem segít azonban, ha egy `long int`-en és egy pointeren hajtunk végre túlterhelést *(overloading)*:

```
#define NULL 0L
void f(long int x);          // Ez az f most egy long-ot vesz fel.
void f(string *p);
f(NULL);                     // Rendben, az f(long int)-et hívja meg.
```

A gyakorlatban ez valószínűleg veszélytelenebb, mint a `NULL`-t `int`-nek definiálni, de a problémát így inkább megkerültük, minthogy kiküszöböltük volna.

A probléma megszüntethető, de ehhez a nyelv egy később keletkezett kiegészítésére van szükségünk: a tagfüggvény sablonokra *(member function templates)* (gyakran egyszerűen csak tagsablonoknak *(member templates)* hívják őket). A tagfüggvény sablonok pont azok, aminek hallatszanak: osztályokon belüli olyan sablonok, amelyek tagfüggvényeket generálnak ezekhez az osztályokhoz. A `NULL` esetében olyan objektum kell nekünk, amelyik minden `T` típusra úgy viselkedik, mint a `static_cast<T*>(0)` kifejezés. Ebből következően a `NULL`-nak egy olyan osztály objektumának kellene lennie, amely minden lehetséges pointertípusra tartalmaz egy implicit konverziós operátort. Ez rengeteg konverziós operátort jelent, de egy tagsablonnal rákényszeríthetjük a C++-t arra, hogy ezeket létrehozza nekünk:

```
// Első próbálkozás egy NULL pointer objektumokat eredményező
// osztályra.
class NullClass {
public:
   template<class T>                         // operator T*-ot
      operator T*() const { return 0; }      // generál minden
};                                           // T típusra; minden
                                             // függvény a null-
                                             // pointert adja
                                             // vissza.

const NullClass NULL;          // A NULL egy NullClass
                               // típusú objektum.

void f(int x);                 // Ua., mint amink eredetileg volt.
void f(string *p);             // Dettó.
f(NULL);                       // Rendben, a NULL-t string*-ra.
                               // Konvertálja, aztán meghívja.
                               // Az f(string*)-ot.
```

Első vázlatnak ez jó, de még sokféleképpen finomítható. Először is, csak egy NullClass objektumra van igazán szükségünk, így az osztályt nem kell elneveznünk – egy név nélküli osztályt használunk és a NULL-t ilyen típusúvá tesszük. Másodszor, ha lehetővé tesszük, hogy a NULL bármilyen típusú pointerre konvertálható legyen, kezelnünk kell a tagokra irányuló pointereket is. Ehhez kell egy második tagsablon, egy olyan, amely a 0-t T C::*-ra („a C osztályban lévő T típusú tagra mutató pointer"-re) konvertálja minden C osztály és minden T típus esetében. (Ha ennek az olvasó semmi értelmét nem látja, vagy sosem hallott a – keveset használt – tagra mutató pointerekről, csak nyugalom. A tagra mutató pointer ritka állat, elvétve látni a vadonban és valószínűleg sosem lesz dolgunk vele. Az igazán kíváncsiak nézzék meg a 30. jó tanácsot, amely a tagra mutató pointereket egy kicsit részletesebben tárgyalja.) Végül, a felhasználókat meg kell óvnunk attól, hogy a NULL címét átvegyék, mert a NULL elvileg nem úgy viselkedik, mint egy *pointer*, hanem mint egy pointer *értéke*, egy pointer értékének (pl. 0x453AB002) pedig nincs címe.

A felturbózott NULL definíció már így néz ki:

```
const                              // Ez egy const objektum...
class {
public:
   template<class T>               // Amely bármilyen típusú
      operator T*() const          // nem tag nullpointerre
      { return 0; }                // konvertálható...
   template<class C, class T>      // vagy bármilyen típusú tag
      operator T C::*() const      // nullpointerre konvertálható...
      { return 0; }
private:
   void operator&() const;         // Amelynek a címét nem lehet át-
                                   // venni (ld. a 27. jó tanácsot)
} NULL;                            // és a neve NULL.
```

Ez már igazi látványosság, bár annyiban engedhetünk a gyakorlatiasság követelményeinek, hogy elnevezzük az osztályt. Ha ezt nem tesszük meg, a fordítónak a NULL típusára vonatkozó üzeneteit valószínűleg elég nehéz lesz megérteni.

Lényeges tudnunk ezekről a működőképes NULL létrehozását célzó törekvésekről, hogy csak akkor segítenek, ha mi vagyunk a *függvényhívók*. Ha mi vagyunk a szerzői a meghívott függvényeknek, egy üzembiztos NULL egyáltalán nem viszi előbbre a dolgokat, mert nem kényszeríthetjük a függvények meghívóit a használatára. Még ha fel is kínáljuk a felhasználóknak például az előbb kifejlesztett űrkorszakalkotó NULL-unkat, akkor sem akadályozhatjuk meg őket abban, hogy elkövessék a következőt:

```
f(0);                              // Még mindig f(int)-et hív,
                                   // mert a 0 még mindig egy int.
```

és ez pont ugyanolyan problémás most, mint a fejezet elején volt.

A túlterhelt függvények tervezésének szempontjából az a legfőbb tanulság tehát, hogy akkor járunk a legjobban, ha kerüljük a numerikus és a pointer típusok túlterhelését, persze a lehetőségek függvényében.

26. JÓ TANÁCS: ÓVAKODJUNK A POTENCIÁLIS TÖBBÉRTELMŰSÉGTŐL

Mindenkinek kell, hogy legyen valamilyen filozófiája. Néhányan a *laissez faire* közgazdaságtanban hisznek, mások a reinkarnációban. Néhányan még abban is képesek hinni, hogy a COBOL egy igazi programozási nyelv. A C++-nak is van filozófiája: hisz abban, hogy a potenciális többértelműség nem hiba.

Íme egy példa a potenciális többértelműségre:

```
class B;                    // a B osztály előzetes deklarációja

class A {
public:
   A(const B&);             // Egy A létrehozható egy B-ből.
};

class B {
public:
   operator A() const;      // Egy B átkonvertálható egy A-ra.
};
```

Ezekkel az osztálydeklarációkkal nincs semmi baj – a legkisebb gond nélkül élhetnek egymás mellett egy programban. De nézzük meg, mi történik, ha ezeket az osztályokat egy olyan függvénnyel kombináljuk, amely egy A objektumot vesz át, de valójában egy B objektumot kap:

```
void f(const A&);
B b;
f(b);                          // Hiba! - Többértelmű
```

Látva f hívását, a fordítók tudják, hogy valahogy egy A típusú objektummal kell előállniuk, még ha egy B típusú objektum van is a kezükben. Két egyformán helyes út kínálkozik erre. Egyrészt meg lehetne hívni az A osztály konstruktorát. Ez egy új A objektumot hozna létre, b-t használva paraméterként. Másrészt b-t át lehetne konvertálni egy A típusú objektummá a B osztályban lévő, felhasználó által definiált konverziós operátor *(conversion operator)* meghívásával. Mivel ez a két megoldás egyformán jó, a fordítók nem fognak választani közülük.

Egy ideig természetesen használhatnánk ezt a programot anélkül, hogy akár egyszer is belefutnánk a többértelműségbe. Pont ez a potenciális többértelműségben alattomosan rejlő veszély. Sokáig meghúzhatja magát tétlenül és észrevétlenül egy programban, míg egy napon egy mit sem sejtő programozó tényleg tesz valami többértelműt, és kitör a zűrzavar. Ez azt a nyugtalanító lehetőséget rejti magában, hogy kiadunk a kezünkből egy nem egyértelműen meghívható könyvtárat anélkül, hogy tudnánk róla.

A nyelv szabvány konverziói is vezethetnek ehhez hasonló többértelműséghez, még osztály sem kell hozzá:

```
void f(int);
void f(char);

double d = 6.02;
f(d);                          // Hiba! Többértelmű
```

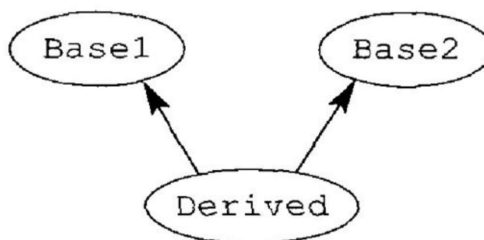
A d-t int-re vagy char-ra konvertáljuk? Mivel mindkét konverzió ugyanolyan jó, a fordítók nem fognak dönteni. Szerencsére ezt a problémát meg lehet oldani egy explicit típuskonverzióval *(explicit cast)*:

```
f(static_cast<int>(d));        // Jó, meghívja f(int)-et
f(static_cast<char>(d));       // Jó, meghívja f(char)-t
```

A többszörös öröklődés (lásd a 43. jó tanácsot) bővelkedik a potenciális többértelműséghez vezető lehetőségekben. A legkonkrétabb esete ennek az, amikor egy származtatott osztály több bázisosztálytól is ugyanazt a tagnevet örökli:

```
class Base1 {
public:
  int doIt();
};

class Base2 {
public:
  void doIt();
};
```




```
class Derived: public Base1,   // A Derived nem deklarál
               public Base2 {  // doIt nevű függvényt.
  ...
};
Derived d;
d.doIt();                      // Hiba! - Többértelmű.
```

Amikor a Derived osztály két ugyanolyan nevű függvényt örököl, a C++ egy kukkot sem szól: ezen a ponton a többértelműség csak lehetőségként áll fenn (potenciális). A doIt függvény hívásakor azonban a fordítóprogramoknak is szembe kell nézniük a problémával. A hívás csak akkor lesz rendben, ha kifejezetten egyértelművé tesszük, melyik bázisosztálybeli függvényt akarjuk:

```
d.Base1::doIt();               // Jó, meghívja Base1::doIt-et.
d.Base2::doIt();               // Jó, meghívja Base2::doIt-et.
```

Ezt kevesen találják idegesítőnek, az a tény viszont, hogy a hozzáférési korlátozások nem jönnek be a képbe, már sok, egyébként pacifista lelket késztetett távolról sem békés akciókra:

```
class Base1 { ... };              // Ua., mint fent.
class Base2 {
private:
   void doIt();                   // Ez a függvény most
};                                // privát.

class Derived: public Base1, public Base2
{ ... };                          // U.a., mint fent.
Derived d;
int i = d.doIt();                 // Hiba! - Még mindig többértelmű!
```

A `doIt` hívása még mindig többértelmű, annak ellenére, hogy csak a `Base1`-ben lévő függvény hozzáférhető! Annak sincs jelentősége, hogy csak a `Base1::doIt` ad vissza olyan értéket, amely egy `int` inicializálásához használható: a hívás ettől még többértelmű marad. Ha végre akarjuk hajtani ezt a hívást, egyszerűen meg *kell* adnunk, melyik osztály `doIt`-jére van szükségünk.

Mint a legtöbb, nem intuíció alapján kialakított C++ szabály esetében, itt is megvan az oka annak, hogy miért nem vesszük számításba a hozzáférési korlátozásokat akkor, amikor a származtatott tagok sokszorosítására szolgáló referenciákat kell egyértelművé tenni. A lényeg az, hogy egy program jelentése sohasem módosulhat azért, mert egy osztálytagjának a hozzáférhetőségét megváltoztatjuk.

Tegyük fel, például, hogy az előző megoldásnál figyelembe vettük a hozzáférési korlátozásokat. Ebben az esetben a `d.doIt()` kifejezés átalakulna egy `Base1::doIt` hívássá, mert a `Base2`-féle verziója nem volt elérhető. Most tegyük fel azt, hogy a `Base1` úgy változott meg, hogy az ő `doIt` verziója nyilvános *(public)* helyett védett *(protected)* lett, a `Base2` pedig úgy, hogy az övé privát helyett nyilvános.

Hirtelen ugyanaz a `d.doIt()` kifejezés egy *teljesen más függvényhívást* eredményezne, noha sem a hívás kódja, sem pedig a függvények nem módosultak! Ha valami, hát *ez* nem magától értetődő és a fordítók még csak figyelmeztetést sem tudnának küldeni. Figyelembe véve a választási lehetőségeket, arra a döntésre is juthatunk, hogy a származtatott tagok sokszorosítására szolgáló referenciák explicit egyértelműsítése egyáltalán nem annyira ésszerűtlen, mint azt eredetileg gondoltuk.

Lévén, hogy ennyiféleképpen lehet potenciális többértelműséget hordozó programokat és könyvtárakat írni, mit tehet a jó szoftverfejlesztő? Először is nem feledkezik meg róluk. A lehetetlennel határos a potenciális többértelműség minden gyökerének kiirtása, különösen, amikor programozók olyan könyvtárakat kombinálnak össze, amelyeket egymástól függetlenül fejlesztettek ki (lásd a 28. jó tanácsot), de ha megértjük, milyen helyzetek vezetnek potenciális többértelműséghez, akkor nagyobb eséllyel csökkenthetjük minimálisra a többértelműség előfordulását az általunk tervezett és fejlesztett szoftverekben.

27. jó tanács: Explicit tiltsuk meg azoknak az implicit függvényeknek a használatát, amelyekre nem tartunk igényt

Tegyük fel, hogy egy olyan `Array` nevű osztálysablont akarunk írni, amelynek generált osztályai azon kívül, hogy értékhatár-ellenőrzést végeznek, minden vonatkozásban úgy viselkednek, mint a beépített C++ tömbök. Tervezéskor szembekerülnénk többek között azzal a problémával, hogy hogyan tiltsuk meg az `Array` objektumok közötti értékadást, mert C++ tömbök számára szabályellenes az értékadás:

```
double values1[10];
double values2[10];

values1 = values2;                // Hiba!
```

A legtöbb függvény esetében ez nem lenne probléma. Ha nem akarnánk engedélyezni egy függvényt, akkor egyszerűen nem tennénk az osztályba. Az értékadó operátor azonban egyike azoknak az előkelő tagfüggvényeknek, amelyeket a C++, a mindig segítőkész szolga, megír helyettünk, ha olyan hanyagok voltunk, hogy nem írtuk meg magunk (lásd a 45. jó tanácsot). Mit tegyünk ebben a helyzetben?

A megoldás az, hogy a függvényt – jelen esetben az `operator=`-t – *privát*nak deklaráljuk. Ha egy tagfüggvényt explicit deklarálunk, akkor a fordítók nem fogják létrehozni a saját verziójukat. Ha a függvényt priváttá tesszük, megakadályozzuk, hogy azt mások meghívják.

A tervünk azonban nem üzembiztos, mert tag- és barátfüggvények *(friend functions)* még mindig meghívhatják a privát függvényünket. *Kivéve* akkor, ha annyira okosak nem vagyunk, hogy nem *definiáljuk* a függvényt. Így, ha véletlenül meg is hívjuk, hibajelzést kapunk a linkelés során (lásd a 46. jó tanácsot).

Az `Array` esetében a sablonunk definiálása valahogy így indulna:

```
template<class T>
class Array {
private:
   // Ne definiáljuk ezt a függvényt!
   Array& operator=(const Array& rhs);
   ...
};
```

Ha most egy felhasználó megpróbál értéket adni az `Array` objektumoknak, a fordítók meghiúsítják ezt a kísérletet. Ha véletlenül mi próbáljuk meg ugyanezt egy tag- vagy egy barátfüggvényben, a linker fog vonítani.

Ne gondoljuk ennek a példának az alapján, hogy ez a jó tanács csak az értékadó operátorokra vonatkozik, mert nem így van. Az összes olyan, fordító által generált függvény-

re érvényes, amely a 45. jó tanácsban szerepel. A gyakorlatban azt fogjuk tapasztalni, hogy az értékadó operátor és a másoló konstruktor (lásd a 11. és 16. jó tanácsot) viselkedése közti hasonlóság majdnem mindig azt jelenti, hogy amikor meg akarjuk tiltani az egyik használatát, akkor meg kell tiltanunk a másikét is.

28. JÓ TANÁCS: PARTICIONÁLJUK A GLOBÁLIS NÉVTERET

Az a legnagyobb probléma a globális hatókörrel *(global scope)*, hogy csak egy van belőle. Egy nagy szoftverprojektben általában egy sereg ember dob be neveket erre az egyetlen területre, ami elkerülhetetlenül névütközéshez *(name conflict)* vezet. Például a `library1.h` definiálhat egy sor konstanst, beleértve a következőt:

```
const double LIB_VERSION = 1.204;
```

Ugyanígy a `library2.h`:

```
const int LIB_VERSION = 3;
```

Nem kell nagy tehetség ahhoz, hogy belássuk, ez problémát fog okozni, ha egy program a `library1.h`-t és a `library2.h`-t is be akarja szerkeszteni. Sajnos, azon kívül, hogy magunkban káromkodunk, gyűlölködő leveleket küldünk a könyvtár szerzőinek, és addig szerkesztgetjük a fejállományokat *(header files)*, amíg a névütközés el nem tűnik, nem sok mindent tehetünk.

Azokon a szegény lelkeken viszont még megkönyörülhetünk, akiknek a nyakába a *mi* könyvtáraink zúdulnak majd. Az olvasó valószínűleg rendszeresen odabiggyeszt a globális szimbólumai elé egy reményei szerint egyedi előtagot, de bizonyára azt is kénytelen elismerni, hogy az eredményül kapott azonosítók nem éppen szemet gyönyörködtetőek. Ennél jobb megoldás egy C++ `namespace` használata. Lényegét tekintve a `namespace` lehetőséget ad arra, hogy az általunk kedvelt és ismert előtagokat szebb formába öntve használhassuk úgy, hogy másoknak nem is kell állandóan látni őket. Tehát e helyett:

```
const double sdmBOOK_VERSION = 2.0; // Ebben a könyvtárban
                                    // minden szimbólum
class sdmHandle { ... };            // „sdm"-mel kezdődik.
sdmHandle& sdmGetHandle(); // Nézzük meg a 47. jó tanácsot,
                           // hogy miért akarhatunk így
                           // deklarálni egy függvényt.
```

írjuk inkább ezt:

```
namespace sdm {
   const double BOOK_VERSION = 2.0;
   class Handle { ... };
   Handle& getHandle();
}
```

Ettől fogva a felhasználók a névterünkben *(namespace)* lévő szimbólumokat a három szokásos mód bármelyikével elérhetik: akár úgy, hogy az egy névtérhez tartozó összes szimbólumot egy hatókörbe importálják, akár úgy, hogy az egyes szimbólumokat importálják egy hatókörbe, vagy egy szimbólumot explicit módon egyszeri használatúvá minősítenek. Íme, néhány példa:

```
void f1()
{
   using namespace sdm;      // Minősítés nélkül tegyen
                             // elérhetővé ebben a hatókörben
                             // minden sdm-ben lévő szimbólumot.

   cout << BOOK_VERSION;     // Rendben, átalakul
                             // sdm::BOOK_VERSION-re.
   ...
   Handle h = getHandle();   // Rendben, a Handle átalakul
                             // sdm::Handle-re, a getHandle
   ...                       // pedig sdm::getHandle-re.
}
void f2()
{
   using sdm::BOOK_VERSION;     // Csak a BOOK_VERSION-t tegye
                                // elérhetővé ebben a hatókörben
                                // minősítés nélkül.
   cout << BOOK_VERSION         // Rendben, átalakul
                                // sdm::BOOK_VERSION-re.
   ...
   Handle h = getHandle();      // Hiba! Sem a Handle,
                                // sem pedig a getHandle nem lett
   ...                          // importálva ebbe a hatókörbe.
}

void f3()
{

   cout << sdm::BOOK_VERSION;   // Rendben, a BOOK_VERSION-t
                                // csak erre az egy esetre
   ...                          // teszi elérhetővé.

   double d = BOOK_VERSION;     // Hiba! A BOOK_VERSION nincs
                                // ebben a hatókörben.

   Handle h = getHandle();      // Hiba! Sem a Handle,
                                // sem pedig a getHandle nem lett
   ...                          // importálva erre a területre.
}
```

A névterekben az is szép, hogy itt a potenciális többértelműség nem hiba (lásd a 26. jó tanácsot). Épp ezért több névtérből is importálhatjuk ugyanazt a szimbólumot úgy, hogy közben gondtalanul éljük az életünket – persze, csak, ha ténylegesen sohasem használjuk ezt a szimbólumot. Ha például az `sdm` névtér mellett az alábbi névteret is használnunk kellene:

```
namespace AcmeWindowSystem {
   ...

   typedef int Handle;
   ...
}
```

akkor gond nélkül használhatnánk mind az `sdm`-et, mind az `AcmeWindowSystem`-et, feltéve, hogy soha nem hivatkozunk a `Handle` szimbólumra. Ha mégis hivatkoznánk, egyértelműen meg kellene adni, melyik névtér `Handle`-jét akarjuk:

```
void f()
{
   using namespace sdm;               // Importáljuk az
                                      // sdm szimbólumokat.

   using namespace AcmeWindowSystem;  // Importáljuk az
                                      // Acme szimbólumokat.

   ...                                // A Handle kivételével
                                      // bármely sdm és Acme
                                      // szimbólumra szabadon
                                      // hivatkozhatunk.

   Handle h;                          // Hiba! Melyik Handle?

   sdm::Handle h1;                    // Rendben, nincs
                                      // többértelműség.

   AcmeWindowSystem::Handle h2;       // Itt sincs
                                      // többértelműség.
   ...
}
```

Hasonlítsuk ezt össze a hagyományos fejállományokra épülő megközelítéssel: ott a fordítók már az `sdm.h` és az `acme.h` puszta beemelésekor panaszkodni kezdenének a `Handle` szimbólum többszörös definíciója miatt.

A szabványosítási játszma során viszonylag későn vették be a C++-ba a névtereket, emiatt azt is gondolhatnánk, hogy nem olyan fontosak és jól megvagyunk nélkülük. Nem egészen. Nem, mert a szabvány könyvtárban majdnem minden (lásd a 49. jó tanácsot) az

`std` névtéren belül van. Ez apró részletnek tűnhet, mégis elég közvetlenül érint minket: emiatt büszkélkedhet most a C++ olyan furcsa kinézetű, kiterjesztés nélküli fejállománynevekkel, mint az `<iostream>`, a `<string>` stb. A részletekért lapozzunk a 49. jó tanácshoz.

Mivel a névterek nem olyan régen kerültek bevezetésre, előfordulhat, hogy a fordítóink még nem támogatják őket. Még ha így is áll a helyzet, akkor sem indokolható a globális névtér összeszemetelése, mert a `namespace`-eket `struct`-okkal is közelíthetjük. Ehhez először létrehozunk egy globális nevek hordozására szolgáló struktúrát *(struct)*, majd ezeket a globális neveket statikus tagokként beletesszük a struktúrába:

```
// A névteret kiváltó struct definíciója.
struct sdm {
   static const double BOOK_VERSION;
   class Handle { ... };
   static Handle& getHandle();
};

const double sdm::BOOK_VERSION = 2.0;   // A statikus adattag
                                        // kötelező definíciója.
```

Ettől kezdve, ha valaki el akarja érni a globális neveinket, egyszerűen eléjük írja a struktúra nevét:

```
void f()
{
   cout << sdm::BOOK_VERSION;
   ...
   sdm::Handle h = sdm::getHandle();
   ...
}
```

Ha globális szinten nincsenek névütközések, a könyvtárunk használói körülményesnek találhatják a minősített nevek *(qualified names)* használatát. Szerencsére azt is tudjuk biztosítani számukra, hogy rendelkezzenek a hatókörök felett, és azt is, hogy figyelmen kívül hagyják őket.

A típusneveinkhez olyan típusdefiníciókat kell rendelkezésre bocsátanunk, amelyek szükségtelenné teszik a hatókör explicit meghatározását. Ez azt jelenti, hogy a névtérre hasonlító `S` struktúránkban lévő `T` típusnévre bocsássunk rendelkezésre egy (globális) típusdefiníciót úgy, hogy `T` az `S::T` szinonimája legyen:

```
typedef sdm::Handle Handle;
```

A struktúránk minden egyes (statikus) `X` objektumához bocsássunk rendelkezésre egy olyan (globális) `X` referenciát, amelyet `S::X` inicializál:

```
const double& BOOK_VERSION = sdm::BOOK_VERSION;
```

Könnyen lehet, hogy az olvasó – ha már megnézte a 47. jó tanácsot – egy BOOK_VERSION-höz hasonló, nemlokális statikus objektum definiálásának a gondolatától rosszul lesz. (Az ilyen objektumokat lecserélné a 47. jó tanácsban leírt függvényekre.)

A függvények kezelése nagyon hasonlít az objektumokéhoz, mégis, annak ellenére, hogy teljesen szabályos dolog függvényre hivatkozó referenciákat definiálni, kódunk jövőbeni karbantartói sokkal kevésbé fognak utálni minket, ha inkább függvényekre hivatkozó pointereket alkalmazunk:

```
sdm::Handle& (* const getHandle)() =
   sdm::getHandle;              // A getHandle egy
                                // sdm::getHandle-re
                                // hivatkozó const pointer
                                // (lásd a 21. jó tanácsot).
```

Vegyük észre, hogy a getHandle egy *konstans* pointer. Azt ugye nem akarjuk, hogy a felhasználók az sdm::getHandle-en kívül másra is mutathassanak ezzel a pointerrel? (Ha az olvasónak életbevágóan fontos tudni, miképpen lehet definiálni egy függvényre vonatkozó referenciát, itt az éltető megoldás:

```
sdm::Handle& (&getHandle)() = // A getHandle referencia
   sdm::getHandle;              // az sdm::getHandle-re.
```

Szerintem ez nagyon király, és annak is megvan az oka, ha az olvasó ezzel esetleg még sohasem találkozott. Az inicializálástól eltekintve a függvényekre irányuló referenciák és a konstans függvénypointerek ugyanúgy viselkednek, a függvénypointereket viszont sokkal könnyebb megérteni.)

Ilyen típusdefiníciók és referenciák mellett azok a felhasználók, akik a globális névütközések problémájával nem szembesülnek, minden további nélkül használhatják a minősítés nélküli *(unqualified)* típus- és objektumneveket, míg a névütközésekkel bajlódó felhasználók figyelmen kívül hagyhatják a típusdefiníciók *(typedef)* és referenciák definícióit, mindenütt a minősített neveket használva helyettük. Elég valószínűtlen, hogy minden felhasználó a rövidített nevet akarja majd használni, ezért a típusdefiníciókat és a referenciákat tegyük egy külön fejállományba, ne legyenek abban, amely a namespace-szel versengő struktúrát tartalmazza.

A struct-ok szép megközelítései a namespace-eknek, de mégsem az igaziak. Egy sor dologban elmaradnak tőlük, amelyek közül a legnyilvánvalóbb az operátorok kezelése. Egyszerűen arról van szó, hogy a struktúrák static tagfüggvényként definiált operátorait csak függvényhívással lehet elérni, a természetes infix jelölési szintakszison keresztül azonban, melyet az operátorok eleve támogatnak, soha:

```
// Definiáljunk egy névteret kiváltó struct-ot, amely típusokat
// és függvényeket tartalmaz a Widget-ekhez. A Widget objektumok
// az operator+-on keresztül támogatják az összeadást.
struct widgets {
   class Widget { ... };
```

```
    // Hogy miért const a visszatérési érték,
    // lásd a 21. jó tanácsot.
    static const Widget operator+(const Widget& lhs,
                                  const Widget& rhs);
    ...
};
// Kísérlet globális (minősítés nélküli) nevek létrehozására.
// A Widget és az operator+ számára a fent leírtak szerint.
typedef widgets::Widget Widget;
const Widget (* const operator+)       // Hiba!
      (const Widget&, const Widget&);  // Az operator+
                                       // nem lehet egy
                                       // pointer neve.
Widget w1, w2, sum;
sum = w1 + w2;                   // Hiba! Ebben a hatókörben nincs
                                 // deklarálva olyan operator+,
                                 // amely Widget-eket venne át.
sum = widgets::operator+(w1, w2);
                                 // Szabályos, de aligha
                                 // „természetes" szintaxis.
```

Az ilyen korlátok miatt arra kell törekednünk, hogy amilyen hamar csak a fordítónk lehetővé teszi, használjuk a valódi névtereket.

OSZTÁLYOK ÉS FÜGGVÉNYEK: IMPLEMENTÁCIÓ

Lévén a C++ erősen típusos, a munka oroszlánrészét általában az teszi ki, hogy megtaláljuk az osztályaink és sablonjaink *(templates)* legmegfelelőbb definícióját és hasonlóképp megfelelő deklarációt határozzunk meg a függvényeink számára. Ha ugyanis ezeket sikerül jól eltalálni, már igazán nehéz a sablonok, osztályok és függvények implementációját elrontani. Nehéz, valahogy mégiscsak sikerülni szokott.

Néha az absztrakció akaratlan megsértése okoz ilyen problémákat, amikor például véletlenül megengedjük, hogy az implementációs részletek kikandikáljanak az őket tartalmazó osztály vagy függvény határai alól. Mások az objektum élettartama körüli félreértésekből adódnak. Aztán vannak olyanok, amelyek az elhamarkodott optimalizálásból erednek, ezek általában visszavezethetők az `inline` kulcsszó csábító erejére. Végezetül, bizonyos, az adott szinten jónak tűnő implementációs stratégiák a forrásfájlok olyan erős összekapcsolódásához vezetnek, hogy ez elfogadhatatlanul drágává teszi nagy rendszerek újrafordítását *(rebuild)*.

Ezen problémák mindegyike – a hozzájuk hasonlókkal együtt – elkerülhető, ha tudjuk, hogy mire kell figyelnünk. Az itt következő jó tanácsok néhány olyan helyzetet mutatnak be, amelyekben különösen ébernek kell lennünk.

29. jó tanács: Kerüljük a belső adatokra hivatkozó „leírók" visszaadását

Jelenet egy objektumorientált szerelmi történetből:

A objektum: – Drágám, sose változz meg!

B objektum: – Ne aggódj édes, `const` vagyok.

S ahogy ez a való életben is megtörténik, A elgondolkozik: Meg lehet bízni B-ben? És mint a való életben, a válasz itt is B természetétől – tagfüggvényeinek összetételétől – függ.

Tegyük fel, hogy B egy konstans `String` objektum:

```
class String {
public:
   String(const char *value); // Lehetséges implementációkért
   ~String();                 // lásd a 11. jó tanácsot
   operator char *() const;   // String -> char* konverzió.
   ...

private:
   char *data;
};
const String B("Hello World");   // B egy const objektum.
```

Mivel a `B` `const`, jobb lenne, ha `B` értéke most és mindörökké a „Hello World" lenne. Természetesen azzal a feltétellel, hogy a `B`-vel dolgozó programozók civilizáltan játszanak. Különösen fontos az a feltétel, hogy senki ne kövessen el az alábbihoz hasonló – a `B` konstansságát eltüntető – aljas kis trükköket (lásd a 21. jó tanácsot):

```
String& alsoB =                 // Legyen alsoB B-nek egy másik
   const_cast<String&>(B);      // neve, de konstansság nélkül.
```

Ha tudjuk, hogy senki nem követ el ilyen gonoszságot, akkor nyugodtan fogadhatunk arra, hogy `B` sosem fog megváltozni. Vagy mégsem? Gondoljuk végig az alábbi utasítássorozatot:

```
char *str = B;                  // B.operator char*() hívása
strcpy(str, "Hi Mom");          // az str által mutatott
                                // tartalom módosítása.
```

Akkor most `B` értéke még mindig a „Hello World", vagy hirtelen valami olyanná változott, amivel édesanyánknak köszönnénk? A válasz csakis a `String::operator char*` implementációjától függ.

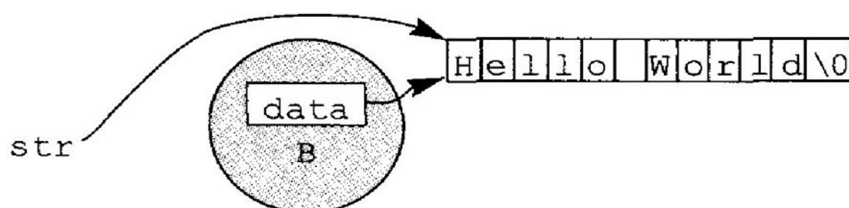
Az itt következő példa egy gondatlanul elkészített implementáció, amely nem azt teszi, amit kellene, de azt annál hatékonyabban, ezért esik sok programozó a csapdájába:

```
// Gyors, de hibás implementáció.
inline String::operator char*() const
{ return data; }
```

Ebben a függvényben az a hiba, hogy olyan információra hivatkozó „leírót" *(handle)* – jelen esetben egy pointert – ad vissza, amelynek abban a `String` objektumban, amelyre a függvény meghívódik, takarva kellene lennie. Ez a leíró a hívóknak korlátlan hozzáférést ad azokhoz az adatokhoz, amelyekre a privát `data` mező mutat. Más szóval a

```
char *str = B;
```

utasítás után a következő helyzet áll elő:



Jól látható, hogy az `str` által mutatott memóriaterület bármilyen megváltoztatása megváltoztatja `B` értékét is. Így aztán igaz, hogy `B`-t `const`-nak deklaráltuk, és csak `const` tagfüggvényeket hívtunk meg rá, `B` mégis különböző értékeket vehet fel a program futása során. Ha megváltozik például az `str` által mutatott terület, akkor megváltozik `B` is.

Alapvetően nincs azzal semmi baj, ha a `String::operator char*`-ot így írjuk meg. A baj az, hogy konstans objektumokra is alkalmazható. Ha a függvény nem `const`-nak lenne deklarálva, semmi gond nem lenne, mert nem lehetne meghívni a `B`-hez hasonló objektumokra.

Mégis teljesen ésszerűnek tűnik egy `String` objektumot – még egy konstans változatot is – a vele ekvivalens `char*`-gá alakítani, ezért szeretnénk ezt a függvényt `const`-nak hagyni. Ha ezt akarjuk, akkor úgy kell átírnunk az implementációt, hogy ne adjon vissza az objektum belső adataira *(internal data)* hivatkozó leírót:

```
// Lassúbb, de biztonságosabb megvalósítás.
inline String::operator char*() const
{
   char *copy = new char[strlen(data) + 1];
   strcpy(copy, data);
   return copy;
}
```

Ez a megvalósítás biztonságos, mert a `String` objektum által mutatott adat egy *másolatát* tartalmazó memóriára hivatkozó pointert ad vissza. Így a `String` objektum értéke már nem változtatható meg a függvény által visszaadott pointeren keresztül. Ahogy az lenni szokott, ennek a biztonságnak ára van: a `String::operator char*()` e változata lassúbb, mint a fenti egyszerű megoldás, és a hívóknak azt sem szabad elfelejteniük, hogy a visszaadott pointerre használniuk kell a `delete`-et.

Ha az `operator char*` e változatát túl lassúnak érezzük, vagy aggódunk a lehetséges memóriaszivárgás *(memory leak)* miatt – amit jól teszünk –, akkor létezik egy kicsit más megoldás is, nevezetesen a *konstans* `char`-okra mutató pointer visszaadása:

```
class String {
public:
   operator const char *() const;
   ...
};
inline String::operator const char*() const
{ return data; }
```

Ez a függvény gyors és biztonságos, és annak ellenére, hogy nem pont ugyanaz, mint amit eredetileg meghatároztunk, az alkalmazások nagy részében elegendő. Sőt, morálisan is egybevág a C++ szabványosítási bizottságnak a `string/char*` talányra adott válaszával: a szabvány `string` típusban van egy `c_str` nevű tagfüggvény, amely a kérdéses `string` objektum `const char*` változatát adja vissza. A 49. jó tanács további részletekkel szolgál a szabvány `string` típust illetően.

Belső adatra hivatkozó leírót nem csak pointereken keresztül lehet visszaadni. A referenciákat ugyanilyen könnyű rosszul használni. Íme egy gyakori megvalósítás, megint csak a `String` osztályt használva:

```
class String {
public:
   ...
   char& operator[](int index) const
   { return data[index]; }
private:
   char *data;
};
String s = "I'm not constant";
s[0] = 'x';                        // Rendben, s nem const.
const String cs = "I'm constant";
cs[0] = 'x';                       // Ez módosítja
                                   // a const string-et,
                                   // de a fordítók nem fogják
                                   // észrevenni.
```

Figyeljük meg, hogy a `String::operator[]` az eredményét referencia szerint adja vissza. Ez azt jelenti, hogy a függvény meghívója a belső `data[index]` elem *egy másik nevét* kapja vissza, és ez a másik név használható arra, hogy a konstansnak szánt objektum belső adatait megváltoztassuk. Ezzel a problémával már találkoztunk korábban, csak a pointer helyett most a referencia a bűnös visszatérési érték.

Egy ilyen probléma megoldási lehetőségei általában ugyanazok, mint pointerek esetében voltak: vagy nem `const`-tá tesszük a függvényt, vagy úgy írjuk meg, hogy ne adjon vissza leírót. Erre a *speciális* problémára – hogyan írjuk meg a `String::opera-`

tor[] függvényt úgy, hogy működjön const és nem const objektumok esetében is – a 21. jó tanács ad megoldást.

Nem csak a const tagfüggvényeknek kell aggódniuk a leírók visszaadása miatt. Még a nem const tagfüggvényeknek is bele kell nyugodniuk abba a ténybe, hogy egy leíró érvényessége a neki megfelelő objektummal együtt szűnik meg. Ez előbb is bekövetkezhet, mint azt a felhasználó várná, különösen akkor, ha a kérdéses objektum egy fordító által generált ideiglenes objektum *(temporary object)*.

Példaként vessünk egy pillantást erre a String objektumot visszaadó függvényre:

```
String someFamousAuthor()          // Véletlenszerűen választja
{                                  // ki és adja vissza
                                   // egy szerző nevét.
  switch (rand() % 3) {            // rand() az <stdlib.h> része
                                   // (és a <cstdlib>-é –
                                   // lásd a 49. jó tanácsot).
  case 0:
    return "Margaret Mitchell";
                                   // Az „Elfújta a szél"
                                   // írója, igazi klasszikus.
  case 1:
    return "Stephen King";         // Az ő történetei miatt
                                   // milliók nem tudnak
                                   // éjszakai aludni.
  case 2:
    return "Scott Meyers";         // Hmm, ez nem olyan,
  }                                // mint a többi...
  return "";                       // Idáig nem juthatunk el, de
                                   // egy értékkel visszatérő
                                   // függvényben minden
                                   // ágnak értéket kell
}                                  // visszaadnia, sajna.
```

Kérem az olvasót, tegye félre a rand visszatérési értékének véletlenszerűségével kapcsolatos kételyeit, és legyen megértő a nagyzási hóbortomat illetően, amellyel igazi írók közé soroltam magam. Vagy koncentráljon inkább arra, hogy a someFamousAuthor függvény visszatérési értéke egy String objektum, egy *ideiglenes* String objektum. Az ilyen objektumok csak átmenetiek – élettartamuk általában arra az időre korlátozódik, amire az őket létrehozó függvény hívását tartalmazó kifejezésé. A fenti konkrét esetben létezésük azzal a pillanattal ér véget, amikor a someFamousAuthor függvény hívását tartalmazó kifejezés megszűnik.

Vizsgáljuk most meg a someFamousAuthor itt következő használatát, ahol feltételezzük, hogy a String egy operator const char* tagfüggvényt deklarál, ahogy ezt korábban már láttuk:

```
const char *pc = someFamousAuthor();
cout << pc;                        // ajaj...
```

Akár hiszi az olvasó, akár nem, nem lehet előre megmondani, hogy ez a kód mit fog tenni, legalábbis semmi biztosat nem mondhatunk róla. Ennek az az oka, hogy mire a `pc` által mutatott karaktersorozatot megpróbáljuk kiírni, a sorozat már definiálatlan lesz. A bajt a `pc` inicializálása során lezajló események okozzák:

1. Létrejön egy ideiglenes `String` objektum a `someFamousAuthor` visszatérési értéke számára.
2. Ez a `String` ezután `const char*`-gá konvertálódik a `String` osztály `operator const char*` tagfüggvényén keresztül, a `pc` pedig az eredményül kapott pointer értékével inicializálódik.
3. Megszűnik az ideiglenes `String` objektum, ami a destruktor meghívásával egyenlő. A destruktorban megtörténik a `data` pointer felszabadítása (a kód a 11. jó tanácsban található). A `data` azonban ugyanarra a memóriaterületre mutat, mint a `pc`, így a `pc` most már egy felszabadított memóriaterületre mutat – egy definiálatlan tartalmú memóriaterületre.

Mivel a példában a `pc`-t egy ideiglenes objektum leírójával inicializáltuk, és az ideiglenes objektumok röviddel létrehozásuk után megszűnnek, a leíró már azelőtt érvénytelenné vált, mielőtt a `pc` bármit kezdhetett volna vele. Akárhonnan nézzük, a `pc` holtan született. Ebben áll az ideiglenes objektumok belsejére hivatkozó leírók veszélye.

Így aztán, az absztrakció megsértése miatt nem tanácsos leírókat visszaadni `const` tagfüggvények esetén. Még a nem `const` tagfüggvények esetén is bajt okozhat a leírók visszaadása, főleg, ha ideiglenes objektumok is odakeverednek. A leírók – pointerekhez hasonlóan – lebeghetnek, gyökértelenné válhatnak, és a gyökértelen *(dangling)* pointerekhez hasonlóan a lebegő, gyökértelen leírók elkerülésére is törekednünk kell.

Persze ez ügyben sem kell fanatikusnak lenni. A nem triviális programokból úgysem lehet kiirtani az összes lehetséges gyökértelen pointert, és persze ugyanez igaz a gyökértelen leírókra is. Kerüljük azonban a leírók visszaadását, hacsak nem kell valami ellenállhatatlan kényszernek engedelmeskednünk. Ez jót tesz a programunknak és a hírünknek is.

30. jó tanács: Kerüljük az olyan tagfüggvényeket, amelyek náluk szigorúbb láthatóságú tagokra hivatkozó nem `const` pointereket vagy referenciákat adnak vissza

Egy tagot azért teszünk privátttá vagy védetté *(protected)*, hogy korlátozzuk az elérhetőségét, ugye? A túlhajszolt és alulfizetett C++ fordítóink nem kevés fáradsággal próbálják biztosítani ezeknek a hozzáférési megszorításoknak *(access restrictions)* a sértetlenségét. Így aztán, ugye nem sok értelme van olyan függvényeket írnunk, amelyek tetszőleges felhasználó számára biztosítanak szabad hozzáférést a korlátozott láthatóságú tagokhoz? Ha valaki úgy gondolja, hogy ennek *van* értelme, az legyen szíves, olvassa el ezt a bekezdést újra és újra, egészen addig, amíg meg nem változik a véleménye.

Könnyű ezt az egyszerű szabályt megsérteni. Íme egy példa:

```
class Address { ... };          // Lakcím.
class Person {
public:
   Address& personAddress() { return address; }
   ...
private:
   Address address;
   ...
};
```

A `personAddress` tagfüggvény biztosítja a hívó számára a `Person` objektumban található `Address` objektumot, de – valószínűleg hatékonysági okokból – az eredmény visszaadása érték helyett referencia szerint történik (lásd a 22. jó tanácsot). Sajnálatos módon ennek a tagfüggvénynek a jelenléte meghiúsítja azt a célt, amelyért a `Person::address` privát láthatóságát bevezettük:

```
Person scott(...);              // Az egyszerűség kedvéért a
                                // paramétereket elhagytuk.
Address& addr =                 // Tegyük fel, hogy
   scott.personAddress();       // az addr globális.
```

Ezután az `addr` globális objektum már a `scott.address` *egy másik neve,* és így szabadon használható a `scott.address` olvasására és írására. Gyakorlati szempontból ez azt jelenti, hogy a `scott.address` már nem privát, hanem nyilvános *(public),* és ez az előbbre lépés a láthatóságban a `personAddress` tagfüggvénytől ered. Ebben a példában persze nem igazán használtuk ki a `private` láthatósági szintet, ha az `address` `protected` lenne, ugyanez a gondolatmenet vonatkozna rá.

Nem elég a referenciák miatt aggódnunk, a pointerek is eljátszhatják ugyanezt a játékot. Nézzük ugyanezt a példát, de most pointerekkel:

```
class Person {
public:
   Address * personAddress() { return &address; }
   ...

private:
   Address address;
   ...
};
Address *addrPtr =
   scott.personAddress();       // Az előzővel megegyező probléma.
```

Pointerek esetében azonban nemcsak az adattagok miatt van félnivalónk, hanem a *tagfüggvények* miatt is. Ugyanis lehet visszaadni tagfüggvényre mutató pointert:

```
class Person;                    // Előre deklarálás.
// PPMF = "pointer to Person member function"(= "Person
// tagfüggvényre mutató pointer")
typedef void (Person::*PPMF)();

class Person {
public:
   static PPMF verificationFunction()
   { return &Person::verifyAddress; }
   ...

private:
   Address address;
   void verifyAddress();
};
```

Ha az olvasó nincs hozzászokva ahhoz, hogy tagfüggvényre mutató pointerekkel és a belőlük képzett típusdefiníciókkal *(typedef)* keveredjen egy társaságba, akkor ijesztőnek tűnhet a `Person::verificationFunction` deklarációja. Nem kell megrémülni. Csak ennyit jelent:

- a `verificationFunction` egy paraméter nélküli tagfüggvény;
- visszatérési értéke a `Person` osztály egy tagfüggvényére mutató pointer;
- a mutatott függvénynek (azaz a `verificationFunction` visszatérési értékének) nincsenek paraméterei, és nem ad vissza semmit, azaz `void`.

Ami a `static` kulcsszót illeti, itt is ugyanazt jelenti, mint amit a tagdeklarációkban mindig is jelentett: az egész osztály számára csak egy példánya van a tagnak, és ez a tag objektum nélkül is elérhető. Ha az olvasó kíváncsi a teljes történetre, akkor lapozza fel kedvenc C++ tankönyvét. (Ha kedvenc C++ tankönyve nem foglalkozik a statikus tagokkal, akkor óvatosan tépje ki belőle a lapokat, és gondoskodjon újrahasznosításukról! A könyv borítójától környezetbarát módon szabaduljon meg, majd pedig kölcsönözzön vagy vásároljon egy jobb könyvet!)

Ebben az utolsó példában a `verifyAddress` egy privát tagfüggvény, ami azt jelzi, hogy valójában csak az osztály egy implementációs részlete, és csak az osztály tagjainak szabad tudni róla (meg persze a barátfüggvényeknek és osztályoknak). A publikus `verificationFunction` tagfüggvény azonban a `verifyAddress`-re mutató pointert ad vissza, ezért aztán a felhasználók megint előszedhetnek egy ilyet:

```
PPMF pmf = scott.verificationFunction();

(scott.*pmf)();                  // U.a., mint a
                                 // scott.verifyAddress hívása.
```

Itt a `PPMF` lett a `Person::verifyAddress` függvény szinonimája azzal a nagy különbséggel, hogy a felhasználása nem korlátozott.

Az iménti gondolatok ellenére eljöhet a nap, amikor olyan helyzettel találjuk szembe magunkat, amelyben – hatékonysági megszorítások miatt – muszáj olyan tagfüggvényt írnunk, amely egy nála szigorúbb láthatóságú tagra *(less-accessible member)* hivatkozó referenciát vagy pointert ad vissza. Ugyanakkor persze nem akarjuk feláldozni azokat a láthatósági megszorításokat sem, amelyeket a `private` és `protected` kulcsszavak tesznek lehetővé. Ezekben az esetekben majdnem mindig elérhetjük mindkét célt úgy, hogy egy `const` objektumra hivatkozó referenciát vagy pointert adunk vissza. A részleteket lásd a 21. jó tanácsban.

31. JÓ TANÁCS: SOHA NE TÉRJÜNK VISSZA LOKÁLIS OBJEKTUMRA, VAGY EGY DEREFERENCIÁLT POINTERRE HIVATKOZÓ REFERENCIÁVAL, AMELY POINTERT A FÜGGVÉNYEN BELÜL `new`-VAL INICIALIZÁLTUNK

Ez a jó tanács bonyolultnak hangozhat, mégsem az. Csak józan megfontolásokon alapul. Tényleg. Becsszóra. *Bízzanak bennem!*

Először vizsgáljuk meg a lokális objektumra hivatkozó referencia visszaadásának kérdését! Itt pont az a probléma, hogy a lokális objektumok *lokálisak*. Ez azt jelenti, hogy a definiálásukkor jönnek létre, és hatókörön *(scope)* kívülre kerülésükkor semmisülnek meg. A hatókörük azonban annak a függvénynek a törzse, amelyben találhatók. Amikor a függvény visszatér, a vezérlés elhagyja a hatókörét és a függvényre nézve lokális objektumok automatikusan megsemmisülnek. Ennek eredményeképp, ha egy lokális objektumra hivatkozó referenciát adunk vissza, akkor az a lokális objektum már megszűnik létezni, mire a függvény meghívója rátehetné számító kezét.

Ez a probléma rendszerint akkor üti fel csúf fejét, amikor egy függvény hatékonyságán úgy próbálunk javítani, hogy az eredményét érték helyett referencia szerint adjuk vissza. A következő példa megegyezik azzal, amely a 23. jó tanácsban szerepel (ez a jó tanács részletezi, hogy mikor adhatunk vissza referenciát és mikor nem):

```
class Rational {                    // Racionális számok osztálya.
public:
   Rational(int numerator = 0, int denominator = 1);
   ~Rational();
   ...

private:
   int n, d;                        // Számláló és nevező.
// Vegyük észre, hogy az operator* (hibásan)
// referenciát ad vissza.
friend const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs);
};
```

```
// Az operator* hibás implementációja.
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
   Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
   return result;
}
```

Itt a `result` lokális objektum az `operator*` törzsébe történő belépéskor jön létre. A lokális objektumok azonban automatikusan felszabadulnak, amikor hatókörön kívülre kerülnek. A `result` a `return` utasítás végrehajtása után kerül hatókörön kívülre, ezért amikor ezt írjuk:

```
Rational two = 2;
Rational four = two * two;      // Ugyanaz, mint az
                                // operator*(two, two).
```

akkor a függvényhívás során az alábbiak történnek:

1. Létrejön a `result` lokális objektum.
2. Egy referencia a `result` másik neveként inicializálódik és ez lesz az `operator*` visszatérési értéke.
3. A `result` lokális objektum megsemmisül, és az a memóriaterület, amelyet a veremben *(stack)* elfoglalt, újra használhatóvá válik a program más részei, vagy más programok számára.
4. A `four` objektum a 2. lépés referenciájának felhasználásával inicializálódik.

Minden rendben is van egészen a negyedik lépésig, ahol azonban nagyon súlyos hiba történik. A 2. lépésben inicializált referencia a 3. lépés végére már egy érvénytelen objektumra hivatkozik, így a `four` objektum inicializálásának a végkifejlete teljesen definiálatlan. A tanulság már könnyen levonható: ne adjunk vissza lokális objektumra hivatkozó referenciát.

– Rendben – mondhatja az olvasó –, az a probléma, hogy a szükséges objektum túl korán kerül hatókörön kívülre. Ezt ki tudom javítani. Lokális objektum helyett meghívom a `new`-t. – Így:

```
// Az operator* egy újabb hibás implementációja.
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
   // Halomban (heap) hoz létre egy új objektumot.
   Rational *result =
      new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
   // Visszaadja.
   return *result;
}
```

Ez a megközelítés tényleg kiküszöböli az előző példában látott hibát, de behoz helyette egy újat. Ahhoz, hogy a szoftverünkben elkerüljük a memóriaszivárgást *(memory leak)*, minden `new`-val létrehozott pointerre biztosítanunk kell a `delete`-et is. De pont ez a bökkenő: ki fogja meghívni a függvénybeli `new` `delete` párját?

Tisztán látszik, hogy az `operator*` *hívójának* kell gondoskodni a `delete` alkalmazásáról. Igen, tisztán látszik, sőt még könnyen dokumentálható is, ennek ellenére az ügy reménytelen. Ennek a pesszimista állításnak két oka is van.

Először is, közismert, hogy a programozó mint faj, felületes. Persze ez nem azt jelenti, hogy az olvasó felületes, vagy én felületes lennék, de ritka az olyan programozó, aki ne dolgozna együtt olyasvalakivel, aki kicsit – hogy is mondjam – szórakozott. Mi az esélye annak, hogy ezek a programozók – mert tudjuk, hogy vannak ilyenek – nem felejtik el, hogy az `operator*` hívása után *venniük kell az eredmény címét* és alkalmazni rá a `delete`-et? Vagyis az `operator*`-ot így kell használniuk:

```
const Rational& four = two * two;
                                // Fogjuk a dereferenciált
                                // (dereferenced)
                                // pointert; eltároljuk
                                // egy referenciában.
   ...
   delete &four;                // Visszakeressük a pointert
                                // és felszabadítjuk.
```

Az esély bizony elég kicsi. Ne felejtsük el, hogy ha az `operator*`-nak akár csak *egyetlen hívója* megfeledkezik erről a szabályról, máris szivárog a memória.

A dereferenciált pointerek visszaadásával van egy másik, ennél is komolyabb probléma. Komolyabb, mert még a leglelkiismeretesebb programozók jelenlétében is előfordul. Az `operator*` eredménye gyakran egy ideiglenes közbülső érték, egy olyan objektum, amely csak egy nagyobb kifejezés kiértékelése céljából jön létre. Pédául:

```
Rational one(1), two(2), three(3), four(4);
Rational product;
product = one * two * three * four;
```

A `product`-nak értékül adandó kifejezés kiértékeléséhez az `operator*` három különböző hívására van szükség. Ez a tény egyértelműbbé válik, ha a kifejezést a vele ekvivalens függvényformában írjuk fel:

```
product = operator*(operator*(operator*(one, two),
                                three), four);
```

Tudjuk, hogy az `operator*` minden egyes hívása olyan objektumot ad vissza, amelyet meg kell semmisíteni, de nincs lehetőségünk a `delete` alkalmazására, mert egyik visszaadott objektum sem lett elmentve.

Erre a nehézségre egyetlen megoldás létezik – a felhasználóktól azt kell kérnünk, hogy így kódoljanak:

```
const Rational& temp1 = one * two;
const Rational& temp2 = temp1 * three;
const Rational& temp3 = temp2 * four;
delete &temp1;

delete &temp2;
delete &temp3;
```

Kérhetünk ilyet, de a legjobb, amit remélhetünk, hogy levegőnek néznek. A realitások felől nézve a dolgot, inkább elevenen megnyúznak, vagy tíz évi, pirítós- és szendvicssütő-mikrokódolással töltendő kényszermunkára ítélnek.

Inkább tanuljuk meg most a leckét: a dereferenciált pointert visszaadó függvény nem más, mint egy lesben álló memóriaszivárgás.

Különben, ha az olvasó úgy gondolja, hogy talált egy megoldást, amellyel elkerülhető a lokális objektumra hivatkozó referencia visszaadásával eleve együtt járó definiálatlan viselkedés, vagy a halomban allokált objektumra történő referencia visszaadását kísértetként bejáró memóriaszivárgás, akkor lapozzon a 23. jó tanácshoz és olvassa el, miért nem működik jól a lokális `static` objektumra hivatkozó referencia visszaadása sem. Így orvoshoz sem kell mennie azért, mert saját magát dicséretből hátbaveregette, és kificamodott a karja.

32. jó tanács: Késleltessük a változók definiálását, amíg csak lehet

Szóval azonosulunk a C-nek azzal a filozófiájával, hogy a változókat a blokk elején illik definiálni. Felejtsük el! C++-ban ez felesleges, természetellenes és költséges.

Ne felejtsük el, hogy ha egy változót konstruktorral és destruktorral rendelkező típusúnak definiálunk, akkor viselnünk kell a létrehozás költségét, amikor a vezérlés a változó definíciójához ér, a megsemmisítés költségét pedig akkor, amikor a változó hatókörön *(scope)* kívülre kerül! Ez azt jelenti, hogy a használaton kívüli változóknak is van költségvonzatuk, ezért ahol csak lehet, kerülni kell őket.

Az olvasó, akinek a programozási módszereiről tudom, hogy kifinomultak és letisztultak, valószínűleg azt gondolja, hogy soha nem definiál használaton kívüli változókat, ezért az ő tömör és takarékos kódolási stílusára ez a jó tanács nem vonatkozik. Azért nem árt ezt újra végiggondolni. Vizsgáljuk meg a következő függvényt. Ez a függvény egy jelszó titkosított változatát adja vissza, feltéve, hogy a jelszó elég hosszú. Ha a jelszó túl rövid, akkor a függvény `logic_error` típusú kivételt dob, amely a szabvány C++ könyvtárban definiált (lásd a 49. jó tanácsot):

```
// Ez a függvény túl korán definiálja az "encrypted" változót
string encryptPassword(const string& password).
{
   string encrypted;
```

```
    if (password.length() < MINIMUM_PASSWORD_LENGTH) {
        throw logic_error("Password is too short");
    }
    a password titkosított változatának előállítása az encrypted
    objektumba;
    return encrypted;
}
```

Ebben a függvényben az `encrypted` objektum nincs *teljesen* használaton kívül, de kivétel dobása esetén nem használjuk. Azaz, akkor is kell fizetnünk az `encrypted` létrehozásáért és megsemmisítéséért, ha az `encryptPassword` függvény kivételt dob. Ezért aztán célszerű kitolni az `encrypted` definícióját addig, amíg *biztosan* nem tudjuk, hogy szükség lesz rá:

```
// Ez a függvény addig késlelteti az "encrypted" definícióját,
// amíg igazán szükség nincs rá.
string encryptPassword(const string& password)
{
    if (password.length() < MINIMUM_PASSWORD_LENGTH) {
        throw logic_error("Password is too short");
    }
    string encrypted;
    a password titkosított változatának előállítása az encrypted
    objektumba;
    return encrypted;
}
```

Ez a kód még mindig lehetne tömörebb, mert az `encrypted`-et inicializáló paraméterek *(initialization arguments)* nélkül definiáltuk. Ez azt jelenti, hogy az alapértelmezett *(default)* konstruktora hívódik meg. Az esetek nagy részében egy objektumnak legelőször valamilyen értéket adunk, leggyakrabban értékadáson *(assignment)* keresztül. A 12. jó tanács elmagyarázza, hogy az alapértelmezett konstruktorral történő létrehozás és az azt követő értékadás helyett miért jóval hatékonyabb az, ha azonnal a megfelelő értékkel történik az inicializálás. Az ott található elemzés erre az esetre is vonatkozik. Tegyük fel például, hogy az `encryptPassword` munkájának nehezét ez a függvény végzi el:

```
void encrypt(string& s);        // s-t helyben titkosítja.
```

Ekkor az `encryptPassword` így is implementálható, bár nem ez a legjobb megoldás:

```
// Ez a függvény addig késlelteti az "encrypted" definícióját,
// míg igazán szükség nincs rá, de még így sem kellően hatékony.
string encryptPassword(const string& password)
{
    ...                          // Hossz ellenőrzése, mint fent.
```

```
    string encrypted;          // Az encrypted létrehozása
                               // alapértelemezett konstruktorral.
    encrypted = password;      // Értékadás az encrypted-nek.
    encrypt(encrypted);
    return encrypted;
}
```

Ennél jobb megközelítés az `encrypted` inicializálása a `password`-del, így a (céltalan) alapértelmezett konstruktorral való létrehozás kihagyható:

```
// végül: az encrypted legjobb definíciója és inicializálása
string encryptPassword(const string& password)
{
    ...                            // A hossz ellenőrzése.
    string encrypted(password);    // Definíció és inicializálás
                                   // másoló konstruktorral.
    encrypt(encrypted);
    return encrypted;
}
```

Ez akar lenni a címben szereplő „ameddig csak lehet" igazi jelentése. Nemcsak az első felhasználásig célszerű egy változó definícióját késleltetni, hanem meg kell próbálni csak akkor definiálni, amikor már az inicializáló paraméterek is rendelkezésre állnak. Ha így teszünk, akkor nemcsak a szükségtelen objektumok létrehozását és felszabadítását takarítjuk meg, de elkerüljük az alapértelmezett konstruktorok felesleges használatát is. Az is igaz továbbá, hogy a változók céljának dokumentálást is jobban segíti az, ha olyan környezetben inicializáljuk őket, ahol a jelentésük már tisztán érthető. Ugye emlékszünk, hogy C-ben tanácsos minden változódefiníció mögé egy-egy rövid megjegyzést tenni, amely jelzi, mire fogjuk használni azt a változót? Nos, ha a megfelelő változónevekkel (lásd a 28. jó tanácsot is) együtt az adott környezetben beszédes inicializáló paramétereket használunk, akkor elérhetjük minden programozó álmát: alapos indokunk lesz bizonyos megjegyzések *elhagyására*.

A változók definiálásának késleltetésével növeljük a program hatékonyságát, átláthatóságát, és kevésbé van szükség a változók jelentésének dokumentálására. Úgy látszik, ideje búcsút intenünk a változók blokk elején megadott definíciójának.

33. JÓ TANÁCS: HASZNÁLJUK AZ `inline`-T MEGFONTOLTAN

Az inline függvények... milyen *csodálatos* ötlet! Úgy néznek ki, mint a függvények, úgy is viselkednek, mint a függvények, sokkal jobbak, mint a makrók (lásd az 1. jó tanácsot), ráadásul függvényhívás költsége nélkül hívhatjuk meg őket. Kérhetünk-e ennél többet?

Valójában többet kapunk, mint gondolnánk, mert a függvényhívás költségének megtakarítása csak a történet egyik fele. A fordítók optimalizáló eljárásait általában úgy ter-

vezik, hogy a függvényhívás nélküli kódrészletekre összpontosítsanak, ezért aztán az inline függvény használatával a fordító számára lehetővé tesszük a függvény törzsének környezetfüggő optimalizálását. „Rendes" függvényhívások esetén az ilyen optimalizálások nem végezhetők el.

Azért ne szaladjon el velünk a ló! A programozásban, akárcsak a való életben, semmit sem adnak ingyen – és ez alól az inline függvények sem kivételek. Az inline függvényt azért találták ki, hogy hívását a függvény törzsével lehessen helyettesíteni. Nem kell hozzá doktori fokozat statisztikából, hogy belássuk, ez valószínűleg megnöveli a tárgykód *(object code)* méretét. Korlátozott memóriával rendelkező gépeken készülhetnek olyan programok az inline függvények túlbuzgó használatával, amelyek túl nagyok a rendelkezésre álló memóriához. Még virtuális memóriakezelés esetén is előfordulhat, hogy az inline miatt felduzzadt kód kóros memórialapozási viselkedéshez vezet, amitől a program csak vánszorog. (A merevlemez-vezérlőnek – *disk controller* – viszont kellemes gyakorlatozási lehetőség.) A túl sok inline lecsökkentheti az utasítás cache *(gyorsítótár)* találati arányát is, ezáltal az utasítás betöltésének sebességét a cache memória sebességéről az elsődleges memória sebességére csökkentve.

Más oldalról nézve viszont, ha az inline függvény törzse *nagyon* rövid, akkor a függvénytörzsből generált kód lehet, hogy kisebb lesz, mint a függvényhívás kódja. Ebben az esetben az inline függvény használata *kisebb* tárgykódot és magasabb cache találati arányt eredményezhet!

Tartsuk szem előtt, hogy az `inline` direktíva – a `register`-hez hasonlóan – csak *útmutatás* a fordítónak, nem parancs. Ez azt jelenti, hogy a fordító akkor hagyja figyelmen kívül az inline direktívákat, amikor csak akarja. És nem is olyan nehéz rávenni, hogy ezt akarja. A legtöbb fordító a „bonyolult" – pl. ciklust tartalmazó vagy rekurzív – függvények esetében például elutasítja a függvény inline-ná tételét, és a legtriviálisabb virtuális függvényhívások kivételével a virtuális függvények sem lesznek inline-ok. (Ez persze nem túl meglepő. A `virtual` azt jelenti: „futási időben döntsük el, melyik függvényt kell meghívni", míg az `inline` jelentése: „fordítási időben cseréljük a hívás helyét a meghívott függvényre". Ha a fordító nem tudja, hogy melyik függvény fog meghívódni, akkor igazán nem hibáztathatjuk azért, hogy nem hajlandó inline meghívni.) Összegzésként megállapíthatjuk: az, hogy egy inline-nak megadott függvény ténylegesen inline-ként valósul-e meg, a fordítóprogramtól függ. Szerencsére a legtöbb fordító rendelkezik olyan felismerő funkcióval, amely figyelmeztetést küld, ha inline kérés ellenére sem sikerül egy függvényt inline-ná tennie (lásd a 48. jó tanácsot).

Tegyük fel, hogy írtunk egy `f` függvényt, és `inline`-nak deklaráltuk. Mi történik, ha valamilyen okból a fordító úgy dönt, hogy mégsem inline-ként valósítja meg ezt a függvényt? A magától értetődő válasz az, hogy `f`-et nem inline függvényként kezeli, azaz `f` kódja úgy áll elő, mintha normális függvény lenne, `f` hívásai pedig rendes függvényhívásként valósulnak meg.

Elméletileg pontosan ez fog történni, de ez egy olyan eset, ahol az elmélet és a gyakorlat útjai el is válhatnak. Ennek az az oka, hogy a „Mit tegyünk a nem inline-ként fordított inline függvényekkel?" kérdésre a C++ szabványosítási folyamatának csak viszonylag kései szakaszában adták meg a fenti tisztességes választ. A nyelv korábbi specifikációi (mint például az *ARM (Annotated Reference Manual)* – lásd az 50. jó tanácsot) a fordítók szállítóinak másfajta viselkedés megvalósítását írták elő. Mivel ez a régebbi viselkedés még mindig elég gyakori, érdemes megismerkednünk vele.

Pár perc gondolkodás után valószínűleg rájövünk, hogy az inline függvények definíciói szinte mindig fejállományba *(header file)* kerülnek. Így több fordítási egység (forrásfájl) is be tudja emelni ugyanazt a fejállományt, és ki tudja használni a benne definiált inline függvények előnyeit. Íme egy példa, amelyben azt a hagyományt követem, hogy a forrásfájlok nevei „.cpp"-re végződnek (a C++ világában a fájlok elnevezésére vonatkozó szabályok közül talán ez a legelterjedtebb):

```
// Ez az example.h fájl
inline void f() { ... }          // f definíciója.
...
// Ez a source1.cpp fájl
#include "example.h"             // Beemeli f definícióját
...                              // f hívásait tartalmazza.

// Ez a source2.cpp fájl
#include "example.h"             // Szintén beemeli
                                 // f definícióját.

...                              // Szintén f-et hívja meg.
```

A régi, nem inline-ként fordított inline függvényekre vonatkozó szabályokat, és azt a feltételezést véve alapul, hogy `f` nem inline-ként fordul le, a `source1.cpp` fordítása során eredményül kapott tárgykód fájl *(object file)* tartalmazni fog egy `f` nevű függvényt, mintha `f`-et sosem deklaráltuk volna `inline`-nak. Hasonlóan, a `source2.cpp` fordításakor a belőle készülő tárgykód fájl is tartalmazni fog egy `f` nevű függvényt. Amikor a két tárgykód fájlt megpróbáljuk összelinkelni, érthetően azt várjuk, hogy a linker hibát üzen, panaszkodik, hogy a programban `f`-nek két definíciója is van.

Ennek a problémának az elkerülésére a régi szabályok kimondták, hogy egy nem inline-ként fordított inline függvényt a fordítóknak úgy kell kezelni, mintha azt `static`-ként – azaz az éppen fordított fájlra nézve lokálisként – deklaráltuk volna. Az imént látott példában a régi szabályok szerint működő fordító `f`-et statikusnak tekintené a `source1.cpp` fájlra nézve, amikor azt fordítja, és hasonlóan statikusnak tekintené a `source2.cpp` fájlra nézve, amikor azt fordítja. Ez a stratégia megszünteti a linkelési problémát, de milyen áron! Minden olyan fordítási egység, amely beemeli `f` definícióját (és amely meghívja `f`-et), `f`-nek egy saját statikus másolatával fog rendelkezni. Ha `f` definiál lokális statikus változókat, akkor `f` minden egyes másolatának *saját példánya* lesz ezekből a változókból, ami biztosan megdöbbenti azokat a programozókat, akik azt hiszik, hogy a függvényen belüli „`static`" azt jelenti: „csak egy példány".

Mindez megdöbbentő következménnyel jár. Akár az új, akár a régi szabályok az érvényesek, ha egy inline függvény nem inline-ként fordul le, a függvényhívás árát akkor is meg kell fizetnünk minden hívás helyén, a régi szabályok fennállása esetén pedig még a *kódméret is* megnő, mert minden olyan fordítási egységnek, amely beemeli és meghívja `f`-et, saját másolata lesz `f` kódjából és `f` statikus változóiból. (Sőt, még ennél is rosszabb a helyzet, mert `f` minden egyes másolata és statikus változóinak minden példánya nagy valószínűséggel más-más virtuális memórialapra kerül, így aztán `f` két különböző másolatának meghívása egy vagy akár több memórialapozást is eredményezhet.)

És még nincs vége! Néha ezeknek a szegény, csatarendbe állított fordítóknak akkor is kell függvénytörzset generálniuk az inline függvényekből, amikor vonakodás nélkül hajlandók őket inline-ként fordítani. Konkrétan arra gondolok, ha a programunk egy inline függvény címét egyszer is használja, a fordítóknak muszáj egy függvénytörzset generálni hozzá. Honnét szedhetnek egy nem létező függvényre mutató pointert?

```
inline void f() {...}          // Mint fent.
void (*pf)() = f;              // pf f-re mutat.
int main()
{
   f();                        // f inline hívása.
   pf();                       // f nem inline hívása.
                               // pf-en keresztül.
   ...
}
```

Ebben az esetben látszólag ellentmondásos helyzetbe kerülünk, mert bár az `f` hívásai inline-ként fordulnak, a régi szabályok szerint minden olyan fordítási egység, amely `f` címét használja, a függvényből még egy statikus másolatot is előállít. (Az új szabályokkal `f`-nek már csak egyetlenegy nem inline változata jön létre, függetlenül attól, hogy hány fordítási egységben fordul elő.)

A nem inline-ként fordított inline függvények e vonatkozása akkor is hatással lehet ránk, ha soha nem használunk függvénypointereket, mert nem feltétlenül csak a programozóknak van szükségük függvényekre mutató pointerekre. Néha a fordítóknak is. Bizonyos esetekben a fordítók azért generálnak a konstruktorokból és a destruktorokból nem inline másolatokat, hogy használhassanak ezekre a függvényekre mutató pointereket egy osztály objektumtömbjeinek *(arrays of objects)* létrehozásakor és felszabadításakor.

A konstruktorok és a destruktorok általában rosszabb jelöltek inline kód generálására, mint azt egy felületes vizsgálódásból gondolnánk. Tekintsük az alábbi példában a `Derived` osztály konstruktorát:

```
class Base {
public:
   ...
private:
   string bm1, bm2;            // Bázis tagok: 1 és 2.
};

class Derived: public Base {
public:
   Derived() {}                // A Derived konstruktora
   ...                         // üres - vagy mégsem?
private:
   string dm1, dm2, dm3;       // Származtatott tagok: 1-3.
};
```

Ez a konstruktor kiváló inline jelöltnek tűnik, mivel nincs benne kód. De a külső gyakran megtévesztő. Az, hogy nincs benne kód, nem feltétlenül jelenti azt, hogy nincs benne kód. Valójában elég komoly kódmennyiséget is tartalmazhat.

A C++ garantálja, hogy az objektumok létrehozásakor és megszüntetésekor bizonyos dolgok megtörténnek. Az 5. jó tanácsból kiderül, hogy a `new` használatakor a dinamikusan létrehozott objektumok a konstruktoraikon keresztül automatikusan inicializálódnak, a `delete` használatakor pedig meghívódnak a megfelelő destruktorok. A 13. jó tanács elmondja, hogy egy objektum létrehozásakor annak minden bázisosztálya és adattagja automatikusan jön létre konstruktorral, míg a folyamat fordítottja zajlik le – természetesen destruktorral – az objektum megszüntetésekor. Ezek a jó tanácsok arról szólnak, hogy a C++ szerint minek kell történni. A C++ csak azt nem mondja el, hogy ezeknek *hogyan* kell megtörténniük. Ez a fordítóprogram készítőin múlik, de gondolom az mindenkinek világos, hogy ezek a dolgok nem csak úgy maguktól történnek. Kell, hogy legyen olyan kód a programunkban, ami miatt ezek az események bekövetkeznek, és ennek a kódnak – amelyet a fordítóprogram készítői írnak, a mi programunkba pedig a fordítás során kerül – megvan a maga helye. Néha a konstruktorokba és a destruktorokba kerül, így aztán bizonyos implementációk az alábbival ekvivalens kódot készítenek a fenti, állítólag üres `Derived` konstruktorból:

```
// A Derived konstruktor egy lehetséges implementációja.
Derived::Derived()
{
   // Halomban foglaljon memóriát (heap memory) az objektum.
   // Számára, ha az objektumot halomban kell létrehozni; az
   // operator new-val kapcsolatos info
   // a 8. jó tanácsban található.
   if (az objektum halomban jön létre)
      this = ::operator new(sizeof(Derived));

   Base::Base();                  // A Base rész inicializálása.

   dm1.string();                  // dm1 létrehozása.
   dm2.string();                  // dm2 létrehozása.
   dm3.string();                  // dm3 létrehozása.
}
```

Azt persze ne reméljük, hogy ez a kód valaha is lefordul, mert az ilyen C++ forrás nem megengedett – legalábbis nem nekünk! Egyrészt egy objektum konstruktorának a belsejéből lehetetlen kideríteni, hogy az objektum halomban van-e. Másrészt nekünk tilos a `this`-nek értéket adni. És persze a konstruktorokat sem hívhatjuk meg egyszerű függvényként. A fordítóprogramok munkájára azonban nem vonatkoznak ilyen megszorítások, ők azt tehetnek, amit csak akarnak. Nem is a kód szabályos volta érdekes most. A lényeg az, hogy a kód, amelynek az `operator new`-t kell meghívnia (ha erre szükség van), vagy a bázisosztály részeit és adattagokat létrehoznia, szép csendben bekerülhet a konstruktorainkba, és ha ez megtörténik, akkor a konstruktoroknak megnő a mérete, így már nem olyan vonzó jelöltek, ha inline megvalósításról van szó. Természetesen

ugyanez a gondolatmenet érvényes a `Base` konstruktorára is, azaz ha a `Base` inline, akkor minden hozzáadott kód belekerül a `Derived` konstruktorába is (a `Derived` konstruktornak a `Base` konstruktorra vonatkozó hívásán keresztül). És ha véletlenül a `string` konstruktor is inline, akkor a `Derived` konstruktora a függvény kódjának *öt másolatával* gyarapszik, a `Derived` objektum mind az öt (a két öröklött plusz az általa deklarált három) sztringje számára eggyel. Ugye most már látszik, miért nem lehet gondolkodás nélkül eldönteni, hogy a `Derived` osztály konstruktora inline legyen-e vagy sem? Természetesen hasonló megfontolások alkalmazhatók a `Derived` destruktorára is, amelynek így vagy úgy, de gondoskodnia kell arról, hogy a `Derived` konstruktora által inicializált objektumok maradéktalanul megsemmisüljenek. És persze arra is szükség lehet, hogy az épp megszüntetett `Derived` objektum által korábban dinamikusan elfoglalt memóriát felszabadítsa.

Könyvtárak tervezésekor végig kell gondolnunk, milyen következményekkel jár az, ha egyes függvényeket `inline`-nak deklarálunk, mert az inline függvények lehetetlenné teszik a könyvtár inline függvényeinek bináris frissítését. Más szóval, ha `f` inline függvény egy könyvtárban, akkor a könyvtár felhasználói `f` törzsét belefordítják az alkalmazásaikba. Ha a könyvtár implementálója később megváltoztatja `f`-et, akkor minden `f`-et használó felhasználót újra kell fordítani. Ez gyakran nagyon kényelmetlen megoldás (lásd még a 34. jó tanácsot). Ha viszont `f` nem inline függvény, akkor `f` módosítása csak azt kívánja a felhasználóktól, hogy újralinkeljenek. Ezt már jóval könnyebb elvégezni, mint az újrafordítást, sőt, ha a függvényt tartalmazó könyvtár dinamikusan linkelt, akkor a változatatás a felhasználók elől teljesen elrejtve is kivitelezhető.

Fontos, hogy ezeket a megfontolásokat a programfejlesztés során mindig szem előtt tartsuk, de tisztán gyakorlati szempontból nézve a kódolás folyamatát, a következő szempont fontosabb az összes többinél: a legtöbb debugger-nek meggyűlik a baja az inline függvényekkel.

Ez persze nem nagy felfedezés. Hogyan tegyünk töréspontot *(breakpoint)* egy olyan függvénybe, ami nincs? Hogyan lépkedjünk végig egy ilyen függvényen? Hogy kapjuk el a hívásait? Hacsak nem vagyunk hihetetlenül okosak – vagy nagyon ravaszak –, akkor sehogy. Szerencsére mindezek ismeretében kidolgozhatunk egy logikus stratégiát arra nézve, hogy mely függvényeket deklaráljuk `inline`-nak, és melyeket ne.

Kezdetben semmi se legyen inline, vagy legalábbis korlátozzuk az inline deklarációkat az igazán triviális függvényekre, mint amilyen például az `age` az alábbi kódrészletben:

```
class Person {
public:
   int age() const { return personAge; }
   ...

private:
   int personAge;
   ...
};
```

Ha óvatosan alkalmazzuk az inline deklarációkat, akkor egyfelől megkönnyítjük a debugger-ünk munkáját, másfelől pedig az inline is az lesz, amire való: kézzel bevihető optimalizálás. Sose feledjük a tapasztalaton alapuló 80-20-as szabályt, amely kimondja: egy átlagos program összes idejének 80 százalékában kódjának mindössze 20%-át hajtja végre. Ez fontos szabály, mert felhívja a figyelmet arra, hogy szoftverfejlesztőként az a célunk, hogy beazonosítsuk a kódnak azt a 20 százalékát, amely képes megnövelni az egész program teljesítményét. Ítéletnapig próbálkozhatunk: inline-ná tehetjük, vagy csiszolgathatjuk bárhogy a függvényeinket, mindez elvesztegett erőfeszítés, ha nem a *megfelelő* függvényekkel tesszük.

Ha sikerül beazonosítani az alkalmazás igazán fontos függvényeit, vagyis azokat, amelyeknek az inline megvalósítása tényleg különbséget jelent (ez a függvényhalmaz függ a futtató architektúrától is), ne habozzunk `inline`-nak deklarálni őket! Ilyenkor se feledkezzünk meg azonban a kódduzzadás miatt kialakuló problémákról és figyeljünk a fordítónak azokra a figyelmeztetéseire (lásd a 48. jó tanácsot), amelyekből az derül ki, hogy az inline függvényeink mégsem inline-ok.

Megfontoltan használva az inline függvények felbecsülhetetlen értékű összetevői lehetnek bármely C++ programozó eszköztárának, de ahogyan az a fenti okfejtésből is kiderül, nem annyira egyszerűek és egyértelműek, mint azt elsőre gondolhattuk volna.

34. JÓ TANÁCS: MINIMALIZÁLJUK A FÁJLOK KÖZÖTTI FORDÍTÁSI FÜGGŐSÉGET

Tegyük fel, hogy vesszük a C++ programunkat és egy kicsit megváltoztatjuk az egyik osztály implementációját. Figyelem, nem a felületét *(interface)*, hanem az implementációját, csak a privát részt! Aztán felkészülünk a program újraépítésére *(rebuild)*, gondolván, hogy a fordítás és a linkelés csak néhány másodpercig fog tartani. Végülis, csak egy osztályt módosítottunk. Ráklikkelünk a „Rebuild"-re, vagy begépeljük a `make` parancsot (vagy az odaillő megfelelőjét), aztán elcsodálkozunk, majd halálra válunk, amikor rájövünk arra, hogy Tolnától Baranyáig minden újrafordul és újralinkelődik!

Ennél *utálatosabb* dolgot...!

Az a baj, hogy a C++ nem igazán ért a felületek és az implementációk szétválasztásához. Ha az osztálydefiníciókat nézzük, azok nemcsak a felületre vonatkozó specifikációt tartalmazzák, hanem elég sok implementációs részletet is. Például:

```
class Person {
public:
   Person(const string& name, const Date& birthday,
          const Address& addr, const Country& country);
   virtual ~Person();

   ...                         // A másoló konstruktort és az
                               // értékadó operátort az
                               // egyszerűség kedvéért elhagytuk.
```

```
    string name() const;
    string birthDate() const;
    string address() const;
    string nationality() const;
private:
    string name_;              // Implementációs részlet.
    Date birthDate_;           // Implementációs részlet.
    Address address_;          // Implementációs részlet.
    Country citizenship_;      // Implementációs részlet.
};
```

Ez az osztályterv aligha kapna Nobel-díjat, de nagyon jól bemutat egy érdekes névadási szokást, amely a privát adatok és a publikus függvények megkülönböztetésére szolgál, ha ugyanannak a névnek mindkét esetben van értelme: a privát adat nevét a végén megtoldjuk egy aláhúzás karakterrel. Fontos azt észrevennünk, hogy a `Person` osztályt csak úgy lehet lefordítani, ha a fordító hozzáfér azokhoz az osztálydefiníciókhoz is, amelyeken keresztül a `Person` implementációja történik. Ezek név szerint a `string`, a `Date`, az `Address`, és a `Country`. Az ilyen definíciókat rendszerint az `#include` direktívával tesszük elérhetővé, így aztán a `Person` osztályt definiáló fájl elején nagy valószínűséggel valami ilyet találhatunk:

```
#include <string>     // A string típusra (l. a 49. jó tanácsot).
#include "date.h"
#include "address.h"
#include "country.h"
```

Balszerencsénkre ez fordítási függőséget hoz létre a `Person`-t definiáló fájl és ezek között az include fájlok között. Ennek eredményeképp, ha valamelyik ilyen segédosztály megváltoztatja az implementációját, vagy valamelyik olyan osztály változtatja meg az implementációját, amelytől egy ilyen segédosztály függ, akkor a `Person` osztályt újra kell fordítani. Sőt, azokat a fájlokat is, amelyek ezt a `Person` osztályt használják. A `Person` felhasználói számára ez a legjobb esetben is idegesítő. De inkább egyszerűen elfogadhatatlan.

Elképzelhető, hogy az olvasó elgondolkozik azon, miért is ragaszkodik a C++ ahhoz, hogy az osztály implementációs részletei is bekerüljenek az osztály definíciójába. Miért ne definiálhatnánk a `Person` osztályt például így,

```
class string;         // Elvileg a string típus előzetes dek-
                      // larációja. Részletek a 49. jó tanácsban.
class Date;           // Előzetes deklaráció.
class Address;        // Előzetes deklaráció.
class Country;        // Előzetes deklaráció.
class Person {
public:
    Person(const string& name, const Date& birthday,
           const Address& addr, const Country& country);
```

```
    virtual ~Person();
    ...                               // Másoló konstruktor, operator=.
    string name() const;
    string birthDate() const;
    string address() const;
    string nationality() const;
};
```

az osztály implementációs részleteit valahol máshol megadva? Ha ez lehetséges volna, akkor a `Person` felhasználóit csak akkor kellene újrafordítani, ha az osztályfelület is megváltozna. Mivel a felületek hajlamosak előbb stabilizálódni, mint az implementációk, a felület és az implementáció ilyen szétválasztása tömérdek újrafordítással és linkeléssel töltött időt spórolhatna meg nekünk egy nagyobb szoftverfejlesztés során.

Sajna a való világ befurakodik ebbe az idilli képbe, ami teljesen érthető, ha az alábbi példát végiggondoljuk:

```
int main()
{
    int x;                            // Egy int definíciója.
    Person p(...);                    // Egy Person definíciója
                                      // (a paramétereket elhagytuk
    ...                               // az egyszerűség kedvéért).
}
```

Amikor a fordítóprogram meglátja az `x` definícióját, tudja, hogy egy `int` tárolásához elegendő memóriát kell lefoglalnia. Semmi gond. Minden fordító tudja, mekkora egy `int`. Amikor azonban a `p` definíciójával találkozik, tudja, hogy egy `Person` objektum számára elegendő helyet kell lefoglalnia, de honnan tudhatná, hogy mekkora egy `Person` objektum? Az egyetlen lehetőség ennek az információnak a megszerzésére az osztálydefiníció vizsgálata. Ha viszont megengednénk az implementációs részletek elhagyását az osztálydefinícióból, akkor a fordítók honnan tudnák, hogy mennyi memóriát kell lefoglalni?

Elvben ez a probléma nem legyőzhetetlen. Az olyan nyelvek, mint a Smalltalk, az Eiffel és a Java mindig találnak rá megoldást. Méghozzá úgy, hogy egy objektum definiálásakor csak egy objektumra mutató *pointer*nek elegendő memóriaterületet foglalnak le. Azaz a fenti kódot úgy kezelik, mintha ezt írtuk volna:

```
int main()
{
    int x;          // Egy int definíciója.
    Person *p;      // Egy Person-ra mutató pointer definíciója.
    ...
}
```

Bizonyára az olvasónak is feltűnt, hogy ez valójában szabályos C++ kódrészlet. Mint az ki fog derülni, a „rejtsük az objektum implementációját egy pointer mögé" játékot mi is eljátszhatjuk.

Lássuk, hogyan választhatjuk el egymástól a `Person` felületét és implementációját ezzel a technikával. Először is, a `Person` osztályt deklaráló fejállományba *(header file)* csak az alábbit írjuk:

```
// A fordítóknak még mindig szükségük van ezekre a típusnevekre
// a Person konstruktorhoz.
class string;                  // Ismét lásd a 49. jó tanácsot,
                               // hogy ez gyakorlatilag miért
                               // nem megy string esetében.

class Date;
class Address;
class Country;
// Egy Person objektum implementációs részleteit
// a PersonImpl osztály fogja tartalmazni;
// Ez csak az osztálynév előzetes deklarációja.
class PersonImpl;

class Person {
public:
   Person(const string& name, const Date& birthday,
          const Address& addr, const Country& country);
   virtual ~Person();
   ...                         // Másoló konstruktor, operator=.

   string name() const;
   string birthDate() const;
   string address() const;
   string nationality() const;
private:
   PersonImpl *impl;           // Pointer az implementációra.
};
```

A `Person` felhasználóit most már teljesen elválasztottuk a sztringek, dátumok, címek, országok és személyek implementációs részleteitől. Ezek az osztályok tetszés szerint változtathatók, és a `Person` felhasználói ezt szerencsére észre sem veszik. És ami sokkal nagyobb szerencse, nem kell őket újrafordítani. Továbbá, mivel a felhasználók nem látják a `Person` osztály implementációs részleteit, egyáltalán nem valószínű, hogy olyan kódot írnak, amely bármilyen módon függ ezektől a részletektől. Ez a felület és az implementáció igazi szétválasztása.

Ennek a szétválasztásnak az a kulcsa, hogy az osztály*definíciók*tól való függést az osztály*deklarációk*tól való függéssel helyettesítjük. A fordítási függőségek minimalizálásáról jegyezzük meg a következőt: ahol csak lehet, tegyük a fejállományainkat önállóvá, ott pedig, ahol nem lehet, csak az osztálydeklarációktól függjenek, az osztálydefinícióktól ne. Minden más már egyenesen következik ebből a tervezési stratégiából.

Íme három közvetlen következmény:

- **Kerüljük az objektumok használatát ott, ahol objektumokra hivatkozó referenciák vagy pointerek is elegendőek!** Egy típusra hivatkozó pointer vagy referencia definiálásához elegendő a típus *deklarációja*. Egy adott típushoz tartozó *objektum* definiálásának elengedhetetlen feltétele a típusdefiníció.

- **Osztálydefiníciók helyett használjunk osztálydeklarációkat, amikor csak lehet!** Jegyezzük meg, hogy *soha sincs* szükség az osztály definíciójára ahhoz, hogy egy olyan függvényt deklaráljunk, amelyik az osztályt használja, még akkor sem, ha a függvény az osztály típusát érték szerint veszi át, vagy érték szerint adja vissza:

```
class Date;                  // Osztálydeklaráció.
Date returnADate();          // Jó — nincs szükség a
void takeADate(Date d);      // Date osztály definíciójára.
```

  Persze az érték szerinti paraméterátadás *(pass-by-value)* általában rossz ötlet (lásd a 22. jó tanácsot), de ha valamilyen okból kénytelenek vagyunk használni, az még nem indok a szükségtelen fordítási függőségek bevezetésére.

  Azokat az olvasókat, akik meglepődnek azon, hogy a `returnADate` és a `takeADate` deklarációi lefordulnak a `Date` definíciója nélkül, üdvözlöm a klubban. Én is meglepődtem. Egyébként ez nem olyan fura, mint amilyennek kinéz, hiszen bárhol, ahol ezeket a függvényeket *meghívják*, a `Date` definíciójának is láthatónak kell lennie. Tudom, most mit gondolnak: minek deklaráljunk olyan függvényeket, amelyeket *senki sem* hív meg? Egyszerű a válasz: nem arról van szó, hogy senki sem hívja meg őket, hanem, hogy *nem mindenki* hívja meg őket. Ha például egy többszáz függvénydeklarációt tartalmazó (valószínűleg több névtérre *(namespace)* kiterjedő – lásd a 28. jó tanácsot) könyvtárat tekintünk, kicsi rá az esély, hogy minden felhasználó meghívja az összes függvényt. Azzal, hogy az osztálydefiníciók – `#include` direktívákon keresztüli – megadásának terhét a *függvénydeklarációkat* tartalmazó fejállományokból a *függvényhívásokat* tartalmazó felhasználói állományokba helyezzük át, megszüntethetjük a felhasználók természetellenes függését olyan típusdefinícióktól, amelyekre nincs is igazán szükségük.

- **Csak akkor `#include`-oljunk fejállományokat a fejállományainkban, ha a nélkül nem fordulnának le!** Deklaráljuk manuálisan a számunkra szükséges osztályokat, és legyen a fejállományok felhasználóinak gondja minden olyan további fejállomány `#include`-olása, amelyre az *ő* kódjuk lefordulása miatt van szükség. Bizonyára lesznek felhasználók, akik morgolódni fognak, hogy ez így kényelmetlen, mi azonban dőljünk hátra nyugodtan abban a tudatban, hogy sokkal több fájdalomtól óvjuk meg őket, mint amennyit okozunk nekik. Ez egy olyannyira elismert technika, hogy még a szabvány C++ könyvtár is nagy becsben tartja (lásd a 49. jó tanácsot). Az `<iosfwd>` fejállomány az iostream könyvtárban található típusok deklarációit (és csak deklarációit) tartalmazza.

Az olyan – `Person`-hoz hasonló – osztályokat, amelyek csak egy, az implementációjukra mutató pointert tartalmaznak, angolszász nyelvterületen gyakran nevezik *Handle* (Kezelő) vagy *Envelope* (Boríték) osztálynak – a mutatott osztályok elnevezésére az első esetben a *Body* (Törzs) osztály, a második esetben a *Letter* (Levél) osztály elnevezés szolgál. Ezeket az osztályokat néha *Cheshire Cat* osztályoknak is hívják. Ez a

név utalás az *Alice Csodaországban*[14] macskájára, amelyik ha akarja, hátra hagyhatja a mosolyát, amikor eltűnik.

Ha az olvasót érdekli, hogy a Handle osztályok hogyan dolgoznak, egyszerű a válasz: minden nekik szóló függvényhívást továbbítanak a megfelelő Body osztálynak, és azok végzik el az igazi munkát. Nézzük meg például, a `Person` tagfüggvényére, hogyan lehetne azokat implementálni:

```
#include "Person.h"          // Mivel a Person osztályt
                             // implementáljuk, #include-olnunk
                             // kell az osztálydefinícióját.

#include "PersonImpl.h"      // A PersonImpl osztály definícióját
                             // is #include-olni kell, különben
                             // nem tudnánk meghívni a tag-
                             // függvényeit. Vegyük észre,
                             // hogy a PersonImpl-nek pontosan
                             // ugyanazok a tagfüggvényei, mint
                             // a Person-nak - a felületük
                             // megegyezik.

Person::Person(const string& name, const Date& birthday,
               const Address& addr, const Country& country)
{
   impl = new PersonImpl(name, birthday, addr, country);
}
string Person::name() const
{
   return impl->name();
}
```

Figyeljük meg, hogy a `Person` konstruktor hogyan hívja meg a `PersonImpl` konstruktort (implicit módon, a `new`-t használva – lásd az 5. jó tanácsot), vagy hogy a `Person::name` hogyan hívja meg a `PersonImpl::name` függvényt. Ez fontos. Ha a `Person`-t `Handle` osztályként valósítjuk meg, az osztály ténykedése nem változik meg, csak az, hogy hol végzi azt.

A Handle osztályt alkalmazó megközelítés alternatívájaként a `Person`-t egy speciális, *protokoll osztálynak (protocol class)* nevezett absztrakt bázisosztályként is létrehozhatjuk. Definíció szerint egy protokoll osztálynak nincs implementációja, egyetlen célja felület meghatározása a származtatott osztályok számára (lásd a 36. jó tanácsot). Ezért aztán rendszerint nincsenek adattagjai és konstruktorai, csak egy virtuális destruktort (lásd a 14. jó tanácsot) és a felületet meghatározó tisztán virtuális függvényeket *(pure virtual functions)* tartalmazza. Egy lehetséges, a `Person`-nak megfelelő protokoll osztály így nézhetne ki:

[14] Lewis Carroll: Alice Csodaországban. Móra Ferenc Ifjúsági Könyvkiadó, Budapest, 1974. Ford.: Kosztolányi Dezső. Az eredeti műben szereplő „Cheshire Cat" Kosztolányi Dezső fordításában „fakutya", tehát ezeket az osztályokat Magyarországon fakutya osztályoknak lehetne hívni. – A szerk.

```
class Person {
public:
   virtual ~Person();
   virtual string name() const = 0;
   virtual string birthDate() const = 0;
   virtual string address() const = 0;
   virtual string nationality() const = 0;
};
```

E `Person` osztály felhasználóinak `Person` pointerekkel és referenciákkal kell programozniuk, ugyanis tisztán virtuális függvényeket tartalmazó osztályokat nem lehet példányosítani. (Mint azt látni fogjuk, a `Person`-ből *származtatott* osztályokat viszont lehet példányosítani.) A Handle osztályok felhasználóihoz hasonlóan a protokoll osztályok felhasználóit is csak akkor kell újrafordítani, ha a protokoll osztály felülete megváltozik.

Természetesen a protokoll osztályok felhasználóinak *valahogyan* új objektumokat is létre kell hozniuk. Ehhez általában egy olyan függvényt hívnak meg, amely a ténylegesen példányosított, eltakart *(hidden)* (származtatott) osztályok konstruktorának szerepét játssza. Ezeknek a függvényeknek több elnevezése is ismeretes (pl. *factory függvény* vagy *virtuális konstruktor*), de mind ugyanúgy viselkednek: pointereket adnak vissza olyan dinamikusan lefoglalt objektumokra, amelyek a protokoll osztály felületét valósítják meg. Egy ilyen függvényt így deklarálhatunk:

```
// A makePerson egy „virtuális konstruktor" (más néven „factory
// függvény") a Person felületét megvalósító objektumokhoz.
Person*
   makePerson(const string& name,        // Az adott paraméterek-
              const Date& birthday,      // kel inicializált
              const Address& addr,       // új Person-ra mutató
              const Country& country);   // pointert ad vissza.
```

A felhasználók pedig így használhatják:

```
string name;
Date dateOfBirth;
Address address;
Country nation;
...
// Egy, a Person felületét megvalósító objektum létrehozása.
Person *pp = makePerson(name, dateOfBirth, address, nation);
...
cout << pp->name()              // Az objektum használata
     << " was born on "         // a Person felületén
     << pp->birthDate()         // keresztül.
     << " and now lives at "
     << pp->address();
...
```

```
delete pp;                      // Az objektum törlése, ha már
                                // nincs rá szükség.
```

Mivel a makePerson-hoz hasonló függvények szorosan kapcsolódnak ahhoz a protokoll osztályhoz, amelynek a felületét az általuk létrehozott objektumok valósítják meg, jó ízlésre vall, ha a protokoll osztályon belül static-ként deklaráljuk őket.

```
class Person {
public:
   ...                          // Mint fent.

// A makePerson most az osztály tagfüggvénye
static Person * makePerson(const string& name,
                           const Date& birthday,
                           const Address& addr,
                           const Country& country);
};
```

Így nem fog a globális (vagy bármely más) névtér zsúfolásig megtelni ilyen természetű függvényekkel (lásd még a 28. jó tanácsot).

Eljön persze a pont, amikor a protokoll osztályfelületet megvalósító konkrét osztályokat kell definiálni, és a valódi konstruktorokat is meg kell hívni. Ez mind a színfalak mögött, a virtuális konstruktorok implementációs fájljaiban történik. A Person protokoll osztálynak a RealPerson lehet például egy konkrét leszármazottja, amely az örökölt virtuális függvények implementációját biztosítja:

```
class RealPerson: public Person {
public:
   RealPerson(const string& name, const Date& birthday,
              const Address& addr, const Country& country)
   :  name_(name), birthday_(birthday),
      address_(addr), country_(country)
   {}

   virtual ~RealPerson() {}
   string name() const;         // E függvények implementációját
   string birthDate() const;    // nem mutatjuk meg,
   string address() const;      // de könnyű elképzelni őket.
   string nationality() const;

private:
   string name_;
   Date birthday_;
   Address address_;
   Country country_;
};
```

Ha adott a RealPerson, akkor már igazán triviális feladat a Person::makePerson megírása:

```
Person * Person::makePerson(const string& name,
                            const Date& birthday,
                            const Address& addr,
                            const Country& country)

{
   return new RealPerson(name, birthday, addr, country);
}
```

A RealPerson jól szemlélteti a protokoll osztályok implementálására használt két leggyakoribb módszer közül az egyiket. Ezek az osztályok felületük specifikációját a – Person – protokoll osztálytól öröklik, majd megvalósítják a felület függvényeit. A protokoll osztályok implementálásának egy másik módszere – a 43. jó tanácsban részletesen vizsgált – többszörös öröklődést *(multiple inheritance)* használja.

Jó, akkor a Handle és a protokoll osztályok szétválasztják a felületet és az implementációt, és így csökkentik a fájlok közötti fordítási függőséget. Amilyen cinikus az olvasó, tudom, hogy már várja a fekete levest. „És ez a nagy hókusz-pókusz mibe fog nekem kerülni?" – dünnyögi. A válasz a számítástechnikában már nem újdonság: némi futási sebességbe és objektumonként egy kicsit több memóriába.

A Handle osztályok esetében a tagfüggvényeknek az implementációs pointeren keresztül kell elérniük az objektum adatait. Ez minden egyes eléréshez hozzáad egy indirekciót. No meg ennek az implementációs pointernek a méretét hozzá kell adnunk az egyes objektumok tárolásához szükséges memóriához. Végül, az implementációs pointert – a Handle osztály konstruktoraiban – egy dinamikusan lefoglalt implementációs objektumra kell inicializálni, így aztán kitesszük magunkat a dinamikus memóriafoglalásban – és felszabadításban – (lásd a 10. jó tanácsot) rejlő plusz költségeknek is.

Protokoll osztályok esetén minden függvényhívás virtuális, úgyhogy minden függvényhíváskor meg kell fizetnünk az indirekt ugrás költségét (lásd a 14. jó tanácsot). A protokoll osztályból származtatott objektumokban kell lennie továbbá egy virtuális táblapointernek *(virtual table pointer)* (szintén lásd a 14. jó tanácsot). Ez a pointer megnövelheti az objektumok tárolásához szükséges memóriát attól függően, hogy az objektumnak csak a protokoll osztályból származnak-e virtuális függvényei, vagy sem.

Végül, sem a Handle, sem a protokoll osztályok nem tudják igazán kihasználni az inline függvényekben rejlő lehetőségeket. Az inline függvények gyakorlati alkalmazása során mindig szükség van az implementációs részletek elérésére, a Handle és protokoll osztályok tervezésekor viszont pont ennek elkerülése volt az elsődleges cél.

Komoly hiba lenne azonban a Handle és protokoll osztályokat csak azért elutasítani, mert költséggel járnak. A virtuális függvények is ilyenek, mégsem akarunk lemondani róluk, ugye? (Ha mégis, akkor az olvasó nem a megfelelő könyvet tartja a kezében.) Nézzük ezeknek a technikáknak az alkalmazását inkább az evolúció szemszögéből. A fejlesztés során azért használjuk a Handle és protokoll osztályokat, hogy minimalizáljuk az implementációk változásainak felhasználókra gyakorolt hatását. A Handle és protokoll osztályokat csak akkor cseréljük le a kész konkrét osztályokra, ha kimutatható, hogy a

sebesség- és/vagy méretbeli eltérés elég jelentős ahhoz, hogy indokolja az egyes osztályok szorosabb összekapcsolását. Reméljük, egyszer lesznek majd olyan eszközeink, amelyek ezt az átalakítást automatikusan elvégzik.

A Handle, protokoll és konkrét osztályok szakszerű keverése lehetővé teszi olyan szoftverrendszerek fejlesztését, amelyek hatékonyan futtathatók, könnyen továbbfejleszthetők, de van egy komoly hátrányuk: lehet, hogy le kell rövidítenünk miattuk a programok újrafordítása alatti kávészüneteket.

Öröklődés és objektumorientált tervezés

Nagyon sokan gondolják azt, hogy az objektumorientált programozás csak az öröklődésről szól. Hogy ez igaz-e, arról lehet vitatkozni, de a többi fejezet jó tanácsainak sokasága elég meggyőzően mutatja, hogy C++-ban nemcsak úgy programozhatunk hatékonyan, hogy megadjuk, mely osztályok mely osztályokból származzanak – ennél jóval több eszköz áll rendelkezésünkre.

Az osztályhierarchiák tervezése és implementálása alapvetően különbözik mindattól, amit a C világában találunk. Kétség sem fér ahhoz, hogy az öröklődés és az objektumorientált tervezés terén kell legradikálisabban újragondolnunk a szoftverrendszerek felépítésére vonatkozó elképzeléseinket. Ráadásul a C++ az objektumorientált építőelemek zavarba ejtő választékát nyújtja: a nyilvános *(public)*, védett *(protected)* és privát bázisosztályokat, a virtuális és nemvirtuális bázisosztályokat, a virtuális és nemvirtuális tagfüggvényeket stb. Ezek az elemek mind kölcsönhatásban állnak egymással, és persze a nyelv többi összetevőjével is. Ezért aztán ijesztő feladatnak tűnhet azt megérteni, hogy ezek az elemek mit jelentenek, mikor használhatók, és hogyan ötvözhetők legjobban a C++ nem objektumorientált eszközeivel.

Tovább bonyolítja a helyzetet az a tény, hogy a nyelv különböző elemeinek működése ránézésre többé-kevésbé megegyezik. Például:

- Olyan osztályok gyűjteményére van szükségünk, amelyek sok közös tulajdonsággal rendelkeznek. Öröklődést használjunk, és származtassunk minden osztályt egy közös bázisosztályból, vagy használjunk inkább sablonokat *(templates)*, és generáltassuk az osztályokat egy közös kódvázból?
- Az A osztályt a B osztály fogalmán keresztül kellene implementálnunk. Legyen A-ban egy B típusú adattag, vagy inkább legyen az A osztály a B privát örököse?
- A szabvány könyvtárban nem szereplő, típushelyes *(type-safe)* homogén tároló osztályt *(container class)* kell terveznünk. (A 49. jó tanács olyan tárolókat sorol fel, amelyek *benne vannak* a szabvány könyvtárban). Sablont használjunk, vagy jobb lenne típushelyes felületeket építeni egy generikus (`void*`) pointerekkel megvalósított osztály köré?

Az itt következő tanácsok útmutatást adnak arra nézve, hogy hogyan válaszolhatunk az efféle kérdésekre. Persze nem remélhetem, hogy az objektumorientált tervezés min-

den szempontját sorra tudom venni. Elmagyarázom inkább a C++ különböző elemeinek igazi *jelentését*, azt, hogy mit mondunk *valójában*, amikor ezt vagy azt a nyelvi eszközt használjuk. A publikus öröklődés például „azegy" *(isa)* relációt jelent (lásd a 35. jó tanácsot), és ha bármilyen más jelentéssel próbáljuk felruházni, hamar bajba jutunk. Hasonlóan, egy virtuális függvény azt jelenti, hogy „felületet kell származtatnunk", míg egy nemvirtuális függvény jelentése az, hogy „*mind* a felületnek, *mind* az implementációnak öröklődnie kell". Már sok C++ programozónak okozott kimondhatatlan szenvedést az, hogy nem tudták ezeket a jelentéseket elkülöníteni egymástól.

Ha értjük a C++ sokszínű elemeinek jelentését, azt vesszük észre, hogy megváltozik az objektumorientált tervezéssel kapcsolatos szemléletünk. Ahelyett, hogy a nyelvi elemek megkülönböztetésén rágódnánk, arról fog szólni a történet, hogy mit akarunk mondani a szoftverrendszerünkről. Ha ugyanis egyszer már tudjuk, hogy mit akarunk mondani, akkor különösebb nehézségek nélkül le tudjuk azt fordítani a megfelelő C++ elemek nyelvére.

Mondjuk azt, amit gondolunk és mindig értsük is, amit mondunk. E két dolog fontosságát nem lehet túlbecsülni. Az itt következő jó tanácsok tüzetesen megvizsgálják, hogy mindezt hogyan tehetjük hatékonyan. A 44. jó tanács összefoglalja a C++ objektumorientált elemei és az elemek jelentése közötti megfeleléseket, ezért egyrészt a rész sarokkövének tekinthető, másrészt pedig a téma jövőbeni tárgyalásának hivatkozási alapjául szolgál.

35. jó tanács: Bizonyosodjunk meg arról, hogy a publikus öröklődés az „azegy" relációt fejezi ki

William Dement *Some Must Watch While Some Must Sleep* című könyvében (W. H. Freeman and Company, 1974) elmesél egy történetet arról, hogyan kísérelte meg kurzusának legfontosabb leckéit diákjai emlékezetébe vésni. Elmondta osztályának, hogy egy átlag brit iskolás állítólag alig jegyez meg történelemből annál többet, mint hogy a hastings-i csata 1066-ban volt. Dement kiemelte, hogy ha egy gyerek alig emlékszik valamire, az 1066-os évszámot akkor is megjegyzi. Aztán azzal folytatta, hogy *ő* az óráit látogató diákoknak csak néhány fontos üzenetet kommunikált, többek között azt – ami elég érdekes –, hogy az altatók álmatlanságot okoznak. Arra kérte hallgatóit, hogy ezt a néhány sarkalatos tényt akkor is jegyezzék meg, ha az órákon elhangzottakból minden mást elfelejtenek, a félév során pedig többször is visszatért rájuk.

A kurzus végén a záróvizsga utolsó feladata ez volt: Írjon egy olyan dolgot a kurzuson tanultakból, amelyre egész életében emlékezni fog! A vizsgalapok értékelésekor Dement nagyon megdöbbent. Majdnem mindenki azt írta: 1066.

Így aztán lázas sietséggel kihirdetem, hogy az objektumorientált programozás legfontosabb (C++-ra vonatkozó) szabálya a következő: a publikus öröklődés *(public inheritance)* jelentése „azegy" *(isa)*. Ezt véssük jól az emlékezetünkbe!

Ha azt írjuk, hogy a D osztály (Derived = származtatott) publikusan származik a B osztályból (Base = bázis), akkor tulajdonképpen azt mondjuk a C++ fordítónak – no meg a kódunkat olvasó embereknek –, hogy minden D típusú objektum egyben B tí-

pusú objektum is, de *ez nem igaz visszafelé.* Azt mondjuk, hogy B a D-nél általánosabb, míg a D pedig a B-nél speciálisabb fogalmat jelöl. Azt is állítjuk, hogy ahol szerepelhet egy B típusú objektum, ott ugyanúgy használható egy D típusú objektum is, mert minden D típusú objektum „azegy" B típusú objektum. Másrészt viszont, ahol egy D típusú objektumra van szükségünk, ott a B típusú nem lesz jó: minden D „azegy" B, de ez fordítva nem igaz.

A C++ kikényszeríti a publikus öröklődés ilyen értelmezését. Tekintsük a következő példát:

```
class Person { ... };
class Student: public Person { ... };
```

Mindennapi tapasztalatainkból tudjuk, hogy minden diák (Student) személy (Person), de nem minden személy diák. Ez a hierarchia éppen ezt állítja. Azt várjuk, hogy minden, ami igaz egy személyre – például, hogy van születési dátuma –, az igaz egy diákra is, de arra nem számítunk, hogy minden, ami egy diákra igaz – mondjuk, hogy beiratkozott valamilyen iskolába –, az általában a személyekre is igaz. A személy fogalma általánosabb, mint a diák fogalma, a diák egy speciális típusú személy.

A C++ birodalmában minden olyan függvény, amely `Person` típusú paramétert (vagyis `Person`-ra hivatkozó pointert vagy referenciát) vár, átvesz helyette egy `Student` objektumot (vagyis egy `Student`-re hivatkozó pointert vagy referenciát):

```
void dance(const Person& p);  // Bárki táncolhat.

void study(const Student& s); // Csak a diákok tanulnak.

Person p;                     // p egy személy (Person).
Student s;                    // s egy diák (Student).

dance(p);                     // Rendben, p egy személy.
dance(s);                     // Rendben, s egy diák és
                              // egy diák „azegy" személy.

study(s);                     // Ez is jó.
study(p);                     // Hiba! p nem diák.
```

Ez csak a *publikus* öröklődésre igaz. A C++ csak akkor fog a fent leírt módon viselkedni, ha a `Student` publikusan származik a `Person` osztályból. A privát öröklődés egészen mást jelent (lásd a 42. jó tanácsot). Azt pedig, hogy a védett öröklődés *(protected inheritance)* mit is jelent, úgy tűnik, senki sem tudja.

A publikus öröklődés és az „azegy" ekvivalenciája egyszerűnek hangzik, de a gyakorlatban a dolgok nem mindig ilyen kézenfekvőek. Az ösztönös megérzéseink néha félrevezethetnek bennünket. Tény például, hogy a pingvin az egy madár, és az is tény, hogy a madarak tudnak repülni. Ha ezt megpróbáljuk naivan C++-ban kifejezni, erőfeszítéseink ezt eredményezik:

```
class Bird {
public:
   virtual void fly();        // A madarak tudnak repülni.
   ...
};
class Penguin:public Bird {   // A pingvin madár.
   ...
};
```

Már bajba is kerültünk, mert ez a hierarchia azt mondja, hogy a pingvinek tudnak repülni, és mi tudjuk, hogy ez nem igaz. Mi történt?

Ebben az esetben a pontatlan (magyar) nyelv áldozatai lettünk. Amikor azt mondjuk, hogy a madarak tudnak repülni, akkor ezen nem azt értjük, hogy *minden* madár tud repülni, hanem csak azt, hogy a madarak általában képesek a repülésre. Ha precízebbek lennénk, akkor elismernénk, hogy valójában sokféle repülni nem tudó madár létezik, és az alábbi, a valóságot sokkal jobban modellező hierarchiával rukkolnánk elő:

```
class Bird {
   ...                  // Nem deklarálunk
};                      // fly (repülni) függvényt.
class FlyingBird: public Bird {
public:
   virtual void fly();
   ...
};
class NonFlyingBird: public Bird {
   ...                  // Nem deklarálunk fly függvényt.
};
class Penguin: public NonFlyingBird {
   ...                  // Nem deklarálunk fly függvényt.
};
```

Ez a hierarchia sokkal hívebben tükrözi ismereteinket, mint az eredeti terv.

Még mindig nem végeztünk teljesen ezekkel a szárnyasokkal, mert bizonyos szoftverrendszerekben teljesen helyénvaló lenne azt mondani, hogy a pingvin „azegy" madár. Ha teszem azt az alkalmazásnak sok köze van csőrökhöz és szárnyakhoz, de semmi a repüléshez, akkor az eredeti hierarchia teljesen jól használható. Bármilyen bosszantónak is tűnik, itt csak arról van szó, hogy nem létezik egy olyan ideális terv, amely minden szoftverre működik. Az, hogy épp melyik terv a legjobb, függ attól, hogy milyen elvárásokat támasztanak a rendszerrel szemben most és a jövőben. Ha az alkalmazásnak nincs tudomása a repülésről, és várhatóan a jövőben sem várják tőle, hogy legyen, akkor a `Penguin` osztály származtatása a `Bird`-ből tökéletesen indokolt tervezési döntés. Sőt, valószínűleg jobb döntés annál is, mint a repülni tudó és a repülni nem tudó madarak megkülönböztetése, mert ez a megkülönböztetés abból a világból is hiányzik, amelyet leképezni próbálunk. Legalább olyan rossz tervezési döntés felesleges osztályokat bevezetnünk egy hierarchiába, mint amilyen rossz az osztályok közötti hibás öröklési relációk meghatározása.

Létezik egy felfogás, amely szerint másképp is megoldható az általam a „Minden madár tud repülni, a pingvinek madarak, a pingvinek nem tudnak repülni, hoppá!”-nak elnevezett probléma. Eszerint a `fly` függvényt pingvinekre úgy definiáljuk át, hogy futási idejű hibát *(runtime error)* generáljon:

```
void error(const string& msg); // Máshol van definiálva.

class Penguin: public Bird {
public:
   virtual void fly() { error("Penguins can't fly!"); }
   ...
};
```

A Smalltalkhoz hasonló, interpretált programozói nyelvek hajlanak e megközelítés elfogadására, de fontos felismernünk, hogy ez a megoldási mód valami egészen másról szól, mint gondolnánk. *Nem* azt mondja, hogy „A pingvinek nem tudnak repülni”, hanem azt, hogy „A pingvinek tudnak repülni, de részükről hiba ezzel próbálkozni.”

Hogy miben áll a kettő közötti különbség? A hiba észlelésének időpontjában. Azt a kijelentést, hogy „A pingvinek nem tudnak repülni”, a fordítók is ki tudják kényszeríteni, míg „A pingvinek részéről hiba a repüléssel próbálkozni” állítás megsértését csak futási időben lehet észlelni.

Az „A pingvinek nem tudnak repülni” kijelentés kifejezésre juttatásához azt kell biztosítanunk, hogy a `Penguin` objektumokra ne legyen definiálva ilyen függvény:

```
class Bird {
   ...                     // Nem deklarálunk fly függvényt.
};
class NonFlyingBird: public Bird {
   ...                     // Nem deklarálunk fly függvényt.
};
class Penguin: public NonFlyingBird {
   ...                     // Nem deklarálunk fly függvényt.
};
```

Ha megpróbálunk egy pingvint röptetni, akkor a fordítóprogram megdorgál bennünket a szabályszegésért:

```
Penguin p;
p.fly();                   // Hiba!
```

Ez a Smalltalk megközelítéstől nagyon eltérő viselkedés, mert azzal a módszerrel a fordítóprogram egy szót sem szólna.

A C++ filozófiája alapjaiban különbözik a Smalltalk filozófiájától, úgyhogy jobb, ha C++ módra tesszük a dolgunkat, legalábbis amíg C++-ban programozunk. Ráadásul a hibák fordítási idejű felismerésének vannak bizonyos gyakorlati előnyei is a futási idejűvel szemben – lásd a 46. jó tanácsot.

Lehet, hogy mindebből az olvasó azt a következtetést vonja le, hogy a madártan területén nem jók a megérzései, de az elemi geometriai tudására azért biztonsággal támaszkodhat, nem? Úgy értem, a téglalapok és a négyzetek már nem olyan bonyolult dolgok, ugye?

Válaszoljon akkor erre az egyszerű kérdésre: a `Square` *(négyzet)* osztály publikusan származzon-e a `Rectangle` *(téglalap)* osztályból?

```
   Rectangle
       ▲
       |
    Square
```

„Naná!" mondja az olvasó, „Persze! Mindenki tudja, hogy a négyzet téglalap, de fordítva általában nem igaz." Ez így is van, legalábbis középiskolában. De mi szerintem már nem vagyunk középiskolások.

Nézzük ezt a kódot:

```
class Rectangle {
public:
   virtual void setHeight(int newHeight);
   virtual void setWidth(int newWidth);
   virtual int height() const;        // Az aktuális értékeket
   virtual int width() const;         // adják vissza.
   ...
};

void makeBigger(Rectangle& r)         // Az r területét
{                                     // megnövelő függvény.
   int oldHeight = r.height();
   r.setWidth(r.width() + 10);        // 10-et ad r
                                      // szélességéhez
   assert(r.height() == oldHeight);   // kijelentjük, hogy r
}                                     // magassága változatlan.
```

Látható, hogy a feltétel mindig teljesül. A `makeBigger` az `r`-nek csak a szélességét változtatja meg. A magassága sosem változik.

Tekintsük most a következő kódot, amely publikus öröklődéssel teszi lehetővé, hogy a négyzeteket téglalapként kezeljük:

```
class Square: public Rectangle { ... };
Square s;
...
assert(s.width() == s.height());      // Ennek minden négyzetre
                                      // teljesülni kell.
```

```
makeBigger(s);                    // Az öröklődés miatt s
                                  // „azegy" téglalap, így
                                  // megnövelhetjük a
                                  // területét.

assert(s.width() == s.height());  // Ennek még mindig igaznak
                                  // kell lenni minden
                                  // négyzetre.
```

Ugyanolyan tiszta sor, mint az előzőekben volt, hogy ennek az utolsó feltételnek is mindig teljesülnie kell. Ugyanis a négyzet szélessége definíció szerint megegyezik a magasságával.

Problémába ütközünk azonban. Hogyan tudjuk összeegyeztetni az alábbi állításokat?

- A `makeBigger` hívása előtt `s` magassága és szélessége megegyezik;
- A `makeBigger` belsejében `s` szélessége megváltozik, de a magassága nem;
- Miután a `makeBigger`-ből visszatérünk, `s` magassága ismét ugyanannyi, mint a szélessége. (Figyeljük meg, hogy `s`-t referencia szerint adjuk át a `makeBigger`-nek, így a `makeBigger` magát s-t módosítja, és nem `s` egy másolatát.)

Nos?

Üdvözlöm az olvasót a publikus öröklődés csodálatos világában, ahol a más tudományterületeken – a matematikát is beleértve – kifejlesztett ösztöneink nem úgy szolgálnak bennünket, ahogy azt várnánk. Ebben az esetben az okozza az alapvető nehézséget, hogy ami egy téglalapra alkalmazható (a szélessége a magasságától függetlenül módosítható), azt nem alkalmazhatjuk egy négyzetre (ahol a magasságnak és a szélességnek meg kell egyeznie). A publikus öröklődés márpedig azt állítja, hogy minden – *minden!* –, ami alkalmazható a bázisosztály objektumaira, az alkalmazható a származtatott osztály objektumaira *(derived class)* is. A téglalapok és négyzetek esetében (vagy a 40. jó tanácsban található hasonló, listákról és halmazokról szóló példában) ez a kijelentés nem állja meg a helyét, így aztán egyszerűen hibás döntés a köztük lévő kapcsolatot publikus öröklődéssel modellezni. A fordítók természetesen megengedik, de ahogy az imént láttuk, semmi garancia sincs arra, hogy a kód megfelelően fog működni. Persze egyszer minden programozónak meg kell tanulnia – kinek előbb, kinek később –, hogy attól, hogy egy program lefordul, még nem feltétlenül működik jól.

Attól azért ne féljünk, hogy az évek hosszú sora alatt kifejlődött szoftverfejlesztési érzékünk cserbenhagy bennünket az objektumorientált tervezéshez közeledve. Ez a tudás még mindig ugyanolyan értékes, csak annyi történt, hogy a tervezési lehetőségeink tárháza kibővült az öröklődéssel. Ezért aztán teret kell engednünk az új megérzéseinknek, hogy ezek irányíthassanak majd bennünket az öröklődés megfelelő alkalmazásában. Idővel a `Penguin`-nek a `Bird` osztályból, vagy a `Square`-nek a `Rectangle` osztályból való származtatásának az ötlete ugyanúgy megmosolyogtat majd minket, ahogyan most egy több oldal hosszúságú függvény teszi. *Lehet,* hogy helyes az a fajta megközelítés is, csak ennek elég kicsi a valószínűsége.

Természetesen az osztályok között nem csak „azegy" reláció állhat fenn. Gyakori a másik két osztályközi kapcsolat is, a „vanegy" *(has-a)* és a „keresztül implementált" *(is-*

implemented-in-terms-of). Ezeket a relációkat a 40. és a 42. jó tanácsban vizsgálom meg közelebbről. Nem ritka, hogy egy C++ osztályterv azért fullad kudarcba, mert a két másik fontos kapcsolatot is, hibásan, „azegy"-ként modellezik. Ezért is kell megértenünk a fenti relációk közötti különbségeket és megtanulni, hogyan lehet őket C++-ban a legjobban modellezni.

36. jó tanács: Tegyünk különbséget a felület öröklése és az implementáció öröklése között

Ha közelebbről megvizsgáljuk a (publikus) öröklődés látszólag magától értetődő fogalmát, akkor azt találjuk, hogy két jól szétválasztható részre bomlik: a függvényfelületek öröklésére és a függvény-implementációk öröklésére. Az öröklődés e két fajtája közötti különbség pontosan megfelel a függvénydeklarációk és függvénydefiníciók közötti – a könyv Bevezetésében tárgyalt – különbségnek.

Osztálytervezőként néha csak azt szeretnénk, ha a származtatott osztályok egy tagfüggvénynek csak a felületét (deklarációját) örökölnék. Máskor meg az lenne jó, ha a függvénynek mind a felülete, mind az implementációja öröklődne a származtatott osztályokban, azt a lehetőséget is megengedve, hogy a származtatottak az általunk megadott implementációt felülírhassák. És persze van úgy, hogy a felület és az implementáció öröklését a felülírás lehetősége nélkül akarjuk megvalósítani.

Hogy jobban érezzük a különbséget a fenti lehetőségek között, tekintsünk egy mértani alakzatokat reprezentáló osztályhierarchiát egy grafikai alkalmazásban:

```
class Shape {
public:
   virtual void draw() const = 0;
   virtual void error(const string& msg);
   int objectID() const;
   ...
};

class Rectangle: public Shape { ... };
class Ellipse: public Shape { ... };
```

A `Shape` egy absztrakt osztály, a tisztán virtuális *(pure virtual)* `draw` függvénye teszi azzá. Ennek eredményeképp a felhasználók nem tudnak példányokat létrehozni a `Shape` osztályból, csak annak származtatott osztályaiból. A `Shape` mindemellett erős hatást gyakorol az összes olyan osztályra, amely belőle publikusan származik, mert:

- A tagfüggvények *felülete mindig öröklődik*. Amint azt a 35. jó tanács kifejti, a publikus öröklődés „azegy" *(isa)* kapcsolatot jelent, így aztán az, ami igaz egy bázisosztályra, igaz kell, hogy legyen annak származtatott osztályaira is. Ennélfogva, ha egy függvény alkalmazható egy osztályra, akkor annak alosztályaira is alkalmazhatónak kell lennie.

A Shape osztályban három függvényt definiáltunk. Az első, a draw, kirajzolja az aktuális objektumot egy adott kijelzőre. A másodikat, az error-t a tagfüggvények akkor hívják meg, ha hibát kell jelezniük. A harmadik, az objectID, az aktuális objektum egy egyedi egész típusú azonosítóját adja vissza. A 17. jó tanács egy ilyen függvény használatára hoz példát. E függvények deklarációja eltér egymástól: a draw egy tisztán virtuális függvény, az error egy egyszerű (tisztátalanul?) virtuális függvény, míg az objectID egy nemvirtuális függvény. Mi következik ezekből az eltérő deklarációkból?

Tekintsük először a draw tisztán virtuális függvényt. A tisztán virtuális függvények egyik legszembeötlőbb tulajdonsága az, hogy minden őket öröklő konkrét osztályban *muszáj* őket újradeklarálni *(redeclare)*, a másik pedig az, hogy absztrakt osztályokban rendszerint nincsenek definiálva. E két jellemzőt egymáshoz illesztve az alábbi felismerésre juthatunk:

- Egy tisztán virtuális függvény deklarációjának célja az, hogy a származtatott osztályok csak a *függvény felületét* örököljék.

Ez teljesen érthető a Shape::draw függvény esetében, ugyanis minden Shape objektummal szemben ésszerű elvárás, hogy rajzolható (drawable) legyen, a Shape osztály azonban nem tud elfogadható alapértelmezett implementációt biztosítani erre a függvényre. Az ellipszis rajzolásának algoritmusa például nagyon is különbözik a téglalap rajzolásának algoritmusától. A Shape::draw deklarációját jól értelmezzük, ha olyan formába írjuk át, mintha az alosztályok tervezőivel a következőt közölnénk: Gondoskodniuk kell egy draw függvényről, de arról már fogalmam sincs, hogyan tudják azt implementálni!

Egy tisztán virtuális függvény esetében azonban definíció is *megadható*. Azaz gondoskodhatunk egy implementációról a Shape::draw függvényhez úgy, hogy a C++ nem fog panaszkodni, meghívni azonban csak úgy lehet, ha a hívást az osztálynévvel együtt adjuk meg:

```
Shape *ps = new Shape;           // Hiba! a Shape absztrakt.
Shape *ps1 = new Rectangle;      // Rendben.
ps1->draw();                     // A Rectangle::draw-t hívja meg.

Shape *ps2 = new Ellipse;        // Jó.
ps2->draw();                     // Az Ellipse::draw-t hívja meg.
ps1->Shape::draw();              // A Shape::draw-t hívja meg.
ps2->Shape::draw();              // A Shape::draw-t hívja meg.
```

Attól eltekintve, hogy koktélpartikon egész jó benyomást tehetünk ezzel a többi programozóra, gyakorlati haszna vajmi csekély. Jól alkalmazható viszont – mint azt a továbbiakban látni fogjuk – egy olyan módszer megvalósításában, amely az egyszerű (tisztátalanul) virtuális függvények számára a szokásosnál biztonságosabb alapértelmezett implementációt biztosít.

Néha hasznos olyan osztályt deklarálnunk, amely *csupán* tisztán virtuális függvényeket tartalmaz. Egy ilyen *protokoll osztály* csak függvényfelületet tud biztosítani a származtatott osztályok számára, implementációt soha. A protokoll osztályokat a 34. jó tanács írja le részletesen, de a 43. jó tanács is kitér rájuk.

Az egyszerű virtuális függvények története egy kicsit másról szól, mint a tisztán virtuálisaké. A származtatott osztályok a megszokott módon öröklik a függvény felületét, de az egyszerű virtuális függvények hagyományosan implementációt is biztosítanak, amelyet a származtatott osztályok választásuk szerint vagy felülírnak, vagy nem. Ha egy pillanatra belegondolunk, akkor rájöhetünk, hogy:

- Az egyszerű virtuális függvények deklarálásának az a célja, hogy a származtatott osztályok a függvény *felületével együtt alapértelmezett implementációt* is örököljenek.

A `Shape::error` esetében a felület azt mondja, hogy minden osztályban kell lenni egy olyan függvénynek, amely hiba előfordulásakor meghívódik, de minden osztály úgy intézheti a hibakezelést, ahogy jónak látja. Ha egy osztály nem akar semmi különöset tenni ez ügyben, akkor egyszerűen a `Shape` osztályban biztosított alapértelmezett hibakezelésre épít. Vagyis a `Shape::error` deklarációja a következőt mondja a származtatott osztályok tervezőinek: – Rendelkezniük kell egy `error` függvénnyel, de ha nem akarnak saját változatot készíteni, akkor számíthatnak a `Shape` osztály alapértelmezett verziójára is.

Aztán az is kiderül, hogy veszélyes lehet megengednünk rendes virtuális függvényeknek, hogy függvénydeklarációt és alapértelmezett implementációt is meghatározhassanak. Hogy belássuk miért, tekintsük egy XYZ légitársaság repülőinek hierarchiáját. Az XYZ-nek csak kétfajta gépe van, az A modell és a B modell, és mindkettőt ugyanúgy repítik. Ezért aztán az XYZ a következő hierarchiát tervezi meg:

```
class Airport { ... };          // Reptereket reprezentál.
class Airplane {
public:
   virtual void fly(const Airport& destination);
   ...
};
void Airplane::fly(const Airport& destination)
{
   alapértelmezett kód, amely elrepíti a gépet a megadott
   célállomásra
}
class ModelA: public Airplane { ... };
class ModelB: public Airplane { ... };
```

Mivel ki akarjuk fejezni, hogy minden repülőnek támogatnia kell egy `fly` függvényt, és tudjuk, hogy a különböző repülőgépmodelleknek – elvben – a `fly` más-más implementációjára lehet szükségük, az `Airplane::fly` függvényt virtuálisnak deklaráljuk. Mivel azonban el akarjuk kerülni, hogy a `ModelA` és `ModelB` osztályban is le kelljen írnunk ugyanazt a kódot, az alapértelmezett repülési viselkedést az `Airplane::fly` törzsében biztosítjuk, amelyet mind `ModelA`, mind `ModelB` örököl.

Ez egy klasszikus objektumorientált osztályterv. Van két osztály, amelynek van egy közös eleme (a `fly` implementálásának módja), ezért aztán ezt a közös elemet kiemeljük egy bázisosztályba, ahonnan mindkét osztály örökli. Ez a tervezés explicitté teszi a közös elemet, elkerüli a kódismétlést, elősegíti a jövőbeni finomításokat és megkönnyí-

ti a hosszútávú karbantartást. Pont ezek azok az adottságok, amelyekért az objektum-orientált technológiát olyan nagyra becsülik. Az XYZ légitársaság büszke lehet!

Most tegyük fel, hogy az XYZ vagyona gyarapodásával elhatározza, hogy egy új típusú gépet, egy C modellt szerez be. A C modell eltér az A és a B modelltől, méghozzá azért, mert nem úgy repítik.

Az XYZ programozói a C modell osztályát hozzáadják a hierarchiához, de az új modell szolgálatba állítása körüli sietségben elfelejtik újradefiniálni a `fly` függvényt:

```
class ModelC: public Airplane {
   ...                           // Nincs deklarálva
                                 // fly függvény.
};
```

A kódjukban aztán valami ehhez hasonlatos fordul elő:

```
Airport JFK(...);                // A JFK egy New York-i
                                 // repülőtér.
Airplane *pa = new ModelC;
...
pa->fly(JFK);                    // Az Airplane::fly-t hívja meg!
```

Katasztrófa van kilátásban: kísérlet történik egy `ModelC` objektum `ModelA` vagy `ModelB` objektumként való repítésére. Ez a viselkedés nem fogja növelni az utazóközönség bizalmát!

A probléma itt nem az, hogy az `Airplane::fly`-nak alapértelmezett viselkedése van, hanem az, hogy a `ModelC` osztály örökölhette ezt a viselkedést anélkül, hogy kifejezetten kérte volna. Szerencsére az alosztályoknak könnyű felajánlani az alapértelmezett viselkedést úgy, hogy csak akkor kapják meg, ha kérik. A trükk az, hogy elvágjuk a kapcsolatot a virtuális függvény *felülete* és az alapértelmezett *implementációja* között. Íme egy lehetséges megoldás:

```
class Airplane {
public:
   virtual void fly(const Airport& destination) = 0;
   ...
protected:
   void defaultFly(const Airport& destination);
};
void Airplane::defaultFly(const Airport& destination)
{
   alapértelmezett kód, amely elrepíti a gépet a megadott
   célállomásra
}
```

Figyeljük meg, hogy az `Airplane::fly` hogyan változott át *tisztán* virtuális függvénnyé. Ez adja meg a repülés felületét. Az `Airplane` osztályban jelen van az alapér-

telmezett implementáció is, de most egy független függvény – a `defaultFly` – formájában. A `ModelA`-hoz és a `ModelB`-hez hasonló olyan osztályok, amelyek az alapértelmezett viselkedést akarják használni, egyszerűen inline meghívják a `defaultFly` függvényt a saját `fly` függvényük törzsében (a virtuális függvények és az inline használata közötti kölcsönhatásról a 33. jó tanácsban található bővebb információ):

```
class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }
   ...
};
class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }
   ...
};
```

A `ModelC` osztály így már véletlenül sem örökölheti a `fly` helytelen implementációját, mert az `Airplane` tisztán virtuálisa rákényszeríti a `ModelC`-t arra, hogy gondoskodjon a saját `fly` verziójáról.

```
class ModelC: public Airplane {
public:
   virtual void fly(const Airport& destination);
   ...
   };
void ModelC::fly(const Airport& destination)
{
   egy ModelC gép megadott célállomásra történő
   eljuttatásának kódja
}
```

Ez a séma nem bolondbiztos, programozók még mindig bajba „kopipésztelhetik" (copy-and-paste) magukat benne, mégis sokkal megbízhatóbb, mint az eredeti osztályterv. Ami az `Airplane::defaultFly`-t illeti, azért védett *(protected)*, mert igazából csak az `Airplane`-re és annak származtatott osztályaira tartozó implementációs részlet. A repülőgépeket használó felhasználóknak csak azzal kell törődniük, hogy a gépeket lehessen röpíteni, azzal már nem, hogy a repülés hogyan valósul meg.

Fontos részlet az is, hogy az `Airplane::defaultFly` egy *nemvirtuális* függvény. Azért nemvirtuális, mert ezt a függvényt az alosztályoknak nem szabad átdefiniálni. Ez egy olyan igazság, amelynek egy egész jó tanácsot, a 37.-et szenteltem. Ha a `defaultFly` virtuális lenne, akkor egy visszatérő problémával állnánk szemben: mi történne, ha valamelyik alosztály elfelejtené átdefiniálni a `defaultFly` függvényt, pedig feladata lenne?

Vannak, akik ellenzik azt az ötletet, hogy más függvényt biztosítsunk egy művelet felületének és megint másikat alapértelmezett implementációjának megadására, mint a `fly`-t és a `defaultFly`-t a fenti esetben. Azzal érvelnek például, hogy ez a technika beszennyezi az osztály névterét *(namespace)* az egymáshoz igen hasonló függvénynevek elburjánzása miatt. Azzal azonban ők is egyetértenek, hogy a felületet és az alapértelmezett implementációt szét kell választani. Hogyan oldják fel ők ezt a látszólagos ellentmondást? Úgy, hogy kihasználják azt a tényt, hogy bár a tisztán virtuális függvényeket újra kell deklarálni az alosztályokban, lehet saját implementációjuk is. Az `Airplane` hierarchia a következőképpen tud például hasznot húzni a tisztán virtuális függvények definiálásának lehetőségéből:

```
class Airplane {
public:
   virtual void fly(const Airport& destination) = 0;
   ...
};
void Airplane::fly(const Airport& destination)
{
   alapértelmezett kód, amely elrepít egy gépet
   a megadott célállomásra
}
class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { Airplane::fly(destination); }
   ...
};
class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { Airplane::fly(destination); }
   ...
};
class ModelC: public Airplane {
public:
   virtual void fly(const Airport& destination);
   ...
};
void ModelC::fly(const Airport& destination)
{
   egy ModelC gép adott célállomásra juttatásának kódja
}
```

Ez az osztályterv majdnem ugyanaz, mint az előző, a különbség csak annyi, hogy a független `Airplane::defaultFly` függvény helyét a tisztán virtuális `Airplane::fly` függvény törzse vette át. Lényegében a `fly` függvényt két alapvető összetevőjé-

re bontottuk szét. A deklaráció a felületét határozza meg (ezt a származtatott osztályoknak *használniuk kell*), a definíció pedig az alapértelmezett viselkedését (ezt a származtatott osztályok *használhatják*, de csak akkor, ha kifejezetten kérik). A `fly` és a `defaultFly` összevonásával azonban elvesztettük annak a lehetőségét, hogy a két függvényhez eltérő védettségi szintet *(protection level)* rendeljünk: a kód, amely valaha `protected` volt (a `defaultFly`-ban), most `public` (mert a `fly`-ba került).

Eljutottunk végül a `Shape` nemvirtuális függvényéhez, az `objectID`-hez. Egy nemvirtuális tagfüggvényről azt feltételezzük, hogy viselkedése a származtatott osztályokban sem változik. Egy nemvirtuális tagfüggvény valójában egy *specializáció feletti invariánst (invariant over specialization)* határoz meg, mert olyan viselkedést azonosít, amelynek függetlenül attól, hogy egy származtatott osztály mennyire lesz speciális, sosem szabad megváltoznia. Ezért aztán:

- Egy nemvirtuális függvény deklarálásának célja az, hogy a származtatott osztályok *függvényfelületet, és vele együtt kötelező implementációt is örököljenek.*

Gondolhatunk úgy is a `Shape::objectID` deklarációjára, mintha a következőt közölné: „Minden `Shape` objektumnak van egy függvénye, amely előállít egy objektumazonosítót, és ennek az objektum-azonosítónak a kiszámítási módja mindig ugyanaz. Ezt a számítási módszert a `Shape::objectID` definíciója határozza meg, és a származtatott osztályoknak nem szabad azt megváltoztatni." Mivel a nemvirtuális függvény egy specializáció feletti *invariánst* azonosít, soha nem szabad az alosztályokban átdefiniálni – erre a kérdésre a 37. jó tanácsban részletesen kitérek.

A tisztán virtuális, az egyszerű virtuális és a nemvirtuális függvények deklarációja közötti különbségek lehetővé teszik számunkra annak pontos meghatározását, hogy mit akarunk a származtatott osztályokba örököltetni: csak a felületet, a felületet és egy alapértelmezett implementációt, vagy a felületet és egy kötelező implementációt. Mivel ezeknek az egymástól eltérő deklarációtípusoknak a jelentése alapvetően különbözik egymástól, megfontoltan kell választanunk közülük tagfüggvényeink deklarálásakor. Ha így teszünk, biztosan elkerüljük azt a két leggyakoribb hibát, amelyet a kevés tapasztalattal rendelkező osztálytervezők rendszeresen elkövetnek.

Az első hibát akkor vétjük, ha az összes függvényt nemvirtuálisnak deklaráljuk. Ez a származtatott osztályokban nem hagy teret a specializációnak, a nemvirtuális destruktorokkal kifejezetten sok baj van (lásd a 14. jó tanácsot). Természetesen teljesen ésszerű dolog olyan osztályt tervezni, amelyet nem bázisosztálynak szánunk. Ebben az esetben egy kizárólag nemvirtuális tagfüggvényekből álló halmaz teljességgel megfelelő. Ilyen osztályt legtöbbször azonban csak azok deklarálnak, akik nincsenek tisztában a virtuális és nemvirtuális függvények közötti különbségekkel, vagy akik indokolatlanul aggódnak a virtuális függvények használatának költségei miatt. A lényeg az, hogy majdnem minden olyan osztályban lesz virtuális függvény, amelynek bázisosztályként kell funkcionálnia (lásd megint a 14. jó tanácsot).

Azok kedvéért, akiket nyugtalanít a virtuális függvények költsége, hadd idézzem fel a 80-20-as szabályt (lásd még a 33. jó tanácsot), amely kimondja, hogy egy átlagos program futási idejének 80 százalékát kódja mindössze 20 százalékának végrehajtásával tölti. Ez fontos szabály, mert azt jelenti, hogy a függvényhívásainknak átlagosan 80 százaléka lehet virtuális anélkül, hogy ennek bármilyen kimutatható hatása lenne a program összteljesítményére. Mielőtt bárki elkezd attól rettegni, hogy megengedheti-e magának egy virtuális

függvény költségét, elővigyázatosságból előbb győződjön meg arról, hogy a programnak azon a 20 százalékán dolgozik-e, ahol ennek a döntésnek egyáltalán hatása lehet.

A másik gyakori hibát akkor követjük el, ha *minden* tagfüggvényt virtuálisnak deklarálunk. Néha ez a helyes döntés – ott vannak például a protokoll osztályok (lásd a 34. jó tanácsot). Máskor viszont olyan osztálytervezőre is utalhat, aki nem eléggé gerinces ahhoz, hogy határozottan kiálljon valami mellett. A származtatott osztályokban vannak függvények, amelyeknek nem szabad átdefiniálhatónak lenni, és ezt mindig ki is kell nyilvánítanunk róluk úgy, hogy nemvirtuálissá tesszük őket. Senkinek sem használ, ha úgy teszünk, mintha az osztályunk bárkinek bármilyen célra megfelelne azzal a feltétellel, hogy időt szakít a függvényeink átdefiniálására. Ne feledjük, hogy ha van egy B bázisosztályunk, egy D származtatott osztályunk és egy `mf` tagfüggvényünk, akkor az `mf` összes alábbi hívásának jól *kell* működnie:

```
D *pd = new D;
B *pb = pd;
pb->mf();                    // mf hívása egy bázisra.
                             // Mutató pointeren keresztül.
pd->mf();                    // mf hívása egy leszármazottra
                             // Mutató pointeren keresztül.
```

Néha muszáj `mf`-et nemvirtuálissá tenni ahhoz, hogy minden úgy működjön, ahogy kell (lásd a 37. jó tanácsot). Ha van egy specializáció feletti invariánsunk, akkor ne vonakodjunk ezt kimondani!

37. jó tanács: Soha ne definiáljunk át egy örökölt nemvirtuális függvényt

Kétféleképpen közelíthetjük meg ezt a kérdést: az elmélet és a gyakorlat oldaláról. Kezdjük a gyakorlattal. Az elméleti emberek már úgyis hozzászoktak ahhoz, hogy türelmesnek kell lenniük.

Tegyük fel, azt mondom az olvasónak, hogy a D osztály publikusan származik a `B` osztályból, és a `B` osztályban definiálva van egy `mf` nyilvános *(public)* tagfüggvény. Az `mf` visszatérési típusa és paraméterei most lényegtelenek, ezért tételezzük fel, hogy mindkettő `void`. Vagyis a következő kijelentést teszem:

```
class B {
public:
   void mf();
   ...
};
class D: public B { ... };
```

Anélkül, hogy bármit tudnánk `B`-ről, `D`-ről vagy `mf`-ről, ha adott egy `D` típusú `x` objektum,

```
D x;                        // x egy D típusú objektum.
```

akkor nagyon meglepődnénk, ha ez:

```
B *pB = &x;                 // Vegyünk egy x-re mutató pointert.
pB->mf();                   // mf hívása a pointeren keresztül.
```

másképp viselkedne, mint ez:

```
D *pD = &x;                 // Vegyünk egy x-re mutató pointert
pD->mf();                   // mf hívása a pointeren keresztül.
```

Ez azért van így, mert az `mf` tagfüggvényt az `x` objektumra hívjuk meg mindkét esetben. De mivel mindkét esetben ugyanaz a függvény és ugyanaz az objektum szerepel, a két hívásnak nem ugyanúgy kellene viselkednie?

De igen. Bár lehet, hogy mégsem fog. Különösen akkor nem, ha `mf` nemvirtuális, `D` pedig definiálta saját `mf` változatát:

```
class D: public B {
public:
   void mf();               // Eltakarja B::mf-et;
                            // lásd az 50. jó tanácsot.
   ...
};
pB->mf();                   // B::mf hívása.
pD->mf();                   // D::mf hívása.
```

Ennek a kétarcú viselkedésnek az az oka, hogy az olyan *nemvirtuális* függvények, mint a `B::mf` és a `D::mf` statikusan kötöttek *(statically bound)* (lásd a 38. jó tanácsot). Ez azt jelenti, hogy mivel `pB` típusa deklaráció szerint `B`-re mutató pointer, a `pB`-n keresztül meghívott nemvirtuális függvények *mindig* a `B` osztályban definiált függvények lesznek, még akkor is, ha `pB` egy `B`-ből származtatott osztály objektumára mutat, mint ebben a példában.

A *virtuális* függvények ugyanakkor dinamikusan kötöttek *(dynamically bound)* (ismét lásd a 38. jó tanácsot), úgyhogy őket ez a probléma nem érinti. Ha `mf` virtuális függvény lenne, akkor akár `pB`-n, akár `pD`-n keresztül hívnánk meg `mf`-et, mindkettő a `D::mf` végrehajtását eredményezné, mert `pB` és `pD` *valójában* egy `D` típusú objektumra mutat.

A lényeg tehát az, hogy ha egy `D` osztály írásakor átdefiniálunk egy `B` osztályból örökölt nemvirtuális `mf` függvényt, akkor a `D` objektumok nagy valószínűséggel tudathasadásos viselkedést fognak mutatni. Különösen akkor viselkedhet egy `D` objektum úgy, hogy akár `B`, akár `D` is lehetne, amikor az `mf`-et meghívják. A döntő tényezőnek ekkor ugyanis semmi köze sem lesz magához az objektumhoz, hanem csak a rá mutató pointer deklarált típusához. A referenciák ugyanazt a zavarbaejtő viselkedést mutatják, mint a pointerek.

Ennyit a gyakorlati érvekről. Tudom, most mindenki valamiféle elméleti megalapozást vár arra, hogy miért nem szabad az örökölt nemvirtuális függvényeket átdefiniálni. Örömmel engedelmeskedem.

A 35. jó tanács elmagyarázza, hogy a publikus öröklődés „azegy" *(isa)* kapcsolatot jelent, a 36. jó tanácsból pedig azt tudhatjuk meg, hogy egy nemvirtuális függvény deklarálása egy osztályban miért határoz meg az osztályra nézve egy a specializáció feletti invariánst *(invariant over specialization)*. Ha ezeket a megfigyeléseket a `B` és `D` osztályra, valamint a `B::mf` nemvirtuális tagfüggvényre alkalmazzuk, az alábbi megállapításokra jutunk:

- Minden, ami alkalmazható a `B` objektumokra, az alkalmazható a `D` objektumokra is, mert minden `D` objektum „azegy" `B` objektum;
- `B` alosztályainak `mf` felületét *és* implementációját is örökölni kell, mert `mf` nemvirtuális a `B`-ben.

Namármost, ha `D` átdefiniálja `mf`-et, akkor ellentmondás van a tervezésben. Ugyanis, ha `D`-nek *valóban* a `B`-től eltérő módon kell implementálnia `mf`-et, és ha minden `B` objektumnak – függetlenül attól, hogy mennyire speciális – *valóban* az `mf` `B`-beli implementációját kell használnia, akkor egyszerűen nem igaz, hogy minden `D` „azegy" `B`. Ebben az esetben `D`-nek nem szabadna publikusan örökölnie `B`-ből. Másrészt, ha `D`-nek *valóban* publikusan kell örökölnie `B`-ből, és `D`-nek *valóban* a `B`-től eltérő módon kell implementálnia az `mf`-et, akkor meg az nem igaz, hogy az `mf` egy specializáció feletti invariánst fejezne ki a `B`-re nézve. Ebben az esetben az `mf`-nek virtuálisnak kellene lennie. Végezetül, ha *valóban* minden `D` „azegy" `B`, és `mf` valóban megfelel egy specializáció feletti invariánsnak `B`-ben, akkor `D`-nek nem igazán van szüksége az `mf` átdefiniálására, még próbálkoznia sem szabad vele.

Függetlenül attól, hogy melyik érvelés érvényes, valamit fel kell áldozni, ez viszont semmilyen körülmények között se az örökölt nemvirtuális függvények átdefiniálásának tilalma legyen!

38. jó tanács: Soha ne definiáljunk át egy örökölt alapértelmezett paraméterértéket

Már az elején egyszerűsítsük le a téma tárgyalását. Alapértelmezett *(default)* paraméter csak egy függvény részeként létezhet, és csak kétféle függvény esetében beszélhetünk öröklésről: virtuális és nemvirtuális függvény esetében. Így aztán egy alapértelmezett paraméterértéket csak úgy tudunk átdefiniálni, ha átdefiniálunk egy örökölt függvényt. Egy örökölt nemvirtuális függvényt azonban mindig hibás döntés átdefiniálni (lásd a 37. jó tanácsot), így aztán bátran korlátozhatjuk jelen okfejtésünket arra a helyzetre, amikor egy alapértelmezett paraméterértékkel rendelkező *virtuális* függvényt öröklünk.

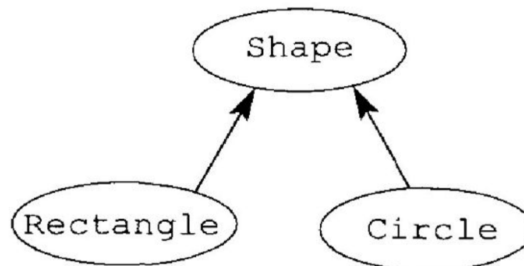
Így már elég egyértelművé válik, miért van egyáltalán szükség erre a jó tanácsra: a dinamikusan kötött *(dynamically bound)* virtuális függvényekkel szemben az alapértelmezett paraméterértékek statikusan kötöttek *(statically bound)*.

Ez meg mi? Az olvasó nem vágja a legfrissebb objektumorientált szakzsargont? Vagy talán a statikus és dinamikus kötés közötti különbség esett ki az amúgy is túlterhelt emlékezetéből? Akkor tekintsük át újra az egészet.

Egy objektum *statikus típusa (static type)* az a típus, amelyet mi deklarálunk számára a program szövegében. Tekintsük az alábbi osztályhierarchiát:

```
enum ShapeColor { RED, GREEN, BLUE };
// Geometriai alakzatok osztálya.
class Shape {
public:
   // Minden alakzatnak biztosítania kell
   // egy önmagát kirajzoló függvényt.
   virtual void draw(ShapeColor color = RED) const = 0;
   ...
};
class Rectangle: public Shape {
public:
   // Vegyük észre az eltérő alapértelmezett
   // paraméter értéket - rossz!
   virtual void draw(ShapeColor color = GREEN) const;
   ...
};
class Circle: public Shape {
public:
   virtual void draw(ShapeColor color) const;
   ...
};
```

Szemléletesen ez így néz ki:



Most nézzük a következő pointereket:

```
Shape *ps;                      // Statikus típus = Shape*
Shape *pc = new Circle;         // Statikus típus = Shape*
Shape *pr = new Rectangle;      // Statikus típus = Shape*
```

Ebben a példában a `ps`, `pc`, és `pr` mindegyike `Shape`-re mutató pointertípusnak lett deklarálva, így aztán mindnek ez a statikus típusa. Vegyük észre, hogy egyáltalán nem számít, mire mutatnak *valójában:* a statikus típusuk ettől függetlenül `Shape*`.

Egy objektum *dinamikus típusát (dynamic type)* az határozza meg, hogy éppen milyen típusú objektumra hivatkozik. Azaz a dinamikus típus azt jelzi, hogy az objektum hogyan fog viselkedni. A fenti példában `pc` dinamikus típusa `Circle*`, `pr` dinamikus típusa pedig `Rectangle*`. Ami `ps`-t illeti, neki nincs dinamikus típusa, merthogy (még) nem mutat semmilyen objektumra sem.

A dinamikus típus – ahogy ezt a neve is sugallja – megváltozhat a program futása során. Ez rendszerint értékadáson keresztül történik:

```
ps = pc;                    // ps dinamikus típusa
                            // most Circle*.
ps = pr;                    // ps dinamikus típusa
                            // most Rectangle*.
```

A virtuális függvények *dinamikusan kötöttek,* ami azt jelenti, hogy a meghívásra kerülő függvény kiválasztása azon objektum dinamikus típusa alapján történik, amelyen keresztül meghívjuk:

```
pc->draw(RED);              // Circle::draw(RED)-et hívja.
pr->draw(RED);              // Rectangle::draw(RED)-et hívja.
```

Tudom, hogy ez szájbarágós duma. Biztosan mindenki érti a virtuális függvényeket. A csavar ott van, amikor a virtuális függvényeinkben alapértelmezett paraméterérték szerepel. Hiszen – mint ahogy már említettem – a virtuális függvények dinamikusan kötöttek, az alapértelmezett paraméterértékek viszont statikusan kötöttek. Ez könnyen oda vezethet, hogy egy *származtatott osztályban* definiált virtuális függvényt úgy hívunk meg, hogy a felhasznált alapértelmezett paraméterérték már egy *bázisosztályból* származik:

```
pr->draw();                 // Rectangle::draw(RED)-et hívja!
```

Ebben az esetben a `pr` dinamikus típusa `Rectangle*`, így a `Rectangle` virtuális függvénye hívódik meg, ahogy az várható is. A `Rectangle::draw`-ban az alapértelmezett paraméter érték a `GREEN`. Mivel azonban a `pr` statikus típusa `Shape*`, a függvényhívás alapértelmezett paraméterértéke a `Shape` osztályból kerül ki, nem a `Rectangle` osztályból! Egy olyan hívást kapunk eredményül, amely a `Shape` és `Rectangle` osztályban is megtalálható `draw`-deklarációk különös és minden bizonnyal előre nem látható kombinációjából áll elő. Higgye el nekem az olvasó, hogy nem szeretné, ha így viselkedne a programja, vagy legalább azt higgye el nekem, hogy a *felhasználók* nem szeretnék, ha a programja így viselkedne.

Mondanom sem kell, hogy jelen esetben a `ps`, `pc`, és `pr` pointer voltának nincs jelentősége. Ha referenciák volnának, a probléma ugyanúgy fennállna. Csak az a fontos, hogy a `draw` virtuális függvény, és az egyik alapértelmezett paraméterérték át lett definiálva egy alosztályban *(subclass).*

Miért ragaszkodik a C++ ehhez a feje tetejére állított viselkedéshez? A válasz a futási idejű *(runtime)* hatékonysággal van kapcsolatban. Ha az alapértelmezett paraméterértékek dinamikusan kötöttek lennének, akkor a fordítóknak arra kellene megoldást találniuk, hogy futási időben a virtuális függvények paramétereinek helyes alapértelmezett értéke(ke)t határozzanak meg valahogy. Ez sokkal bonyolultabb és lassúbb lenne, mint a jelenlegi, fordítási időben alkalmazott mechanizmus. A döntés a sebesség és az egyszerű implementálás javára született meg, amelynek eredményeként olyan futási viselkedést élvezhetünk, amely hatékony, de ha nem fogadjuk meg ennek a jó tanácsnak az intelmeit, akkor zavarba ejtő.

39. jó tanács: Az öröklődési hierarchiában kerüljük a lefelé irányuló konverziót

A mai, gazdasági szempontból felfokozott tempójú világban nem árt, ha szemmel tartjuk pénzügyi intézményeinket. Tekintsünk hát egy bankszámlákat leíró protokoll osztályt (lásd a 34. jó tanácsot):

```
class Person { ... };
class BankAccount {
public:
   BankAccount(const Person *primaryOwner,
               const Person *jointOwner);
   virtual ~BankAccount();
   virtual void makeDeposit(double amount) = 0;
   virtual void makeWithdrawal(double amount) = 0;
   virtual double balance() const = 0;
   ...
};
```

Manapság a bankok elképesztően sokféle számlatípust kínálnak, de az egyszerűség kedvéért tegyük fel, hogy csak egyféle számlatípus létezik, a betéti számla:

```
class SavingsAccount: public BankAccount {
public:
   SavingsAccount(const Person *primaryOwner,
                  const Person *jointOwner);
   ~SavingsAccount();
   void creditInterest();     // Kamat jóváírása a számlán.
   ...
};
```

Hát ez nem egy tekintélyes betéti számla, de mi számít manapság annak? A jelenlegi céljainknak mindenesetre megfelel.

Egy bank valószínűleg nyilvántartja a nála vezetett összes számlát – talán éppen a szabvány könyvtár `list` osztálysablonját (*class template*) használva (lásd a 49. jó tanácsot). Tételezzük fel, hogy ezt a listát nagyon leleményesen `allAccounts`-nak nevezték el:

```
list<BankAccount*> allAccounts;      // A bankban vezetett
                                     // összes számla.
```

Mint minden szabvány tároló, a `list` is *másolatokat* tárol a belerakott dolgokról. Mivel a bank nem akar többszörös másolatot tárolni minden egyes `BankAccount`-ról, úgy dönt, hogy az `allAccounts` a `BankAccount`-ok helyett inkább a `BankAccount`-okra mutató *pointereket* tartalmazzon.

Képzeljük azt, hogy egy olyan kódot kell írnunk, amely az összes számlát bejárja, és mindegyiken jóváírja a kamatot. Próbálkozhatunk ezzel,

```
// Ez a ciklus nem fordul le (aki még sosem látott „bejáró"-t
// (iterator) használó kódot, az nézze meg az itt következőket).
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
   (*p)->creditInterest();     // hiba!
}
```

de a fordítónk gyorsan észhez térít minket: az `allAccounts` a `BankAccount` objektumokra, és nem a `SavingsAccount` objektumokra mutató pointereket tartalmaz, úgyhogy `p` a ciklus minden lefutásakor egy `BankAccount`-ra mutat. Ez a `creditInterest` hívását érvénytelenné teszi, mert a `creditInterest`-et csak `SavingsAccount` objektumokra deklaráltuk, `BankAccount` objektumokra nem.

Akinek a „`list<BankAccount*>::iterator p = allAccounts.begin()`" kifejezésről előbb jut eszébe az adatátviteli zaj, mint a C++, az úgy látszik még sosem részesült abban az örömben, hogy találkozott volna a szabvány könyvtár tároló osztálysablonjaival. A könyvtárnak ezt a részét rendszerint szabvány sablonkönyvtárnak (*Standard Template Library*-nek, azaz „*STL*"-nek) hívják. A 49. jó tanács ad róla egy áttekintést. Nekünk most elég annyit tudnunk, hogy a `p` változó úgy viselkedik, mint egy olyan pointer, amely végiglépked az `allAccounts` elemein, az elejétől a végéig. Azaz `p` úgy tesz, mintha `BankAccount**` típusú volna, a lista elemei pedig egy tömbben lennének tárolva.

Elég kiábrándító, hogy a fenti ciklus nem fordul le. Persze az `allAccounts`-ot úgy definiáltuk, hogy `BankAccount*`-okat tartalmazzon, de mi *tudjuk*, hogy a fenti ciklusban valójában `SavingsAccount*`-okat tartalmaz, mert a `SavingsAccount` az egyetlen olyan osztály, amelyet példányosítani *(instantiate)* lehet. Az ostoba fordítók! Ezért azán úgy döntünk, hogy elmondjuk nekik azt is, ami számunkra nyilvánvaló, de számukra nem, mert túl sötétek ahhoz, hogy maguktól kitalálják, most például azt, hogy az `allAccounts` igazából `SavingsAccount*`-okat tartalmaz:

```
// Ez a ciklus lefordul, ennek ellenére veszedelmes.
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
   static_cast<SavingsAccount*>(*p)->creditInterest();
}
```

Minden problémánk megoldódott! Méghozzá tisztán, elegánsan és tömören, egy egyszerű konverzió *(cast)* használatával. Mi tudjuk, hogy az `allAccounts` milyen pointert tartalmaz valójában, a kicsit agyatlan fordítók viszont nem, ezért egy konverzióval eláruljuk nekik. Mi lehetne ennél logikusabb?

Van egy bibliai analógiám, amivel most előhozakodnék. Egy C++ programozónak olyan a konverzió, mint Évának volt az alma.

Az ilyen konverziót – amely egy bázisosztály pointeréből egy származtatott osztály pointerére irányul – *lefelé konvertálásnak (downcast)* nevezzük, mert az öröklődési hierarchiában lefelé konvertálunk. A lefelé konvertálás az imént látott példában véletlenül pont működik, de ennek a kódnak a karbantartása, amint azt hamarosan látni fogjuk, rémálomba illő.

De térjünk vissza a bankhoz! Tegyük fel, hogy a bank a betéti számla sikerén felbuzdulva elhatározza, hogy folyószámlát is kínál ezentúl. Tételezzük fel azt is, hogy a folyószámlához ugyanúgy tartozik kamat, mint a betétihez:

```
class CheckingAccount: public BankAccount {
public:
   void creditInterest();      // Kamat jóváírása.
   ...
};
```

Mondanom sem kell, hogy az `allAccounts` most már olyan lista, amely tartalmaz betéti és folyószámlára mutató pointert is. A fent megírt kamatjóváíró ciklus máris komoly bajba került.

Először is azért, mert ez a kód továbbra is lefordul anélkül, hogy bármi olyat is változtatni kellet volna rajta, ami a `checkingAccount` létezésére utalna. Ez azért van így, mert a fordítók ostobán hisznek nekünk, ha azt mondjuk – `static_cast`-tal, hogy `*p` igazából egy `SavingsAccount*`-ra mutat. Mi vagyunk a főnökök! Ez az Egyes Számú Karbantartási Rémálom. A Kettes Számú Karbantartási Rémálom eléréséhez kísértésbe kell esnünk, hogy ezt a problémát megoldjuk, ami általában valami ilyen kód írásába torkollik:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
   if (*p SavingsAccount-ra mutat)
      static_cast<SavingsAccount*>(*p)->creditInterest();
   else
      static_cast<CheckingAccount*>(*p)->creditInterest();
}
```

Ha bármikor olyan kód írásán kapjuk magunkat, amely a „ha az objektum T1 típusú, akkor ez és ez történjen, de ha T2 típusú, akkor valami más" alakot ölti, akkor adjunk magunknak egy pofont. Ez nem „a" C++-féle módszer. C-ben, vagy Pascalban ez a stratégia teljesen logikus, igen, C++-ban viszont nem. C++-ban virtuális függvényeket használunk.

Ne felejtsük el, hogy virtuális függvényeknél a *fordítóprogramok* felelősek azért, hogy – az éppen használt objektum típusától függően – a megfelelő függvény hívódjon meg. Ne szemeteljük tele a kódunkat feltételekkel vagy switch utasításokkal: hagyjuk, hogy ezt a munkát a fordítók végezzék el helyettünk! Így:

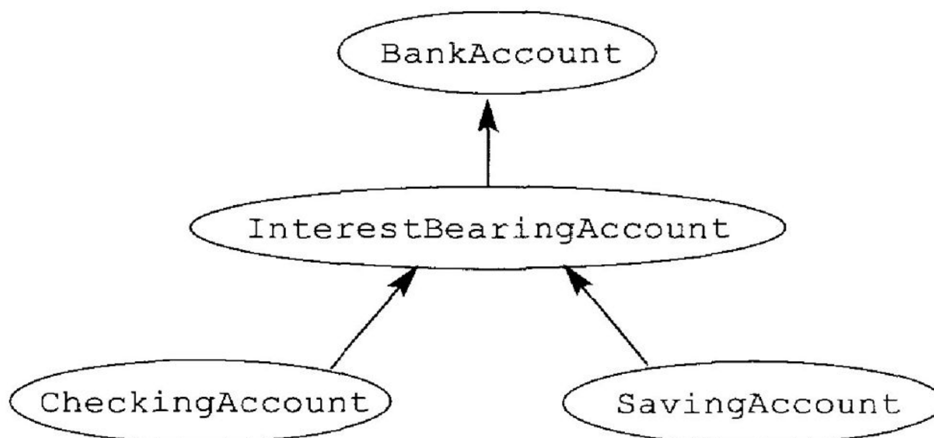
```
class BankAccount { ... };     // Mint fent.
// Új osztály a kamatozó számlák reprezentálására.
```

```
class InterestBearingAccount: public BankAccount {
public:
   virtual void creditInterest() = 0;
   ...
};

class SavingsAccount: public InterestBearingAccount {
   ...                          // Mint fent.
};

class CheckingAccount: public InterestBearingAccount {
   ...                          // Mint fent.
};
```

Szemléletesen ez így néz ki:



Mivel mind a betéti, mind a folyószámlák kamatoznak, természetes módon merül fel bennünk az igény arra, hogy ezt a közös vonást egy közös bázisosztályba emeljük át. Azzal a feltételezéssel élve azonban, hogy a banknak nem minden számlája kamatozik szükségszerűen (tapasztalataim alapján ennek a feltételezésnek van valóságalapja), nem tehetjük e viselkedést a `BankAccount` osztályba. Ezért bevezetjük a `BankAccount` egy új alosztályát, amelyet `InterestBearingAccount`-nak nevezünk el, és megoldjuk, hogy a `SavingsAccount` és a `CheckingAccount` ebből származzon.

A betéti számlák és a folyószámlák kamatozásának tényét az `InterestBearing-Account` tisztán virtuális *(pure virtual)* `creditInterest` függvénye fejezi ki, amelyet a `SavingsAccount` és `CheckingAccount` nevű alosztályok feltételezhetően átdefiniálnak.

Az új osztályhierarchia segítségével a ciklust a következőképpen tudjuk átírni:

```
// Jobb, de még mindig nem tökéletes.
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
   static_cast<InterestBearingAccount*>(*p)->creditInterest();
}
```

Annak ellenére, hogy ebben a ciklusban még mindig van egy fránya konverzió, sokkal életrevalóbb, mint a korábbi változat, ugyanis akkor is jól működik, ha az alkalmazásunk újabb `InterestBearingAccount` alosztályokkal bővül.

Ahhoz, hogy teljesen megszabaduljunk a konverziótól, további változtatásokat kell végrehajtanunk az osztálytervünkön. Az egyik megközelítésben ez a számlalista specifikációjának szigorításával érhető el. Ha ugyanis `BankAccount` objektumok listája helyett `InterestBearingAccount` objektumokból álló listánk lenne, akkor minden flottul működne:

```
// A bank összes kamatozó számlája.
list<InterestBearingAccount*> allIBAccounts;
// Egy ciklus, amely lefordul és működik, most és mindörökké.
for (list<InterestBearingAccount*>::iterator p =
        allIBAccounts.begin();
     p != allIBAccounts.end();
     ++p) {
   (*p)->creditInterest();
}
```

Ha a lista szigorítására nincs lehetőségünk, akkor logikus megoldás lehet, hogy a `creditInterest` műveletet *minden* bankszámlára érvényessé tesszük úgy, hogy a nem kamatozó számlák esetében ne csináljon semmit. Ezt ekképpen fejezhetjük ki:

```
class BankAccount {
public:
   virtual void creditInterest() {}
   ...
};
class SavingsAccount: public BankAccount { ... };
class CheckingAccount: public BankAccount { ... };
list<BankAccount*> allAccounts;
// Figyeled? nincs konverzió.
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
   (*p)->creditInterest();
}
```

Vegyük észre, hogy a `BankAccount::creditInterest` virtuális függvény gondoskodik egy üres alapértelmezett implementációról *(default implementation)*. Ezzel kényelmesen kifejezhető az, hogy igaz, a függvény alapértelmezésben nem csinál semmit, mégis okozhat előre nem látható nehézségeket pusztán létezése jogán. A miérttel összefüggő kulisszatitkok és a veszély kiküszöbölésének mikéntje a 36. jó tanácsban található. Azt is vegyük észre, hogy a `creditInterest` (implicit módon) inline függvény. Nincs ezzel semmi baj, de mivel virtuális is, az inline direktíva valószínűleg észrevétlen marad. A 33. jó tanács elmagyarázza az okokat.

Amint láthattuk, a lefelé konvertálás többféle módon is kiküszöbölhető. A legjobb módszer az, ha ezeket a konverziókat virtuális függvények hívására cseréljük, az egyes virtuális függvényeket pedig például üres törzzsel írjuk meg minden olyan osztályra, amelyre nem igazán érvényesek. Egy másik lehetőség az, hogy megszigorítjuk a típusaink kiosztását úgy, hogy ne jöjjön létre többértelműség a pointer deklarált típusa és a tényleges – általunk ismert – pointertípus között. Bármennyi erőfeszítést is igényel a lefelé konvertálás elkerülése, megéri a munkát, mert a lefelé konvertálás csúnya, és hibalehetőségeket rejt magában, ráadásul olyan kódot eredményez, amely nehezen érthető, nehezen fejleszthető és nehezen karbantartható.

Az előbb az igazat és csakis az igazat írtam. Nem ez a teljes igazság azonban. Vannak esetek, amikor *kifejezetten* a lefelé konvertálást kell alkalmaznunk.

Tegyük fel például, hogy a jó tanács elején bemutatott helyzettel kerültünk szembe, azaz az `allAccounts` `BankAccount` pointereket tartalmaz, a `creditInterest` csak a `SavingsAccount` objektumokra van definiálva, és egy olyan ciklust kell írnunk, amely minden számlán jóváírja a kamatot! Tegyük fel továbbá azt is, hogy ezek a dolgok nem a mi ellenőrzésünk alatt állnak, azaz nem változtathatjuk meg a `BankAccount`, a `SavingsAccount`, vagy az `allAccounts` definícióját! (Például olyan esetben, ha egy számunkra csak olvasható könyvtárban vannak definiálva.) Ha ez a helyzet állna elő, akkor lefelé konvertálást *kellene* alkalmaznunk, bármennyire irtóznánk még a gondolatától is.

Van azért erre a korábban látott nyers konverziónál jobb megoldás is. Ezt a jobb módszert „biztonságos lefelé konvertálásnak *(safe downcasting)*" hívják, és a C++ `dynamic_cast` operátorán keresztül implementálható. Amikor a `dynamic_cast`-ot egy pointerre alkalmazzuk, az kísérletet tesz a konvertálásra, és ha sikerrel jár (azaz, ha a pointer dinamikus típusa (lásd a 38. jó tanácsot) összeegyeztethető azzal a típussal, amire konvertáljuk), akkor egy új típusú, érvényes pointert kapunk vissza. Ha a `dynamic_cast` nem sikerül, akkor nullpointert.

Íme, a biztonságos lefelé konvertálással megtoldott bankos példa:

```
class BankAccount { ... };          // Ua., mint a jó tanács elején.

class SavingsAccount:               // Dettó.
   public BankAccount { ... };
class CheckingAccount:              // Dettó ez is.
   public BankAccount { ... };

list<BankAccount*> allAccounts;     // Ennek ismerősnek kell tűnnie
void error(const string& msg);      // hibakezelő függvény,
                                    // lásd lejjebb.

// Na, így legalább a konverzió biztonságos...
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
   // Megpróbáljuk biztonságosan lefelé konvertálni a *p-t
   // egy SavingsAccount*-gá; a psa definíciójáról további
   // info lejjebb található.
```

```
    if (SavingsAccount *psa =
            dynamic_cast<SavingsAccount*>(*p)) {
      psa->creditInterest();
    }
    // Megpróbáljuk biztonságosan lefelé konvertálni
    // CheckingAccount*-ra.
    else if (CheckingAccount *pca =
              dynamic_cast<CheckingAccount*>(*p)) {
      pca->creditInterest();
    }
    // Hohó! — ismeretlen számlatípus.
    else {
      error("Unknown account type!");
    }
  }
```

Ez a séma messze nem az ideális, de legalább nyomon tudjuk követni azt, ha a lefelé konvertálások nem sikerülnek. Ez a `dynamic_cast` használata nélkül nem volna lehetséges. Jegyezzük meg azonban, hogy óvatosságból mindig érdemes azt az esetet is megvizsgálnunk, hogy az *összes* lefelé konvertálás sikertelen-e. Pont ez a célja az utolsó `else` ágnak a fenti kódban. Virtuális függvényeknél nem lenne szükség egy ilyen ellenőrzésre, mert minden virtuális hívásnak valamilyen *függvényt* kell végrehajtania. Ha azonban elkezdünk lefelé konvertálni, akkor nincs esélyünk. Ha valaki például egy új számlatípust ad a hierarchiához, de elfelejti frissíteni a fenti kódot, akkor az összes lefelé konvertálás sikertelen lesz. Ezért fontos kezelnünk ezt az eshetőséget is. Figyelembe véve az összes lehetőséget, nem igazán szabadna az összes lefelé konvertálásnak egyszerre sikertelennek lenni, de ha megengedjük a lefelé konvertálást, akkor jó programozókkal rossz dolgok kezdenek történni.

Nem kezdte az olvasó pánikba esve a szemüvegét keresni, amikor valami változó definíciójára hasonlító dolgot pillantott meg a fenti `if` utasítások feltétel részében? Ha igen, semmi ok az aggodalomra, nincs baj a szemével. Az ilyen változók definiálásának lehetőségét a `dynamic_cast`-tal egy időben adták hozzá a nyelvhez. Ezzel a nyelvi elemmel elegánsabb kódot lehet írni, mert a `psa`-ra vagy a `pca`-ra valójában csak akkor van szükség, ha az őket inicializáló `dynamic_cast`-ok sikerülnek, és ezzel az új szintaxissal nem kell ezeket a változókat a konverziókat tartalmazó feltételeken kívül definiálni. (A 32. jó tanácsból megtudhatjuk, hogy általában miért kerülendők a feleslegesen definiált változók.) Ha a fordítónk még nem támogatja a változók definiálásának ezt az új módját, akkor még mindig használhatjuk a régi megoldást:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
       p != allAccounts.end();
       ++p) {
   SavingsAccount *psa;         // Hagyományos definíció.
   CheckingAccount *pca;        // Hagyományos definíció.
   if (psa = dynamic_cast<SavingsAccount*>(*p)) {
      psa->creditInterest();
```

```
    }
    else if (pca = dynamic_cast<CheckingAccount*>(*p)) {
       pca->creditInterest();
    }
    else {
       error("Unknown account type!");
    }
}
```

Az igazán nagy dolgok szempontjából természetesen nem számít, hogy a psa-hoz és a pca-hoz hasonló változóinkat hol definiáljuk. Lényeges viszont a következő: az if-then-else stílusú programozás, amelyet a lefelé konvertálás alkalmazása mindig kivált, sokkal alacsonyabb rendű, mint a virtuális függvények használata, és azokra a helyzetekre kell csak tartalékolni, amikor tényleg nincs más lehetőség. Kis szerencsével sosem kell a programozás e zord és elhagyatott vidékére tévednünk.

40. jó tanács: Modellezzük a „vanegy" vagy a „keresztülimplementált" kapcsolatokat rétegekkel

A *rétegelés (layering)* az a folyamat, amelynek során az egyik osztályt úgy építjük a másik fölé, hogy a feljebb levő osztály adattagként tartalmazza az alatta levő osztály egy objektumát. Például:

```
class Address { ... };              // Lakcím.
class PhoneNumber { ... };
class Person {
public:
   ...

private:
   string name;                     // Tartalmazott objektum.
   Address address;                 // Dettó.
   PhoneNumber voiceNumber;         // Dettó.
   PhoneNumber faxNumber;           // Dettó.
};
```

Azt mondjuk, hogy ebben a példában a Person osztály a string, Address, és PhoneNumber osztályok fölé lett rétegelve, mert ezekhez a típusokhoz tartozó adattagokat tartalmaz. A *rétegelésnek* sok szinonímája van. Használatos rá a *kompozíció (composition)*, a *tartalmazás (containment)*, és a *beágyazás (embedding)* kifejezés is.

A 35. jó tanács elmagyarázza, hogy a publikus öröklődés „azegy" *(isa)* kapcsolatot jelent. Ezzel szemben a rétegelés vagy „vanegy" *(has-a)*, vagy „keresztülimplementált" *(is-implemented-in-terms-of)* relációt modellez.

A fenti Person osztály a „vanegy" kapcsolatot szemlélteti. Egy `Person` objektumnak van egy neve, egy címe, és telefon-, illetve faxszáma. Azt nem igazán mondjuk, hogy egy személy *az* egy név, vagy hogy egy személy *az* egy cím. Azt viszont már igen, hogy egy személynek *van* egy neve, *van egy* címe stb. A legtöbb embernek nem okoz gondot ez a megkülönböztetés, így aránylag ritkán van félreértés az „azegy" és a „vanegy" szerepe körül.

Az „azegy" és a „keresztül implementált" közötti különbség már kicsit problémásabb terület. Tegyük fel például, hogy egy olyan sablonra (*template)* van szükségünk, amely tetszőleges objektumok halmazait (azaz másolatoktól mentes gyűjteményeket) reprezentáló osztályokat ír le. Mivel az újrafelhasználás csodálatos dolog, és mivel bölcsen elolvastuk a 49. jó tanács áttekintését a szabvány C++ könyvtárról, ösztönösen hajlunk arra, hogy a könyvtár `set` sablonját használjuk. Végül is minek írjunk újat, ha egyszer használhatjuk a mások által már megírt, bevált sablont is?

A `set` dokumentációjában vájkálva felfedezhetünk azonban egy olyan korlátozást, amellyel az alkalmazásunk nem tud megbirkózni: egy `set` megköveteli, hogy a benne tárolt elemekre legyen egy *teljes rendezés,* azaz a halmaz minden `a` és `b` objektumpárjáról egyértelműen el lehessen dönteni, hogy `a<b`, vagy `b<a`. Sok típusnál könnyű megfelelni ennek az elvárásnak, az objektumok közötti teljes rendezés megléte pedig lehetővé teszi a `set` számára, hogy a teljesítményét tekintve vonzó garanciákat adjon. (A szabvány könyvtár teljesítményi garanciáiról a 49. jó tanácsban találhatunk további információt.) Nekünk azonban valami ennél általánosabbra van szükségünk: egy `set`-szerű osztályra, ahol az objektumoknak nem kell teljesen rendezettnek lenniük, elegendő ha a C++ szabvány színes megfogalmazásában „egyenlőségükben összehasonlíthatók" *(EqualityComparable)*, azaz ugyanahhoz a típushoz tartozó minden `a` és `b` objektumra eldönthető, hogy `a==b` fennáll-e. Ez a szerényebb követelmény jobban illeszkedik az olyan típusokhoz, amelyek például színeket reprezentálnak. A piros kisebb mint a zöld, vagy a zöld kisebb mint a piros? Úgy tűnik, muszáj lesz egy saját sablont írni az alkalmazásunk számára.

Az újrafelhasználás ettől még csodálatos dolog. Az adatszerkezetek nagy ismerőjeként tudjuk, hogy halmazok implementálására a szinte korlátlan számú lehetőség közül a láncolt listák *(linked lists)* alkalmazása az egyik legegyszerűbb. Nem fogja elhinni a kedves olvasó! A `list` sablon, amely láncolt lista osztályokat generál, ott *üldögél* a szabvány könyvtárban! Ezért aztán a(z újra)felhasználása mellett döntünk.

Pontosabban úgy döntünk, hogy a születendő `Set` sablonunk a `list`-ből fog származni. Azaz a `Set<T>` származni fog a `list<T>`-ből. Végül is, a mi implementációnkban egy `Set` objektum ténylegesen egy `list` objektum lesz. Így aztán a `Set` sablonunkat így deklaráljuk:

```
// A list hibás használata a Set-re.
template<class T>
class Set: public list<T> { ... };
```

Minden jónak és pompásnak tűnhet ezen a ponton, valami azonban teljesen el van rontva. Ahogy azt a 35. jó tanácsból megtudhatjuk, ha D „azegy" B, akkor minden, ami igaz B-re, az igaz D-re is. Egy `list` objektum tartalmazhat azonban másolatokat is, ezért aztán ha egy `list<int>`-be kétszer kerül a 3051 értéke, akkor a lista a 3051-nek

két másolatát fogja tartalmazni. Ezzel szemben a `Set`-ben nem lehetnek másolatok, ezért ha a 3051 értékét egy `Set<int>`-be kétszer tesszük bele, a halmazban az értéknek akkor is csak egy másolata lesz. Ezért aztán az, hogy a `Set` „azegy" `list`, szemenszedett hazugság, mert bizonyos dolgok, amelyek igazak a `list` objektumokra, nem teljesülnek a `Set` objektumokra.

Mivel a két osztály közötti kapcsolat nem „azegy", hibás dolog a kapcsolatot publikus öröklődéssel modellezni. Akkor járunk el helyesen, ha észrevesszük, hogy egy `Set` objektum *implementálható* egy `list` objektumon *keresztül:*

```
// A list helyes használata a Set-re.
template<class T>
class Set {
public:
   bool member(const T& item) const;
   void insert(const T& item);
   void remove(const T& item);
   int cardinality() const;
private:
   list<T> rep;                  // Egy halmaz reprezentációja.
};
```

A `Set` tagfüggvényei bátran támaszkodhatnak a `list` által már biztosított funkcionalitásra és a szabvány könyvtár más részeire, ezért aztán az implementációt megírni sem nehéz, és elolvasni sem horror:

```
template<class T>
bool Set<T>::member(const T& item) const
{ return find(rep.begin(), rep.end(), item) != rep.end(); }
template<class T>
void Set<T>::insert(const T& item)
{ if (!member(item)) rep.push_back(item); }
template<class T>
void Set<T>::remove(const T& item)
{
   list<T>::iterator it =
      find(rep.begin(), rep.end(), item);
   if (it != rep.end()) rep.erase(it);
}
template<class T>
int Set<T>::cardinality() const
{ return rep.size(); }
```

Ezek a függvények elég egyszerűek, ezért ésszerű dolog lenne őket inline függvényként megvalósítani, de persze tudom, hogy mielőtt az olvasó bármit határozottan eldöntene inline téren, szeretné átnézi a 33. jó tanács értekezését a témáról. (A fenti kódban szereplő függvények, mint például a `find`, `begin`, `end`, `push_back` stb. a szabvány

könyvtárban a `list`-hez hasonló tároló sablonok *(container templates)* alkalmazását támogató vázhoz tartoznak, amelyről a 49. jó tanácsban található áttekintés.)

Érdemes megjegyezni, hogy a `Set` osztály felülete *(interface)* nem felel meg a teljesség és minimalitás követelményének (lásd a 18. jó tanácsot). A teljességet tekintve a legnagyobb hiányosság az, hogy nincs mód a halmaz elemeinek bejárására *(iterate)*, amire pedig sok alkalmazásnak szüksége lehet (és amivel a szabvány könyvtár minden eleme szolgál – a `set`-et is beleértve). A `Set` egy másik hátránya az, hogy nem követi a szabvány könyvtár tároló osztályainak konvencióit (lásd a 49. jó tanácsot), és ez megnehezíti a könyvtár más részeiből származó előnyök kihasználását a `Set` felhasználói számára.

A `Set` felületének hiányosságai azonban nem árnyékolhatják be azt, amit a `Set` vitathatatlanul jól old meg, mégpedig a `Set` és a `list` közötti kapcsolatot. Ez a kapcsolat nem az „azegy" (bár az elején annak tűnhetett), hanem a „keresztül implementált", és ha ezt a kapcsolatot rétegeléssel sikerül megvalósítani, az már olyan teljesítmény, amire minden osztálytervező joggal lehet büszke.

Mellesleg, ha rétegeléssel kapcsolunk össze két osztályt, akkor fordítási idejű függőséget hozunk létre az osztályok között. Hogy ezzel miért kell foglalkoznunk, és hogy hogyan tudjuk csillapítani a felmerülő aggályainkat, a 34. jó tanács témája.

41. JÓ TANÁCS: TEGYÜNK KÜLÖNBSÉGET AZ ÖRÖKLÉS ÉS A SABLONOK KÖZÖTT

Gondoljuk végig az alábbi két tervezési problémát:

- Mivel a számítástechnika elkötelezett diákjai vagyunk, olyan osztályokat akarunk készíteni, amelyek objektumokat tartalmazó vermeket *(stack)* reprezentálnak. Számos egymástól különböző osztályra van szükségünk, mert minden veremnek homogénnek kell lennie, azaz csak egyetlen objektumtípust tartalmazhat. Lehet például egy osztályunk `int`-ek vermére, egy másik osztályunk `string`-ek vermére, egy harmadik `string` vermeket tartalmazó vermekre stb. Az osztály egy minimális felületének *(minimal interface)* (lásd a 18. jó tanácsot) a megvalósítása érdekel csak bennünket, ezért az alábbiakra korlátozzuk a műveleteket: verem létrehozása, verem megsemmisítése, objektum berakása *(push)* a verembe, objektum eltávolítása *(pop)* a veremből, és annak meghatározása, hogy a verem üres-e. A gyakorlás kedvéért most figyelmen kívül hagyjuk a szabvány könyvtár osztályait (köztük a `stack`-et; lásd a 49. jó tanácsot), mert nagyon szeretnénk a kód megírását mi magunk megtapasztalni. Az újrafelhasználás csodálatos dolog, de ha alaposan meg akarjuk érteni valaminek a működését, akkor legjobb, ha fejest ugrunk bele, és nem kíméljük magunkat.
- A macskafélék elkötelezett híveként macskákat reprezentáló osztályokat akarunk tervezni. Több különböző osztályra van szükségünk, mert minden macskafajta eltér egy kicsit a többitől. Mint minden objektum, a macska is létrehozható és megsemmisíthető, de – ahogy azt minden macskaszerető ember tudja – a macskák ezen kívül csak esznek és alszanak. Minden macskafajta a maga elbűvölő módján eszik és alszik azonban.

Ez a két probléma-specifikáció hasonlónak hangzik, mégis egymástól teljesen eltérő szoftvertervet eredményez. Hogy miért?

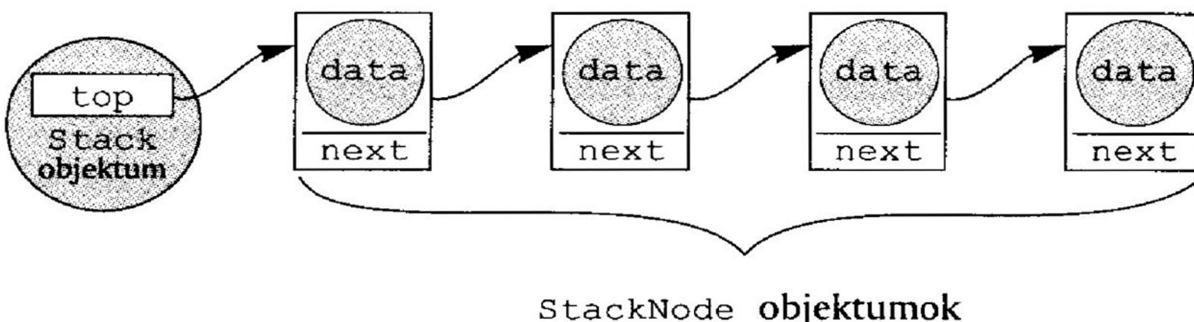
A válasz az egyes osztályok viselkedése és a kezelt objektumok *típusa* közötti összefüggéssel kapcsolatos. Mind vermeknél, mind macskáknál a típusok többféle változatával dolgozunk (`T` típusú objektumokat tartalmazó verem, illetve a `T` fajtához tartozó macska), a következő kérdést azonban mindenképp fel kell tennünk magunknak: befolyásolja-e a `T` típus az osztály *viselkedését*? Ha `T` *nincs* hatással a viselkedésre, akkor használhatunk sablont *(template)*. Ha viszont `T` *hatással van* a viselkedésre, akkor virtuális függvények *(virtual functions)* kellenek, ezért aztán az öröklődést használjuk.

Íme egy lehetőség a `Stack` osztály láncolt listás implementációjának definiálására, feltételezve, hogy a verembe rakható objektumok típusa `T`:

```
class Stack {
public:
   Stack();
   ~Stack();
   void push(const T& object);
   T pop();
   bool empty() const;  // Üres a verem?
private:
   struct StackNode {   // A láncolt lista csomópontja (node)
      T data;           // adat a csomópontban.
      StackNode *next;  // A lista következő csomópontja.

      // A StackNode konstruktor mindkét mezőt inicializálja.
      StackNode(const T& newData, StackNode *nextNode)
      : data(newData), next(nextNode) {}
   };
   StackNode *top;      // A verem teteje.
   Stack(const Stack& rhs);
                        // A másolás és értékadás.
   Stack& operator=(const Stack& rhs);
                        // Letiltása
                        // (lásd a 27. jó tanácsot).
};
```

A `Stack` objektumok ehhez hasonló adatstruktúrákat építenének:



Maga a láncolt lista `StackNode` objektumokból épül fel, de ez a `Stack` osztály egy implementációs részlete, ezért a `StackNode`-ot a `Stack` privát típusaként deklaráltuk. Vegyük észre, hogy a `StackNode`-nak van egy konstruktora, amely biztosítja, hogy minden mező megfelelően inicializálódjon. Csak mert álmunkból felébresztve is meg tudunk írni egy láncolt listát, még használhatjuk az olyan technológiai újításokat, mint például a konstruktorok.

Íme egy elfogadható első próbálkozás a `Stack` tagfüggvények implementálására. Sok implementáció-prototípushoz (és a kelleténél jóval több szoftvertermékhez) hasonlóan, itt sincs hibaellenőrzés, mert a prototípusok világában soha nem romlik el semmi.

```
Stack::Stack(): top(0) {}      // A top nullára inicializálása.
void Stack::push(const T& object)
{
   top = new StackNode(object, top);    // Rakjon egy új csomó-
                                        // pontot a lista
                                        // elejére.
T Stack::pop()
{
   StackNode *topOfStack = top;  // Jegyezze meg a
   top = top->next;              // legfelső csomópontot.
   T data = topOfStack->data;    // Jegyezze meg a csomópontban
   delete topOfStack;            // lévő adatot.
   return data;
}

Stack::~Stack()                // A verem kiürítése.
{
   while (top) {
      StackNode *toDie = top; // Egy ptr a legfelső cs.p.-ra.
      top = top->next;        // Lépjen a következő cs.p.-ra.
      delete toDie;           // Korábbi legfelső cs.p. törlése.
   }
}
bool Stack::empty() const
{ return top == 0; }
```

Nincs semmi különös ezekben az implementációkban. Az egyetlen érdekességük az, hogy minden tagfüggvényt meg tudunk úgy írni, hogy lényegében *semmit* sem tudunk `T`-ről. (Azzal a feltételezéssel élünk, hogy meg tudjuk hívni a `T` másoló konstruktorát, de ahogy a 45. jó tanácsból is kiderül, ez egy elég logikus feltételezés.) A létrehozásra, megsemmisítésre, berakásra, eltávolításra és a verem ürességének eldöntésére írt kód ugyanaz, függetlenül attól, hogy `T` micsoda. Azon feltételezés kivételével, hogy meghívhatjuk a `T` másoló konstruktorát, egy stack viselkedése semmilyen módon sem függ `T`-től. Pont ez a sablonosztály *(template class)* ismérve, azaz, hogy a viselkedés nem függ a típustól.

Ha már itt tartunk, a `Stack` osztályból annyira egyszerű sablont készíteni, hogy Mézga Géza is meg tudná csinálni:

```
template<class T> class Stack {
   ...                          // Pontosan ugyanaz, mint fent.
};
```

De térjünk át a macskákra. Miért nem akarnak a sablonok macskákkal működni?

Olvassuk újra a specifikációt, és koncentráljunk az alábbi követelményre: „minden macskafajta a maga elbűvölő módján eszik és alszik". Ez annyit jelent, hogy minden macskatípushoz *más-más viselkedést* kell implementálnunk. Nem tudunk egy olyan függvényt írni, amely minden macskát kezel, a legtöbb, amit tehetünk, hogy *meghatározzuk* egy olyan függvény *felületét*, amelyet minden macskatípusnak implementálni kell. Aha! A függvénynek *csak a felületét* pedig úgy tudjuk bevezetni, hogy egy tisztán virtuális függvényt *(pure virtual function)* deklarálunk (lásd a 36. jó tanácsot):

```
class Cat {
public:
   virtual ~Cat();                // Lásd a 14. jó tanácsot.
   virtual void eat() = 0;        // Minden macska eszik.
   virtual void sleep() = 0;      // Minden macska alszik.
};
```

A `Cat` alosztályainak – mondjuk a `Siamese`-nek és a `BritishShortHaired-Tabby`-nek természetesen újra kell definiálnia az örökölt `eat` és `sleep` függvényfelületet:

```
class Siamese: public Cat {
public:
   void eat();
   void sleep();
   ...
};
class BritishShortHairedTabby: public Cat {
public:
   void eat();
   void sleep();
   ...
};
```

Jó, most már tudjuk, hogy miért alkalmazhatók sablonok a `Stack` osztálynál, és miért nem alkalmazhatók a `Cat` osztálynál. És azt is tudjuk, hogy az öröklődés miért működik a `Cat` osztály esetében. Egyetlen kérdés maradt, méghozzá az, hogy vajon az öröklődés miért nem működik a `Stack` osztályra. Hogy ezt megértsük, próbáljunk meg gyökérosztályt *(root class)* deklarálni egy `Stack` hierarchiához, azaz egy olyan osztályt, amelyből minden további verem osztály származik:

```
class Stack {                    // Akármit tartalmazó verem.
public:
   virtual void push(const ??? object) = 0;
   virtual ??? pop() = 0;
   ...
};
```

A nehézség most már tisztán látszik. Milyen típusokat deklaráljunk a `push` és `pop` tisztán virtuális függvények számára? Ne feledjük, hogy az örökölt virtuális függvényeket minden alosztálynak az eredetivel *pontosan* megegyező paramétertípussal, és a bázisosztály deklarációkkal összhangban álló visszatérési érték típussal kell újradeklarálnia. Balszerencsénkre egy `int`-ekből álló verem `int` objektumokat akar berakni és eltávolítani, míg mondjuk egy `Cat`-ekből álló verem `Cat` objektumokkal teszi ugyanezt. Hogy tudja a `Stack` osztály a tisztán virtuális függvényeit úgy deklarálni, hogy a felhasználók int vermet és `Cat` vermet is képesek legyenek létrehozni? A kegyetlen, kőkemény igazság az, hogy sehogy, így aztán vermek létrehozására az öröklődés nem a megfelelő eszköz.

És mi van, ha trükkösek vagyunk? Túljárhatunk-e a fordítók eszén, ha generikus (`void*`) pointereket használunk? Hamar kiderül, hogy a generikus pointerek itt nem segítenek. Egyszerűen nem lehet megkerülni azt a kitételt, hogy egy virtuális függvény származtatott osztálybeli deklarációi nem mondhatnak ellent a függvény bázisosztálybeli deklarációjának. Egy másik – a sablonokból generált osztályok hatékonyságával kapcsolatos – probléma megoldásában azonban segítségünkre lehetnek a generikus pointerek. A részletek a 42. jó tanácsban találhatók.

Most, hogy már sikerült letudnunk a vermeket és a macskákat, az alábbiakban foglalhatjuk össze ennek a jó tanácsnak a tanulságait:

- Ahol az objektumok típusa *nincs hatással* az osztály függvényeinek viselkedésére, ott sablont ajánlatos használnunk osztálygyűjtemény generálására.
- Ahol az objektumok típusa *hatással van* az osztály függvényeinek viselkedésére, ott az öröklődést ajánlatos használnunk osztálygyűjtemények számára.

Elég magunkévá tenni ezt a két kis szabályt, és máris elérhető közelségbe kerül az, hogy megbirkózzunk az öröklődés és a sablonok közötti választás kényszerével.

42. jó tanács: Használjuk a privát öröklődést megfontoltan

Amint azt a 35. jó tanács szemlélteti, a C++ a publikus öröklődést „azegy" *(isa)* kapcsolatként kezeli. Egy olyan osztályhierarchia esetén, amelyben egy `Student` osztály publikusan örököl egy `Person` osztálytól, a fordítók a `Student`-eket implicit módon `Person`-okká konvertálják, ha erre van szükség egy függvényhívás sikeres végrehajtásához. Érdemes az ott szereplő példa egy részletét megismételni úgy, hogy a publikus öröklődést privátra cseréljük:

```
class Person { ... };
class Student:                      // Ez alkalommal privát
   private Person { ... };          // öröklődést használunk.

void dance(const Person& p);        // Mindenki tud táncolni.
void study(const Student& s);       // Csak a diákok tanulnak.

Person p;                           // p egy személy.
Student s;                          // s egy diák.
dance(p);                           // Jó, p egy személy.
dance(s);                           // Hiba! Egy Student nem
                                    // egy Person.
```

Jól látható, hogy a privát öröklődés nem „azegy" kapcsolatot jelent. De akkor mit? – Hát – mondhatná az olvasó –, akkor mielőtt a jelentéssel törődnénk, nézzük a viselkedést! Hogy viselkedik a privát öröklődés? – A privát öröklődést vezérlő első szabályt épp most láttuk működés közben: a publikus öröklődéssel ellentétben a fordítók általában *nem* fognak egy származtatott osztálybeli objektumot (mint amilyen a `Student`) bázisosztálybeli objektummá (jelen esetben `Person`-ná) konvertálni, ha a két osztály közötti öröklési reláció privát. Ezért nem működik a `dance` hívása az `s` objektumra. A második szabály az, hogy egy privát bázisosztályból örökölt tagok a származtatott osztály privát tagjaivá válnak akkor is, ha a bázisosztályban publikusak vagy védettek *(protected)* voltak. Ennyit a viselkedésről.

Akkor most jöjjön a jelentés. A privát öröklődés a „keresztül implementált" *(is-implemented-in-terms-of)* kapcsolatot jelenti. Ha egy `D` osztályt privát módon származtatunk egy `B` osztályból, akkor azt azért tesszük, mert fel akarjuk használni a `B` osztályban már megírt kód egy részét, nem pedig azért, mert valamilyen fogalmi kapcsolat lenne a `B` típus objektumai és a `D` típus objektumai között. A privát öröklődés mint olyan, egyszerűen csak egy implementációs technika. A 36. jó tanácsban bevezetett kifejezésekkel élve a privát öröklődés azt jelenti, hogy csak az *implementációt* kell származtatni, a felületet figyelmen kívül kell hagyni. Ha `D` privát módon örököl `B`-ből, az semmi mást nem jelent, minthogy a `D` objektumokat a `B` objektumokon keresztül implementáljuk. A privát öröklődésnek nincs értelme a *szoftvertervezés* során, csak *implementáláskor*.

Az, hogy a privát öröklődés a „keresztül implementált" kapcsolatot jelenti, egy kicsit zavaró, hiszen – ahogy arra a 40. jó tanács rámutat – a rétegelés *(layering)* is jelentheti ugyanezt. Akkor most mi alapján válasszunk közülük? A válasz egyszerű: a rétegelést akkor használjuk, amikor csak lehet, a privát öröklődést pedig akkor, ha muszáj. Hogy mikor muszáj? Amikor a védett tagok *(protected members)* és/vagy a virtuális függvények *(virtual functions)* is képbe kerülnek – de erről mindjárt többet is megtudhatunk.

A 41. jó tanács egy olyan `Stack` sablon *(template)* írására mutat be példát, amely egymástól eltérő típusú objektumokat tartalmazó osztályokat generál. Ha szükséges, most nyugodtan vissza lehet lapozni ehhez a jó tanácshoz. A sablon az egyik leghasznosabb eszköz C++-ban, de amint elkezdjük rendszeresen használni, rájövünk, hogy ha egy sablont tucatszor példányosítunk, akkor valószínűleg a *kódját* is tucatszor példányosítjuk. A `Stack` sablon esetében például a `Stack<int>` tagfüggvényeit megvalósító kód teljesen különválik attól a kódtól, amely a `Stack<double>` tagfüggvényeit való-

sítja meg. Az ilyen kódismétlés néha elkerülhetetlen, de akkor is elő szokott fordulni, ha a sablon függvényei akár osztozhatnának is ezen a kódon. Az ebből származó objektumkód-növekedésnek külön neve is van angolszász nyelvterületeken: sablon miatti *code bloat (kódduzzadás)*. Ha ilyen jelenséggel van dolgunk, az nem jó.

Az osztályok bizonyos fajtáinál generikus pointereket használhatunk a kódduzzadás elkerülésére. Ez a megközelítés azoknál az osztályoknál alkalmazható, amelyek objektumok helyett *pointereket* tárolnak és implementációjuk az alábbiak szerint történik:

- Létrehozunk egy önálló osztályt, amely objektumokra mutató `void*` pointereket tárol.
- Létrehozunk olyan további osztályokat is, amelyek egyetlen célja az erős típusosság kikényszerítése. Ezek az osztályok mind az első lépésben létrehozott generikus osztályt használják a tényleges munka elvégzésekor.

Íme egy példa, amely a 41. jó tanácsban szereplő nem sablon `Stack` osztályt használja annyi változtatással, hogy most objektumok helyett generikus pointereket tárol:

```
class GenericStack {
public:
   GenericStack();
   ~GenericStack();
   void push(void *object);
   void * pop();
   bool empty() const;

private:
   struct StackNode {
      void *data;             // Adat ebben a csomópontban.
      StackNode *next;        //(node) a lista
                              // következő cs.pontja.
      StackNode(void *newData, StackNode *nextNode)
      : data(newData), next(nextNode) {}
   };
   StackNode *top;                              // A verem teteje.

   GenericStack(const GenericStack& rhs); // Másolás és érték-
   GenericStack&                                // adás letiltása
      operator=(const GenericStack& rhs); // (lásd a 27. jó
                                                // tanácsot).
};
```

Mivel ez az osztály objektumok helyett pointereket tárol, elképzelhető, hogy egy objektumra több verem *(stack)* is mutat (azaz az objektum több verembe is bekerült). Ezért aztán döntő fontosságú, hogy az osztály destruktora és a `pop` *ne szabadítsa fel* egyetlen megsemmisítendő `StackNode` objektum `data` pointerét se, noha magát a

StackNode objektumot továbbra is fel kell szabadítania. Végül is a StackNode objektumok a GenericStack osztályon belül jönnek létre, így aztán ott is kell megsemmisíteni őket. Ennek eredményeképpen a 41. jó tanács Stack osztályának implementációja szinte teljesen jó a GenericStack osztály számára is. Csak annyi változtatásra van szükség, hogy a T-t void*-ra cseréljük.

Önmagában a GenericStack osztály kevés haszonnal kecsegtet, elég könnyű rosszul használni. Előfordulhat például, hogy egy felhasználó egy Cat objektumra mutató pointert tévedésből olyan verembe tesz, amely csak int-ekre mutató pointerek számára készült, a fordítók pedig vígan elfogadják. Végül is void* paraméterek esetén az egyik pointer olyan, mint a másik.

Ha vissza akarjuk szerezni azt a típushelyességet, amelyhez már hozzászoktunk, *felületosztályokat (interface classes)* kell létrehoznunk a GenericStack-hez az alábbi módon:

```
class IntStack {                    // Felületosztály int-ekre.
public:
   void push(int *intPtr) { s.push(intPtr); }
   int * pop() { return static_cast<int*>(s.pop()); }
   bool empty() const { return s.empty(); }
private:
   GenericStack s;                  // Implementáció.
};
class CatStack {                    // Felületosztály macskákra.
public:
   void push(Cat *catPtr) { s.push(catPtr); }
   Cat * pop() { return static_cast<Cat*>(s.pop()); }
   bool empty() const { return s.empty(); }
private:
   GenericStack s;                  // Implementáció.
};
```

Láthatjuk, hogy az IntStack és a CatStack osztályok csak az erős típusosság kikényszerítését szolgálják. Egy IntStack-be csak int pointereket tehetünk és csak azokat szedhetünk ki belőle, míg egy CatStack-be csak Cat pointerek rakhatók be és csak azok vehetők ki belőle. Mind az IntStack, mind a CatStack a GenericStack osztályon keresztül implementált, és ez a kapcsolat rétegelésen keresztül jut kifejezésre (lásd a 40. jó tanácsot). Az IntStack és a CatStack megosztozik azoknak a GenericStack-beli függvényeknek a kódján, amelyek a viselkedésüket implementálják. Az a tény továbbá, hogy az IntStack és CatStack osztályok összes tagfüggvénye (implicit módon) inline, azt jelenti, hogy ezen felületosztályok használatának futási idejű költsége nulla, semmi, zéró.

De mi van akkor, ha a leendő felhasználók erre nem jönnek rá? Mi van, ha tévesen úgy gondolják, hogy a GenericStack használata hatékonyabb? Vagy egyszerűen csak vadságból és vakmerőségből arra jutnak, hogy csak a nyúlbéláknak van szükségük a típusosság biztonsági hálójára? Mi akadályozhatja meg őket abban, hogy az IntStack-et és a CatStack-et megkerülve ne egyenesen a GenericStack-et használják, ahol

is már szabadon elkövethetik azokat a típusokat érintő hibákat, amelyek megelőzésére a C++ tervezésekor különös hangsúlyt fektettek?

Sehogy. Ezt semmi sem akadályozza meg. Bár lehet, hogy mégis van erre lehetőség.

E jó tanács elején említettem, hogy osztályok között a „keresztül implementált" kapcsolat kifejezésének egy alternatívája lehet a privát öröklődés. Itt ez a technika előnyben van a rétegeléssel szemben, mert lehetővé teszi annak a gondolatnak a kifejezését, hogy a `GenericStack` osztály általános használata nem elég biztonságos, és maga az osztály csak arra való, hogy más osztályokat implementáljunk rajta keresztül. Ezt úgy mondhatjuk ki, hogy védetté tesszük a `GenericStack` tagfüggvényeit:

```
class GenericStack {
protected:
   GenericStack();
   ~GenericStack();
   void push(void *object);
   void * pop();
   bool empty() const;
private:
   ...                        // Ua., mint fent.
};
GenericStack s;               // Hiba! a konstruktor védett.

class IntStack: private GenericStack {
public:
   void push(int *intPtr) { GenericStack::push(intPtr); }
   int * pop() { return static_cast<int*>
                           (GenericStack::pop()); }
   bool empty() const { return GenericStack::empty(); }
};
class CatStack: private GenericStack {
public:
   void push(Cat *catPtr) { GenericStack::push(catPtr); }
   Cat * pop() { return static_cast<Cat*>
                           (GenericStack::pop()); }
   bool empty() const { return GenericStack::empty(); }
};
IntStack is;                  // Jó.
CatStack cs;                  // Ez is jó.
```

A rétegeléses megközelítéshez hasonlóan a privát öröklődésen alapuló implementáció is elkerüli a kódismétlést, mert a típushelyes felületosztályok semmi mást nem tartalmaznak, csak a nekik megfelelő `GenericStack` függvények inline hívásait.

Elég ügyes húzás típushelyes felületeket készíteni a `GenericStack` osztály fölé, de szörnyen kényelmetlen kézzel begépelni az összes felületosztályt. Szerencsére erre nincs szükség. Sablonokkal automatikusan generálhatók. Íme egy privát öröklődést használó sablon típushelyes veremfelületek generálására:

```
template<class T>
class Stack: private GenericStack {

public:
   void push(T *objectPtr) { GenericStack::push(objectPtr); }
   T * pop() { return static_cast<T*>(GenericStack::pop()); }
   bool empty() const { return GenericStack::empty(); }
};
```

Ez a kód bámulatos, bár ezt első ránézésre nem könnyű észrevenni. A sablon miatt a fordítók automatikusan annyi felületosztályt generálnak, amennyire csak szükség van. Mivel ezek az osztályok típushelyesek, a felhasználói típusokat érintő hibák fordítási időben észlelhetők. Mivel a `GenericStack` tagfüggvényei védettek, a felületosztályok pedig privát bázisosztályukként használják, a felhasználók nem tudják megkerülni a felületosztályokat. Mivel a felületosztályok összes tagfüggvénye (implicit módon) `inline`-ként van deklarálva, a típushelyes osztályok használata nem okoz futási idejű költséget; a generált kód pontosan ugyanaz, mintha a felhasználó közvetlenül a `GenericStack`-et használta volna a programjában (feltéve, hogy a fordító figyelembe veszi az inline kérést – lásd a 33. jó tanácsot). És mivel a `GenericStack` osztály `void*` pointereket használ, a vermeket kezelő kódnak csak egy másolatával kell számolnunk, nem számít, hogy ténylegesen hány különböző típusú vermet használunk a programban. Röviden összefoglalva, ez a konstrukció olyan kódhoz vezet, amely egyszerre maximálisan hatékony és maximálisan típushelyes. Elég nehéz lenne ennél többet elérni.

Könyvem egyik alapgondolata az, hogy a C++ elemei csodálatos módon kapcsolódnak egymásba. Ez a példa – remélem, egyetértünk – tényleg csodálatos.

A példából levonható legfőbb tanulság az, hogy ugyanezt rétegeléssel nem tudtuk volna elérni. Csak öröklődésen keresztül érhetők el a védett tagok, és a virtuális függvényeket is csak az öröklődés használatával tudjuk átdefiniálni. (A 43. jó tanácsban példát láthatunk arra, hogy a virtuális függvények jelenléte hogyan mozdítja elő a privát öröklődés használatát.) A védett tagok és a virtuális függvények létezése okán néha a privát öröklődés az egyetlen kivitelezhető megoldás a „keresztül implementált" kapcsolat osztályok közötti kifejezésére. Végeredményben azt mondhatjuk, hogy használjuk bátran a privát öröklődést, amikor a rendelkezésre álló implementációs technikák közül az a legkézenfekvőbb. Általánosságban ugyanakkor a rétegelés a kívánatosabb módszer, ezért ahol csak lehet, használjuk inkább azt.

43. jó tanács: Használjuk a többszörös öröklődést megfontoltan

Attól függően, hogy épp ki beszél róla, a többszörös öröklődés *(multiple inheritance)* vagy isteni kinyilatkoztatás, vagy magának az ördögnek a műve. Támogatói a való világ problémáinak természetes modellezéséhez elengedhetetlen eszközként üdvözlik, míg kritikusai azzal érvelnek, hogy lassú, nehezen implementálható és semmivel sem hatékonyabb, mint az egyszeres öröklődés *(single inheritance)*. Elég nyugtalanító módon még

az objektumorientált programnyelvek világa is megosztott ebben a kérdésben: a C++, az Eiffel és a Common LISP Object System (CLOS) megengedi a többszörös öröklődést; a Smalltalk, az Objective C, és az Object Pascal nem; a Java pedig csak egy egyszerűsített formáját támogatja. Akkor most szegény, küzdelmes életű programozók kinek higgyenek?

Mielőtt bármiben is hinni kezdenénk, tisztáznunk kell a tényeket. Az egyetlen, megkérdőjelezhetetlen tény a C++ többszörös öröklődésével kapcsolatban az, hogy olyan bonyodalmaknak a Pandora szelencéjét nyitja ki, amelyek egyszeres öröklődés esetén egyszerűen nem léteznek. Ezek közül a bonyodalmak közül kiemelkedik a többértelműség *(ambiguity)* (lásd a 26. jó tanácsot). Ha egy származtatott osztály egy tagnevet egynél több bázisosztályból is megörököl, akkor az arra a névre hivatkozó összes referencia többértelmű lesz, ezért mindig explicit módon ki kell fejeznünk, hogy melyik tagra gondolunk. Íme egy példa, amely az *ARM (Annotated Reference Manual* – lásd az 50. jó tanácsot) egy idevágó részletén alapul:

```
class Lottery {
public:
   virtual int draw();
   ...
};

class GraphicalObject {
public:
   virtual int draw();
   ...
};

class LotterySimulation: public Lottery,
                         public GraphicalObject {
   ...                        // Nem deklarál draw-t.
};
LotterySimulation *pls = new LotterySimulation;
pls->draw();                  // Hiba! — Többértelmű.
pls->Lottery::draw();         // Jó.
pls->GraphicalObject::draw(); // Jó.
```

Ez így elég esetlennek néz ki, de legalább működik. Balszerencsénkre az esetlenséget nehéz eltüntetni belőle. A többértelműség még akkor is megmaradna, ha az örökölt `draw` függvények egyike privát – és így hozzáférhetetlen – lenne. (Ennek megvan a maga oka, de mivel a 26. jó tanácsban a teljes magyarázat megtalálható, itt most nem ismétlem meg.)

A tagok explicit minősítése nemcsak, hogy esetlen, de határokat is szab. Ha egy virtuális függvényt explicit módon osztálynévvel minősítünk, akkor az a függvény többé már nem viselkedik virtuálisként. Ehelyett a meghívott függvény pontosan az általunk megadott függvény lesz akkor is, ha az objektum, amelyre meghívjuk, egy származtatott osztályhoz tartozik:

```
class SpecialLotterySimulation: public LotterySimulation {
public:
   virtual int draw();
   ...
};
pls = new SpecialLotterySimulation;
pls->draw();                       // Hiba! — Még mindig többértelmű.
pls->Lottery::draw();              // A Lottery::draw-t hívja.
pls->GraphicalObject::draw();      // A GraphicalObject::draw-t
                                   // hívja.
```

Vegyük észre, hogy ebben az esetben annak ellenére, hogy a `pls` egy `SpecialLotterySimulation` objektumra mutat, az osztályban definiált `draw` függvényt (a lefelé konvertálástól eltekintve – lásd a 39. jó tanácsot) sehogy sem lehet meghívni.

És ez még nem minden! A `Lottery` és a `GraphicalObject` osztály `draw` függvényeit azért deklaráltuk virtuálisnak, hogy az alosztályok átdefiniálhassák őket (lásd a 36. jó tanácsot). De mi van akkor, ha a `LotterySimulation` szeretné *mindkettőt* átdefiniálni? A kegyetlen igazság az, hogy ezt nem teheti meg, mert egy osztályban csak egy olyan `draw` nevű függvény lehet, amelynek nincs paramétere. (Van egy kivétel ez alól a szabály alól, méghozzá az az eset, amikor az egyik függvény `const`, a másik pedig nem az – lásd a 21. jó tanácsot.)

Eljött egyszer a pillanat, amikor ez a nehézség már elég komoly problémának számított ahhoz, hogy indokolttá tegye a nyelv megváltoztatását. Az *ARM* foglalkozik az örökölt virtuális függvények „átnevezésének" lehetőségével, ám időközben kiderült, hogy a probléma megkerülhető egy új osztálypár bevezetésével:

```
class AuxLottery: public Lottery {
public:
   virtual int lotteryDraw() = 0;
   virtual int draw() { return lotteryDraw(); }
};

class AuxGraphicalObject: public GraphicalObject {
public:
   virtual int graphicalObjectDraw() = 0;
   virtual int draw() { return graphicalObjectDraw(); }
};

class LotterySimulation: public AuxLottery,
                         public AuxGraphicalObject {
public:
   virtual int lotteryDraw();
   virtual int graphicalObjectDraw();
   ...
};
```

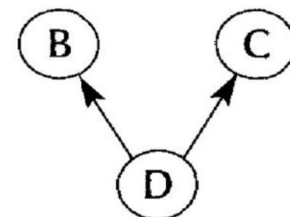
Mindkét új osztály – az `AuxLottery` és az `AuxGraphicalObject` – lényegében új nevet deklarál az általuk örökölt `draw` függvénynek. Ez az új név tisztán virtuális *(pure virtual)* függvényformát ölt, ebben az esetben a `lotteryDraw`-t és a `graphicalObjectDraw`-t. Ezek a függvények azért tisztán virtuálisak, hogy a konkrét alosztályoknak muszáj legyen őket átdefiniálni. Ráadásul mindegyik osztály úgy definiálja át az örökölt `draw` függvényt, hogy az hívja meg az új tisztán virtuális függvényt. Ennek következtében az egyetlen, többértelmű `draw` nevet sikerült lényegében két egyértelmű, de működését tekintve egyenértékű névre bontanunk ebben az osztályhierarchiában, a `lotteryDraw`-ra és a `graphicalObjectDraw`-ra:

```
LotterySimulation *pls = new LotterySimulation;
Lottery *pl = pls;
GraphicalObject *pgo = pls;
// Ez a LotterySimulation::lotteryDraw-t hívja meg.
pl->draw();
// Ez a LotterySimulation::graphicalObjectDraw-t hívja meg.
pgo->draw();
```

Jegyezzük meg jól ezt a stratégiát, amely nincs híján a tisztán virtuális, a virtuális és az inline függvények (lásd a 33. jó tanácsot) leleményes alkalmazásának. Először is egy olyan problémát old meg, amellyel egy nap mindannyian szembe kerülhetünk. Másodszor, nem hagyja, hogy megfeledkezzünk azokról a bonyodalmakról, amelyek a többszörös öröklődés jelenlétében felmerülhetnek. Igen, ez az eljárás működik, de akarunk-e kényszerűségből új osztályokat csak azért létrehozni, mert újra akarunk definiálni egy virtuális függvényt? Ennek a hierarchiának a helyes működéséhez alapvető fontosságú az `AuxLottery` és az `AuxGraphicalObject` osztály, de sem a problématérben, sem az implementációs térben nincs nekik megfeleltethető absztrakció. Egyszerűen csak implementációs eszközök, semmi mások. Azt már tudjuk, hogy a jó szoftver „eszközfüggetlen". Ez az alapelv itt is érvényes.

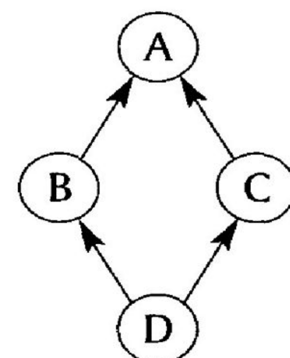
A többértelműség problémája, bármennyire is érdekes dolog, csak a jéghegy csúcsát jelenti, ha a többszörös öröklődéssel flörtölünk. Egy következő probléma alapjául az a tapasztalati megfigyelés szolgál, hogy egy eredetileg ilyen kinézetű öröklődési hierarchia:

```
class B { ... };
class C { ... };
class D: public B, public C { ... };
```



aggasztó hajlandóságot mutat arra, hogy ilyenné váljon:

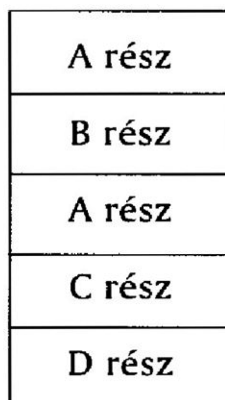
```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```



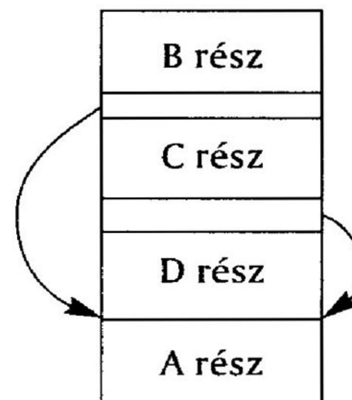
Azon lehet vitatkozni, hogy a gyémántok a lányok legjobb barátai-e vagy sem[15], az azonban bizonyos, hogy a fentihez hasonló gyémántalakú öröklődési hierarchia nem valami barátságos. Amikor egy ilyen hierarchiát létrehozunk, egyből szembesülünk azzal a kérdéssel, hogy A-t virtuális bázisosztállyá tegyük-e, azaz, hogy A-ból virtuálisan származtassunk-e. A gyakorlatban a válasz szinte mindig igenlő, csak nagyon ritkán akarjuk azt, hogy egy D típusú objektum az A adattagjainak több példányát is tartalmazza. Ezt az igazságot felismerve a fenti B és C osztályok az A-t virtuális bázisosztálynak deklarálják.

B és C definiálásakor sajnos előfordulhat, hogy nem tudjuk előre, lesz-e majd olyan osztály, amely szeretne mindkettőből örökölni, de a helyes definiálásukhoz ezt nem is kell tudnunk. Osztálytervezőként azonban ez szörnyű bizonytalanságba taszít minket. Ha *nem* deklaráljuk A-t a B és a C virtuális bázisosztályának, akkor annak, aki később tervez egy D-t, szüksége lehet a B és a C definíciójának módosítására ahhoz, hogy hatékonyan tudja őket használni. Ez azonban sok esetben elfogadhatatlan megoldás, gyakran azért, mert az A, B, és C osztályok definíciója csak olvasható. Jó példa erre az, ha A, B, és C egy könyvtár része, D pedig a könyvtár egy kliensprogramjában található.

Másrészről, ha A-t *mégis* a B és C virtuális bázisaként deklaráljuk, akkor hely és idő tekintetében rendszerint további költségeket kényszerítünk az osztályok felhasználóira. Ez azért van, mert a virtuális bázisosztályok implementálása általában objektumokra mutató *pointerekkel* történik, nem magukkal az objektumokkal. Mondanom sem kell, hogy az objektumok memóriabeli elhelyezkedése fordítófüggő, de ettől még az olyan D típusú objektum által használt memória, amelynél az A egy nemvirtuális bázis, általában memóriahelyek egybefüggő sorozata lesz. Egy olyan D típusú objektum által használt memória viszont, amelynél az A egy virtuális bázis, néha olyan memóriahelyek egybefüggő sorozata lesz, amelyből kettő a virtuális bázisosztály adattagjait tartalmazó memóriahelyekre mutató pointereket tartalmaz:



egy A nemvirtuális bázisosztályra épülő D objektum szokásos memóriabeli elhelyezkedése



egy A virtuális bázisosztályra épülő D objektum memóriabeli elhelyezkedése bizonyos fordítóknál

A virtuális öröklődés használatát általában még azok a fordítók is megbosszulják tárhely tekintetében, amelyek nem ezt a szokatlan implementációs stratégiát alkalmazzák.

[15] Utalás Jule Styne és Leo Robin „Gentlemen Prefer Blondes" című musicaljének nagy sikerű betétdalára: „Diamonds Are A Girl's Best Friend". – A ford.

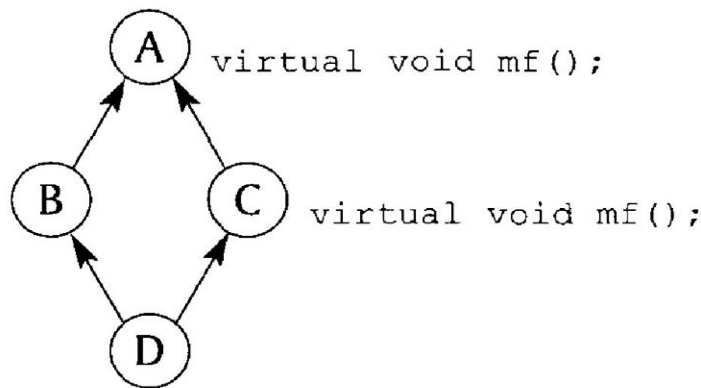
E meggondolások tükrében úgy tűnhet, hogy akkor tud egy könyvtár tervezője hatékonyan osztályokat tervezni a többszörös öröklődés jelenlétében, ha a jövőbe lát. Mivel a józan paraszti ész egyre ritkább árucikk manapság, rosszul tennénk, ha túlságosan nagy mértékben építenénk egy olyan nyelvi elemre, amely nemcsak azt kívánja meg a tervezőktől, hogy előre lássák a jövőbeni szükségleteket, hanem azt is, hogy mindjárt próféták legyenek.

Természetesen ugyanezt mondhatnánk a bázisosztálybeli virtuális és nemvirtuális függvények közötti választásról is, van azonban egy döntő különbség. A 36. jó tanács elmagyarázza, hogy a virtuális függvénynek van egy jól definiált magasszintű jelentése, amely eltér a nemvirtuális függvény ugyanilyen jól definiált magasszintű jelentésétől, így tehát lehet választani a kettő közül attól függően, hogy mit akarunk kommunikálni az alosztályok írói felé. Annak a döntésnek azonban, hogy egy bázisosztály virtuális legyen-e vagy sem, már nem létezik jól definiált magasszintű jelentése. Inkább alapul a teljes öröklődési hierarchia struktúráján, és mint ilyen döntés, egészen addig nem hozható meg, amíg a teljes struktúra nem ismeretes. Ha az osztályunk helyes definiálásához pontosan kell tudnunk, hogyan fogják azt felhasználni, akkor nagyon nehéz lesz hatékony osztályokat terveznünk.

Ha egyszer túljutottunk a többértelműség problémáján és azt a kérdést is eldöntöttük, hogy a bázisosztályainkból virtuális legyen-e az öröklődés, még mindig várnak ránk bonyodalmak. Anélkül, hogy ebbe nagyon mélyen belemennénk, nézzünk két olyan problémás területet, amelyet nem árt szemmel tartanunk:

- **Konstruktorparaméterek átadása virtuális bázisosztályoknak.** Nemvirtuális öröklődés során a bázisosztály konstruktorának paramétereit a bázisosztályból közvetlenül származtatott osztályok konstruktorának taginicializáló listájában *(member initialization list)* kell megadni. Mivel az egyszeres öröklődésű hierarchiáknak nemvirtuális bázisosztályokra van csak szükségük, a paraméterek, a dolgok természetes módján, felfelé adódnak az öröklődési hierarchiában: a hierarchia *n.* szintjének osztályai az *n-1.* szinten található osztályoknak adják át a paramétereket. Egy virtuális bázisosztály konstruktorainak paraméterei viszont azoknak az osztályoknak a taginicializáló listáiban vannak megadva, amelyek a bázistól számított *legtávolabbi* származtatott osztályok. Ennek eredményeképp a virtuális bázis és az azt inicializáló osztály tetszőleges távolságra lehet egymástól az öröklődési gráfban, az inicializálást végző osztály pedig meg is változhat, ahogy új osztályok kerülnek a hierarchiába. (Jó megoldás erre a problémára az, ha elérjük, hogy ne kelljen konstruktorparamétereket átadni a virtuális bázisoknak. Legkönnyebben úgy, ha nem teszünk adattagokat ezekbe az osztályokba. A probléma Java-beli megoldásának is ez a lényege: Java-ban a virtuális bázisosztályok (értsd: Interface-ek) nem tartalmazhatnak adatot.)

- **A virtuális függvények közötti dominancia.** Amikor végre úgy érezzük, hogy a többértelműséget már teljesen kiismertük, a szabályok hirtelen mégsem állják meg a helyüket. Tekintsünk megint egy `A`, `B`, `C` és `D` osztályokból álló gyémántalakú öröklődési hierarchiát! Tegyük fel, hogy `A` definiál egy `mf` virtuális tagfüggvényt, `C` átdefiniálja azt, `B` és `D` viszont nem:



Korábbi fejtegetéseink alapján azt várnánk, hogy az alábbi kód többértelmű lesz:

```
D *pd = new D;
pd->mf();                    // A::mf vagy C::mf?
```

A `D` objektumra melyik `mf` függvényt kell meghívni? Azt, amelyik közvetlenül `C`-ből öröklődik, vagy azt, amelyik (a `B`-n keresztül) indirekt módon az `A`-ból? A válasz az, hogy attól függ, *a `B` és a `C` hogyan származik az `A`-ból.* Jelen esetben, ha `A` egy nemvirtuális bázisa a `B`-nek és a `C`-nek, akkor a hívás többértelmű, ha viszont `A` a `B` és a `C` virtuális bázisa is, akkor azt mondjuk, hogy `mf` átdefiniálása `C`-ben *dominálja* az eredeti `A`-beli definíciót, és az `mf` függvény `pd`-n keresztüli meghívása (egyértelműen) a `C::mf` lesz. Ha leülünk és végigjátsszuk az egészet, kiderül, hogy pont ezt a viselkedést akarjuk, de igazából gyötrődés, hogy le kell ülni, és ki kell dolgozni valamit ahhoz, hogy értelmet nyerjen.

Az olvasó most már talán egyetért azzal, hogy a többszörös öröklődés bonyodalmakhoz vezethet. Talán meggyőződött már arról is, hogy épeszű ember sosem használná. Talán már készül azt javasolni a C++ szabványosítási bizottságnak, hogy a többszörös öröklődést egy az egyben vegyék ki a nyelvből, vagy legalábbis kész előterjeszteni a főnökének azt, hogy a cégnél dolgozó programozókat fizikailag meg kell akadályozni abban, hogy a többszörös öröklődést használják.

Kicsit azért talán gyors ez a tempó.

Tartsuk szem előtt, hogy a C++ tervezője a többszörös öröklődést nem úgy alkotta meg eleve, hogy nehezen lehessen használni, egyszerűen csak kiderült, hogy az összes részlet többé-kevésbé elfogadható egésszé kovácsolása szükségszerűen vonja magával a megoldások bonyolódását! Talán észrevettük, hogy a téma fenti tárgyalásában e bonyolult megoldások nagyobbik része a virtuális bázisosztályok használatával függ össze. Ha el tudjuk kerülni a virtuális bázisok alkalmazását – azaz, ha el tudjuk kerülni a halálos, gyémántalakú öröklődési gráf létrehozását –, akkor a dolgok sokkal kezelhetőbbé válnak.

A 34. jó tanács bemutatja például a *protokoll osztályt,* amely csak azért létezik, hogy deklarálja a származtatott osztályok felületét: nincs benne sem adattag, sem konstruktor, csak egy virtuális destruktor (lásd a 14. jó tanácsot) és a felületet meghatározó tisztán virtuális függvények egy halmaza. Egy protokoll `Person` osztály valahogy így nézhetne ki:

```
class Person {
public:
   virtual ~Person();
   virtual string name() const = 0;
   virtual string birthDate() const = 0;
   virtual string address() const = 0;
   virtual string nationality() const = 0;
};
```

A fenti osztály felhasználóinak `Person` pointerekben és referenciákban gondolkodva kell programozniuk, mert az absztrakt osztályok nem példányosíthatók.

A `Person` objektumként kezelhető objektumok létrehozásához a `Person` felhasználói *factory függvényeket* (lásd a 34. jó tanácsot) használnak, amellyekkel az osztály konkrét alosztályait példányosítják:

```
// factory függvény egy Person objektum létrehozására.
// Egyedi adatbázis azonosító (ID) alapján.
Person * makePerson(DatabaseID personIdentifier);

DatabaseID askUserForDatabaseID();

DatabaseID pid = askUserForDatabaseID();
Person *pp = makePerson(pid); // A Person felületét támogató
                              // objektum létrehozása.
...                           // *pp-t a Person tagfüggvényein
                              // keresztül kezeljük.
delete pp;                    // Az objektum törlése, ha már
                              // nincs rá szükség.
```

E példa alapján automatikusan felvetődik a kérdés: a `makePerson` hogyan hozza létre azokat az objektumokat, amelyekre pointert ad vissza? Egyértelmű, hogy léteznie kell a `Person`-ból származtatott olyan konkrét osztálynak, amelyet a `makePerson` példányosítani tud.

Tegyük fel, hogy ezt az osztályt `MyPerson`-nak hívják. Konkrét osztály lévén, a `MyPerson`-nak implementációt kell biztosítania azokhoz a tisztán virtuális függvényekhez, amelyeket a `Person`-ból örököl. Megírhatja ezeket teljesen nulláról, de sokkal színvonalasabb szoftvermérnöki munkát eredményezne az, ha kihasználná azoknak a már meglévő komponenseknek az előnyeit, amelyek részben, vagy akár teljes egészében elvégzik a szükséges tennivalókat. Tegyük fel például, hogy van már egy öreg, rozsdásodó adatbázis-specifikus `PersonInfo` osztályunk, amely lényegében azt biztosítja, amire a `MyPerson`-nak szüksége van:

```
class PersonInfo {
public:
   PersonInfo(DatabaseID pid);
   virtual ~PersonInfo();
```

```
    virtual const char * theName() const;
    virtual const char * theBirthDate() const;
    virtual const char * theAddress() const;
    virtual const char * theNationality() const;

    virtual const char * valueDelimOpen() const;    // Lásd
    virtual const char * valueDelimClose() const;   // lent.
    ...
};
```

Nyilvánvaló, hogy ez egy régebbi osztály, mert a tagfüggvények `const char*`-ot adnak vissza `string` objektumok helyett. De ha egy cipő illik a lábunkra, miért ne hordanánk? Az osztályban található tagfüggvények neve azt sejteti, hogy az eredmény elég kényelmes lesz.

Arra persze rájövünk, hogy a `PersonInfo` osztály azzal könnyíti meg bizonyos adatbázis mezők különféle formában történő nyomtatását, hogy minden mezőérték kezdetét és végét speciális sztringekkel határolja. Alapértelmezés szerint *(by default)* a mezőértékek nyitó és záró határolójele a szögletes zárójel, ezért aztán a „Ring-tailed Lemur" mezőérték formázása így nézne ki:

```
[Ring-tailed Lemur]
```

Abból kiindulva, hogy ezeket a zárójeleket a `PersonInfo` nem minden felhasználója követeli meg, a `valueDelimOpen` és a `valueDelimClose` virtuális függvény megengedi a származtatott osztályoknak, hogy saját nyitó és záró határolójel sztringet adjanak meg. A `PersonInfo` osztály `theName`, `theBirthDate`, `theAddress` és `theNationality` implementációi ezeket a virtuális függvényeket hívják meg a célból, hogy az általuk visszaadott értékekhez hozzátegyék a megfelelő határolójelet. A `PersonInfo::name`-mel például, a kód így néz ki:

```
const char * PersonInfo::valueDelimOpen() const
{
    return "[";        // Alapértelmezett nyitó határolójel.
}
const char * PersonInfo::valueDelimClose() const
{
    return "]";        // Alapértelmezett záró határolójel.
}
const char * PersonInfo::theName() const
{
    // puffer (buffer) foglalás a visszatérési érték számára.
    // Mivel ez statikus, automatikusan csupa nullára
    // inicializálódik.
    static char value[MAX_FORMATTED_FIELD_VALUE_LENGTH];
    // Nyitó határolójel kiírása.
    strcpy(value, valueDelimOpen());
```

```
    az objektum névmezőjének hozzáfűzése a value-ban található
    stringhez

    // Záró határolójel kiírása.
    strcat(value, valueDelimClose());
    return value;
}
```

Bele lehetne kötni a `PersonInfo::theName` megvalósításába (különösen a rögzített méretű statikus pufferbe – lásd a 23. jó tanácsot), de tegyük félre a szőrszálhasogatást, és összpontosítsunk inkább a következőre: a `theName` meghívja a `valueDelimOpen`-t a visszaadandó sztring nyitó határolójelének létrehozásához, majd létrehozza magát a név értékét, és aztán meghívja a `valueDelimClose`-t. Mivel a `valueDelimOpen` és a `valueDelimClose` virtuális függvény, a `theName` által visszaadott eredmény nemcsak a `PersonInfo`-tól függ, hanem a belőle származtatott osztályoktól is.

Mint a `MyPerson` implementálóinak, nekünk ez jó hír, mert ha gondosan elolvassuk a `Person` dokumentáció apróbetűs részeit, kiderül, hogy a `name`-nek és nővéreinek díszítés nélküli értékeket kell visszaadniuk, azaz semmilyen határolójel nem megengedett. Vagyis, ha egy személy Madagaszkárról származik, akkor az ő `nationality` függvényének „Madagascar"-t kell visszaadnia, nem „[Madagascar]"-t.

A `MyPerson` és a `PersonInfo` között annyi a kapcsolat, hogy a `PersonInfo`-nak véletlenül vannak olyan függvényei, amelyek a `MyPerson`-t könnyebben implementálhatóvá teszik. Ez minden. Sehol a láthatáron egy „azegy" *(isa)*, vagy „vanegy" *(has-a)* kapcsolat. A köztük lévő viszony tehát a „keresztül implementált" *(is-implemented-in-terms-of)*, ez pedig, mint tudjuk, kétféleképp fejezhető ki: rétegeléssel *(layering)* (lásd a 40. jó tanácsot) és privát öröklődéssel (lásd a 42. jó tanácsot). A 42. jó tanács rámutat arra, hogy általában a rétegelés a kívánatos megközelítés, de privát öröklődést kell használni akkor, ha virtuális függvények átdefiniálására van szükség. A mi esetünkben a `MyPerson`-nak át kell definiálnia a `valueDelimOpen`-t és a `valueDelimClose`-t, tehát a rétegelés nem jó, privát öröklődés kell: a `MyPerson`-nak privát módon kell származnia a `PersonInfo`-ból.

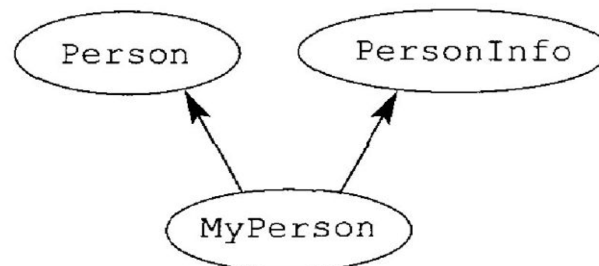
A `MyPerson`-nak implementálnia kell azonban a `Person` felületét is, ami már publikus öröklődésért kiált. Itt ésszerű dolog többszörös öröklődést alkalmazni – egy felület publikus öröklését egy implementáció privát öröklésével kapcsoljuk össze:

```
class Person {                      // Ez az osztály adja meg
public:                             // az implementálandó
   virtual ~Person();               // felületet.
   virtual string name() const = 0;
   virtual string birthDate() const = 0;
   virtual string address() const = 0;
   virtual string nationality() const = 0;
};
class DatabaseID { ... };           // Lent használjuk;
                                    // a részletei lényegtelenek
class PersonInfo {                  // Ez az osztály a Person
public:                             // felület implementálásakor
```

```
    PersonInfo(DatabaseID pid);     // használható függvényeket
    virtual ~PersonInfo();          // tartalmaz.

    virtual const char * theName() const;
    virtual const char * theBirthDate() const;
    virtual const char * theAddress() const;
    virtual const char * theNationality() const;
    virtual const char * valueDelimOpen() const;
    virtual const char * valueDelimClose() const;
    ...
};
class MyPerson: public Person,           // Figyelem! Többszörös
                private PersonInfo {     // öröklődést használunk.
public:
    MyPerson(DatabaseID pid): PersonInfo(pid) {}
    // Az örökölt virtuális határoló függvények átdefiniálása
    const char * valueDelimOpen() const { return ""; }
    const char * valueDelimClose() const { return ""; }
    // A szükséges Person tagfüggvények implementációi
    string name() const
    { return PersonInfo::theName(); }
    string birthDate() const
    { return PersonInfo::theBirthDate(); }
    string address() const
    { return PersonInfo::theAddress(); }
    string nationality() const
    { return PersonInfo::theNationality(); }
};
```

Szemléletesen ez így néz ki:



Ebből a példából jól látszik, hogy a többszörös öröklődés lehet hasznos és érthető is egyszerre, persze nem véletlen, hogy a rettegett gyémánt alakú öröklődési gráf teljesen hiányzik belőle.

A kísértésnek így is ellen kell állnunk. Beleeshetünk néha abba a csapdába, hogy többszörös öröklődéssel akkor is gyorsan helyre teszünk valamit az öröklődési hierarchiában, amikor egy alaposabb áttervezés sokkal inkább a hierarchia javára válna. Tegyük fel például, hogy rajzfilmfigurák hierarchiájával dolgozunk. Elméleti síkon bármely figura esetében van értelme a táncolásnak és az éneklésnek, bár a különböző típusú figurák ezt különbözőképp teszik. Továbbá az éneklés és a táncolás alapértelmezésben azt jelenti, hogy a rajzfilmfigura nem csinál semmit.

Mindezt C++-ban így tudjuk elmondani:

```
class CartoonCharacter {
public:
   virtual void dance() {}
   virtual void sing() {}
};
```

A virtuális függvények természetüknél fogva modellezik azt a megszorítást, hogy a táncolásnak és az éneklésnek minden `CartoonCharacter` objektum esetében van értelme. Az alapértelmezett semmittevő viselkedést az osztály ezen függvényeinek üres definíciójával fejezzük ki (lásd a 36. jó tanácsot).

Tegyük fel, hogy a szöcske egy speciális rajzfilmfigura típus amely a saját rá jellemző módján táncol és énekel:

```
class Grasshopper: public CartoonCharacter {
public:
   virtual void dance();        // A definíció máshol van.
   virtual void sing();         // A definíció máshol van.
};
```

Most tételezzük fel, hogy a `Grasshopper` osztály implementálása után úgy döntünk, szükség van egy tücsök osztályra is:

```
class Cricket: public CartoonCharacter {
public:
   virtual void dance();
   virtual void sing();
};
```

Ahogy nekiülünk a `Cricket` osztály implementálásának, észrevesszük, hogy a `Grasshopper` osztályhoz írt kód nagy része itt is felhasználható. Itt-ott azért trükköznünk kell, mert a szöcske és a tücsök táncolása és éneklése között vannak eltérések. Hirtelen eszünkbe ötlik egy okos megoldás a meglévő kód újrafelhasználására: a `Cricket` osztályt a `Grasshopper` osztályon *keresztül* fogjuk implementálni, és virtuális függvényeket fogunk használni arra, hogy a `Cricket` osztály testreszabhassa a `Grasshopper` viselkedését!

Azonnal rájövünk arra is, hogy ez a kettős követelmény – a „keresztülimplementált" kapcsolat és a virtuális függvények átdefiniálásának lehetősége – azt jelenti, hogy a `Cricket`-nek privát módon kell örökölnie a `Grasshopper`-től. Természetesen a tücsök ettől még rajzfilmfigura marad, így aztán a `Cricket` osztályt úgy definiáljuk át, hogy mind a `Grasshopper`-ből, mind a `CartoonCharacter`-ből örököljön:

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
```

```
public:
   virtual void dance();
   virtual void sing();
};
```

Ezután a `Grasshopper` osztályon elvégezzük a szükséges módosításokat. Kell például deklarálni néhány új virtuális függvényt, amelyet a `Cricket` át fog definiálni:

```
class Grasshopper: public CartoonCharacter {
public:
      virtual void dance();
      virtual void sing();
protected:
      virtual void danceCustomization1();
      virtual void danceCustomization2();
      virtual void singCustomization();
};
```

A szöcskék táncának definiálása most így néz ki:

```
void Grasshopper::dance()
{
   közös táncgyakorlatok előadása;

   danceCustomization1();

   további közös táncgyakorlatok előadása;

   danceCustomization2();

   közös záró táncgyakorlat előadása;
}
```

A szöcskék éneklése hasonlóan hangszerelt.

Persze a `Cricket` osztályt is frissítenünk kell, hogy figyelembe vehesse azokat az új virtuális függvényeket, amelyeket át kell definiálnia:

```
class Cricket:public CartoonCharacter,
              private Grasshopper {
public:
   virtual void dance() { Grasshopper::dance(); }
   virtual void sing() { Grasshopper::sing(); }
protected:
   virtual void danceCustomization1();
   virtual void danceCustomization2();
   virtual void singCustomization();
};
```

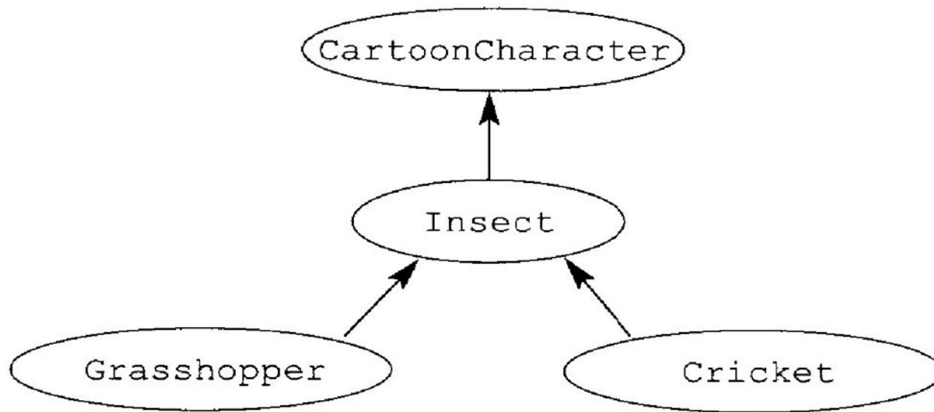
Ez jónak tűnik. Amikor egy `Cricket` objektumnak azt mondják, hogy táncoljon, végre fogja hajtani a `Grasshopper` osztállyal közös `dance` kódot, majd elvégzi a `Cricket` osztálybeli `dance` testreszabásának kódját, aztán megint a `Grasshopper::dance`-beli kóddal folytatja, és így tovább.

Ennek az osztálytervnek van azonban egy komoly hibája, mégpedig az, hogy hanyatt-homlok nekirohantunk Ockham borotvájának. Semmilyen borotvának nem jó nekirohanni, de különösen rossz a helyzet, ha ez a borotva Ockhami Vilmos tulajdona. Az ockhamizmus azt hirdeti, hogy szükségtelenül ne többszörözzük a lényeges dolgokat, és ebben az esetben a kérdéses dolgok az öröklődési kapcsolatok. Ha azt hisszük, hogy a többszörös öröklődés sokkal bonyolultabb, mint az egyszeres (és remélem, mindenki ezt hiszi), akkor a `Cricket` osztály kialakítása feleslegesen lett túlbonyolítva.

Alapjában véve az a baj, hogy *nem* igaz az az állítás, hogy a `Cricket` osztály a `Grasshopper` osztályon „keresztül implementált". Sokkal inkább az a helyzet, hogy a `Cricket` osztálynak és a `Grasshopper` osztálynak *van közös kódja*. Jelen esetben azon a kódon osztoznak, amely meghatározza a szöcskék és a tücskök közös táncolási és éneklési viselkedését.

Azt, hogy két osztályban van valami közös, nem úgy fejezzük ki, hogy az egyiket származtatjuk a másikból, hanem úgy, hogy *mindkettőt* egy közös bázisosztályból származtatjuk. A szöcskékre és tücskökre nézve a közös kód éppúgy nem tartozik a `Grasshopper` osztályhoz, mint ahogy a `Cricket` osztályhoz sem. Ez egy olyan különálló osztályhoz tartozik, amelyből mindkettő örököl, és amit pl. `Insect`-nek nevezhetnénk:

```
class CartoonCharacter { ... };
class Insect: public CartoonCharacter {
public:
   virtual void dance();        // A szöcskék és a tücskök
   virtual void sing();         // közös kódja.
protected:
   virtual void danceCustomization1() = 0;
   virtual void danceCustomization2() = 0;
   virtual void singCustomization() = 0;
};
class Grasshopper: public Insect {
protected:
   virtual void danceCustomization1();
   virtual void danceCustomization2();
   virtual void singCustomization();
};
class Cricket: public Insect {
protected:
   virtual void danceCustomization1();
   virtual void danceCustomization2();
   virtual void singCustomization();
};
```



Vegyük észre, mennyivel átláthatóbb ez az osztályterv. Csak egyszeres öröklődés van benne, ráadásul csak *publikus* öröklődést használunk. A `Grasshopper` és a `Cricket` csak a testreszabásért felelős függvényeket definiálja, a `dance` és `sing` függvényeket változatlan formában örökli az `Insect`-ből. Ockhami Vilmos büszke lenne!

Annak ellenére, hogy ez a megoldás tisztább, mint a többszörös öröklődést tartalmazó, első ránézésre alsóbbrendűnek tűnhetett. Végülis a többszörös öröklődéses megközelítéshez viszonyítva ez az egyszeres öröklődéses arhchitektúra egy teljesen új osztály bevezetését igényli, egy olyan osztályét, amelyre többszörös öröklődés használatakor nincs szükség. Miért vezetnénk be egy új osztályt, ha nem muszáj?

Máris szemtől-szembe kerültünk a többszörös öröklődés csábító természetével. Látszólag a többszörös öröklődés az egyszerűbb út. Nem vezet be új osztályokat, csak megkívánja új virtuális függvények bevezetését a `Grasshopper` osztályba, amelyeket viszont amúgy is be kellene vezetnünk valahol.

Képzeljünk el most egy programozót, aki egy nagy C++ osztálykönyvtárat tart karban. Egy olyan könyvtárat, amelyhez egy új osztályt kell hozzáadnunk, épp úgy, ahogy a `Cricket` osztályt kellett a már meglévő `CartoonCharacter/Grasshopper` hierarchiához. Ez a programozó tudja, hogy sok felhasználó dolgozik a meglévő hierarchiával, így aztán minél nagyobb a változás a könyvtárban, az annál jobban elszakad a felhasználóktól. A programozó mindenképpen a minimumra akarja csökkenteni az ilyen kimaradást. A lehetséges megoldásokon rágódva a programozó rájön, hogy ha egyetlen privát öröklődési kapcsolatot létrehoz a `Grasshopper` és a `Cricket` között, akkor a hierarchián semmit sem kell változtatni. A programozó ennek puszta gondolatára elmosolyodik, mert örül annak a lehetőségnek, hogy az alkalmazhatóságot úgy is bővítheti, hogy a program alig lesz bonyolultabb.

Most képzelje az olvasó azt, hogy ez a karbantartással foglalkozó programozó saját maga. Ne hagyja, hogy elcsábítsák!

44. jó tanács: Azt mondjuk, amit gondolunk, és értsük, amit mondunk

Ennek az öröklődésről és objektumorientált tervezésről szóló résznek a bevezetőjében hangsúlyoztam, milyen fontos megértenünk a C++ különböző objektumorientált konstrukcióinak *jelentését*. Ez egész más dolog, mint a nyelv szabályainak az ismerete. A C++ szabályai például kimondják, hogy ha a `D` osztály publikusan származik

(public inheritance) a `B` osztályból, akkor van egy szabvány típuskonverzió egy `D` pointerből egy `B` pointerbe; hogy a `B` nyilvános tagfüggvényei a `D` nyilvános tagfüggvényeiként öröklődnek stb. Ez mind igaz, de szinte használhatatlan, amikor a programtervünket megpróbáljuk C++ nyelvre átírni. Azt kell inkább megértenünk, hogy a publikus öröklődés „azegy"-et *(isa)* jelent, és hogy ha `D` publikusan származik a `B`-ből, akkor minden `D` típusú objektum „azegy" `B` típusú objektum is. Így, ha a programtervünkben valamit „azegy" kapcsolatnak szánunk, akkor tudjuk, hogy publikus öröklődést kell használnunk.

Ha azt mondjuk, amit gondolunk, az még csak fél siker. Azt megérteni, amit mondunk: ez az érem másik oldala, és legalább annyira fontos. Felelőtlenség például – ha nem egyenesen erkölcstelen –, a tagfüggvényeket könnyelműségből nemvirtuálisnak deklarálni anélkül, hogy tudnánk, ezzel bizonyos megszorításokat erőltetünk az alosztályokra. Egy nemvirtuális tagfüggvény deklarálásával valójában azt mondjuk, hogy a függvény specializáció feletti invariánst *(invariant over specialization)* reprezentál. Végzetes hiba lenne, ha ezt nem tudnánk.

A publikus öröklődés és az „azegy" reláció, vagy a nemvirtuális tagfüggvény és a specializáció feletti invariáns ekvivalenciája jól példázza, hogy a C++ bizonyos elemei hogyan felelnek meg a tervezés szintjén születő fogalmaknak. Az alábbi lista a legfontosabb ilyen leképezéseket foglalja össze:

- **Egy közös bázisosztály közös jellemzőket jelent.** Ha a `D1` és `D2` osztályok mindegyike bázisnak deklarálja a `B` osztályt, akkor `D1` és `D2` közös adattagokat és/vagy közös tagfüggvényeket örököl a `B`-ből. Lásd a 43. jó tanácsot.
- **A publikus öröklődés „azegy" kapcsolatot jelent.** Ha a `D` osztály publikusan örököl a `B` osztályból, akkor minden `D` típusú objektum egyben `B` típusú objektum is lesz, de ez fordítva nem teljesül. Lásd a 35. jó tanácsot.
- **A privát öröklődés „keresztülimplementált"** *(is-implemented-in-terms-of)* **kapcsolatot jelent.** Ha a `D` osztály privát módon örököl a `B` osztályból, akkor a `D` típusú objektumok egyszerűen a `B` típusú objektumokon keresztül implementáltak, nincs semmiféle fogalmi kapcsolat a `B` és `D` típusú objektumok között. Lásd a 42. jó tanácsot.
- **A rétegelés** *(layering)* **„vanegy" vagy „keresztülimplementált" kapcsolatot jelent.** Ha az `A` osztály tartalmaz egy `B` típusú adattagot, akkor az `A` típusú objektumoknak vagy van egy `B` típusú komponensük, vagy pedig a `B` típusú objektumokon keresztül implementáltak. Lásd a 40. jó tanácsot.

A következő leképezések csak a publikus öröklődés használata esetén érvényesek:

- **Egy tisztán virtuális függvény** *(pure virtual function)* **azt jelenti, hogy csak a függvény felülete** *(interface)* **öröklődik.** Ha a `C` osztály deklarál egy `mf` tisztán virtuális tagfüggvényt, akkor a `C` alosztályainak örökölniük kell az `mf` felületét, és a `C` konkrét alosztályainak saját implementációt is kell adniuk hozzá. Lásd a 36. jó tanácsot.
- **Egy egyszerű virtuális függvény** *(simple virtual function)* **azt jelenti, hogy a függvény felülete és egy alapértelmezett implementáció öröklődik.** Ha a `C` osztály deklarál egy `mf` egyszerű (nem tisztán) virtuális függvényt, akkor a `C` alosztályainak örökölniük kell az `mf` felületét, és örökölhetnek egy alapértelmezett implementációt is, ha úgy döntenek. Lásd a 36. jó tanácsot.

- **Egy nemvirtuális függvény azt jelenti, hogy a függvény felülete, és vele együtt egy kötelező implementáció is öröklődik.** Ha a `C` osztály deklarál egy `mf` nemvirtuális tagfüggvényt, akkor a `C` alosztályainak örökölniük kell az `mf` felületét és implementációját is. Valójában az `mf` a `C` specializációja feletti invariánst definiál. Lásd a 36. jó tanácsot.

Egyebek

Léteznek olyan, a hatékony C++ programozást érintő iránymutatások, amelyeket nehéz kategorizálni. Ezek a jó tanácsok ebben a fejezetben kaptak helyet. Ez persze nem csökkenti a fontosságukat. Ha hatékony szoftvert akarunk írni, tisztában kell lennünk azzal, hogy a fordítók mit tesznek értünk (velünk?) a tudtunk nélkül, hogyan biztosíthatjuk, hogy a nemlokális statikus objektumok inicializálódjanak használatba vétel előtt, mit várhatunk a szabvány könyvtártól, vagy hová forduljunk, ha a nyelv tervezési filozófiájának alapelveiről szeretnénk többet megtudni. Ebben az utolsó részben többek között ezekkel a kérdésekkel foglalkozom.

45. jó tanács: Tudjunk róla, milyen függvényeket ír és hív meg titokban a C++

Mikor nem üres egy üres osztály? Miután a C++ végez vele. A fordítóprogramok figyelmesek, ezért a másoló konstruktorból, az értékadó operátorból, a destruktorból és egy cím operátor *(address-of operator)* párból saját változatot deklarálnak, ha mi magunk nem tesszük ezt meg. Sőt, ha nem deklarálunk konstruktort, akkor még egy alapértelmezett konstruktort is deklarálnak nekünk. Ezek a függvények mind nyilvánosak lesznek. Más szóval, ha ezt írjuk:

```
class Empty{};
```

az ugyanaz, mintha ezt írtuk volna:

```
class Empty {
public:
    Empty();                          // Alapértelmezett konstruktor.
    Empty(const Empty& rhs);          // Másoló konstruktor.
    ~Empty();                         // Destruktor — lásd lejjebb,
                                      // hogy virtuális-e.
    Empty&
    operator=(const Empty& rhs);      // Értékadó operátor.
    Empty* operator&();               // Cím operátorok.
    const Empty* operator&() const;
};
```

Ezek a függvények csak akkor jönnek létre, ha szükség van rájuk, márpedig ehhez nem kell sok. Az alábbi kód az összes fenti függvényt előállítja:

```
const Empty e1;                   // Alapértelmezett konstruktor
                                  // és destruktor.
Empty e2(e1);                     // Másoló konstruktor.
e2 = e1;                          // Értékadó operátor.
Empty *pe2 = &e2;                 // Cím operátor (nem const).
const Empty *pe1 = &e1;           // Cím operátor (const).
```

A fordítók tehát írnak nekünk függvényeket, de vajon mit csinálnak ezek a függvények? Az alapértelmezett konstruktor és a destruktor tulajdonképpen semmit sem csinál, egyszerűen lehetővé teszik számunkra, hogy létrehozzuk és megszüntessük az osztály objektumait. (Kényelmesek implementáláshoz is, mert megfelelő helyet biztosítanak annak a kódnak, amely a kulisszák mögött végbemenő viselkedés végrehajtásáról gondoskodik – lásd a 33. jó tanácsot.) Jegyezzük meg jól, hogy a létrejött destruktor nem lesz virtuális, (lásd a 14. jó tanácsot), kivéve, ha olyan osztály számára készül, amely egy virtuális destruktort deklaráló bázisosztályból származik. Az alapértelmezett cím operá-

torok egyszerűen visszaadják az objektum címét. Ezek a függvények hatékonyan így definiálhatók:

```
inline Empty::Empty() {}
inline Empty::~Empty() {}
inline Empty * Empty::operator&() { return this; }
inline const Empty * Empty::operator&() const
{ return this; }
```

Ami a másoló konstruktort és az értékadó operátort illeti, a hivatalos szabály a következő: az alapértelmezett másoló konstruktor (értékadó operátor) az osztály nemstatikus adattagjain tagonként hajt végre másolva létrehozást (értékadást). Azaz, ha `m` egy nemstatikus, `T` típusú adattag a `C` osztályban, és `C` nem deklarál másoló konstruktort (értékadó operátort), akkor `m` másolva létrehozása (értékadása) a `T`-re definiált másoló konstruktoron (értékadó operátoron) keresztül történik, feltéve, hogy van egyáltalán ilyen. Ha nincs, akkor ez a szabály ismétlődik rekurzívan az `m` adattagjaira addig, amíg elő nem kerül egy másoló konstruktor (értékadó operátor), vagy egy beépített típus (pl. `int`, `double`, pointer stb.). A beépített típusok objektumainak a másolva létrehozása (értékadása) alapértelmezés szerint a kiindulási objektum célobjektumba történő bitenkénti másolásával megy végbe. A más osztályokból származó osztályokra az öröklődési hierarchia minden szintjén vonatkozik ez a szabály, ezért a felhasználó által definiált másoló konstruktorok és értékadó operátorok a deklarációjuk szintjén hívódnak meg.

Remélem, ez kristálytiszta.

Nézzünk egy példát, hátha mégsem az. Tekintsük a `NamedObject` sablon *(template)* definícióját, amelynek példányosított osztályai megengedik, hogy az objektumokhoz neveket rendeljünk:

```
template<class T>
class NamedObject {
public:
   NamedObject(const char *name, const T& value);
   NamedObject(const string& name, const T& value);
   ...

private:
   string nameValue;
   T objectValue;
};
```

Mivel a `NamedObject` osztályok legalább egy konstruktort deklarálnak, a fordítók nem fognak alapértelmezett konstruktort gyártani. Mivel azonban nem deklarálnak másoló konstruktort és értékadó operátort, ezeket a függvényeket (szükség esetén) a fordítók állítják elő.

Gondoljuk végig a másoló konstruktor alábbi hívását:

```
NamedObject<int> no1("Smallest Prime Number", 2);
NamedObject<int> no2(no1);    // Másoló konstruktor hívása.
```

A fordítónk által generált másoló konstruktornak a `no2.nameValue`-t és a `no2.objectValue`-t a neki megfelelő `no1.nameValue`-val és `no1.objectValue`-val kell inicializálni. A `nameValue` típusa `string`, a `string`-nek pedig van másoló konstruktora (amit a szabvány könyvtárban található `string` vizsgálatával is ellenőrizhetünk – lásd a 49. jó tanácsot), így a `no2.nameValue` a `string` másoló konstruktorának meghívásával inicializálódik úgy, hogy a `no1.nameValue`-t kapja meg paraméterként. Másrészt, a `NamedObject<int>::objectValue` típusa `int` (mert `T` ebben a sablon-példányosításban `int`), és mivel az `int`-ekhez nincs definiálva másoló konstruktor, a `no2.objectValue` a `no1.objectValue` bitjeinek átmásolásával inicializálódik.

A fordító által a `NamedObject<int>` számára generált értékadó operátor is így viselkedne, általában azonban az igaz, hogy a fordító által generált értékadó operátorok csak akkor viselkednek az itt leírtak szerint, ha az eredményül kapott kód érvényes és jó eséllyel értelme is van. Ha csak ezek egyike nem teljesül, a fordító nem fogja létrehozni az `operator=` függvényt az osztályunkban, és majd valamilyen kedves hibaüzenetet kapunk a fordítás során.

Tegyük fel például, hogy a `NamedObject` úgy van definiálva, hogy a `nameValue` egy sztringre hivatkozó *referencia*, az `objectValue` pedig egy `const T`:

```
template<class T>
class NamedObject {
public:
   // Ez a konstruktor már nem egy const nevet vár
   // paraméterként, mert a nameValue most egy nem const
   // string-re hivatkozó referencia. A char* konstruktor
   // eltűnt, mivel olyan string objektumra van szükség,
   // amelyre hivatkozni lehet.
   NamedObject(string& name, const T& value);
   ...                         // Mint fent, feltesszük, hogy
                               // nincs operator= deklarálva.

private:
   string& nameValue;          // Ez most egy referencia.
   const T objectValue;        // Ez most const.
};
```

Gondoljuk végig, hogy itt minek kellene történnie:

```
string newDog("Persephone");
string oldDog("Satch");
NamedObject<int> p(newDog, 2);    // Amikor ezt írom, a
                                  // kutyánk, Perszephoné
                                  // már majdnem két éves.
```

```
NamedObject<int> s(oldDog, 29);  // A család kutyája, Satch
                                 // (gyerekkoromból)
                                 // 29 éves lenne, ha
                                 // még élne.

p = s;                           // Mi történjen a p-beli
                                 // adattagokkal?
```

Az értékadás előtt a `p.nameValue` valamilyen `string` objektumra hivatkozik, `s.nameValue` szintén egy – bár az előbbitől különböző – `string`-re. Az értékadásnak milyen hatással kellene lennie a `p.nameValue`-ra? Az értékadás után vajon az `s.nameValue` által hivatkozott `string`-re kellene-e mutatnia a `p.nameValue`-nak, azaz módosulnia kellene-e magának a referenciának? Ha igen, akkor érintetlen területre tévedtünk, mert a C++ nem ad módot arra, hogy egy referencia másik objektumra hivatkozzon. Vagy módosítsuk talán helyette a `p.nameValue` által hivatkozott `string` objektumot, így befolyásolva a többi olyan objektumot, amelyek pointereket vagy referenciákat tartalmaznak ugyanerre a `string`-re, azaz így befolyásolva az olyan objektumokat, amelyek nem is szerepelnek ebben az értékadásban? Tényleg ezt kellene tennie a fordító által létrehozott értékadó operátornak?

Amikor a C++ ilyen talánnyal kerül szembe, megtagadja a kód lefordítását. Ha támogatni szeretnénk az értékadást egy referenciatagot tartalmazó osztályban, akkor ott nekünk kell az értékadó operátort definiálnunk. A `const` adattagokat – mint amilyen pl. az `objectValue` a fenti módosított osztályban – tartalmazó osztályok esetén is hasonlóan viselkednek a fordítók. Szabályellenes a `const` tagokat módosítani, ezért aztán a fordítók nem tudnak mit kezdeni velük egy implicit módon generált értékadó függvényben. Továbbá, a fordítók nem generálnak értékadó operátort az olyan bázisosztályból származó osztályok számára, amely bázisok a szabvány értékadó operátort `private`-nak deklarálják. Elvégre a fordító által a származtatott osztályokhoz generált értékadó operátoroknak boldogulniuk kell a bázisosztálybeli részekkel is (lásd a 16. jó tanácsot), de ennek során természetesen nem szabad meghívniuk olyan tagfüggvényt, amelyre a származtatott osztálynak nincs hívási jogosultsága.

A fordító által létrehozott függvények fenti tárgyalásából logikusan következik a kérdés: mit tegyünk, ha le akarjuk tiltani ezeknek a függvényeknek a használatát? Azaz, mi van, ha azért nem deklarálunk szándékosan mondjuk egy `operator=`-t, mert *sohasem* akarjuk megengedni objektumok értékadását az osztályunkban? A fejtörő megoldása a 27. jó tanács témája. A pointer tagok és a fordító által generált másoló konstruktorok és értékadó operátorok közötti kölcsönhatást, amely gyakran elkerüli az ember figyelmét, a 11. jó tanácsban tárgyalom.

46. JÓ TANÁCS: RÉSZESÍTSÜK ELŐNYBEN A FORDÍTÁSI ÉS LINKELÉSI IDEJŰ HIBÁKAT A FUTÁSI IDEJŰEKKEL SZEMBEN

Attól a néhány esettől eltekintve, amikor a C++ kivételt dob (például, ha elfogy a memória – lásd a 7. jó tanácsot), a futási idejű hiba *(runtime error)* fogalma éppoly idegen a C++-tól, mint a C-től. Nem észlelhető az alul- és túlcsordulás, a nullával való osztás, senki sem ellenőrzi a tömbök indexhatárainak megsértését stb. Ha a program túljut egyszer a fordítón és a linkeren, egyedül maradunk, nincs védőháló, amely megóvna minket a lehetséges következményektől. Ugyanúgy, mint egy ejtőernyős ugrásnál, vannak, akiket teljesen feldob egy ilyen helyzet, mások viszont megbénulnak a félelemtől. E filozófia hátterében meghúzódó motiváció természetesen megint a hatékonyság: futási idejű ellenőrzések nélkül a programok kisebbek és gyorsabbak.

Máshogyan is meg lehet közelíteni a dolgokat. Az olyan nyelvek, mint pl. a Smalltalk és a LISP általában nem vesznek észre olyan sokféle hibát fordítási és linkelési időben, nagyon erős futtató rendszereik vannak viszont, amelyek elkapják a végrehajtás során előforduló hibákat. A C++-tól eltérően ezek a nyelvek szinte mindig interpretáltak, és az általuk nyújtott többletrugalmasságért a végrehajtási sebesség csökkenésével kell fizetnünk.

Sose felejtsük el, hogy C++-ban programozunk. Bármennyire vonzónak is találjuk a Smalltalk/LISP filozófiát, verjük ki a fejünkből. Sokat lehetne beszélni a „C++ párt" melletti kitartásról, ami a mi esetünkben azt jelenti, hogy lemondunk a futási idejű hibákról. Amikor csak lehet, toljuk vissza a hibaellenőrzést futási időből a linkelés, vagy ideális esetben a fordítás idejére.

Egy ilyen módszer nemcsak a program mérete és a sebesség, hanem a megbízhatóság miatt is kifizetődő. Ha a programunk hibaüzenet nélkül jut át a fordításon és a linkelésen, akkor biztosak lehetünk abban, hogy nincsenek benne a fordító és a linker által felismerhető hibák, pont. (A másik lehetőség természetesen az, hogy hibás a fordító vagy a linker, de ne hagyjuk magunkat elkedvetleníteni azzal, hogy behódolunk ennek a lehetőségnek.)

A futási idejű hibákkal egészen más a helyzet. Csak azért, mert a programunk egy adott futtatás során nem ad futási idejű hibát, hogyan lehetünk biztosak abban, hogy egy ettől különböző futtatás során sem ad, amikor esetleg a dolgokat más sorrendben végezzük, vagy más adatokat használunk, vagy éppen hosszabb, vagy rövidebb ideig futtatjuk a programot? Tesztelhetjük a programunkat, amíg bele nem zöldülünk, sosem fogjuk lefedni az összes lehetőséget. Emiatt aztán egyszerűen kevésbé biztonságosan járunk el, ha a hibákat futási időben keressük, ahelyett, hogy a fordítás vagy a linkelés során kapnánk el őket.

Sokszor már azzal is elkaphatunk – egyébként futási idejű – hibát fordítási időben, ha az osztálytervünkön viszonylag kis változtatást végzünk. Ez gyakran jelenti új típusok bevezetését. Tegyük fel például, hogy olyan osztályt írunk, amely dátumokat reprezentál. Első megközelítésként ezt írhatjuk:

```
class Date {
public:
   Date(int day, int month, int year);
   ...
};
```

Ha implemetálnunk kellene ezt a konstruktort, probléma lenne például azt a vizsgálatot elvégeznünk, hogy a nap és a hónap értéke értelmes-e. Lássuk, hogyan küszöbölhetjük ki ennek a vizsgálatnak szükségességét, a hónapnak megadott érték esetében.

Egy nyilvánvaló megoldás lehet az integrális típus helyett egy enumerátor *(felsoroló típus)* használata:

```
enum Month { Jan = 1, Feb = 2, ... , Nov = 11, Dec = 12 };

class Date {
public:
   Date(int day, Month month, int year);
   ...
};
```

Sajnos ez nem segít túl sokat, mert az enumerátorokat nem kell inicializálni:

```
Month m;
Date d(22, m, 1857);            // m definiálatlan.
```

Emiatt aztán a `Date` konstruktornak továbbra is ellenőriznie kell, hogy a `month` paraméter értéke érvényes-e.

Hogy elérjünk egy olyan biztonsági szintet, ahol már mellőzhetjük a futási idejű ellenőrzéseket, a hónapok reprezentálására egy osztályt kell használnunk, és biztosítanunk kell, hogy csak érvényes hónapokat lehessen létrehozni:

```
class Month {
public:
   static const Month Jan() { return 1; }
   static const Month Feb() { return 2; }
   ...
   static const Month Dec() { return 12; }

   int asInt() const              // A kényelmes felhasználás
   { return monthNumber; }        // érdekében megengedjük egy
                                  // Month int-té konvertálását.

private:
   Month(int number): monthNumber(number) {}
   const int monthNumber;
};
```

```
class Date {
public:
   Date(int day, const Month& month, int year);
   ...
};
```

Ebben a tervben számos olyan tényező van, amelyek egymásra épülve biztosítják az adott működés megvalósulását. Először is, a `Month` konstruktor privát. A felhasználók tehát nem tudnak új hónapokat létrehozni. Csak azok érhetők el, amelyeket a `Month` statikus taggfüggvényei adnak vissza, meg ezek másolatai. Másodszor, minden `Month` objektum `const`, így nem lehet megváltoztatni. (Különben előfordulhatna, hogy nem tudunk ellenállni a kísértésnek, hogy pl. a januárból júniust csináljunk – legalábbis az északi féltekén.) Végül, csak úgy készíthetünk új `Month` objektumot, ha meghívunk egy függvényt, vagy pedig lemásolunk egy már létező `Month` objektumot (a `Month` implicit másoló konstruktorán keresztül – lásd a 45. jó tanácsot). Így már bárhol és bármikor használhatunk `Month` objektumokat, és nem kell amiatt aggódnunk, hogy véletlenül az inicializálásuk előtt használjuk őket (a 47. jó tanács elmagyarázza, hogy ezzel mi is lehet a baj.)

Ilyen osztályok mellett már majdnem lehetetlen, hogy egy felhasználó érvénytelen hónapot adjon meg. Teljesen lehetetlen lenne, ha nem létezne ez az utálatos lehetőség:

```
Month *pm;                    // Inicializálatlan ptr def.
Date d(1, *pm, 1997);         // pfuj! használjuk!
```

Ez azonban egy inicializálatlan dereferenciálását *(dereferencing)* tartalmazza, aminek a végeredménye definiálatlan. (Lásd a 3. jó tanácsot a definiálatlan viselkedés iránt viseltetett érzelmeimről.) Sajnos nem ismerek olyan módszert, amellyel az effajta eretnekséget meg lehetne előzni vagy fel lehetne deríteni. Ha viszont abból indulunk ki, hogy ez sosem fordul elő, vagy nem törődünk azzal, mit csinál a szoftverünk akkor, ha mégis megtörténik, akkor a `Date` konstruktornak nem kell vizsgálnia, hogy a `Month` paramétere értelmes-e. A konstruktornak azonban még ekkor is ellenőriznie kell a `day` paraméter érvényességét – hány napot is számlál „az szeptember, az április, az június vagy az november hava"?

A fenti `Date` példa a futási idejű ellenőrzéseket fordítási idejűekkel helyettesíti. Az olvasó bizonyára elgondolkodik azon, hogy vajon mikor használhatunk linkelési idejű ellenőrzést. Az az igazság, hogy nem túl gyakran. A C++ annak biztosítására használja a linkert, hogy minden szükséges függvény pontosan egyszer legyen definiálva (lásd a 45. jó tanácsot arról, hogy mivel jár az, ha egy függvényre „szükség" van). Ezen kívül arra is, hogy a statikus objektumok (lásd a 47. jó tanácsot) is pontosan egyszer legyenek definiálva. Nekünk is érdemes a linkert ugyanígy használni. A 27. jó tanács például bemutatja, hogy a linker által elvégzett ellenőrzésekre építve miért lehet hasznos az, ha egy explicit módon deklarált függvényt szándékosan nem definiálunk.

Azért ne essünk át a ló túlsó oldalára. Nem célszerű kiküszöbölnünk az *összes* futási idejű ellenőrzés szükségességét. Például bármely programnak, amely interaktív inputot fogad, nagy valószínűséggel ellenőriznie kell az input érvényességét. Hasonlóan, egy olyan osztályban, amely az indexhatárok ellenőrzését végző tömböket implementál (lásd

a 18. jó tanácsot), rendszerint a tömb minden elérésekor ellenőrizni kell, hogy az index a határok közé esik-e. Mindig megéri az ellenőrzéseket futási időből fordítási vagy linkelési időbe áttolni, és amikor csak megoldható, tűzzük ki magunk elé ezt a célt. Ha így teszünk, jutalmul kisebb, gyorsabb és megbízhatóbb programokat kapunk.

47. jó tanács: Biztosítsuk, hogy a nemlokális statikus objektumok inicializálódjanak felhasználásuk előtt

Ugye nem kell elmagyaráznom, milyen vakmerőség volna egy objektumot az inicializálást megelőzően használni. Maga a gondolat is teljesen abszurdnak tűnhet. A konstruktorok biztosítják, hogy az objektumok a létrehozásukkor inicializálódjanak, *n'est-ce pas*?

Nos, igen is, meg nem is. Egy adott fordítási egységen (azaz forrásfájlon) belül minden a legnagyobb rendben működik, de a dolgok trükkösebbé válnak, ha az egyik fordítási egységben található objektum inicializálása függ egy másik, egy eltérő fordítási egységben található objektum értékétől, *és* azt a másik objektumot is inicializálni kell.

Tegyük fel, hogy olyan könyvtárat készítettünk, amely egy fájlrendszer absztrakcióját valósítja meg. Olyan lehetőségek lehetnek benne, mint például az interneten található fájlok helyi fájlként történő kezelése. Mivel könyvtárunk az egész világot egyetlen fájlrendszerként láttatja, létrehozhatunk könyvtárunk névterében *(namespace)* (lásd a 28. jó tanácsot) egy speciális `theFileSystem` objektumot azért, hogy a felhasználók ezen keresztül bármikor használhassák a könyvtár által nyújtott fájlrendszer absztrakcióját:

```
class FileSystem { ... };   // Ez az osztály a könyvtárunk része
FileSystem theFileSystem;   // a könyvtár felhasználói ezzel az
                            // objektummal állnak kapcsolatban.
```

Mivel a `theFileSystem` valami nagyon összetett dolgot reprezentál, nem meglepő, hogy a létrehozása egyszerre elengedhetetlen és nem triviális. *Nagyon* definiálatlan viselkedést kapnánk, ha a `theFileSystem`-et azelőtt használnánk, hogy előállítottuk volna.

Tegyük fel most azt, hogy a könyvtárunk egyik felhasználója létrehoz egy osztályt egy fájlrendszer könyvtárai számára. Ez az osztály természetesen a `theFileSystem` objektumot használja:

```
class Directory {         // A könyvtár felhasználója hozza létre.
public:
   Directory();
   ...
};
Directory::Directory()
{
   egy Directory objektum létrehozása a theFileSystem objektum-
   ra meghívott tagfüggvények segítségével
}
```

Tételezzük fel továbbá, hogy ez a felhasználó úgy dönt, létrehoz egy megkülönböztetett globális `Directory` objektumot az ideiglenes fájlok számára:

```
Directory tempDir;              // Ideiglenes fájlokat
                                // tartalmazó könyvtár.
```

Az inicializálás sorrendjének problémája most már nyilvánvaló: a `tempDir` konstruktora megpróbálja a `theFileSystem` objektumot az inicializálása előtt használni, kivéve, ha a `theFileSystem` a `tempDir` előtt inicializálódik. De a `theFileSystem` és a `tempDir` objektumot mások, más időpontban, más fájlban hozták létre. Hogy lehetünk akkor biztosak abban, hogy a `theFileSystem` létre fog jönni a `tempDir` előtt?

Effajta kérdések mindig felmerülnek, ha olyan *nemlokális statikus objektumaink* vannak, amelyek más-más fordítási egységben találhatók, és amelyek helyes viselkedése attól függ, hogy egy adott sorrendben lettek-e inicializálva. A nemlokális statikus objektumok olyan objektumok, amelyek

- globális vagy névtér hatókörben *(scope)* definiáltak (pl. a `theFileSystem` és a `tempDir`),
- egy osztályban `static`-ként deklaráltak, vagy
- fájl hatókörben `static`-ként definiáltak.

Sajnálatos módon nincs rövidebb elnevezés a „nemlokális statikus objektum"-ra, úgyhogy kénytelenek vagyunk hozzászokni ehhez a kicsit kényelmetlen kifejezéshez.

Nem akarjuk, hogy a szoftverünk viselkedése a különböző fordítási egységekhez tartozó nemlokális statikus objektumok inicializálási sorrendjétől függjön, mivel sehogyan sem tudjuk ezt a sorrendet befolyásolni. Hadd ismételjem ezt meg: *semmilyen módon nem lehet befolyásolni azt a sorrendet, amely meghatározza a különböző fordítási egységekben található nemlokális statikus változók inicializálását.*

Indokolt a kérdés, hogy miért van ez így.

Azért, mert a nemlokális statikus objektumok inicializálásának „megfelelő" sorrendjét nehéz meghatározni. Nagyon nehéz. Olyan nehéz, mint a holtpont probléma *(halting-problem)* megoldása. A legáltalánosabb formájában – több fordítási egységgel és implicit sablonpéldányosítások *(template instantiations)* által generált nemlokális statikus objektumokkal (ahol az implicit példányosítások maguk is más implicit sablonpéldányosításokon keresztül jöhetnek létre) – nemcsak lehetetlen az inicializálás helyes sorrendjét meghatározni, hanem még olyan speciális eseteket sem érdemes keresni, amikor *meghatározható* a helyes sorrend.

A káoszelméletben van egy „pillangóeffektus"-nak nevezett alapelv. Ez az elv azt állítja, hogy az a kis légmozgás, amelyet egy pillagó szárnycsapásai okoznak a világ egyik részén, mélyreható időjárási változásokhoz vezethetnek attól távol eső vidékeken. Kicsit szigorúbban fogalmazva, azt állítja, hogy bizonyos típusú rendszerekben az inputok parányi zavarai az outputok gyökeres megváltozásához vezethetnek.

A szoftverrendszerek fejlesztése is bizonyságot tehet saját pillangóeffektusának létezése mellett. Bizonyos rendszerek nagyon érzékenyen viseltetnek a velük szemben támasztott követelmények részleteivel szemben, és a követelmények apró változtatásai is

nagymértékben befolyásolhatják azt, hogy milyen könnyen lehet egy rendszert implementálni. A 29. jó tanács például leírja, hogy ha a „String-ből char*" implicit konverzió specifikációját „String-ből const char*" konverzióra változtatjuk, ez hogyan teszi lehetővé, hogy egy lassú és hibalehetőségeket magában rejtő függvényt gyorsabbra és biztonságosabbra cseréljünk.

A probléma, hogy hogyan biztosítsuk a nemlokális statikus objektumok inicializálását használatuk előtt, ugyanilyen érzékeny kapcsolatban áll az elérni kívánt cél részleteivel. Ha a nemlokális statikus objektumok elérésének igénye helyett hajlandóak vagyunk beérni olyan objektumok elérésével, amelyek úgy *tesznek*, mintha nemlokális statikus objektumok lennének (persze az inicializálás körüli fejfájás nélkül), akkor eltűnik egy nehéz probléma. Annyira könnyen megoldható probléma marad a helyén, amelyet már nem is igazán nevezhetünk problémának.

A technika lényege – amelyet néha *Singleton mintaként* is emlegetnek – maga az egyszerűség. Először is, minden nemlokális statikus objektumot a saját függvényébe teszünk át, ahol static-nak deklaráljuk őket. Majd pedig úgy intézzük, hogy a függvény a benne található objektumra hivatkozó referenciát adjon vissza. Az objektumra való hivatkozás helyett a felhasználók a függvényt hívják meg. Vagyis a nemlokális statikus objektumokat függvényen belüli static objektumokra cseréljük.

Ennek a megközelítésnek az a megfigyelés az alapja, hogy bár a C++ nem mond szinte semmit arról, hogy a nemlokális statikus objektumok mikor inicializálódnak, azt nagyon is pontosan meghatározza, hogy egy függvényen belüli statikus objektum (azaz egy *lokális* statikus objektum) mikor inicializálódik: akkor, amikor először fordul elő az objektum definíciója a függvény hívása során. Így, ha kicseréljük a nemlokális statikus objektumok közvetlen elérését olyan függvények meghívására, amelyek a bennük lévő statikus objektumokra hivatkozó referenciákat adnak vissza, akkor garantált, hogy a függvények által visszaadott referenciák inicializált objektumokra hivatkoznak. És még egy ajándék: ha sosem hívunk meg egy nemlokális statikus objektumot szimuláló függvényt, akkor sohasem kell kifizetnünk az objektum létrehozásának és megszüntetésének a költségét, ami az igazi nemlokális statikus objektumokról már nem mondható el.

Nézzük e technika alkalmazását a theFileSystem és a tempDir mindegyikére:

```
class FileSystem { ... };          // Mint korábban
FileSystem& theFileSystem()        // ez a függvény helyettesíti
{                                  // a theFileSystem objektumot.
   static FileSystem tfs;          // Egy lokális statikus objektum
                                   // definíciója és inicializálása
                                   // (tfs = "the file system").
   return tfs;                     // Referencia visszaadása.
}
class Directory { ... };           // Mint eddig.
Directory::Directory()
{
   ugyanaz, mint eddig, azzal a kivétellel, hogy
   a theFileSystem-re történő hivatkozásokat a
   theFileSystem()-re történő hivatkozásokra cseréljük;
}
```

```
Directory& tempDir()          // Ez a függvény helyettesíti
{                             // a tempDir objektumot.
   static Directory td;       // Lokális statikus objektum
                              // definíciója/inicializálása.
   return td;                 // Referencia visszaadása.
}
```

E módosított rendszerprogram felhasználói továbbra is ugyanúgy programozhatnak, mint eddig tették, csak most a `theFileSystem` és a `tempDir` helyett a `theFileSystem()`-re és a `tempDir()`-re kell hivatkozniuk. Azaz, mindig csak az objektumokra hivatkozó referenciákat visszaadó függvényekre hivatkoznak, sohasem magukra az objektumokra.

A séma szerinti, referenciát visszaadó függvények mindig nagyon egyszerűek: az első sorban definiálunk és inicializálunk egy lokális statikus objektumot, a második sorban pedig ezt adjuk vissza. Ennyi. Mivel ilyen egyszerűek, kísértésbe eshetünk, hogy `inline`-nak deklaráljuk őket. A 33. jó tanácsból kiderül, hogy a C++ nyelv-specifikációjának legfrissebb átdolgozásai szerint ez egy teljesen helytálló implementációs stratégia, de azt is megtudjuk, hogy az alkalmazása előtt miért ajánlatos ellenőriznünk, hogy a fordítónk megfelel-e a szabványnak ebből a szempontból, vagy sem. Ha egy olyan fordítóval próbálkozunk, amely még nincs összhangban a szabvány idevonatkozó részével, akkor azt kockáztatjuk, hogy mind az elérést biztosító függvénynek, mind a benne definiált statikus objektumnak több példánya lesz. Ez már elegendő ahhoz, hogy egy felnőtt programozó elsírja magát.

Nincs itt semmiféle varázslat. Ahhoz, hogy ez a technika hatékonyan működjön, egy elfogadható inicializálási sorrendet kell meghatároznunk az objektumaink számára. Ha valaki úgy alakítja a dolgokat, hogy az A objektumnak a B előtt kell inicializálódnia, és ezzel egyidejűleg az A inicializálását arra alapozza, hogy a B már inicializálva lett, akkor bizony bajba kerül, és őszintén szólva, meg is érdemli. Ha azonban óvakodunk az ilyen patologikus helyzetektől, akkor az itt bemutatott séma igencsak jó szolgálatot tehet nekünk.

48. JÓ TANÁCS: FIGYELJÜNK ODA A FORDÍTÓ FIGYELMEZTETÉSEIRE

Sok programozó megszokásból hagyja figyelmen kívül a fordító figyelmeztetéseit. Végülis, ha komoly lenne a probléma, akkor hibaüzenet lenne, nem? Más nyelveknél ez viszonylag ártalmatlan gondolatnak számít, C++-ban azonban jó esély van arra, hogy a fordító készítői jobban átlátják, hogy mi történik, mint mi. Az itt következő hibát például mindenki elköveti előbb-utóbb:

```
class B {
public:
   virtual void f() const;
};
```

```
class D: public B {
public:
   virtual void f();
};
```

Az alapötlet az, hogy `D::f` újradefiniálja a `B::f` virtuális függvényt, de belecsúszott egy hiba: `B`-ben `f` egy `const` tagfüggvény, `D`-ben viszont nem `const`-nak lett deklarálva. Egy általam ismert fordító erre a következőt mondja:

```
warning: D::f() hides virtual B::f()
```

Kellő tapasztalat híján túl sok programozó így rendezi ezt magában: – Hát *persze*, hogy a `D::f` eltakarja a `B::f`-et, hiszen ez a *dolga!* – Helytelenül. A fordító azt próbálja közölni velünk, hogy a `B`-ben deklarált `f` nem lett újradeklarálva a `D`-ben, hanem teljesen el lett takarva (az 50. jó tanácsban leírom, hogy miért van ez így). Ha nem figyelünk oda a fordítónak erre a figyelmeztetésére, szinte biztos, hogy a program hibásan fog működni. Azán jön a sok-sok „debug"-olás, amivel azt derítjük ki, amit ez a fordító már az első pillanatban észrevett.

Ha már kezdjük megismerni egy adott fordító figyelmeztetéseit, természetesen lassanként azt is megértjük, hogy mit jelentenek az egyes üzenetek (ami sajnos gyakran egészen más, mint aminek az üzenet *látszik)*. Ha már birtokában vagyunk e tapasztalatnak, akkor a figyelmeztetések akár egész soráról eldönthetjük, hogy nem veszünk tudomást róluk. Ez rendben is van, de fontos, hogy csak akkor hessegessünk el egy figyelmeztetést, amikor már pontosan értjük, mit akar nekünk mondani.

Ha már a figyelmeztetések témájánál tartunk, ne feledkezzünk meg arról, hogy ezek a figyelmeztető üzenetek természetüknél fogva implementációfüggők. Így aztán nem túl jó ötlet elnagyolni a programozást, arra számítva, hogy majd a fordítók elénk hozzák a hibáinkat. A fenti, függvénytakarást tartalmazó kód például egy másik – széles körben használt – fordítón zokszó nélkül átsuhan. A fordítóknak annyi a dolguk, hogy a C++-t futtatható formára fordítsák, nem pedig az, hogy személyes védőhálóként funkcionáljanak. Vagy az olvasónak ilyen védőháló kell? Programozzon Adában!

49. jó tanács: Ismerjük meg a szabvány könyvtárat

A C++ szabvány könyvtára nagy. Nagyon nagy. Hihetetlenül nagy. Hogy mennyire? Hadd fogalmazzam meg így: a specifikáció több, mint 300 sűrűn teleírt oldal a C++ szabványban, és ez még nem tartalmazza a szabvány C könyvtárat, amely szintén része a C++ könyvtárnak, méghozzá „referencián keresztül". (Becsszóra ezt a kifejezést használják!)

Természetesen a nagyobb nem mindig jobb, de ebben az esetben a nagyobb jobb, mert egy nagy könyvtár sok funkcionalitást tartalmaz. Minél több funkcionalitás van a szabvány könyvtárban, annál többre támaszkodhatunk az alkalmazásaink fejlesztése során. A C++ könyvtár nem ad eszközt *mindenre* (a párhuzamosság kezelése és a grafikus felhasználói felület *(user interface)* támogatása például hiányzik belőle), de a kínálat így is nagy. Majdnem mindenben támaszkodhatunk rá.

Mielőtt összefoglalnám, miből áll a könyvtár, mondanom kell pár szót arról, hogyan épül fel. Mivel a könyvtárban oly sok minden található, elég nagy az esély arra, hogy az olvasó – vagy egy ismerőse – olyan osztály- vagy függvénynevet választ, amely megegyezik a szabvány könyvtárban található valamelyik névvel. A szabvány könyvtárban jóformán minden az `std` névtérbe *(namespace)* lett beágyazva (lásd a 28. jó tanácsot), épp azért, hogy megvédjen minket az ilyen névütközésektől *(name conflicts)*. Ez azonban egy új problémát vet fel. Megszámlálhatatlanul sok, már létező C++ kódsor támaszkodik az álszabvány könyvtárban fellelhető, évek óta használatban lévő funkcionalitásra, például arra, amelyik az `<iostream.h>`, `<complex.h>`, `<limits.h>` stb. fejállományokban *(header files)* deklarált. Ezeket a létező szoftvereket nem tervezték a névterek használatára, de kár lenne, ha a meglévő kódok a szabvány könyvtár `std`-be csomagolása miatt használhatatlanná válnának. (Ha az ily módon használhatatlanná váló kód szerzői alól kirántanánk a megszokott könyvtárszőnyeget, valószínűleg a „kár" szónál valamivel durvább kifejezést használnának az érzéseik leírására.)

Tekintetbe véve a felbőszült programozók lázadó bandáinak pusztító erejét, a szabványosítási bizottság úgy határozott, hogy az `std`-be csomagolt összetevőknek új fejállománynevet talál ki. Az új fejállománynevek generálásához annyira triviális algoritmust választottak, mint amennyire kellemetlen benyomást kelt az általa produkált eredmény: a meglévő C++ fejállományok nevének végéről egyszerűen elhagyták a `.h`-t. Így az `<iostream.h>`-ból `<iostream>`, a `<complex.h>`-ból `<complex>` stb. lett. A C fejállományokra ugyanezt az algoritmust alkalmazták, de mindegyik eredményül kapott név elejére még odabiggyesztettek egy bevezető `c`-t. Így aztán a C `<string.h>`-jából `<cstring>`, az `<stdio.h>`-ból `<cstdio>` lett stb. Váratlan fordulat volt, hogy a régi C++ fejállományokról hivatalosan is kinyilvánították, hogy a használatuk *nem javasolt* (azaz megszűnt a támogatottságuk), a régi C-s fejállományokról viszont nem (a C-kompatibilitás fenntartása végett). A gyakorlatban a fordítók szállítóinak nem áll érdekében az ügyfelek már meglévő szoftvereit figyelmen kívül hagyni, ezért a régi C++ fejállományok támogatottsága várhatóan még évekig fenn fog maradni.

Ezért aztán gyakorlati szempontból a C++ fejállományok frontján a helyzet a következő:

- A régi C++ fejállományneveket, mint amilyen az `<iostream.h>`, nagy valószínűséggel továbbra is támogatják, habár nincsenek benne a hivatalos szabványban. Ezeknek a fejállományoknak a tartalma *nincs* az `std` névtérben.
- Az új C++ fejállománynevek, mint amilyen az `<iostream>`, ugyanazt az alapvető funkcionalitást tartalmazzák, mint a nekik megfelelő régi fejállományok, de a fejállományok tartalma *benne van* az `std` névtérben. (A szabványosítás során néhány könyvtárkomponens bizonyos részleteit megváltoztatták, így aztán nem feltétlenül van pontos illeszkedés egy régi és egy új C++ fejállomány tartalma között.)
- A szabvány C fejállományok, mint az `<stdio.h>`, továbbra is támogatottak. Ezeknek a fejállományoknak a tartalma *nincs* az `std`-ben.
- A C könyvtár funkcionalitása számára fenntartott új C++ fejállományok olyan nevet kaptak, mint pl. a `<cstdio>`. Ugyanazt a tartalmat kínálják, mint a nekik megfelelő régi C fejállományok, de a tartalmuk *benne van* az `std`-ben.

Mindez elsőre kicsit furcsának tűnhet, de egyáltalán nem olyan nehéz hozzászokni. A legnagyobb kihívás az összes sztring fejállomány rendben tartása: a `<string.h>` a `char*`-alapú, sztringeket kezelő függvényekhez tartozó régi C fejállomány, a `<string>` az új sztring osztályokhoz tartozó (lásd lejjebb), az `std`-be csomagolt C++ fejállomány, a `<string>` pedig a régi C fejállomány `std`-be csomagolt változata. Ha az olvasó ezt el tudja sajátítani – és én tudom, hogy képes erre –, a könyvtár többi része már egyszerű.

Következő lépésként azt kell tudnunk a szabvány könyvtárról, hogy majdnem minden sablon (*template)* benne. Gondoljunk csak a jó öreg iostreams barátunkra! (Aki az iostreams-szel nem áll baráti kapcsolatban, az lapozzon a 2. jó tanácshoz és megtudja, miért ajánlatos az ilyen kapcsolatokat ápolni.) Az iostreams segít a karakterfolyamok *(streams of character)* kezelésében, de mi az a karakter? Egy `char`? Vagy egy `wchar_t`? Vagy egy Unicode karakter? Vagy valamilyen más, több-bájtos karakter? Nincs magától értetődően helyes válasz, ezért a könyvtár megengedi, hogy válasszunk. Az összes adatfolyamosztály *(stream class)* valójában osztálysablon, és a karaktertípust az adatfolyamosztály példányosításakor adjuk meg. A szabvány könyvtár például a `cout`-ot `ostream` típusúnak definiálja, de az `ostream` valójában egy típusdefiníció *(typedef)* a `basic_ostream<char>`-ra.

Hasonló megfontolások igazak a szabvány könyvtár többi osztályának nagy részére is. A `string` nem osztály, hanem osztálysablon, amelyben egy típusparaméter definiálja a karakterek típusát az egyes `string` osztályokban. A `complex` sem osztály, hanem osztálysablon, ahol a típusparaméter az egyes `complex` osztályok valós és képzetes részének típusát definiálja. A `vector` sem osztály, hanem osztálysablon. És ez így megy tovább.

Nem kerülhetjük ki a szabvány könyvtárban található sablonokat, de ha megszoktuk, hogy csak `char`-okból álló adatfolyamokkal és sztringekkel dolgozunk, akkor általában figyelmen kívül hagyhatjuk őket. Ezt azért tehetjük meg, mert a szabvány könyvtár `typedef`-eket definiál ezeknek a könyvtárkomponenseknek a `char`-ral történő példányosítására, így aztán továbbra is a `cin, cout, cerr` stb. objektumokkal és az `istream`, `ostream`, `string` stb. típusokkal programozhatunk. És egyáltalán nem kell azzal foglalkoznunk, hogy a `cin` valódi típusa `basic_istream<char>`, a `string`-é pedig `basic_string<char>`.

A szabvány könyvtár sok összetevője még annál is jobban sablonosított, mint amennyire ez az itt foglaltakból kiderül. Tekintsük megint a sztring látszólag egyértelmű fogalmát. Persze, paraméterezhető a benne lévő karakterek típusával, de a különböző karakterhalmazok részleteikben eltérnek egymástól, például a speciális fájlvége karakterben, a karaktertömbök *(arrays of characters)* leghatékonyabb másolásában stb. Ezeket a jellemzőket a szabvány *trait*-eknek nevezi, és a `string` példányosítások során egy további sablonparaméterrel adhatók meg. Továbbá, a `string` objektumok valószínűleg végeznek dinamikus memóriafoglalást és -felszabadítást, de ennek a feladatnak még nagyon sok megközelítése létezik (lásd a 10. jó tanácsot). Melyik a legjobb? Választáshoz érkezünk. A `string` sablon `Allocator` paramétert vesz fel, a `string` objektumok által használt memóriaterület lefoglalására és felszabadítására pedig `Allocator` típusú objektumok használatosak.

Íme a `basic_string` sablon teljesen kifejtett deklarációja és a ráépülő `string` typedef. Ezt találhatjuk – vagy valami vele ekvivalenset – a `<string>` fejállományban:

```
namespace std {
   template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
      class basic_string;
   typedef basic_string<char> string;
}
```

Vegyük észre, hogy a `basic_string`-ben a `traits` és `Allocator` paramétereknek alapértelmezett értékei vannak. Ez jellemző a szabvány könyvtárra. Rugalmasságot biztosít azoknak, akiknek arra van szükségük, de egy „tipikus" felhasználó, aki csak „normális" dolgokat szeretne elvégezni, figyelmen kívül hagyhatja azt a bonyolultságot, amely ezt a rugalmasságot lehetővé teszi. Más szóval, ha csak olyan sztring objektumokra van szükségünk, amelyek többé-kevésbé úgy viselkednek, mint a C-beli sztringek, akkor használhatunk `string` objektumokat, és egyáltalán nem is kell tudatában lennünk annak, hogy valójában `basic_string<char, char_traits<char>, allocator<char> >` típusú objektumokkal dolgozunk.

Nos, ezt általában megtehetjük. Máskor viszont be kell kukucskálnunk a kulisszák mögé. A 34. jó tanács azt mutatja be például, hogy miért előnyös egy osztályt a definíció megadása nélkül deklarálnunk. Ugyanott szerepel a `string` típus itt következő, helytelen deklarációja:

```
class string;              // Lefordul, de ez nem az, amit akarunk.
```

A névtérrel kapcsolatos megfontolásokat egy pillanatra félretesszük. Itt az az igazi probléma, hogy a `string` nem egy osztály, hanem egy típusdefiníció. Jó lenne, ha az alábbi módon meg tudnánk oldani a problémát:

```
typedef basic_string<char> string;
```

– de ez nem fordul le. – Milyen `basic_string`-ről van itt szó? – csodálkoznának a fordítók, bár valószínűleg egészen másként fogalmaznák meg a kérdést. Nem. Ahhoz, hogy a `string`-et deklaráljuk, deklarálnunk kell az összes olyan sablont is, amelytől függ. Ha meg tudnánk tenni, az valahogy így nézne ki:

```
template<class charT> struct char_traits;
template<class T> class allocator;
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
   class basic_string;
typedef basic_string<char> string;
```

Mégsem deklarálhatunk azonban `string`-et. Vagy legalábbis nem volna szabad. Azért nem, mert a könyvtár implementálóinak akkor megengedett a `string`-et – vagy bármi mást az `std` névtérben – a szabványban megadottól eltérően definiálni, ha ez a

definíció a szabványhoz illeszkedő viselkedést eredményez. Egy `basic_string` implementáció például bevezethetne egy negyedik sablonparamétert, de e paraméter alapértelmezett értékének olyan kódot kellene eredményeznie, amely úgy viselkedik, ahogy egy „díszítetlen" `basic_string`-nek a szabvány szerint viselkednie kell.

Az eredmény? Ne próbáljuk `string` – vagy a szabvány könyvtár bármely más részének – deklarációját manuálisan elkészíteni! Töltsük be helyette a megfelelő fejállományt, például a `<string>`-et!

A fejállományokról és sablonokról szóló háttérinformációval a hónunk alatt most már olyan helyzetbe kerültünk, hogy áttekinthetjük a szabvány C++ könyvtár elsődleges összetevőit:

- **A szabvány C könyvtár.** Még mindig megvan, és még mindig lehet használni. Néhány apróság itt-ott megváltozott benne, de szándékait és célját illetően ez ugyanaz a C könyvtár, amelyik már évek óta létezik.

- **Iostreams.** A „hagyományos" iostream implementációkhoz képest ez sablonosítva lett, módosult az öröklődési hierarchiája, kiegészült a kivételek dobásának lehetőségével, és úgy korszerűsítették, hogy támogatja a sztringeket (a `stringstream` osztályokon keresztül) és a nemzetköziesítés megvalósítását locale-eken keresztül – lásd lejjebb. Mégis, továbbra is megvan benne szinte minden, amit az iostream könyvtártól várunk. Azaz, még mindig támogatja az adatfolyam puffereket *(stream buffers)*, formázó objektumokat és manipulátorokat, a fájlokat, valamint a `cin`, `cout`, `cerr`, és `clog` objektumokat. Ez azt jelenti, hogy a `string`-eket és a fájlokat kezelhetjük adatfolyamként, és teljeskörűen szabályozhatjuk az adatfolyamok viselkedését, beleértve a pufferelést és a formázást is.

- **Sztringek.** A `string` objektumokat arra tervezték, hogy az alkalmazások nagy részében ne legyen szükség a `char*` pointerek használatára. Támogatják azokat a műveleteket, amelyeknél ezt el is várnánk (pl. a sztringek összefűzését, az egyes ka-rakterek konstans idejű elérését az `operator[]`-n keresztül stb.), `char*`-gá konvertálhatók, így kompatibilisek maradnak a már meglévő kódokkal, és automatikusan végzik a memóriakezelést. Bizonyos `string` implementációk referenciaszámlálót használnak, ami *jobb* teljesítményt eredményez – mind idő, mind tárhely tekintetében –, mint a `char*`-alapú sztringek.

- **Tárolók** *(Containers)*. Ezentúl a kulcsfontosságú tároló osztályokat ne mi írjuk magunknak! A könyvtár hatékony vektor-implementációkat nyújt – ezek úgy viselkednek, mint a dinamikusan kiterjeszthető tömbök –, listákat (kétirányú, láncolt listákat), sorokat *(queues)*, vermeket *(stacks)*, kétvégű sorokat *(deques)*, asszociatív tömböket *(maps)*, halmazokat és bithalmazokat biztosít a számunkra. Sajnos nincsenek hasítótáblák *(hash tables)* a könyvtárban (bár jónéhány szállító kiterjesztésként biztosítja őket), valamennyi kárpótlást nyújt azonban az, hogy a `string`-ek is tárolók. Ez fontos, mert azt jelenti, hogy minden, ami egy tárolóval elvégezhető (lásd lejjebb), az elvégezhető egy `string`-gel is.

  Hogyan? Az olvasó tudni szeretné, *honnan* tudom, hogy a könyvtár implementációi hatékonyak? Egyszerű: a könyvtár meghatározza az egyes osztályok felületét, és minden felület specifikációjához hozzátartozik a teljesítményre vonatkozó garanciák egy csoportja. Például, függetlenül attól, hogyan történik a vector implementáci-

ója, önmagában nem elég, ha csak az elemeinek az *elérése (access)* biztosított, az is kell, hogy ez az elérés *konstans idejű* legyen. Ha nem ez történik, akkor a `vector` implementáció nem lesz szabványos.

Sok C++ programban a `new` és a `delete` használata legtöbbször a dinamikusan allokált tömbökhöz és sztringekhez köthető, a `new/delete` használata körüli hibák – különösen a `new`-val lefoglalt memória felszabadításának elmulasztása miatti memóriaszivárgások *(memory leaks)* – pedig aggasztóan gyakoriak. Ha `char*`-ok és dinamikusan allokált tömbökre mutató pointerek helyett `string` és `vector` objektumokat használunk – mindkettő saját maga végzi a memóriakezelését –, a `new`-ink és `delete`-jeink nagy része el fog tűnni, a használatukkal együtt járó nehézséggekkel együtt (lásd pl. a 6. és a 11. jó tanácsot).

- **Algoritmusok.** Jó dolog, ha vannak szabvány tárolóink, de még jobb, ha egyszerű módszerekkel tudunk bizonyos dolgokat elvégezni velük. A szabvány könyvtár több, mint két tucat ilyen egyszerű módszert – azaz előre definiált függvényt, hivatalosan *algoritmust* (amely valójában függvénysablon) – biztosít a számunkra, melyeknek java része a könyvtár *összes* tárolójával és a beépített tömbökkel együtt is használható!

  Az algoritmusok egy tároló tartalmát sorozatnak tekintik, és vagy a tároló összes elemének megfelelő sorozatra, vagy egy részsorozatra mindegyik algoritmus alkalmazható. A szabvány algoritmusok között megtalálható a `for_each` (egy függvényt alkalmaz a sorozat minden elemére), a `find` (az első, egy adott értéket tartalmazó pozíciót keresi meg), a `count_if` (megszámolja a sorozat azon elemeit, amelyek eleget tesznek egy adott feltételnek), az `equal` (meghatározza, hogy két sorozat egyenlő értékű elemeket tartalmaz-e), a `search` (megkeresi az első olyan pozíciót egy sorozatban, ahol egy második sorozat részsorozatként fordul elő), a `copy` (egy sorozatot átmásol egy másikba), a `unique` (a többször előforduló elemeket eltávolítja egy sorozatból), a `rotate` (egy sorozatban az értékeket elforgatja) és a `sort` (egy sorozat értékeit rendezi). Ne feledjük, hogy ez csak egy *mintavétel* a rendelkezésre álló algoritmusokból, a könyvtárban ezeken kívül még sok van.

  Az algoritmusokhoz ugyanúgy tartoznak teljesítménybeli garanciák, mint a tároló műveletekhez. Előírás például, hogy a `stable_sort` algoritmus nem végezhet O(N log N)-nél több összehasonlítást. (Nem kell izgulni, ha az előző mondatban található „nagy ordó" jelölés ismeretlen valakinek. Nagyjából valójában annyit jelent, hogy a `stable_sort`-nak ugyanolyan szintű teljesítményt kell nyújtania, mint a leghatékonyabb, általános sorozatrendező algoritmusoknak.)

- **A nemzetköziesítés támogatása.** A különböző kultúrákban a dolgok különbözőképpen történnek. A C könyvtárhoz hasonlóan a C++ könyvtár is biztosít olyan szolgáltatásokat, amelyek elősegítik a nemzetköziesített szoftverek készítését. A C++ megközelítése azonban – bár elvi alapon rokon a C-belivel – más. Gondolom senkit sem lep meg, hogy például a C++ nemzetköziesítést támogató eszközei széleskörűen hasznosítják a sablonok adta lehetőségeket, és kihasználják az öröklődés és a virtuális függvények előnyeit is.

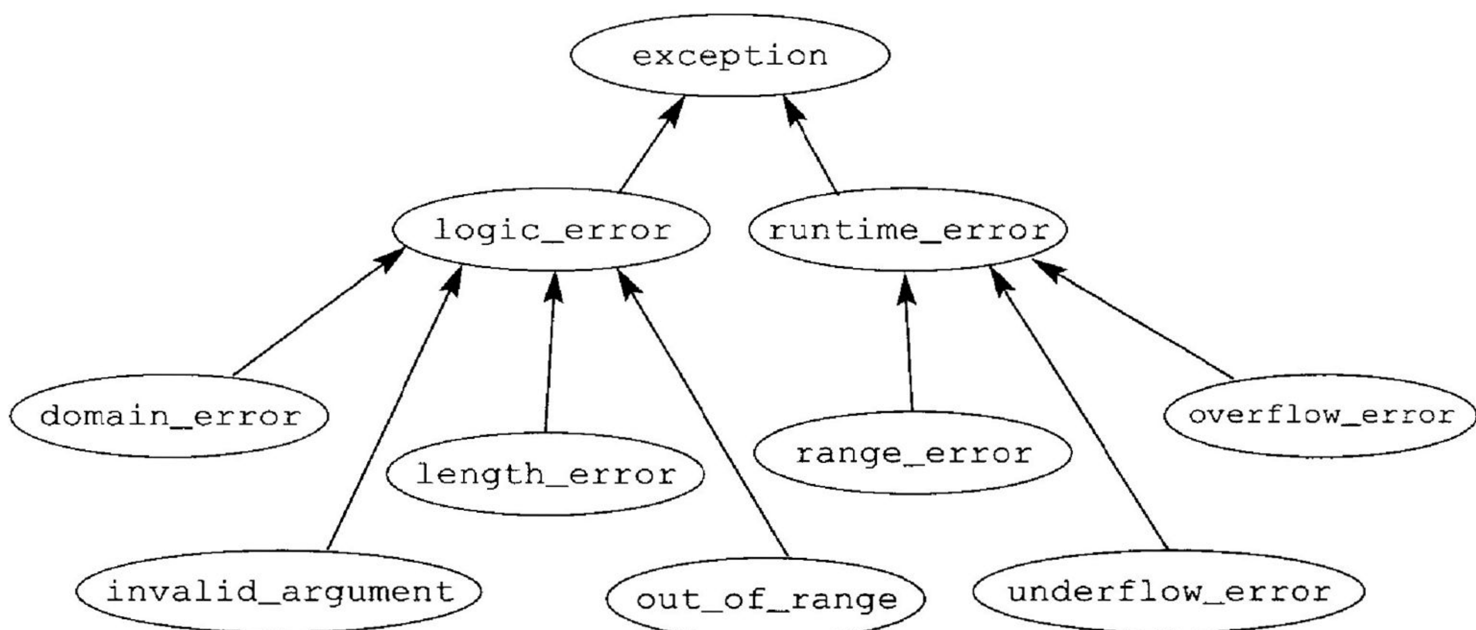
  A nemzetköziesítést támogató elsődleges könyvtárelemek a *facet*-ek és a *locale*-ek. A facet-ek azt írják le, hogy egy kultúra bizonyos tulajdonságait hogyan kell kezelni. Ilyenek például az összehasonlítás szabályai (azaz, hogy a helyi karakterkész-

letből képzett sztringeket hogyan kell rendezni), a dátumok és időpontok megjelenítési módja, a számok és pénzmennyiségek megjelenítése, illetve az üzenetazonosítók leképezése (természetes) nyelv-függő üzenetekre stb. A locale-ek az ilyen facetek halmazait csomagolják egybe. Az Egyesült Államokhoz tartozó locale például olyan facet-eket tartalmazna, amelyek leírják, hogy hogyan kell az amerikai angol nyelvű sztringeket sorba rendezni, hogyan kell írni és olvasni a dátumokat és az időpontokat, a pénzegységeket és a numerikus értékeket stb., úgy, hogy az USA-ban élő embereknek az megfelelő legyen. A Franciaországhoz tartozó locale ugyanakkor azt írná le, hogy hogyan kell ezeket a feladatokat úgy elvégezni, ahogy azt a franciák megszokták. Egy programon belül a C++ több aktív locale-t is megenged, ezért aztán egy alkalmazás különböző részei más-más konvenciókat alkalmazhatnak.

- **Numerikus számítások támogatása.** A FORTRAN ideje lassan lejár. A C++ könyvtár osztálysablont biztosít egyrészt a komplex számokat reprezentáló osztályokhoz (a valós és a képzetes rész pontossága lehet `float`, `double`, vagy `long double`), másrészt az olyan speciális tömbtípusokhoz, amelyeket kifejezetten a numerikus számítások megkönnyítésére terveztek. A `valarray` típusú objektumokat például úgy definiálták, hogy álnevek *(alias)* nélküli elemeket tartalmazzanak. Ez lehetővé teszi a fordítóknak, hogy sokkal erőszakosabbak legyenek az optimalizálás során, különösen vektor-processzoros gépek esetén. A könyvtár támogat továbbá két különböző tömbszelettípust, és többek között algoritmusokat ad a skaláris szorzat, a parciális összeg és a különbségsorozat számítására.

- **Diagnosztikai támogatás.** A szabvány könyvtár három lehetőséget is ad a hibák jelzésére: C állításokat *(assertions)* (lásd a 7. jó tanácsot), hibakódokat és kivételeket. Hogy segítséget nyújtson a kivétel típusok rendszerezéséhez, a könyvtár a kivételosztályok alábbi hierarchiáját definiálja:

  A `logic_error` típusú – vagy annak alosztályaihoz tartozó – kivételek a szoftver logikájában előforduló hibákat reprezentálják. Elméletileg az ilyen hibák gondosabb programozási munkával elkerülhetők lettek volna. A `runtime_error` típusú – vagy annak alosztályaihoz tartozó – kivételek csak futási időben észlelhető hibákat reprezentálnak.



Használhatjuk ezeket az osztályokat úgy, ahogy vannak, vagy származtatással létrehozhatunk belőlük saját kivételosztályokat is. Vagy figyelmen kívül is hagyhatjuk őket. A használatuk nem kötelező.

Ez a felsorolás nem terjed ki a szabvány könyvtár egészére. Ne feledjük, a specifikáció több, mint 300 oldalnyi! Ez a felsorolás azért híven tükrözi az alapvető struktúrát.

A könyvtár tárolókra és algoritmusokra vonatkozó részének közismert neve: *Standard Template Library* – az STL *(Szabvány Sablon Könyvtár)*. Valójában az STL-nek van egy harmadik összetevője is, méghozzá a bejárók (iterátorok), amelyekről még nem beszéltem. A bejárók olyan pointer-szerű objektumok, amelyek lehetővé teszik, hogy az STL algoritmusai és tárolói együttműködjenek. A szabvány könyvtár itt közölt leírásához nincs szükség a bejárók megértésére. Akit érdekel a téma, a 39. jó tanácsban találhat példát a használatukra.

Az STL a szabvány könyvtár legforradalmibb része. Nem az általa nyújtott tárolók és algoritmusok miatt (bár ezek tagadhatatlanul hasznosak), hanem a felépítése miatt. Röviden, a felépítése tekintetében ugyanis kiterjeszthető: az STL-hez hozzáadhatunk új elemeket. Természetesen a szabvány könyvtár összetevői rögzítettek, de ha követjük azokat a konvenciókat, amelyek mentén az STL-t készítették, akkor készíthetünk olyan saját tárolókat, algoritmusokat és bejárókat, amelyek ugyan-olyan jól együtt tudnak működni a szabvány STL komponensekkel, mint azok egymással. Kihasználhatjuk a mások által írt, az STL-nek megfelelő tárolókat, algoritmusokat és bejárókat, vagy éppen mások tudják felhasználni a miénket. Az STL-t az teszi forradalmivá, hogy ez nem egy szoftver, hanem *konvenciók* halmaza. A szabvány könyvtárban található STL komponensek egyszerűen annak a jónak a megnyilvánulásai, amelyek ezeknek a konvencióknak a követéséből származnak.

Ha a szabvány könyvtár összetevőit használjuk, akkor általánosságban felhagyhatunk azzal, hogy saját – a legalacsonyabb szinttől felépített – mechanizmusokat tervezzünk az adatfolyam I/O-ra, a sztringekre, a tárolókra (beleértve a bejárást és a szokásos műveleteket), a nemzetköziesítésre, a numerikus adatstruktúrákra és a diagnosztikára. Ezáltal sokkal több időnk és energiánk marad a szoftverfejlesztés igazán lényeges részére: azokra az implementációkra, amelyek megkülönböztetik a mi termékünket versenytársainkétól.

50. JÓ TANÁCS: TÖKÉLETESÍTSÜK C++-TUDÁSUNKAT

Sok cucc van a C++-ban. C-s cuccok. Túlterheléses cuccok. Objektumorientált cuccok. Sablon cuccok. Kivétel cuccok. Névtér cuccok. Cucc, cucc, cucc! Néha ez már nyomasztó. Hogy fogjuk ezt az összes cuccot megérteni?

Nem olyan nehéz, ha egyszer tudatosítjuk azokat a tervezési célokat, amelyek azzá kovácsolták a C++-t, ami. E célkitűzések között voltak mindenekelőtt az alábbiak:

- **Kompatibilitás a C-vel.** Sok-sok C kód létezik, és sok-sok C programozó. A C++ kihasználja ezt a bázist és épít rá – ööö, úgy értem, „továbbfejleszti".
- **Hatékonyság.** Bjarne Stroustrup, a C++ tervezője és első implementálója a kezde-

tektől fogva tudta, hogy azok a C programozók, akiket meg akar nyerni, oda sem pillantanak másodszor, ha a nyelvváltásért cserébe teljesítménnyel kell fizetniük. Ennek eredményeképp biztosította, hogy a C++ hatékonyság tekintetében versenyképes legyen a C-vel – úgy 5%-on belül.

- **Kompatibilitás a hagyományos eszközökkel és környezetekkel.** Időnként láthatunk különleges fejlesztői környezeteket, de fordítók, linkerek és szövegszerkesztők szinte mindenütt vannak. A C++-t úgy tervezték, hogy az egeres környezettől a mainframe-ekig mindenhol működjön, úgyhogy annyira kicsi a poggyásza, amennyire csak lehet. Portolni akarja valaki a C++-t? Elég egy *nyelvet* portolni, és kihasználni a célplatform már meglévő eszközeit. (Gyakran azonban jobb implementációt is lehet készíteni, ha a linker például módosítható úgy, hogy a helyben kifejtés *(inlining)* és a sablonok terén felmerülő megerőltetőbb feladatok kezelhetővé válnak.)
- **Alkalmazhatóság valós problémák megoldására.** A C++-t nem arra tervezték, hogy egy kellemes, tiszta nyelv legyen, amely arra való, hogy diákokat a programozás alapjaira tanítsunk. Úgy tervezték, hogy professzionális programozók hathatós eszköze legyen a sokféle területen felmerülő valódi problémák megoldásában. A való világnak vannak durva peremvidékei, így aztán egyáltalán nem meglepő, hogy horzsolások csúfítják el azoknak az eszközöknek a végső formáját, amelyekre a profik támaszkodnak.

Ezek a célok magyarázatot adnak a nyelv nagyon sok olyan részletére, amely egyébként csak idegesítő lenne. Miért viselkednek az implicit módon generált másoló konstruktorok *(copy constructors)* és értékadó operátorok *(assignment operators)* úgy, ahogy, főleg pointerek esetén (lásd a 11. és a 45. jó tanácsot)? Azért, mert a C így másolja és adja értékül a `struct`-okat, és a C-vel való kompatibilitás fontos. Miért nem lesznek a destruktorok automatikusan virtuálisak (lásd a 14. jó tanácsot), és miért kell az implementációs részleteknek az osztálydefiníciókban megjelenniük (lásd a 34. jó tanácsot)? Azért, mert ha nem így lenne, akkor azért teljesítménycsökkenéssel kellene fizetnünk, és a hatékonyság fontos. Miért nem képes a C++ a nemlokális statikus objektumok közötti inicializálási függőségeket felderíteni (lásd a 47. jó tanácsot)? Mert a C++ támogatja a külön fordítást – azaz a forrásmodulok egymástól elkülönülő fodításának, majd a tárgykódok *(object files)* futtatható programmá történő összelinkelésének lehetőségét –, meglévő linkerekre támaszkodik, és nem követeli meg programadatbázisok létezését. Ennek eredményeképp a C++ fordítók szinte sosem tudnak mindent egy teljes programról. És végül, vajon miért nem szabadítja meg a programozókat a C++ az olyan fárasztó feladatoktól, mint a memóriakezelés (lásd a jó tanácsokat 5–10-ig), vagy az alacsonyszintű pointer műveletek? Azért, mert bizonyos programozóknak szükségük van ezekre a lehetőségekre, és az igazi programozók szükségletei mindenek felett állnak.

E példák csupán sugallják, hogy a C++ mögötti tervezési célok hogyan formálják a nyelv viselkedését. Ahhoz, hogy mindent végignézzünk, egy egész könyvre lenne szükség, még jó, hogy Stroustrup írt egyet. Ez a könyv a *The Design and Evolution of C++* (Addison-Wesley, 1994.), amit néha csak „D&E"-nek neveznek. Ha elolvassuk, kiderül, hogy milyen új elemek kerültek a C++-ba, milyen sorrendben, és miért. Olyan elemekről is olvashatunk, indoklással együtt, amelyet elutasítottak. Még azt a bennfentes történetet is megtudhatjuk, hogy hogyan vették fontolóra és vetették el, majd hogyan gondolták át megint és fogadták végül el a `dynamic_cast` (lásd a 39. jó tanácsot) lehetőségét – és

persze, hogy mindez miért történt. Ha valaki gondban van a C++ megértésével, akkor a D&E sok félreértést eloszlathat.

A *The Design and Evolution of C++* nagyszerű betekintést nyújt abba a folyamatba, amelynek során a C++ azzá vált, ami, ráadásul a szöveg egyáltalán nem olyan, mint a nyelv egy hivatalos specifikációja. Akinek az kell, az forduljon a C++ nemzetközi szabványához, ami a maga 700 körüli oldalszámával a formalizmus egy lenyűgöző példája. Abban olyan magával ragadó prózai részeket találunk, mint az alábbi:

> Egy virtuális függvény hívása az objektumot jelölő pointer vagy referencia statikus típusa által meghatározott virtuális függvény deklarációjában található alapértelmezett paramétereket használja. Egy származtatott osztályban lévő felüldefiniálást végző függvény nem veszi át az általa felüldefiniált függvény alapértelmezett paramétereit.

Ez a bekezdés a 38. jó tanács alapja („Soha ne definiáljunk át egy örökölt alapértelmezett paraméterértéket"), de remélem, hogy a téma általam bemutatott értelmezése valamivel befogadhatóbb, mint a fenti szöveg.

A szabvány nem éppen lefekvés előtti olvasnivaló, de a legjobb segédeszköz – a *szabvány* segédeszköz – akkor, ha az olvasó nem ért egyet valakivel (mondjuk a fordító szállítójával, vagy egy forráskódot feldolgozó másik eszköz fejlesztőjével) abban, hogy mi C++, és mi nem az. Egy szabvány célja mindig az, hogy olyan döntő fontosságú információkkal szolgáljon, amelyek az efféle vitákat eldöntik.

A szabvány hivatalos címe elég hosszú, de aki mindenáron tudni akarja, annak itt van, tessék: *International Standard for Information Systems – Programming Language C++ (Informatikaiációs rendszerek nemzetközi szabványa – A C++ programozási nyelv)*. A Nemzetközi Szabványügyi Szervezet *(International Organization of Standardization – ISO)* 21-es számú munkacsoportja publikálta. (Ha igazán precízek akarunk lenni, akkor a publikáló – nem én találtam ki – az ISO/IEC JTC1/SC22/WG21.) A hivatalos szabvány megrendelhető az adott ország nemzeti szabványügyi hivatalától (ez az USA-ban az ANSI, az *American National Standards Institute*, Magyarországon pedig a *Magyar Szabványügyi Hivatal*), de a szabvány egykori vázlatainak másolatai – amelyek eléggé hasonlítanak a végső dokumentumra (bár nem azonosak azzal) – letölthetők az internetről. Érdemes például a `http://www.cygnus.com/misc/wp/` címen keresgélni, de a cybervilág változásának sebességét figyelembe véve ne csodálkozzunk azon, ha ez a link már hibát ad, amikor próbálkozunk vele. Ha azt ad, akkor a kedvenc keresőoldalunk biztos felhoz néhány olyan URL-t, ami működik.

Mind mondottam, a *The Design and Evolution of C++* jó arra, hogy betekintést nyerjünk a nyelv tervezési eljárásaiba, míg a szabvány tökéletes a nyelv részleteinek leszögezésére. De azért jó lenne, ha létezne valami kellemes közbülső terület a D&E 10 000 méter távolságból történő elemzése és a szabvány mikron-pontosságú vizsgálódása között. A tankönyveknek kellene ezt a rést betölteniük, de azok általában a szabvány távlatai felé tendálnak, így sokkal nagyobb figyelmet kap az, hogy mi a nyelv, mint az, hogy miért olyan.

Nézzük az ARM-et. Az ARM egy másik – Margaret Ellis és Bjarne Stroustrup által írt – könyv, a *The Annotated C++ Reference Manual* (Addison-Wesley, 1990.). Már megjelenésekor „a C++ forrásmű"-vé vált, és a nemzetközi szabvány is az ARM-ből indult ki (és a meglévő C szabványból). Az azóta eltelt években a szabványban meghatározott nyelv valamennyire eltért az ARM-ben leírtaktól, így az ARM már nem az a szaktekintély, ami valaha volt. Most

is hasznos hivatkozási alap azonban, mert amit mond, annak a nagy része még mindig igaz. És az sem ritka, hogy a C++ fordítók szállítói a C++-nak azokon a területein, ahol a szabvány csak mostanában tisztult le, még ragaszkodnak az ARM specifikációjához.

Ami az ARM-et igazán hasznossá teszi, az nem az RM (*Reference Manual* – a kézikönyv) része, hanem az A rész, a jegyzetek (Annotations). Az ARM kimerítő magyarázatokat közöl arról, hogy a C++ sok eleme miért úgy viselkedik, ahogy. Ezen információk egy része megtalálható a D&E-ben, de a többsége nem, mi viszont meg *akarjuk* ismerni azokat is. Itt van például valami, ami megőrjíti az embereket, amikor először találkoznak vele:

```
class Base {
public:
   virtual void f(int x);
};
class Derived: public Base {
public:
   virtual void f(double *pd);
};
Derived *pd = new Derived;
pd->f(10);                    // Hiba!
```

Az a baj, hogy a `Derived::f` eltakarja a `Base::f`-et, annak ellenére, hogy különböző paramétertípust vesznek fel, így aztán a fordítók azt várják, hogy `f` hívása egy `double*`-ot kap paraméterül, a `10` literál viszont bizonyosan nem az.

Ez így kényelmetlen, de az ARM magyarázatot ad erre a viselkedésre. Tegyük fel, hogy az `f` hívásakor mi tényleg a `Derived`-beli változatot akartuk meghívni, de véletlenül rossz paraméter típust adtunk meg! Tegyük fel továbbá, hogy a `Derived` nagyon mélyen van az öröklődési hierarchiában, és mi nem is tudtuk, hogy a `Derived` valamilyen `BaseClass` bázisosztály távoli leszármazottja, és hogy a `BaseClass` deklarál egy olyan `f` nevű virtuális függvényt, amely egy `int` paramétert vár! Ebben az esetben szándékaink ellenére a `BaseClass::f`-et hívnánk meg, egy olyan függvényt, amelynek a létezéséről nem is tudtunk! Az ilyen típusú hiba ott fordulhatna elő gyakran, ahol nagy osztályhierarchiákat használnak. Így aztán Stroustrup úgy döntött, hogy csírájában elfojtja a hibalehetőséget azzal, hogy a származtatott osztályok tagjai névegyezés alapján takarják el a bázisosztály tagjait.

Vegyük észre azt is, hogy ha a `Derived` írója elérhetővé akarja tenni a `Base::f`-et a felhasználók számára, akkor ezt könnyen megteheti a `using` deklaráció használatával:

```
class Derived: public Base {
public:
   using Base::f;             // Base::f importálása a
                              // Derived hatókörébe.
   virtual void f(double *pd);
};
Derived *pd = new Derived;
pd->f(10);                 // Jó, Base::f-et hívja.
```

Azon fordítók esetén, amelyek még nem támogatják a `using` deklarációkat, egy inline függvény használata jó alternatívának bizonyulhat:

```
class Derived: public Base {
public:
   virtual void f(int x) { Base::f(x); }
   virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                  // Jó, Derived::f(int)-et hívja, amely
                            // meghívja Base::f(int)-et.
```

A D&E és az ARM oldalait lapozgatva olyan ismereteket szerezhetünk a C++ tervezéséről és implementációjáról, amelyekkel felvértezve értékelni tudjuk a barokkos külső mögött megbúvó egészséges, „nem is értelmetlen" architektúrát. Erősítsük meg ezeket az ismereteket a szabvány részletes információival, és olyan szoftverfejlesztési tudás alapjait fektethetjük le, amely az igazán *hatékony C++*-hoz vezet.

Utószó

Ha az olvasónak sikerült megemésztenie ezt az 50 jó tanácsot, amellyel javíthat programterveín és programjain, de még mindig nem csillapodott a C++ irányelvek utáni étvágya, lehet, hogy érdekelni fogja az e témakörben írt második könyvem, a *More Effective C++: 35 New Ways to Improve Your Programs and Designs* (Még hatékonyabb C++: 35 újabb jó tanács programjaink és programterveink javítására). A Hatékony C++-hoz hasonlóan a *More Effective* C++ is olyan témákat dolgoz fel, amelyek elengedhetetlenek a hatékony szoftverfejlesztéshez C++-ban. A *Hatékony C++* inkább az alapokra összpontosít, a *More Effective C++* viszont kitér a nyelv új elemeire és a haladó programozási technikákra is.

A *More Effective C++*-ról részletes információ – négy teljes jó tanács, a könyvben ajánlott könyvek listája, és még sok minden más – található a *More Effective C++* weboldalán, a `http://www.awl.com/cp/mec++.html` címen[16]. Azoknak a kedvéért, akiknek kifúrja az oldalát a kíváncsiság, álljon itt a *More Effective C++* tartalomjegyzéke:

Alapok

1. jó tanács: Tegyünk különbséget a pointerek és a referenciák között
2. jó tanács: Részesítsük előnyben a C++ stílusú konverziókat
3. jó tanács: Soha ne kezeljük a tömböket polimorfikusan
4. jó tanács: Kerüljük a fölösleges alapértelmezett konstruktorokat

Operátorok

5. jó tanács: Ne bízzunk a felhasználó által definiált konverziós függvényekben
6. jó tanács: Tegyünk különbséget a növelő és csökkentő operátor prefix és postfix alakja között
7. jó tanács: Soha ne terheljük túl a `&&`-t, a `||`-t, és a ,-t
8. jó tanács: Legyünk tisztában a `new` és a `delete` különböző jelentéseivel

[16] Scott Meyers More Effective C++: 35 New Ways to Improve Your Programs and Designs című könyve angolul már megjelent: Addison-Wesley, 1996. Az itt közölt címen már nem található meg a könyv weboldala. – A szerk..

Kivételek

9. jó tanács: Az erőforrás lyukak kialakulásának megakadályozására használjuk a destruktorokat
10. jó tanács: Előzzük meg erőforrás lyukak kialakulását a konstruktorokban
11. jó tanács: Előzzük meg azt, hogy a kivételek elhagyhassák a destruktorokat
12. jó tanács: Tudjuk, miben különbözik egy kivétel dobása egy paraméter átadásától vagy egy virtuális függvény hívásától
13. jó tanács: A kivételeket referencia szerint kapjuk el
14. jó tanács: Használjuk a kivétel-specifikációkat megfontoltan
15. jó tanács: Legyünk tisztában a kivételkezelés költségeivel

Hatékonyság

16. jó tanács: Ne feledjük a 80-20-as szabályt
17. jó tanács: Vegyük fontolóra a lusta kiértékelés használatát
18. jó tanács: Csökkentsük a várható számítások költségét
19. jó tanács: Legyünk tisztában az ideiglenes objektumok keletkezésével
20. jó tanács: Segítsük elő a visszatérési érték optimalizálást
21. jó tanács: Az implicit típuskonverziók elkerülésére alkalmazzunk túlterhelést
22. jó tanács: Az op használata helyett vegyük fontolóra az `op=` használatát
23. jó tanács: Fontoljuk meg az alternatív könyvtárak használatát
24. jó tanács: Legyünk tisztában a virtuális függvények, a többszörös öröklődés, a virtuális bázisosztályok és az RTTI költségeivel

Technikák

25. jó tanács: Konstruktorok és nemtagfüggvények virtuálissá tétele
26. jó tanács: Az objektumok számának korlátozása egy osztályban
27. jó tanács: Halom alapú objektumok megkövetelése vagy letiltása
28. jó tanács: Okos pointerek
29. jó tanács: Referenciaszámlálás
30. jó tanács: Proxy osztályok
31. jó tanács: Függvények virtuálissá tétele egynél több objektumra nézve

Egyebek

32. jó tanács: Programozzunk jövő időben
33. jó tanács: Tegyük absztrakttá a nemlevél osztályokat
34. jó tanács: Legyünk tisztában azzal, hogyan lehet a C++-t és a C-t ugyanazon a programon belül kombinálni
35. jó tanács: Ismerkedjünk meg a nyelv szabványával

Ajánlott irodalom

Egy `auto_ptr` implementáció

viselkedés
definiálatlan, lásd definiálatlan viselkedés
testreszabása virtuáli függvényeken keresztül 218–219
visszatérési érték élettartama 143–144
visszatérési típus
`const` 108, 141
leírók elérhetetlen adattagokra 140
`operator=`-nek 82
`operator[]`-nek 109
`void*` pointerek, lásd generikus pointerek
`vptr` 79
`vtbl` 79, 81

W

Wait, John 16
Weaver, Sarah 16
web site-ja a *More Effective* C++-nak 249
web site-ja ennek a könyvnek 11
When Bad Things Happen to Good People, utalás 195
Wildlife Presentation Trust International (WPTI)
lásd `http://www.columbia.edu/cu/cerc/wpti.html`
Williams, Russ 15
WPTI, lásd Wildlife Presentation Trust International
Wright, Kathy 16
WWW site-ja a *More Effective* C++-nak 249
WWW site-ja a ennek a könyvnek 11

X

X akták 80
utalás 69
XYZ légitársaság 178, 179

Z

Zabluda, Oleg 15
Zell, Adam 15

Mini szótár (angol–magyar)

access restriction – hozzáférési megszorítás
address-of operator – cím operátor
alias – álnév
aliasing – álnevesítés
ambiguity – többértelműség
argument – paraméter
array – tömb
assertion – állítás
assignment – értékadás
assignment operator – értékadó operátor
assignment to self – önértékadás

bitwise constness – bitenkénti konstansság
bitwise copy – bitenkénti másolás
breakpoint – töréspont
built-in – beépített

cache – gyorsítótár
cast – konverzió
casting – konvertálás
class template – osztálysablon
client – felhasználó
code bloat – kódduzzadás
conceptual constness – fogalmi konstansság
construction – létrehozás
constructor – konstruktor
container class – tároló osztály
conversion operator – konverziós operátor
copy constructor – másoló konstruktor

dangling pointer – gyökértelen pointer
data member – adattag
declaration – deklaráció
default – alapértelmezett
default constructor – alapértelmezett konstruktor
definition – definíció
deque – kétvégű sor
dereferenced – dereferenciált
derived – származtatott
destruction – felszámolás
destructor – destruktor
downcasting – bázisirányú konverzió, lefelé konvertálás
dynamic type – dinamikus típus
dynamically bound – dinamikusan kötött
enumerator – felsoroló típus
exception – kivétel
exception handling – kivételkezelés

file – fájl
fragmentation – töredezettség
friend – barát
friend function – barátfüggvény
free list – szabad lista
function – függvény
function overloading – függvénytúlterhelés

global – globális
global variable – globális változó

handle – leíró
has-a relation – „vanegy" kapcsolat
hash table – hasító tábla
header file – fejállomány
heap – halom
hidden – eltakart

implementation – implementáció
inheritance – öröklődés, származás
initialization – inicializálás, kezdeti értékadás
initialization argument – inicializáló paraméter
inlining – függvények helyben kifejtése, inline-olás
inner scope – belső hatókör
instantiate – példányosít
instantiation – példányosítás
interface – felület
interface class – felületosztály
invariant over specialization – specializáció feletti invariáns
isa relation – „azegy" kapcsolat
is-implemented-in-terms-of relation – keresztül implementált kapcsolat
iterate – bejár
iterator – bejáró

layering – rétegelés, beágyazás
library – könyvtár
linked list – láncolt lista
local – lokális
local variable – lokális változó

map – asszociatív tömb
member – tag
member function – tagfüggvény
member initialization list – taginicializáló lista
memberwise copy – tagonkénti másolás
memory leak – memória szivárgás
mixin-style – kevert stílusú
multiple inheritance – többszörös öröklődés

name conflict – névütközés
named array – nevesített tömb
namespace – névtér
nested – beágyazott
newsgroup – hírcsoport
non-local – nemlokális
non-member function – nemtagfüggvény
non-public – nempublikus, nemnyilvános
nonvirtual – nemvirtuális

object – objektum
object assignment – objektum-értékadás
object code – tárgykód
object file – tárgykód fájl
operation – művelet
operator – operátor
overload – túlterhelés
overloaded – túlterhelt
outer scope – külső hatókör

pass-by-reference – referencia szerinti átadás
pass-by-value – érték szerinti átadás
pointer aliasing – álnevesítő pointer
preprocessor – előfordító, preprocesszor
private – privát
protected – védett
protection level – védettségi szint
public – publikus, nyilvános
pure virtual – tisztán virtuális

qualification – minősítés
qualified name – minősített név
queue – sor

rebuild – újrafordítás, újraépítés
recursion – rekurzió
redeclare – újradeklarálni
reference – referencia
runtime error – futási idejű hiba

scope – hatókör
simple virtual – egyszerű virtuális
single inheritance – egyszeres öröklődés
slicing – szeletelődés
stack – verem
static – statikus
statically bound – statikusan kötött
static type – statikus típus
stream – folyam
string – sztring
struct – struktúra
subclass – alosztály
template – sablon
temporary – ideiglenes
type – típus
type conversion – típuskonverzió
typedef – típusdefiníció
type-safe – típushelyes

unqualified name – minősítés nélküli név
undefined behavior – definiálatlan viselkedés
unnamed – névtelen
variable – változó
virtual – virtuális
virtual table – virtuális tábla
virtual table pointer – virtuális táblapointe

Mini szótár (magyar–angol)

adattag – data member
alapértelmezett – default
alapértelmezett konstruktor – default constructor
állítás – assertion
álnév – alias
álnevesítés – aliasing
álnevesítő pointer – pointer aliasing
alosztály – subclass
asszociatív tömb – map
„azegy” kapcsolat – isa relation

barát – friend
barátfüggvény – friend function
bázisirányú konverzió – downcasting
beágyazás – layering
beágyazott – nested
beépített – built-in
bejár – iterate
bejáró – iterator
belső hatókör – inner scope
bitenkénti konstansság – bitwise constness
bitenkénti másolás – bitwise copy

cím operátor – address-of operator

deklaráció – declaration
definiálatlan viselkedés – undefined behavior
definíció – definition
dereferenciált – dereferenced
destruktor – destructor
dinamikus típus – dynamic type
dinamikusan kötött – dynamically bound

egyszeres öröklődés – single inheritance
egyszerű virtuális – simple virtual
előfordító – preprocessor
eltakart – hidden
értékadás – assignment
értékadó operátor – assignment operator
érték szerinti átadás – pass-by-value

fájl – file
fejállomány – header file
felhasználó – client
felsoroló típus – enumerator
felszámolás – destruction
felület – interface
felületosztály – interface class
fogalmi konstansság – conceptual constness
folyam – stream

futási idejű hiba – runtime error
függvény – function
függvények helyben kifejtése – inlining
függvénytúlterhelés – function overloading

globális – global
globális változó – global variable
gyorsítótár – cache
gyökértelen pointer – dangling pointer

halom – heap
hasító tábla – hash table
hatókör – scope
hírcsoport – newsgroup
hozzáférési megszorítás – access restriction

ideiglenes – temporary
implementáció – implementation
inicializálás – initialization
inicializáló paraméter – initialization argument
inline-olás – inlining

keresztül implementált kapcsolat –
is-implemented-in-terms-of relation
kétvégű sor – deque
kevert stílusú – mixin-style
kezdeti értékadás – initialization
kivétel – exception
kivételkezelés – exception handling
kódduzzadás – code bloat
konstruktor – constructor
konvertálás – casting
konverzió – cast
konverziós operátor – conversion operator
könyvtár – library
külső hatókör – outer scope

láncolt lista – linked list
lefelé konvertálás – downcasting
leíró – handle
létrehozás – construction
lokális – local
lokális változó – local variable

másoló konstruktor – copy constructor
memória szivárgás – memory leak
minősítés – qualification
minősítés nélküli név – unqualified name
minősített név – qualified name
művelet – operation

nemlokális – non-local
nemnyilvános – non-public
nempublikus – non-public
nemtagfüggvény – non-member function
nemvirtuális – non-virtual
nevesített tömb – named array
névtelen – unnamed
névtér – namespace
névütközés – name conflict
nyilvános – public

objektum – object
objektum-értékadás – object assignment
operátor – operator
osztály – class
osztálysablon – class template
önértékadás – assignment to self
öröklődés – inheritance

paraméter – argument
példányosít – instantiate
példányosítás – instantiation
preprocesszor – preprocessor
privát – private
publikus – public

referencia – reference
referencia szerinti átadás – pass-by-reference
rekurzió – recursion
rétegelés – layering

sablon – template
sor – queue
specializáció feletti invariáns – invariant over specialization
statikus – static
statikusan kötött – statically bound
statikus típus – static type
struktúra – struct

szabad lista – free list
származás – inheritance
származtatott – derived
szeletelődés – slicing
sztring – string

tag – member
tagfüggvény – member function
taginicializáló lista – member initialization list
tagonkénti másolás – memberwise copy
tárgykód – object code
tárgykód fájl – object file
tároló osztály – container class
típus – type
típusdefiníció – typedef
típushelyes – type-safe
típuskonverzió – type conversion
tisztán virtuális – pure virtual
többértelműség – ambiguity
többszörös öröklődés – multiple inheritance
tömb – array
töredezettség – fragmentation
töréspont – breakpoint
túlterhelés – overload
túlterhelt – overloaded

újradeklarálni – redeclare
újraépítés – rebuild
újrafordítás – rebuild

változó – variable
„vanegy” kapcsolat – has-a relation
védett – protected
védettségi szint – protection level
verem – stack
virtuális – virtual
virtuális tábla – virtual table
virtuális táblapointer – virtual table pointer