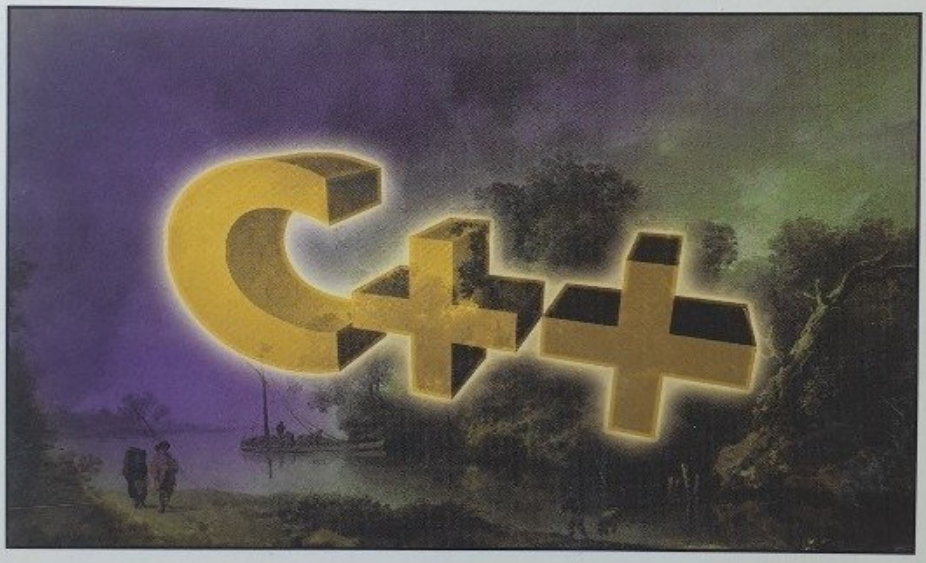
Stephen C. Dewhurst

C++ hibaelhárító



KISKAPU

Addison-Wesley

Stephen C. Dewhurst

C++ hibaelhárító

Addison-Wesley

Tartalomjegyzék

Előszó

Ez a könyv csaknem két évtized apró frusztrációinak, súlyos hibáinak, valamint a miattuk átvirrasztott éjszakáknak és programozással töltött hétvégéknek az eredménye. Összegyűjtöttem benne a C++ programozási nyelvvel kapcsolatos 99 leggyakoribb, legsúlyosabb vagy legérdekesebb hibát, amelyeket (sajnálom, hogy ezt kell mondanom, de így igaz) jobbára én magam is elkövettem.

A könyvben végig használt „hiba" kifejezés meghatározása kissé ködös, sőt nem is teljesen egyértelmű. A legpontosabb megfogalmazás talán az, hogy ezek a bizonyos „hibák" a C++ nyelvű programozás során elkövethető és feltétlenül kerülendő tévedések. Persze ezek skálája az apró formai tévedésektől az alapvető tervezési problémákon át egészen a „közveszélyes" viselkedésig terjed.

Körülbelül tíz évvel ezelőtt elkezdtem efféle hibákkal kiegészíteni a C++ nyelvről tartott előadásaim jegyzeteit. Úgy éreztem, ha látványosan szembeállítom ezeket az általános hibákat a helyes megoldásokkal, azzal „beoltom" az egyetemi hallgatókat ellenük, és megakadályozom, hogy a programozók egy újabb nemzedékének ugyanazoktól a hibáktól kelljen szenvednie. Az elképzelés nagyjában-egészében működött, ezért elkezdtem csoportokba szervezni a hasonló programozási hibákat, eredetileg azzal a céllal, hogy konferenciákon „mutogathassam" őket. Ezek az előadások is népszerűnek bizonyultak (úgy tűnik, a baj vonzza az embereket) és többen arra bátorítottak, hogy előadásaim anyagát könyv formájában foglaljam össze. A C++-szal kapcsolatos hibák, illetve elkerülésük tárgyalása szinte mindig összefügg más témákkal. Ilyenek például az általános tervezési elvek, a nyelvjárások, valamint a C++ működésével kapcsolatos műszaki részletek.

Ez a könyv alapvetően nem a tervezési módszerekről szól, viszont a legtöbb programozási hiba tárgyalása során valamilyen tervezési módszernél lyukadunk ki. E módszerek nevét mindenütt nagy kezdőbetűvel írtam, mint például a „Sablonok módszere" vagy a „Híd módszer". A megfelelő helyeken megemlítem a kérdéses tervezési módszert, majd – feltéve, hogy viszonylag egyszerű – leírom a módszer lényegét is, a részletes tárgyalást azonban minden esetben olyan könyvekre hagyom, amelyek kifejezetten ezzel a témával foglalkoznak. Hacsak másként nem említem, az összes tervezési módszer részletes leírása megtalálható Erich Gamma és szerzőtársai *Design patterns (Tervezési módszerek)* című könyvében. A „Nem ciklikus felülvizsgálat" az „Egyenállapot módszere", valamint a „Null objektumok módszere" leírása Robert Martin *Agile Software Development (Gyors szoftverfejlesztés)* című könyvében található meg. A programozási hibák szemszögéből nézve a tervezési módszereknek két fontos tulajdonságát kell kiemelnünk. Először is ezek

bizonyítottan sikeres tervezési módszereket adnak, amelyek „környezetfüggő módon" könnyen átvihetők bármilyen konkrét programozási feladatra. Másodsorban – bár esetünkben talán ez a fontosabb – egy adott tervezési módszer megemlítése nemcsak az alkalmazott megoldást „dokumentálja", hanem az esetek többségében arra is rávilágít, hogy miért pont ezt a módszert alkalmaztuk és milyen hatása volt ennek a teljes munkafolyamatra.

Ha például azt olvassuk valahol, hogy a tervezés során a szerző a „Híd módszert" használta, egyrészt rögtön tudjuk, hogy elvont adattípusokat hozott létre, de ezek tényleges megvalósítását egy megfelelő felület (interfész) segítségével „elszigetelte" a felhasználótól. Ez egyrészről hasznos, mivel a belső megvalósítás módosítása a felület felhasználóit egyáltalán nem befolyásolja. Ugyanakkor az is nyilvánvaló, hogy ezért a függetlenségért a futásidőben igényelt számítási teljesítmény növekedésével kell fizetnünk, az elvont adattípusokat leíró forráskódot pedig valamivel nehezebb lesz áttekinthetően elrendezni. És ez még csak két lényeges részlet azok közül, amelyekre egy ilyen tervezési minta kihathat. Egy tervezési módszer neve tehát rengeteg tervezéssel kapcsolatos információt és előzetes technikai tapasztalatot takar. Ha a módszer elnevezéseit és elveit következetesen és körültekintően alkalmazzuk mind a munka, mind annak dokumentálása során, azzal számos hiba felbukkanását eleve elkerülhetjük.

A C++ kétségtelenül összetett programozási nyelv. Márpedig minél összetettebb egy nyelv, annál nagyobb jelentőséget kapnak a sajátos kifejezésmódok (nyelvi változatok, nyelvjárások, idiómák – vagyis a stílus). A stílus az alacsony szintű nyelvi elemek kombinálásának olyan módja, amelynek a magasabb szintű tervezésre nézve alapvető hatása van. Ez ugyanolyan összefüggés, mint amilyen az alacsonyabb és magasabb szintű tervezési módszerek, minták között fedezhető fel. Ennek megfelelően a C++ nyelvvel kapcsolatban anélkül tárgyalhatjuk például a másolási műveleteket, a függvényobjektumokat, az „okos" mutatókat, vagy a kivételkezelést, hogy konkrétan megadnánk vagy ismernénk e műveletek és elemek alacsonyszintű megvalósítását.

Fontos kihangsúlyozni, hogy a stílus nemcsak a nyelvi elemek kombinálási módját, hanem a felhasználó által elvárt hatást is magában foglalja. A stílus tehát pontosan leírja, mit várhatunk egy másolási művelettől, vagy mi történik, ha kivétel keletkezik a program végrehajtása során. Az ebben a könyvben található legtöbb tanács a stílussal – annak figyelembe vételével és alkalmazásával – kapcsolatos. Az itt leírt hibák közül számos egyszerűen egy adott stílustól való eltérést jelent, a hiba elkerülésére vagy javítására pedig nem nagyon tudunk mást mondani, mint hogy alkalmazzuk az adott stílust. (Lásd például a 10. hibát.)

A könyv jelentős részét a C++ nyelv egyes területeinek aprólékos, részletekbe menő leírásának szenteltem. Tettem ezt azért, mert gyakran ezek az árnyalatnyi eltéré-

sek azok, amelyek a legtöbb félreértést és hibát eredményezik. Bár ezek a részek időnként „ezoterikusnak" tűnhetnek, e finom részletek ismeretének hiánya számos probléma forrása lehet, és mindenképpen meggátolja az Olvasót abban, hogy a C++ nyelvet mesterfokon használhassa. E „sötét sarkok" felderítése önmagában is érdekes kaland. Bármennyire meglepő, ezek nem gondatlanságból vagy a programozó bosszantására kerültek be a nyelv leírásába. Mindnek igen jó oka van, és a gyakorlott programozók a megmondhatói, hogy különleges helyzetekben mennyire hasznosak tudnak lenni.

A programozási hibák és tervezési módszerek egy másik kapcsolódási pontja a viszonylag egyszerű dolgok pontos leírása. Az egyszerű tervezési módszerek nagyon fontosak. Gyakran fontosabbak még a nagy és összetett eljárásoknál is, mivel – éppen egyszerűségük miatt – sokkal gyakrabban alkalmazzuk őket. Egy jól meghatározott, ám egyszerű tervezési módszernek tehát a teljes munkára sokkal nagyobb hatása lehet, mivel sokkal több helyen bukkanhat fel. Hasonló módon a könyvben tárgyalt hibák a lehetséges tévedések széles spektrumát ölelik fel. Ennek megfelelően a megoldások és tanácsok is az egyszerű intelmektől a súlyos félreértések kihangsúlyozásáig terjednek. Az elsőre a 12. hiba példa, ahol a tanulság csupán annyi, hogy ha lehet, kerüljük el a profizmus látszatát, ha nem vagyunk biztosak a dolgunkban. A 79. leckében már valami sokkal fontosabbról, a virtuális öröklődés elsőbbségi viszonyairól és azok gyakori félreértéséről esik szó. Ugyanakkor a kisebb hiba („szakértőt játszunk") sokkal gyakoribb a mindennapi életben, mint az utóbbi, sokkal súlyosabb hiba.

Két fontos szempont az egész tárgyalás során újra és újra felbukkan. Az egyik a hagyományok tiszteletben tartása. Ennek különösen nagy jelentősége van egy olyan összetett nyelv esetében, mint a C++. Ha ragaszkodunk a hagyományokhoz, az áttekinthetővé teszi munkánkat, és ez mindenképpen megkönnyíti a többi programozóval való együttműködést. A másik lényeges dolog annak felismerése és észben tartása, hogy programunkat nem feltétlenül mi magunk fogjuk használni vagy karbantartani, hanem más programozók. A karbantarthatóság biztosítása lehet közvetlen vagy közvetett: az előbbi esetben arra kell ügyelnünk, hogy az általunk írt kódot bármely általános képzettséggel és tapasztalatokkal rendelkező programozó megérthesse, a közvetett karbantarthatóság pedig azt jelenti, hogy előre gondoskodunk a kód helyes működéséről, akkor is, ha azt vagy annak környezetét a jövőben valaki módosítani fogja.

A könyvben bemutatott hibatípusok leírása mindig egy-egy rövid „esszé", amely körülírja magát a jelenséget vagy jelenségcsoportot, valamint ötleteket ad a kérdéses hiba elkerülésére, illetve kijavítására. Azt hiszem, bármelyik ehhez hasonló, hibakereséssel foglalkozó könyv szükségszerűen kissé szétfolyó, egyszerűen a feldolgozott téma „anarchisztikus" természete miatt. Ugyanakkor megkíséreltem

a logikailag összetartozó hibákból fejezeteket képezni. A rendezőelv a természetes hasonlóság és a félreértések közös eredete volt.

Egy-egy hiba tárgyalása során gyakran elkerülhetetlenül más lehetséges hibák bukkannak fel. Ezeket a kapcsolatokat, ahol ez lehetséges volt, igyekeztem megemlíteni. Ugyanakkor érzésem szerint egy-egy önállónak tekintett téma belső összetartó ereje időnként meglehetősen meglazult. Gyakran volt szükség arra, hogy egy hiba tényleges tárgyalása előtt bemutassam azt a környezetet, amelyben a kérdéses probléma egyáltalán felmerülhet. Ez a bevezetés azonban úgyszintén gyakran igényli egy-egy tervezési módszer, nyelvi finomság, eljárás vagy írásmód bemutatását, ami aztán igen sok előzetes információt eredményez, mielőtt a címben említett hibára rátérhetnénk. Amennyire ez egyáltalán lehetséges volt, igyekeztem az efféle kanyarokat levágni. Ugyanakkor azt gondolom, hogy ezek teljes mellőzése nagyban csökkentette volna a könyv értékét és használhatóságát. Ha hatékonyan akarjuk a C++ nyelvet használni, eleve számos különböző területre, témára kell egyszerre összpontosítanunk, ezért tévedés volna azt gondolni, hogy hasznos tanácsokat adhatunk gyakorló programozóknak anélkül, hogy legalább egy kicsit belebonyolódnánk az efféle eklektikus témák boncolgatásába.

Természetesen szükségtelen – sőt attól tartok, kifejezetten ellenjavallt – ezt a könyvet elejétől a végéig, az első hibától a kilencvenkilencedikig végigolvasnunk. Ez felérne egy súlyos testi sértéssel, és egyeseket örökre eltántoríthatna a C++ használatától. Sokkal használhatóbb módszer, ha csak azokkal a hibákkal foglalkozunk, amelyekbe ténylegesen beleütközünk, amelyek hasonlítanak az általunk tapasztaltakra, vagy egyszerűen amelyeket érdekesnek találunk valamiért. Ezek a leírások aztán úgyis tartalmaznak majd utalásokat számos más kapcsolódó hibalehetőségre. Egy másik lehetséges módszer az, hogy véletlenszerűen „mazsolázunk" a leírtakból.

A szövegben alkalmaztunk néhány, a megértést könnyítő jelölést. A hibás vagy csupán „stilárisan sérült" programrészletek háttere mindig szürke, míg a helyes kódnak nincs háttere. A szövegben előforduló kódrészletek leírásánál a cél az áttekinthetőség és nem a teljesség volt. Ez egyben azt is jelenti, hogy ezek a programrészletek nem fordíthatók le anélkül, hogy teljes programmá egészítenénk ki őket. A nem triviális kódrészletek elektronikus formában is hozzáférhetők a szerző weblapján, melynek címe *www.semantics.org*, illetve a *www.kiskapu.hu/kiado* címről is letölthetők. A szövegben minden ilyen kódrészlet mellett szerepel az elérési útvonal és a fájl neve is.

Végezetül azt hiszem, kívánkozik ide még egy fontos figyelmeztetés. Egyetlen dolgot ne tegyünk soha az itt felsorolt hibajavításokkal: nem emeljük a stílus vagy a programozási minták szintjére, vagyis ne próbáljuk őket gondolkodás nélkül használni. Annak a legbiztosabb jele, hogy valóban jól alkalmazzuk a különböző

stílusokat az, hogy azok természetes módon, önmaguktól bukkannak fel a megfelelő helyeken, nem azért, mert erőltettük a megjelenésüket. A hibák felismerésének készsége amúgy nagyban hasonlít a veszélyhelyzetek kezeléséhez: aki egyszer megégette magát, az másodszorra már fél a tűztől. Ugyanakkor nem feltétlenül kell megégetnünk vagy főbe lőnünk magunkat ahhoz, hogy képesek legyünk elkerülni a tűz vagy a lőfegyverek okozta veszélyeket. Általában az is elég, ha valaki felhívja a figyelmünket a lehetséges buktatókra. Ez a könyv éppen ezt a figyelmeztető szerepet igyekszik betölteni a C++ nyelvvel kapcsolatban.

Stephen C. Dewhurst
Carver, Massachusetts

Köszönetnyilvánítás

Gyakori jelenség, hogy a szerkesztők könyveik köszönetnyilvánításában valami ilyesféle köszönetet kapnak szeretett szerzőjüktől: „... és köszönetemet fejezem ki szerkesztőmnek is, aki gondolom szintén csinált valamit, amíg én a könyv megírásán robotoltam." E könyv megszületése elsősorban szerkesztőm, Debbie Lafferty lelkén szárad. Egyszer régen megkerestem őt egy őszintén szólva középszerű programozási tankönyv úgyszintén középszerű tervezetével. Ennek megírása helyett ő azt javasolta, hogy az egyik szakaszát, ami történetesen a programozási hibákról szólt, részletezzem kicsit tovább, amíg könyv nem lesz belőle. Én természetesen kapásból megtagadtam ezt. Ő meg természetesen kitartott. És győzött. Szerencsére Debbie meglehetősen irgalmas győztes, így minden joga megvan rá, hogy egy szerkesztői „na ugye megmondtam" legyen a jutalma. Ja, egyébként eléggé úgy fest, hogy ő is csinált valamit, amíg én a kézirat megírásán robotoltam.

Szintén köszönettel tartozom lektoraimnak, akik nem kevés idejüket és szaktudásukat áldozták arra, hogy ez a könyv még jobb lehessen. Egy távolról sem tökéletes kézirat lektorálása meglehetősen időt rabló, gyakran unalmas, néha idegesítő, és összességében meglehetősen háládatlan feladat. Amolyan áldozat a szakértők részéről (lásd a 12. hibát), ezért lektoraim éleslátásról tanúskodó, ám néha metsző kritikáját mindvégig türelemmel fogadtam. Steve Clamage, Thomas Gschwind, Brian Kernighan, Patrick McKillen, Jeffrey Oldham, Dan Saks, Matthew Wilson és Leor Zolman voltak azok, akik tanácsokkal, szakmai megjegyzésekkel, kódrészletekkel és javításokkal segítették munkámat, vagy éppen rámutattak egy-egy téves megállapításomra.

Kétségtelenül Leor volt az első lektorom, ő ugyanis már jóval a kézirat megírása előtt számos „letisztított" hibát küldött nekem, amiket a webes levelezési listákon talált. Gyakran ezek képezték a könyvben bemutatott problémák első változatát. Sarah Hewins, legjobb barátom és legkeményebb kritikusom mindkét említett cí-

mére rászolgált, miközben a kézirat különböző változatait olvasta és javítgatta. David R. Dewhurst gyakran segített nekem abban, hogy bizonyos dolgokat a megfelelő szemszögből vizsgáljak. Greg Comeau rendelkezésemre bocsátotta kiváló C++ fordítóját, hogy azzal ellenőrizhessem a bemutatott kódokat.

Mint bármelyik, a C++ nyelvről szóló „nem triviális" könyv, az enyém is számos ember munkájának eredménye. A hosszú évek során számos tanítványom, kollégám és ügyfelem gyarapította az általam ismert programozási hibák sorát. Ezeket az apróbb dolgokat persze lehetetlen volna egyenként megköszönni ezen a helyen, lehetőségem van azonban arra, hogy a nagyobb hozzájárulások „tulajdonosait" név szerint is megemlítsem.

A 11. hiba leírásában szereplő `Select` sablon, valamint a 70. hiba kapcsán említett `OpNewCreator` szabály Andrei Alexandrescu *Modern C++ Design* (*Modern C++ tervezés*) című könyvében jelent meg először. Az állandókra vonatkozó hivatkozások visszaadásával kapcsolatos problémával (lásd a 44. hibát) a Cline és szerzőtársai által írt *C++ FAQ*-ban találkoztam először. Szintén ebben a műben találtam a 73. hiba kapcsán említett módszert a túlterhelt virtuális függvények megkerülésére. A 83. hiba leírásában említett `Cptr` sablon a Nicolai Josutti által írt, *The C++ Standard Library (A C++ szabványos könyvtára)* című könyvben említett `CountedPtr` egy változata.

Scott Meyer tudna bővebben mesélni a `&&`, `||` és `,` (vessző) műveleti jelek nem megfelelő túlterheléséről, amiről a 14. hiba leírásában esik szó. Ő ezt a témát a *More Effective C++* (*Még hatékonyabb C++*) című könyvében érinti. Szintén ő részletezi a bináris műveletek által visszaadott értékek létezésének szükségességét (lásd a 58. hibát), *Effective C++* (*Hatékony C++*) című könyvében. A 68. hiba kapcsán említett, az `auto_ptr` nem megfelelő használatával kapcsolatos problémát is ő említette először *Effective STL* (*Hatékony STL*) című művében. Végül a 87. hiba kapcsán említett, a növelő és csökkentő műveletekkel kapcsolatos megoldás is nála jelenik meg először a már említett, *More Effective C++* című könyvében.

Dan Saks mutatta meg nekem először, miért olyan fontos a 8. hiba kapcsán említett előzetes bevezetés. Szintén ő mutatta meg nekem először a 17. hiba kapcsán említett „testőr operátort", és ő győzött meg arról is, hogy felesleges a felsoroló típusokkal kapcsolatos növelésnél a tartományellenőrzés (87. hiba).

Herb Sutter *More Exceptional C++* (*Még különlegesebb C++*) könyvének 36. szakasza indított arra, hogy újra elolvassam a C++ szabvány 8.5. pontját, és átértékeljem a paraméterek formális előkészítéséről alkotott nézeteimet (57. hiba).

A 10., 27., 32., 33., 38-41., 70., 72-74., 89., 90., 98. és 99. hiba első változatát a *C++ Report* Common Knowledge című részében, illetve később a *The C/C++ Users Journal*-ben magam jelentettem meg.

1

Alapvető hibák

Az, hogy egy hiba alapvető, nem azt jelenti, hogy nem súlyos, vagy hogy nem gyakori. Ami azt illeti, az ebben a fejezetben tárgyalt alapvető hibák nagyobb aggodalomra adnak okot, mint a későbbi fejezetekben bemutatandó műszakilag bonyolultabbak. Ennek pedig egyszerűen az az oka, hogy e hibák éppen alapvető természetükből adódóan bármely C++ kódban előfordulhatnak.

1. hiba: Túl sok megjegyzés használata

A forráskódokban szereplő megjegyzések közül nagyon sok szükségtelen. Nehezítik a kód olvasását és megértését, sőt gyakran félre is vezetik az olvasót. Nézzük például a következő programsort:

```
a = b;  // b értékét a-hoz rendeljük
```

Ez a megjegyzés semmi többet nem nyújt, mint a kód maga, vagyis szükségtelen. Ami azt illeti, nem is egyszerűen szükségtelen, inkább kifejezetten rossz. Rettenetes! Először is elvonja az olvasó figyelmét a tényleges kódról, így az több szöveget lesz kénytelen átnyálazni ahhoz, hogy kihámozza belőle a kód értelmét. Másodszor az ilyen megjegyzéseket a kóddal együtt kell karbantartanunk és szükség esetén módosítanunk, vagyis megnöveltük saját teendőink számát. Harmadszor, a megjegyzéseket általában elfelejtjük módosítani:

```
c = b; // b értékét a-hoz rendeljük
```

Egy figyelmes programozó, ha megkap egy ilyen kódot karbantartás végett, nem tételezheti fel egyszerűen, hogy a megjegyzés hibás. Kénytelen lesz hosszasan elemezni a teljes kódot, hogy eldöntse, hibáról, félreértelmezhető megjegyzésről (c egy a-ra vonatkozó hivatkozás) vagy valamiféle szövevényes kapcsolatrendszerről (a c változó értékét valamikor később tényleg megkapja az a változó) van szó. Mindez elkerülhető lenne, ha a kérdéses sort mindennemű megjegyzés nélkül írtuk volna le:

```
a = b;
```

Ez a kód teljesen világos az elrontható megjegyzés nélkül is. Amúgy természetét tekintve ez a megállapítás ugyanolyan, mint az a gyakran hangoztatott szólás, miszerint a leghatékonyabb kód az, ami nem is létezik. Így van ez a megjegyzésekkel is: az a legjobb, ha nem kell leírnunk semmit, mert a kód maga annyira világos, hogy „önmagát dokumentálja".

A szükségtelen megjegyzések másik gyakori megjelenési helye az osztályok bevezetése (deklarálása). Ennek vagy valamilyen rosszul elképzelt és alkalmazott kódolási szabvány az oka, vagy az, hogy a programozó kezdő, és félelmében a „saját megjegyzéseibe kapaszkodik".

```
class C {
 // Nyilvános felület
 public:
   C(); // Alapértelmezett konstruktor
   ~C(); // Destruktor
   // . . .
};
```

Az ilyen kódot olvasva az embernek az az érzése támad, hogy valakinek a sebtében papírra vetett megjegyzéseit olvassa. Ha egy programozót emlékeztetni kell a `public:` kulcsszó jelentésére, annak az embernek kár volt odaadni ezt a kódot karbantartásra. Egy gyakorlott C++ programozó számára ezek a megjegyzések használhatatlanok; csak felesleges munkát jelentenek neki, hiszen kénytelen lesz ezeket is karbantartani, ha módosítja a kódot.

```
class C {
 // Nyilvános felület
 protected:
   C( int ); // Alapértelmezett konstruktor
 public:
   virtual ~C(); // Destruktor
   // . . .
};
```

Számos programozóban él az a törekvés, hogy ne „vesztegesse a sorokat" a forráskódban. A „közhiedelem" szerint, ha egy szerkezeti elem (függvény, egy osztály nyilvános felülete és így tovább) szokványos, közérthető formában leírható egyetlen lapon, vagyis körülbelül 30–40 sorban, akkor az a kód érthető lesz bárki számára. Ha csak kicsit is átnyúlik a következő lapra, kétszer olyan nehéz lesz megérteni. Ha netán még a harmadik lapra is szükség lesz hozzá, akkor a megértése négyszeres erőfeszítést igényel majd.

Az egyik legutálatosabb szokás, ha a végrehajtott változtatások „naplóját" a forráskód elejére vagy végére szúrja be a programozó megjegyzések formájában:

```
/* 6/17/02 SCD kijavította a gaforniflat hibát */
```

Ilyenkor az ember soha nem lehet benne egészen biztos, hogy a megjegyzésnek tényleg van valami jelentősége a kívülálló számra, vagy a program írója egyszerűen henceg, és dobálózik a szakkifejezésnek tűnő értelmetlen mondatokkal. Egy ilyen megjegyzés egy vagy két hét múlva már annak sem mond semmit, aki odaírta. Viszont esetleg hosszú évekre bennragad a forráskódban, idegesítve és félrevezetve a fejlesztők újabb generációit. Az efféle megjegyzések kezelését célszerűbb a változatkövető rendszerre bízni, elvégre egy program forráskódjában semmi keresnivalója a heti bevásárlólistánknak.

A valóban szükséges megjegyzések száma jelentősen csökkenthető, a kód pedig tisztává és könnyen karbantarthatóvá tehető, ha a programozási egységek (változók, függvények, osztályok) jelölésére értelmes neveket használunk, illetve jól átgondolt elnevezési szabályokat alkalmazunk. A deklarációkban szereplő formális paraméterek világos elnevezése különösen fontos. Vegyünk például egy függvényt, ami három azonos típusú paramétert vesz át:

```
/*
Végrehajtja a megadott műveletet a forráson, és az eredményt
elhelyezi a célként megadott helyen. Arg1 a művelet kódja, arg2
a forrás, arg3 a cél.
*/
void perform( int, int, int );
```

Nos igen. Ez végül is nem olyan borzalmas, de képzeljük csak el, hogy nézne ki ugyanez mondjuk hét vagy nyolc ilyen paraméterrel. Csináljuk tehát jobban:

```
void perform( int actionCode, int source, int destination );
```

Ez már jobb. Persze azért egy egysoros megjegyzésre még mindig szükségünk lesz, ha tudatni akarjuk az utókorral, hogy ez a függvény tulajdonképpen mit csinál (és akkor azt még el se mondtuk, hogyan csinálja). A szépen elnevezett formális paraméterek egyik legvonzóbb tulajdonsága az, hogy az ember önkéntelenül karbantartja őket a kód többi részével együtt. Ugyanezt a megjegyzésekkel nem mindenki teszi meg. Ami azt illeti, nem tudok elképzelni olyan programozót, aki felcserélné a fenti függvény első és második paraméterét anélkül, hogy megváltoztatná a nevüket is. Olyanból viszont egy hadseregnyit ismerek, aki ugyanezt szívfájdalom nélkül megtenné egy megjegyzéssel.

Mindezt a legvilágosabban talán Kathy Stark írta le *Programming in C++* (*Programozás C++ nyelven*) című könyvében: „Ha értelmes vagy legalább értelmezhető neveket használunk egy programban, általában semmi szükség nincs a további megjegyzésekre. Ha viszont a használt nevek értelmetlenek, nincs az a megjegyzés, ami a dolog egészét világosabbá tenné."

A megjegyzések számát úgy is csökkenthetjük, ha szabványos vagy közismert programozási elemeket használunk:

```
printf( "Hello, World!" ); // A „Helló világ!" szöveg kiírása
➥ a képernyőre
```

Ez a megjegyzés egyrészt szükségtelen, másrészt csak időnként helyes. Itt most nem arról van szó, hogy az efféle szabványos programelemek önmagukért beszélnek, sokkal inkább arról, hogy az ilyen megoldások közismertek és jól dokumentáltak.

```
swap( a, a+1 );
sort( a, a+max );
copy( a, a+max, ostream_iterator<T>(cout,"\n") );
```

Mivel ebben a példában a `swap`, a `sort` és a `copy` szabványos függvények, bármilyen további megjegyzés beszúrása szükségtelen, viszont szinte szükségszerűen pontatlan lenne.

A megjegyzések nem eredendően rosszak, és gyakran tényleg szükség van rájuk. Viszont nem szabad elfelejteni, hogy a megjegyzéseket ugyanúgy karban kell tartani, mint a kódot magát, sőt ezek karbantartása rendszerint nehezebb. A megjegyzések ne tartalmazzanak nyilvánvaló állításokat, vagy olyasmit, aminek máshol a helye. A cél tehát nem az, hogy a megjegyzéseket bármi áron elkerüljük, hanem az, hogy a mennyiségüket a lehető legalacsonyabban tartsuk, illetve hogy általuk érthető és jól karbantartható kód keletkezzen.

2. hiba: Mágikus számok

A „mágikus számok" jelen összefüggésben olyan számállandók (literálok), amelyek helyett az adott környezetben célszerűbb lenne nevesített állandókat használni.

```
class Portfolio {
   // . . .
   Contract *contracts_[10];
   char id_[10];
};
```

A mágikus számokkal kapcsolatban az a legnagyobb baj, hogy értékükön túl nincs semmilyen egyéb jelentésük, amit kötni tudnánk bármihez. Azok amik: számok. Az hogy `10`, semmi egyebet nem jelent, csak annyit, hogy `10`. Nem derül ki belőle, hogy ez a megköthető szerződések számának felső határa, vagy egy azonosító kód legnagyobb hossza. Ezért ha mágikus számokat tartalmazó kódot kell karbantartanunk, kénytelenek vagyunk kitalálni a programozó eredeti szándékát. Ez pedig egy kívülálló számára munka. Szükségtelen és gyakran pontatlan munka.

A fenti, meglehetősen ügyetlenül megtervezett példában a megköthető szerződések száma tíz, és ez a határ egy mágikus szám segítségével van beállítva. Tíz szerződés az nem túl sok. Ha netán úgy döntünk, hogy felemeljük a határt mondjuk 32-re, akkor kénytelenek leszünk átnyálazni az összes állományt, ami használja a `Portfolio` osztályt, ugyanis bármelyikben szerepelhet a korábbi mágikus tízes. Ráadásul az sem biztos, hogy mindenütt minden mágikus tízes a szerződések legnagyobb számát jelenti. (És akkor még nem is említettük azt a problémát, hogy a pontosság és biztonság érdekében eleve nem ártott volna a szabványos `vector` típust használni.)

Ami azt illeti, a valós helyzet néha még ennél is rosszabb lehet. Egy igazán nagy és hosszú időtartamú munka során egyesek esetleg kijelenthetik, hogy a megköthető szerződések legnagyobb száma tíz. Ezt mindenki hallja, tudomásul veszi, és a mágikus tízes beépül olyan kódrészletekbe is, amelyek egyáltalán nem hivatkoznak a `Portfolio` osztályra, egyszerűen azért, mert valamilyen más szempontból kezelik a szerződéseket:

```
for( int i = 0; i < 10; ++i )
    // . . .
```

Hát ez a tízes itt mire vonatkozhat? A szerződések legnagyobb számára, vagy egy azonosító hosszára? Esetleg valami teljesen másra?

A mágikus számok véletlen összekeverésének lehetősége néha a legrosszabbat hozza ki a programozókból:

```
if( Portfolio *p = getPortfolio() )
    for( int i = 0; i < 10; ++i )
        p->contracts_[i] = 0, p->id_[i] = '\0';
```

A program karbantartójának immár valahogyan szét kell válogatnia a `Portfolio` egyes összetevőinek kezdeti beállításait, figyelve arra, hogy csupán véletlen egybe-

esés, hogy két eltérő elem értéke ugyanaz. Erre a szükségtelen bonyodalomra pedig egyszerűen nincs mentség, ha a megoldás ilyen egyszerű:

```
class Portfolio {
    // . . .
    enum { maxContracts = 10, idlen = 10 };
    Contract *contracts_[maxContracts];
    char id_[idlen];
};
```

A felsoroló típusok nem foglalnak helyet, nem követelnek nagyobb számítási teljesítményt futásidőben, viszont teljesen tisztává teszik, hogy mi mit jelent.

A mágikus számok kevésbé nyilvánvaló, de annál veszélyesebb mellékhatása, hogy meghatározatlan típusúak, s így tárolási méretük is változhat. A literális `40000` típusa például szinte biztosan rendszerfüggő. Ha az érték elfér egy egész típusban, akkor egész (`int`) lesz. Ha nem, akkor `long`. Ha el akarjuk kerülni az ezzel kapcsolatos hibákat, például a túlterhelések feloldása körüli gondokat, jobb, ha mi magunk mondjuk meg, hogy pontosan mit is akarunk, ahelyett, hogy ezt a döntést a fordítóra és az adott rendszerre bíznánk:

```
const long patienceLimit = 40000;
```

A literális értékekekkel kapcsolatos másik gond, hogy nincs címük. Bár ez nem túl gyakran okoz fejfájást, néha kifejezetten hasznos lehet, ha képesek vagyunk egy állandó értéket címző mutatót vagy hivatkozást használni:

```
const long *p1 = &40000; // Hiba!
const long *p2 = &patienceLimit; // Rendben.
const long &r1 = 40000; // Rendben, de nézzük meg a 44. hibát.
const long &r2 = patienceLimit; // Rendben.
```

Összefoglalva tehát, a mágikus számok használatának egyetlen előnye sincs, viszont számos problémát okozhatnak, ezért használjunk helyettük felsoroló típust, vagy kezdőértékkel ellátott állandókat.

3. hiba: Globális változók

Általában nincs mentség a „nyers" globális változók használatára. Ezek nehezen karbantarthatóvá teszik a kódot, és a program újrahasznosíthatósága is korlátozott

lesz. Az újrahasznosíthatóság azért sérül, mert a globális változó „összenő" a kóddal. A kettő csak együtt használható, külön-külön egyiknek sincs értelme. Az efféle általánosság a karbantartást is nehezíti, hiszen az adott változó bárhonnan hozzáférhető, így semmi sem utal arra, hogy mely kódrészletek használják.

A globális változók erősítik az egyes kódrészletek összetartozását, egymásra utaltságát, és éppen ez a baj velük. Ezek a változók ugyanis gyakran egyfajta üzenetközvetítő szerepet töltenek be a különböző programelemek között. A módszer persze tökéletesen működhet, de egy globális változót eltávolítani egy nagyobb programból gyakorlatilag lehetetlen. Ugyanakkor az általános érvényűség miatt ezek az üzenetközvetítők semmiféle védelmet nem élveznek, így bármelyik kezdő programozó, akit megbíztak a kód karbantartásával, a rendszer teljes összeomlását idézheti elő.

Akik mindezek ellenére globális változókat alkalmaznak, általában a kényelmes használatot említik fő érvként. Ez persze csalóka, önigazoló érv, hiszen a fejlesztés általában kevesebb időt igényel, mint a karbantartás, a globális változók viszont ez utóbbit elképesztően megnehezíthetik. Tegyük fel például, hogy programunknak állandóan hozzá kell férnie egy globálisan elérhető „környezethez", amelyből (megígérték nekünk, hogy így lesz) mindig pontosan egy létezik. Sajnálatos módon ezt a kapcsolatot a program és a környezete között egy globális változó segítségével oldottuk meg:

```
extern Environment * const theEnv;
```

Sajnos az ígéreteket gyakran nem tartják be. Pár nappal a szállítás előtt valaki mégis rájön, hogy két lehetséges környezete is lehet a programnak. Vagy három. Vagy az is lehet, hogy a környezetek számát majd a felhasználó fogja megmondani induláskor. Vagy ő sem fog mondani semmit, hanem a környezetek száma lesz „dinamikus": hol ennyi, hol annyi. Ez az a bizonyos utolsó pillanatban bejelentett változás. Még egy nagy projektben is, ahol egyébként gondosan felépített forráskód-kezelő rendszert használnak, meglehetősen időrabló minden egyes fájlt módosítani, még akkor is, ha a módosítás egyszerű. A folyamat napokat vagy heteket vehet igénybe. Persze ha nem használtunk volna globális változót, az egész meglenne úgy öt perc alatt:

```
Environment *theEnv();
```

Ha a környezethez való hozzáférést egy függvénybe rejtjük, probléma esetén lehetőségünk van annak túlterhelésére, vagy valamelyik paraméterének módosítására. Ehhez nem lesz szükség a kód nagyobb arányú módosítására, csak a függvényt magát kell átszabni:

```
Environment *theEnv( EnvCode whichEnv = OFFICIAL );
```

Van a globális változókkal kapcsolatban egy másik, nem annyira nyilvánvaló gond is: gyakran futásidőben, statikusan kell nekik értéket adni. Ha a kezdőérték fordításkor nem számítható ki, futásidőben kell beállítani, aminek esetenként katasztrofális következményei lehetnek:

```
extern Environment * const theEnv = new OfficialEnv;
```

Ha a globális információhoz való hozzáférést egy függvény vagy osztály „felügyeli", a tényleges értékadás késleltethető addig, amíg kiderül, hogy biztonságosan elvégezhető-e:

➡ 3. hiba/environment.h

```
class Environment {
 public:
    static Environment &instance();
    virtual void op1() = 0;
    // . . .
 protected:
    Environment();
    virtual ~Environment();
 private:
    static Environment *instance_;
    // . . .
};
```

➡ 3. hiba/environment.cpp

```
// . . .
Environment *Environment::instance_ = 0;

Environment &Environment::instance() {
    if( !instance_ )
        instance_ = new OfficialEnv;
    return *instance_;
}
```

Ebben az esetben a Singleton modell egy egyszerű megvalósítását alkalmaztuk, ami „lusta kezdőérték-adást" hajt végre a statikus környezeti mutatón, ezzel biztosítva, hogy soha ne lehessen egynél több érvényes környezet (`Environment` objektum). Az `Environment` objektumnak nincs nyilvános konstruktora, így a felhasználó kénytelen az `instance` tagot használni, ha hozzá akar férni a statikus mutatóhoz. Ez pedig lehetőséget ad arra, hogy az `Environment` objektum tényleges létrehozását elhalasszuk az első hozzáférésig:

```
Environment::instance().op1();
```

Ennél persze sokkal lényegesebb előny, hogy rendszerünk rugalmassága nő, hiszen a későbbiekben továbbra is szabadon használhatjuk a Singleton megközelítést a kód jelentősebb átírása nélkül. Ha később például többszálú felépítésre akarunk áttérni, vagy több környezetet akarunk használni, ugyanolyan egyszerűen módosíthatjuk a Singleton modell megvalósítását, mint amilyen egyszerű volt korábban a „burkoló függvény" átírása.

Összefoglalva tehát kerüljük a globális változók használatát. Léteznek biztonságosabb és sokkal rugalmasabb módszerek, amelyekkel ugyanazt a célt elérhetjük.

4. hiba: A túlterhelés és az alapértelmezett beállítás összemosása

A függvények túlterhelésének (overloading) és a paraméterek kezdőérték-beállításának nem sok köze van egymáshoz. Ennek ellenére ezt a két különböző nyelvi elemet gyakran összetévesztik, mivel segítségükkel nagyon hasonlóan használható felület készíthető. A hasonlóság ellenére azonban a kétféle felület teljesen más:

➡ 4. hiba/c12.h

```
class C1 {
 public:
   void f1( int arg = 0 );
   // . . .
};
```

➡ 4. hiba/c12.cpp

```
// . . .
C1 a;
a.f1(0);
a.f1();
```

A `C1` osztály tervezője úgy döntött, hogy az `f1` függvény bevezetésében alapértelmezett paraméterbeállítást használ. Ennek megfelelően a `C1` osztályt használó alkalmazásnak módjában áll `f1`-et egy darab paraméterrel, vagy paraméter nélkül meghívni, mely esetben a paraméter nulla (vagyis az alapértelmezett érték) lesz. A `C1::f1` függvény fenti kódrészletben bemutatott két meghívása pontosan ugyanazt a végrehajtási sorozatot eredményezi.

➡ 4. hiba/c12.h

```
class C2 {
 public:
```

```
    void f2();
    void f2( int );
    // . . .
};
```

➡ 4. hiba/c12.cpp

```
// . . .
C2 a;
a.f2(0);
a.f2();
```

A `C2` osztály megvalósítása már teljesen más. A felhasználónak itt arra van lehetősége, hogy `f2` néven két különböző függvényt hívjon meg. Hogy melyik hajtódik végre, azt a paraméterek száma dönti el. Míg az előző példában a két függvényhívás eredménye teljesen azonos volt, itt akár eltérő is lehet, hiszen két különböző függvényről van szó.

A két felület közti eltérés még nyilvánvalóbb, ha a `C1::f1` és a `C2::f2` osztályelemek címét próbáljuk használni:

➡ 4. hiba/c12.cpp

```
void (C1::*pmf)() = &C1::f1; // Hiba!
void (C2::*pmf)() = &C2::f2;
```

A `C2` osztály megvalósításából következőleg a `pmf` nevű tag mutatója a paraméterek nélküli `f2`-re fog mutatni. Mivel a `pmf` a bevezetés szerint egy paraméterek nélküli függvény mutatóját tartalmazza, a fordító helyesen a paraméterek nélküli `f2`-t választja. A `C1` osztály esetében ugyanez a logika már nem működik. Olyannyira nem, hogy fordítási hibát kapunk. Az ok nyilvánvaló: ez az osztály csak egy `f1` függvényt tartalmaz, annak pedig pontosan egy egész típusú paramétere van. Amire tehát hivatkozni próbálunk, az egyszerűen nem létezik.

A túlterhelés általában azt hivatott jelezni, hogy a függvények egy csoportja azonos elvont jelentéssel bír, de megvalósításuk különböző. A paraméterek alapértelmezett beállítása ugyanakkor inkább kényelmi szolgáltatás, ami a függvények felületének leírását teszi egyszerűbbé. Ennek megfelelően a túlterhelés és az alapértelmezett paraméterbeállítás két különböző nyelvi szolgáltatás, különböző céllal, és különböző mellékhatásokkal. (A témával kapcsolatban érdemes még megnézni a 73. és 74. hibát.)

5. hiba: A hivatkozások félreértelmezése

A hivatkozásokkal (referencia) kapcsolatban két általános hibát lehet megemlíteni. Az egyik az, hogy gyakran összekeverik őket a mutatókkal (pointer), a másik, hogy nem használják őket eléggé. A legtöbb, a mai C++ programokban használt mutató tulajdonképpen a C korszak öröksége, és nyugodtan lehetne hivatkozással helyettesíteni.

A hivatkozás nem mutató. A hivatkozás tulajdonképpen egy álnév (másodnév, alias), amivel semmi egyebet nem lehet tenni, mint beállítani. Miután ezt megtettük, a hivatkozás pontosan ugyanúgy viselkedik, mint az, amire hivatkozunk vele. (Ámbátor nézzük meg ezzel kapcsolatban a 44. hibát.) A hivatkozásnak nincs címe, sőt az is lehetséges, hogy tárterületet sem foglal:

```
int a = 12;
int &ra = a;
int *ip = &ra; // ip az a változót címzi
a = 42; // ra == 42
```

A fentiek miatt a C++-ban nem megengedett mutatót vagy hivatkozást adni egy hivatkozásra, és a hivatkozásokból tömb sem alkotható. (Ugyanakkor a C++ szabványosításával foglalkozó bizottság már tárgyalt a hivatkozásra vonatkozó hivatkozások bizonyos környezetekben való engedélyezéséről, de ez egyelőre még a jövő zenéje.)

```
int &&rri = ra; // Hiba!
int &*pri; // Hiba!
int &ar[3]; // Hiba!
```

Egy hivatkozás nem lehet `const` vagy `volatile`, mivel az álnevek sem lehetnek ilyenek. Ugyanakkor annak semmi akadálya, hogy egy hivatkozás `const` vagy `volatile` típusú elemre mutasson. A hivatkozás közvetlen bevezetése `const`-ként vagy `volatile`-ként azonban fordítási hibát eredményez:

```
int &const cri = a; // Hibásnak kellene lennie...
const int &rci = a; // Rendben
```

Furcsa módon nem tilos a `const` vagy `volatile` minősítő használata olyan típusnévvel kapcsolatban, amely lényegét tekintve hivatkozás. Persze az előbbiek most is érvényben maradnak: nem kapunk ugyan hibaüzenetet, de a fordító figyelmen kívül hagyja a minősítőt.

```
typedef int *PI;
typedef int &RI;
const PI p = 0; // Állandó mutató
const RI r = a; // Csak egy hivatkozás!
```

Nem létezik „nullhivatkozás", sőt `void` sincs:

```
C *p = 0; // Nullmutató
C &rC = *p; // Meghatározatlan viselkedés
extern void &rv; // Hiba!
```

Röviden: a hivatkozás egy álnév, és az álnévnek kötelezően vonatkoznia kell valamire.

Ugyanakkor fontos kihangsúlyozni, hogy egy hivatkozásnak nem feltétlenül kell egy adott változóra vonatkozni. Néha egyszerűbb a hivatkozást egy balértékhez (lvalue) kötni (lásd a 6. hibát), ami lényegesen összetettebb kifejezést eredményez:

```
int &el = array[n-6][m-2];
el = el*n-3;
string &name = p->info[n].name;
if( name == "Joe" )
   process( name );
```

Ha egy függvény visszatérési értéke hivatkozás, az lehetővé teszi, hogy egy függvényhívás eredményéhez rendeljünk hozzá valamit. A szokványos példa erre a lehetőségre egy elvont tömb indexfüggvénye:

➡ 5. hiba/array.h

```
template <typename T, int n>
class Array {
 public:
   T &operator [](int i)
        { return a_[i]; }
   const T &operator [](int i) const
        { return a_[i]; }
   // . . .
 private:
   T a_[n];
};
```

A hivatkozás típusú visszatérési érték lehetővé teszi a természetes írásmód használatát a tömbelemek értékének beállítása során:

```
Array<int,12> ia;
ia[3] = ia[0];
```

A hivatkozások természetesen arra is felhasználhatók, hogy egy függvénnyel egynél több értéket adjunk vissza eredményként:

```
Name *lookup( const string &id, Failure &reason );
// . . .
string ident;
// . . .
Failure reasonForFailure;
if( Name *n = lookup( ident, reasonForFailure ) ) {
   // A keresés sikeres volt.
}
else {
   // A keresés sikertelen volt. Nézzük, miért...
}
```

Egy objektumnak hivatkozás típusra való átalakítása egészen mást eredményez, mint ha nem hivatkozás típusra alakítunk át:

```
char *cp = reinterpret_cast<char *>(a);
reinterpret_cast<char *&>(a) = cp;
```

Az első esetben egy egész értéket alakítunk mutatóvá. (Célszerűbb a régi típusátalakítás – például `(char *)a` – helyett a `reinterpret_cast`-ot használni. Lásd a 40. hibát.) Az eredmény az egész értékű változó tartalmának másolata, amit mutatóként értelmezünk.

A második esetben, amikor hivatkozás típusra vonatkozik a típusátalakítás, az eredmény már egészen más. Itt az egész változót úgy értelmezzük át, hogy a továbbiakban mutatónak tekintjük. Ami itt keletkezik, az egy balérték, amihez értéket rendelhetünk. (Az már más kérdés, hogy utána valószínűleg azonnali programleállás lesz a jutalmunk. A `reinterpret_cast` használatával egyébként általában automatikusan lemondunk a program hordozhatóságáról.) Egy hasonló átalakítás nem hivatkozás típusra már nem megy, mivel az eredmény itt jobbérték (rvalue), nem bal:

```
reinterpret_cast<char *>(a) = 0; // Hiba!
```

A tömbre vonatkozó hivatkozások megőrzik a tömb határait. Az ugyanilyen mutatók nem:

```
int ary[12];
int *pary = ary; // Az első elemre mutat
int (&rary)[12] = ary; // A teljes tömböt címzi
```

```
int ary2[3][4];
int (*pary2)[4] = ary2; // Az első elemre mutat
int (&rary2)[3][4] = ary2; // A teljes tömböt címzi
```

Ezt a tulajdonságot esetenként ki is használhatjuk, ha tömböt kell függvénynek paraméterként átadnunk (lásd a 34. hibát).

Annak sincs semmi akadálya, hogy egy hivatkozás célja függvény legyen:

```
int f( double );
int (* const pf)(double) = f; // függvényt címző állandó mutató
int (&rf)(double) = f; // függvényre mutató hivatkozás
```

Gyakorlati szempontból nincs sok különbség egy függvényt címző állandó mutató és egy ugyanilyen hivatkozás között. Az egyetlen lényegi eltérés az, hogy a mutató közvetlenül visszakövethető (dereferencia), a hivatkozás pedig – lévén álnév – nem. Ugyanakkor lehetőségünk van rá, hogy egy függvényhivatkozást függvénymutatóvá alakítsunk, majd így visszakövessük:

```
a = pf( 12.3 ); // mutatót használ
a = (*pf)(12.3); // mutatót használ
a = rf( 12.3 ); // hivatkozást használ
a = f( 12.3 ); // függvényt használ
a = (*rf)(12.3); // hivatkozást mutatóvá alakít, majd azt követi
a = (*f)(12.3); // függvényt mutatóvá alakít, majd követi
```

Összefoglalva: különböztessük meg a mutatókat és a hivatkozásokat.

6. hiba: Az állandók félreértelmezése

Az állandók (`const`, konstans) értelmezése a C++ nyelvben meglehetősen egyszerű, de ez az értelmezés nem feltétlenül felel meg annak, amit az állandókról amúgy gondolnánk.

Először is nézzük, mi a különbség egy állandóként (`const`) bevezetett változó és egy literál között:

```
int i = 12;
const int ci = 12;
```

A `12` mint egész literál nem állandó, hanem literál. Ebbéli minőségében nincs címe, és az értéke sem változik soha. Az `i` nevű egész változó ezzel szemben objektum. Van címe, és az értéke is változhat. Az állandóként bevezetett `ci` változó szintén objektum és szintén egész. Van címe, az értéke azonban ebben az esetben nem változhat.

A különbséget tömören úgy fogalmazhatjuk meg, hogy az `i` és `ci` változók lehetnek balértékek, a literális 12 azonban csak jobbérték lehet. Ez a fura elnevezés a `J=B` álkifejezésből („pszeudo-kifejezésből") származik, ahol a `J` jobbérték az értékadásnak mindig a jobb, míg a `B` balérték mindig a bal oldalán szerepel. Ez a meghatározás persze nem teljesen alkalmazható a C++, sőt a C nyelvre sem. Jelen esetben ugyanis a `ci`-t ugyan balértéknek tekintjük, de soha nem szerepelhet értékadás bal oldalán, mert értékének megváltoztatását megtiltottuk. Kicsit árnyaltabban megfogalmazva a különbséget, a balértékek olyan dolgok, amik valamit tárolnak és van címük, a jobbértékek viszont olyanok, amiknek van ugyan értéke, de az nem változhat, és címük sincs.

```
int *ip1 = &12; // hiba!
12 = 13; // hiba!
const int *ip2 = &ci; // rendben
ci = 13; // hiba!
```

A fenti példában az `ip2` bevezetésében szereplő `const` kulcsszó azt írja elő, miként kezelhetjük a `ci` tartalmát ezen a mutatón keresztül, és nem azt, hogy a `ci` tartalma miként változhat általában. Nézzünk talán egy másik példát, amikor egy `const`-ként bevezetett mutató egy amúgy nem állandó egészre mutat:

```
const int *ip3 = &i;
i = 10; // rendben
*ip3 = 10; // hiba!
```

Ebben az esetben viszonylag egyértelmű a helyzet: a `const` kulcsszó azt írja elő, hogy az `ip3` mutatón keresztül történő hivatkozással az `i` tartalma nem módosítható. Ugyanakkor `i` nem állandó, tehát egy értékadással bármikor módosíthatjuk a tartalmát.

A `const` és a `volatile` kombinációi még ravaszabbak:

```
extern const volatile time_t clock;
```

A `const` módosító jelenléte itt azt jelenti, hogy nem módosíthatjuk a `clock` nevű külső változó tartalmát. Ugyanakkor a `volatile` felbukkanása azt jelzi, hogy `clock` tartalma változhat, sőt esetünkben nyilván változni is fog.

7. hiba: Az alapvető nyelvi finomságok ismeretének hiánya

A legtöbb C++ programozó meg van róla győződve, hogy ismeri a C++ nyelv alapjait, vagyis azt a részét, amit a C-től örökölt. Ugyanakkor számos esetben még a gyakorlott C++ programozók sem ismerik a nyelv rejtettebb részleteit, még akkor sem, ha azok történetesen az előbb említett alapvető utasításokkal és műveletekkel kapcsolatosak.

A logikai műveletekről (operátorokról) például elsőre valószínűleg senki sem gondolná, hogy rejtett részletnek minősülnek, pedig újabban a fiatal C++ programozók egyre kevésbé használják őket. Hát nem irritáló olyan kódot látni, mint amilyen ez itt?

```
bool r = false;
if( a < b )
   r = true;
```

Ehelyett?

```
bool r = a<b;
```

Netán a Nyájas Olvasó is elszámol tízig, ha ezt látja?

➡ 7. hiba/bool.cpp

```
int ctr = 0;
for( int i = 0; i < 8; ++i )
   if( options & 1<<(8+i) )
       if( ctr++ ) {
           cerr << "Too many options selected";
           break;
       }
```

Ehelyett?

➡ 7. hiba/bool.cpp

```
typedef unsigned short Bits;
inline Bits repeated( Bits b, Bits m )
   { return b & m & (b & m)-1; }
// . . .
if( repeated( options, 0XFF00 ) )
   cerr << "Too many options selected";
```

No de tulajdonképpen mi történt a jó öreg Boole-féle logikával?

Számos programozó azt is figyelmen kívül hagyja, hogy a feltételes művelet eredménye balérték (lásd a 6. hibát) ha mindkét lehetséges eredmény balérték. Ennek aztán ilyen kód az eredménye:

```
// 1. megoldás
if( a < b )
   a = val();
else if( b < c )
   b = val();
else
   c = val();

// 2. megoldás
a<b ? (a = val()) : b<c ? (b = val()) : (c = val());
```

Az alábbi megoldás, balértékként kezelt feltétellel, kétségtelenül sokkal rövidebb és elegánsabb:

```
// 3. megoldás
(a<b?a:b<c?b:c) = val();
```

Bár ez a kis „ezoterikus" tudás elsőre talán nem tűnik túl fontosnak, hiszen végső soron nem egyéb, mint a Boole-féle logika egyszerű alkalmazása, ne felejtsük el, hogy a C++ számos helyen csak kifejezéseket enged használni (tagbeállítási lista a konstruktorban, szabályozó kifejezések).

Figyeljük meg azt is, hogy a `val` objektumra való hivatkozás az első és második változatban többször fordul elő, a harmadikban viszont csak egyszer. Ha a `val` egy előfeldolgozó makró, a többszöri behelyettesítésnek nem kívánt mellékhatásai lehetnek (lásd a 26. hibát). Ezekben a helyzetekben az `if`-es szerkezetnek egy megfelelően megfogalmazott feltételes művelettel való helyettesítése létkérdés. Azt ugyan nem javaslom, hogy a fenti szerkezetet általánosan használjuk valamennyi programban, azt viszont igen, hogy tartsuk észben ezt a lehetőséget. Néhány –

meglehetősen ritka – esetben nagy hasznát veheti bármely gyakorló C++ programozó, ha más megoldás nem használható vagy egyszerűen csak kevésbé hatékony. Nem véletlen, hogy ez a lehetőség bekerült a C++ nyelvbe.

Meglepő módon még a készen kapott indexjelet is gyakran félreértik. Azt valamennyien tudjuk, hogy a tömb és a mutató egyaránt indexelhető:

```
int ary[12];
int *p = &ary[5];
p[2] = 7;
```

Az indexjel (index operátor) tulajdonképpen nem egyéb, mint a mutatóműveletek egyfajta rövidítése. A fenti példában szereplő `p[2]` kifejezés hatása szó szerint megegyezik a `*(p+2)` kifejezésével. A „C-háttérrel" rendelkező C++ programozók általában azt is szokták tudni, hogy indexként akár negatív számok is használhatók. Ennek megfelelően nem lepődnek meg a `p[-2]` kifejezésen, ami ismét teljesen azonos a `*(p-2)`, illetve a `*(p+-2)` kifejezésekkel. Arra azonban már kevesen gondolnak, hogy az összeadás kommutatív művelet (tényezői felcserélhetők), így nemcsak mutatót lehet egésszel indexelni, hanem egészet mutatóval is, valahogy így:

```
(-2)[p] = 6;
```

Ez amúgy nem egyéb, mint egy egyszerű átalakítás: `p[-2]` egyenértékű a `*(p+-2)` formával, ez egyenértékű `*(-2+p)`-vel, ez pedig megfelel a `(-2)[p]` formának. (A kerek zárójelekre szükségünk van, mert a `[]` jelnek magasabb a rangja, mint az egytényezős mínusznak.)

Ez eddig nyilvánvaló, de miért kell erre kitérni? – kérdezheti az Olvasó. Először is, meg kell jegyeznünk, hogy a tényezők felcserélhetősége csak az indexjel „készen kapott", mutatókkal való használatra vonatkozó meghatározására érvényes. Ha tehát azt látjuk, hogy `6[p]`, biztosak lehetünk benne, hogy az ismert, „beépített" indexjellel van dolgunk és nem egy túlterhelt `operator []` taggal. (Bár a `p` nem feltétlenül mutató vagy tömb.) Aztán bevethetjük ezt a trükköt, mondjuk találós kérdés formájában egy koktélpartin, ha lankadni látszik a társalgás... Mielőtt viszont „élesben" használni kezdenénk, feltétlenül olvassuk el a 11. hiba leírását.

A legtöbb C++ programozó természetesen tudja, hogy a `switch` utasítás alapvető. Csak azt nem sejtik, mennyire. A `switch` szerkezet elvont formája a következő:

```
switch( expression ) statement
```

Ennek az egyszerű írásmódnak a következményei néha egészen meglepőek lehetnek.

A `switch` kifejezést követő utasítás általában egy blokk, amely a megfelelő címkék segítségével több esetet ír le. Ez nem egyéb, mint egy számított `goto` parancs, amelynek célja mindig valahol e blokkon belül van. Az első furcsaság, amit a kezdő C és C++ programozóknak meg kell emészteniük, az „átesés" (fallthrough) jelensége. Ez azt jelenti, hogy eltérően számos más programozási nyelvtől, a `switch` munkája gyakorlatilag befejeződik az első találattal. Az pedig, hogy a megfelelő blokk végrehajtása után a program hol folytatódik, csak a programozótól függ:

```
switch( e ) {
default:
theDefault:
   cout << "default" << endl;
   // „Átesés"...
case 'a':
case 0:
   cout << "group 1" << endl;
   break;
case max-15:
case Select<(MAX>12),A,B>::Result::value:
   cout << "group 2" << endl;
   goto theDefault;
}
```

Általános gyakorlat, hogy ha szándékosan használjuk az említett átesést – szemben a sokkal gyakoribb véletlen „használattal" –, ezt egy megjegyzés formájában illik megemlíteni az utókornak. Ellenkező esetben ugyanis a program karbantartói ellenállhatatlan vágyat fognak érezni néhány szükségtelen, viszont kifejezetten káros `break` utasítás beiktatására.

Jegyezzük meg, hogy az eseteket azonosító címkéknek egészeknek, vagy egész értéket eredményező állandó kifejezéseknek kell lenniük. Másként fogalmazva, a fordítónak képesnek kell lennie meghatározni ezeket az értékeket fordítási időben. Ugyanakkor, amint azt a fenti kissé laza példa is mutatja, az állandó kifejezések megadásának igencsak sok módja létezik. Az elágaztatáshoz használt kifejezésnek egésznek, vagy egésszé alakítható objektumnak kell lennie. Ha például `e` egy osztály, amely egy egészre átalakító függvényt vezet be, akkor `e` szerepelhet egy ilyen szerkezetben kifejezésként.

Érdemes azt is megjegyezni, hogy a `switch` szerkezet formailag még annyira sem kötött, mint a bemutatott példában. A `case` címkék gyakorlatilag bárhol előfordulhatnak a programblokkban, még az sincs megkötve, hogy azonos szerkezeti szinten kell lenniük:

```
switch( expr )
   default:
```

```
if( cond1 ) {
    case 1: stmt1;
    case 2: stmt2;
}
else {
    if( cond2 )
        case 3:stmt2;
    else
        case 0: ;
}
```

Mindez meglehetősen bután néz ki (nem véletlenül, ugyanis példánk tényleg meglehetősen elvetemült), az efféle finom részleteknek azonban nagy szerepe lehet néhány szokatlan helyzetben. A `switch` fenti tulajdonságát például felhasználtuk egyszer egy C++ fordítóprogramban, egy összetett adatszerkezet hatékony belső bejárásához:

➡ 7. hiba/iter.cpp

```
bool Postorder::next() {
  switch( pc )
  case START:
  while( true )
     if( !lchild() ) {
        pc = LEAF;
        return true;
  case LEAF:
        while( true )
           if( sibling() )
              break;
           else
              if( parent() ) {
                 pc = INNER;
                 return true;
  case INNER:    ;
              }
              else {
                 pc = DONE;
  case DONE:     return false;
              }
     }
}
```

Ebben a kódrészletben a `switch` említett tulajdonságát arra használtuk, hogy segítségével egy fák között ugró `next` utasítást valósítsunk meg.

Ami azt illeti, gyakran volt már részem időnként erős, sőt néha kifejezetten támadó negatív kritikában amiatt, hogy programjaimban ilyen és ehhez hasonló megoldá-

sokat használtam. Természetesen készséggel egyetértek bárkivel abban, hogy ezek nem arra valók, hogy általuk kezdők számára is világos kódot írjunk. Ugyanakkor a környező programtól megfelelően elzárva és megfelelően dokumentálva az ilyen rendhagyó megoldásoknak igenis helyük van a programozás gyakorlatában, mert kiélezett helyzetekben életmentők lehetnek. A programnyelvek rejtett finomságainak ismerete tehát igenis hasznos.

8. hiba: A hozzáférhetőség és a láthatóság összetévesztése

A C++ nyelvben nincs adatrejtés, a nyelv csak az adatokhoz való hozzáférést szabályozza. Az osztályok privát és védett elemei nem láthatatlanok a külvilág számára, csupán nem hozzáférhetők. Mint megannyi más látható, de hozzáférhetetlen objektum (gondoljunk például az igazgatókra), ezek is számos probléma okozói lehetnek.

A legnyilvánvalóbb gond talán az a megkötés, hogy akkor is újra kell fordítanunk a kódot, ha csak egy, a külvilág számára nem látható része változott meg. Vegyünk például egy egyszerű osztályt, amiben egy újabb tagot vezettünk be:

```
class C {
 public:
   C( int val ) : a_( val ),
          b_( a_ ) // új
   {}
   int get_a() const { return a_; }
   int get_b() const { return b_; } // új
 private:
   int b_; // új
   int a_;
};
```

Ebben az esetben az osztály számos tulajdonsága megváltozott. Ezek némelyike hozzáférhető elemekkel kapcsolatos, mások nem hozzáférhetőkkel.

Világos módon megváltozott az osztály mérete, hiszen egy újabb elemet hoztunk létre benne. Ez befolyásolja az összes olyan külső kódrészletet, ami az osztály elemeit használja, ezeket ez elemeket címző mutatókat kezel, bármilyen módon az

osztály méretére, vagy a tagok neveire támaszkodik. Vegyük észre azt is, hogy az új elem felvételével megváltozott `a_` eltolási címe (offset), amely természetesen azonnal érvényteleníti az összes ezt címző mutatót, sőt azokat is, amelyek valamilyen módon közvetve mutatnak ide. Ráadásul a konstruktor tagbeállító listája most már hibás, hiszen `b_` kezdőértéke nem meghatározott (lásd az 52. hibát).

A láthatatlan változások kihatnak a másoló konstruktorra (copy constructor) és a másoló értékadás (copy assignment) műveletre is, amelyeket a fordító még az előző deklarációnak megfelelően hozott létre. Alapértelmezés szerint ezek helyben kifejtett függvények (inline függvények), így minden olyan helyre bekerülnek, ahol a `C` osztály elemeit kezeljük (lásd a 49. hibát).

A `C` osztály módosításának legfőbb hatása tehát az, hogy újra kell fordítanunk minden olyan kódrészletet, ami `C`-re hivatkozik. Nagyobb programok esetében ez az újrafordítás meglehetősen időrabló lehet. Ha `C`-t egy fejállományban (header) vezettük be, minden olyan forráskódot újra kell fordítani, ami erre a deklarációra hivatkozik. Részleges megoldást nyújthat az „előzetes bevezetés" (forward declare) módszere, amikor egy befejezetlen deklarációt helyezünk el minden olyan forrásfájlban, amely hivatkozik ugyan a `C` osztályra, de ténylegesen nem használ ilyen objektumokat, így több információra nincs szüksége:

```
class C;
```

Egy ilyen nem teljes bevezetés lehetővé teszi, hogy mutatókat és hivatkozásokat használjunk `C` típusú objektumokra, egészen addig, amíg nem végzünk velük olyan műveleteket, amelyeknél a fordítónak ismernie kell bizonyos elemek pontos címét vagy nevét, vagy az objektum méretét (lásd a 39. hibát).

Ez a megközelítés hatékony lehet, a karbantartási problémák elkerülése végett azonban a hiányos deklarációt mindenképpen abból a fájlból kell kiemelnünk, amelyik az osztály meghatározását (definícióját) tartalmazza. Ez azt jelenti, hogy ha a programunkban van egy olyan rész, amely meglehetősen összetett szolgáltatásokat nyújt, akkor ehhez a részhez írnunk kell egy „előbevezető" fejállományt, ami tartalmazza a megfelelő mennyiségű információt.

Ha a `C` osztály teljes bevezetését például a `c.h` fájl tartalmazza, akkor célszerű ebből egy `cfwd.h` nevű fájlba emelni azokat a részeket, amelyek feltétlenül szükségesek a sikeres fordításhoz. Azokon a helyeken, ahol nincs szükség a `C` teljes meghatározására, elég az utóbbi fejállományra hivatkozni. A hiányos bevezetés ilyen elkülönült formában való biztosítására azért van szükség, mert a `C` osztály meghatá-

rozása a jövőben esetleg úgy változhat, hogy az új forma már nem felel meg a régi, hiányos bevezetésnek. Megeshet például, hogy az osztályt nevesített felhasználói típussá (typedef) alakítjuk:

```
template <typename T>
class Cbase {
   // . . .
};
typedef Cbase<int> C;
```

Bár a `c.h` fájl megalkotója nyilván igyekezne elkerülni a felesleges újrafordítást, ebben a helyzetben az összes olyan lefordított kódrészlet, ami a hiányos bevezetésre hivatkozott, hibás lesz:

```
#include "c.h"
// . . .
class C; // Hiba! C egy típusnév
```

Ha viszont mindig előállítjuk a megfelelő `cfwd.h` állományt is, a probléma nem merülhet fel. Ezt a módszert használták az `iostream` szabványos könyvtár megvalósítása során. Itt az `iosfwd.h` állomány tartalmazza az `iostream.h` szükséges részeit.

A `C` osztályt tartalmazó kódrészletek újrafordításának szükségessége általában meggátolja, hogy egyszerű hibajavítást („foltozást") alkalmazzunk a már telepített programösszetevőkre. A leghatékonyabb módszer, amivel egy osztály megvalósítását elhatárolhatjuk annak programozási felületétől, általában a Híd módszer alkalmazása.

A Híd módszer lényege, hogy az osztályt két részre, a felületre és a megvalósításra bontjuk:

➡ 8. hiba/cbridge.h

```
class C {
 public:
   C( int val );
   ~C();
   int get_a() const;
   int get_b() const;
 private:
   Cimpl *impl_;
};
```

➡ 8. hiba/cbridge.cpp

```
class Cimpl {
 public:
   Cimpl( int val ) : a_( val ), b_( a_ ) {}
   ~Cimpl() {}
   int get_a() const { return a_; }
   int get_b() const { return b_; }
 private:
   int a_;
   int b_;
};
C::C( int val )
   : impl_( new Cimpl( val ) ) {}
C::~C()
   { delete impl_; }
int C::get_a() const
   { return impl_->get_a(); }
int C::get_b() const
   { return impl_->get_b(); }
```

A felület a C osztály eredeti felületének leírását tartalmazza, a tagok megvalósítása azonban már egy másik osztályban kap helyet, ami a külső felhasználó számára nem látható. A C új változata csupán egy mutatót tartalmaz a saját függvényeit megvalósító másik osztályra, ennek az osztálynak a tagjai pedig rejtve maradnak a felhasználó számára. Mindez azt eredményezi, hogy a C minden olyan változása, amely a programozási felületet nem érinti, egyetlen forrásállományra korlátozódik.

A Híd módszernek persze futásidőben fizetjük meg az árát. Minden egyes C objektum valójában két különálló objektum létét igényli majd, a függvényhívások pedig valamennyien közvetettek lesznek, a helyben kifejtés lehetősége nélkül. Ugyanakkor ezt a hátrányt feltehetőleg bőven ellensúlyozza a sokkal rövidebb fordítási idő, valamint az a lehetőség, hogy az ügyfélprogramok által használt kódot bizonyos keretek között azok újrafordítása nélkül javíthatjuk. A módszert meglehetősen régóta használják a programozói gyakorlatban, és több különböző neve is született („pimpl idiom", „Cheshire Cat technique").

A kívülről nem hozzáférhető elemek is befolyásolhatják a származtatott osztályok és alaposztályok jelentését, ha egy származtatott osztály felületén keresztül férünk hozzájuk. Vegyük például a következő alap- és származtatott osztályt:

```
class B {
 public:
   void g();
```

```
  private:
    virtual void f(); // Új
};
class D : public B {
  public:
    void f();
  private:
    double g; // Új
};
```

Azzal, hogy a `B` alaposztályba felveszünk egy privát virtuális függvényt, a származtatott osztály korábban nem virtuális függvényét virtuálissá tesszük. Ha a `D` osztályba veszünk fel egy privát adattagot, azzal elrejtünk egy, a `B`-től örökölt függvényt. Az öröklődést ezért gyakran nevezik „fehér dobozos" (vakon történő) újrahasznosításnak, ugyanis az osztályokon végzett változtatások alapvetően befolyásolhatják az alap- és származtatott osztályok jelentését.

E probléma kiküszöbölésének egyik módja az lehet, ha valamilyen egyszerű nevezéktant használunk, amelyben a függvények neve feladatuk szerint tagolódik. Általában az is hasznos, ha más elnevezési szabályokat használunk a típusnevekre, a privát adattagokra, és egy harmadikat minden egyébre. Ebben a könyvben az összes típusnév nagybetűvel kezdődik, az adattagok neve után (ezek valamennyien privát tagok!) egy aláhúzás karakter található, az összes egyéb név (néhány kivételtől eltekintve) pedig kisbetűvel kezdődik. Ez a nevezéktan megakadályozta volna, hogy a fenti példában a `D` osztály `g` nevű függvényét óvatlanul elrejtsük az új tag bevezetésével. Mindazonáltal ha lehet, óvakodjunk valamilyen bonyolult nevezéktan bevezetésétől, ezt ugyanis kényelmetlensége miatt senki nem fogja később követni.

Soha ne próbáljuk az objektum típusát „beleszőni" a nevébe. Ha például egy egész típusú, indexként használatos változót `iIndex`-nek hívunk, azzal nehezítjük a kód olvasását, megértését és karbantartását. Egy névnek mindenekelőtt azt kell tükröznie, hogy az adott objektumot mire használjuk a programban, nem azt, hogy miként valósítottuk meg a szolgáltatást. (Az adatelvonatkoztatást a beépített típusokra is alkalmazhatjuk.) Másrészt soha nem zárható ki, hogy idővel egy objektum típusa változni fog. Ilyenkor persze a legtöbben elfelejtik megfelelően módosítani a nevet, így aztán félrevezetik a későbbi programozókat és karbantartókat.

Az elnevezésekkel kapcsolatban a 70., 73., 74. és 77. hibánál több más megközelítés is olvasható.

9. hiba: A nyelv hibás használata

Amikor néhány évvel ezelőtt a külvilág behatolt a C++ addig meglehetősen zárt köreibe, elterjedt néhány meglehetősen helytelen kódolási szokás. Ebben a részben megkísérlem felsorolni ezeket, és egyben megpróbálom a helyes stílus felé vezetni a tisztelt Olvasót, ha netán ő is az ismertetett hibás megoldásokat használók táborához tartozna.

Nyelvezet

Az 1.1. táblázatban felsoroltam a leggyakoribb megfogalmazási hibákat, a helyes megfelelőikkel együtt.

Nem létezik olyan fogalom, hogy „tisztán virtuális" alaposztály. Van tisztán virtuális függvény, de az osztály, ami ezt tartalmazza vagy nem írja felül, az nem virtuális, hanem elvont.

A C++ nyelvnek nincsenek metódusai. Ilyenek a Javában és a Smalltalkban tényleg léteznek, de a C++-ban nem. Ha általában beszélünk valamilyen objektumközpontú tervezési módszerről, és kísértést érzünk az „üzenet" és „metódus" szavak használatára, megtehetjük. Ha azonban egy C++ nyelven írt rendszerről beszélünk, ilyen helyzetekben a „függvényhívás" és a „tagfüggvény" szavakat kell használnunk.

Sok, egyébként megbízható C++ szakértő (ugye most mindenki tudja, hogy róla beszélek?) gyakran használja az angol szakirodalomban a „destructed" kifejezést, mint a „constructed" ellentétét. Ez nem szakmai hiba, egyszerűen az angol nyelv hibás használata. A megfelelő szó ugyanis a „destroyed".

A C++-ban valóban léteznek típusátalakító (cast) műveletek, egészen pontosan négy darab: a `static_cast`, a `dynamic_cast`, a `const_cast` és a `reinterpret_cast`. Ugyanakkor a „típusátalakító művelet" (cast operator) kifejezést gyakran (és hibásan) olyan tagátalakító műveletekre (conversion operator) is használják, amelyek azt írják elő, hogy egy osztály egy objektumát miként kell beleértett (rejtett, implicit) módon más típusúvá alakítani.

```
class C {
   operator int *()const; // tagátalakító művelet
   // . . .
};
```

1.1. táblázat A legáltalánosabb megfogalmazási hibák és helyes megfelelőik

Hibás	Helyes
Tisztán virtuális alaposztály	Elvont osztály
Metódus	Tagfüggvény
Virtuális metódus	???
Destructed	Destroyed
Típusátalakító művelet	Tagátalakító művelet

Annak természetesen semmi akadálya, hogy kifejezetten meghívjunk egy tagátalakító műveletet egy típusátalakító műveleti jellel, feltéve, hogy pontosan tudjuk, melyik micsoda.

Érdemes megnézni még a 31. hibát az „állandó mutató" (const pointer) és az „állandót címző mutató" (pointer-to-const) körüli felületességgel kapcsolatban.

Nullmutatók

Egyszer volt, hol nem volt, volt egyszer egy olyan probléma a C++ programokkal, hogy ha a `NULL` előfeldolgozó-szimbólumot használtuk a nullmutatóknak történő értékadás során, akkor idővel nagy bajba kerülhettünk:

```
void doIt( char * );
void doIt( void * );
C *cp = NULL;
```

A gondot az okozta, hogy a NULL egyes rendszereken nem pontosan ugyanazt jelentette:

```
#define NULL ((char *)0)
#define NULL ((void *)0)
#define NULL 0
```

Ennek aztán katasztrofális következményei voltak, ha programunkat megpróbáltuk átvinni egyik rendszerről a másikra:

```
doIt( NULL ); // rendszerfüggő, nem egyértelmű
C *cp = NULL; // Hibás?
```

Ami azt illeti, a C++-ban sehogyan sem tudunk biztonságosan megadni egy nullmutatót. Azt viszont garantálja a szabvány, hogy a literális nulla mindig átalakít-

ható nullmutatóvá, akármilyen típusú mutatóról legyen is szó. Így aztán a C++ programozók hagyományosan ezt a módszert használják programjaik hordozhatóságának biztosítása végett. Ma a nyelvet leíró szabvány már pontosan jelzi, hogy az olyan megoldások, mint a `(void *)0` nem megengedettek, a `NULL` azonban egy előfeldolgozó makró, így elvileg használható, de mindenki ferde szemmel fog ránk nézni miatta. Az igazi C++ programozók továbbra is a literális nullát használják a nullmutató megadására. Minden más próbálkozás reménytelenül divatjamúlt.

Betűszavak

A C++ programozók általában „betűszó-betegségben" szenvednek, bár azt a szintet még biztosan nem érték el, mint az igazgatók. Az 1.2. táblázat nagy hasznunkra lehet, ha legközelebb valamelyik kollégánk lakonikusan közli, hogy az RVO-t nem fogják a POD-re alkalmazni, így jobban tesszük, ha megadunk egy copy ctor-t.

1.2. táblázat A leggyakrabban használt betűszavak és azok jelentése

Betűszó	Angol jelentés	Magyarázat
POD	Plain old data	Régi, C értelemben vett adatok, vagy C szerkezet (struct)
POF	Plain old function	Régi, C értelemben vett függvény
RVO	Return value optimization	Visszatérési érték finomhangolása
NRV	Named RVO	Névvel jelölt RVO
ctor	constructor	konstruktor (létrehozó függvény)
dtor	destructor	destruktor (megsemmisítő függvény)
ODR	One definition rule	Az egyszeri meghatározás szabálya

10. hiba: A stílus figyelmen kívül hagyása

„Régi megfigyelés, hogy egyes kiváló írók néha figyelmen kívül hagyják a fogalmazás szabályait. Ilyen esetekben persze az olvasó mindig talál valamiféle pluszt a szövegben, amely kompenzálja a nyelvi szabadosságot. Mindazonáltal az olvasó maga jobban teszi, ha a továbbiakban is ragaszkodik a szabályokhoz, hacsak nem biztos benne teljesen, hogy utánoznia kell az említett írót."

(Strunk és White, *The Elements of Style*)[1]

[1] Az idézett szöveg egyedüli szerzője William Strunk. A részlet már a könyv eredeti kiadásában is szerepelt, mielőtt White 1959-ben felújította volna.

Ez a helyes angol prózával kapcsolatban gyakran idézett tanács szinte változtatás nélkül átvihető a programozási nyelvekre is. Természetesen én magam is mélységesen egyetértek mind a leírtakkal, mind a mögöttük megbúvó érzéssel. Ugyanakkor mégis érzek valami kis elégedetlenséget ezzel a sommás kijelentéssel kapcsolatban. Végül is nem derül ki belőle, mire jó az nekünk, ha ragaszkodunk a fogalmazás szabályaihoz, az meg végleg nem, honnan a csudából származnak ezek a szabályok. Ami azt illeti, és jobban szeretem White „ökörcsapásos hasonlatát", mint Strunk fent idézett isteni kinyilatkoztatását:

„Az élő nyelv olyan, mint az ökörcsapás. Az ökörcsapást az ökrök csinálják maguknak, és mint alkotóknak, jogukban áll rajta járni, vagy letérni róla, ha kedvük vagy érdekeik épp úgy kívánják. A mindennapos használattól az ösvény eleve változik. Az ökröt semmi sem kényszeríti arra, hogy azon a szűk csapáson maradjon, amit történetesen ő maga segített megalkotni, miközben a domborzatnak megfelelően bandukolt. Ugyanakkor gyakran származik haszna abból, hogy rajta marad, néha meg kifejezetten elveszettnek érzi magát, ha nagyon letér róla, és hirtelen nem tudja, merre van az arra."

(E.B. White, *Writings from The New Yorker*)

A programozási nyelvek távolról sem olyan bonyolultak, mint a beszélt nyelv. Tiszta, világos programkódot írni sokkal egyszerűbb, mint szép mondatokat megfogalmazni. Ugyanakkor a C++ van annyira bonyolult nyelv, hogy a hatékony programozáshoz elengedhetetlenül szükséges bizonyos szabályokhoz, kifejezésmódokhoz ragaszkodnunk. A C++ nem előíró jellegű, vagyis lehetővé teszi, hogy a nyelvi elemeket és szolgáltatásokat viszonylag szabadon kombináljuk. Ugyanakkor bizonyos megállapodások következetes követése egyértelműen jelzi a tiszta és átgondolt tervezést. Ezek szándékos vagy hanyag figyelmen kívül hagyása viszont biztos út a katasztrófa felé.

Nagyon sok, ebben a könyvben megfogalmazott tanács kapcsolódik valamilyen módon a szokványos írásmódhoz és tervezési módszerekhez való ragaszkodáshoz. Számos olyan hiba van, amit egyértelműen a helyes írásmódtól való eltérés okoz, a javítás pedig mindössze annyi, hogy vissza kell térnünk a megfelelő stílushoz. Annak, hogy a használható vagy javasolt stílusok ennyire pontosan meghatározhatók, természetesen jó oka van. Ezeket a stílusokat hosszú éveken át fejlesztette a C++ nyelvet használó közösség. Ők már kipróbáltak sokféle tervezési és megvalósítási módszert, és ma már csak azokat javasolják, amelyek kiállták az idő próbáját. Ezek a módszerek biztosan megállják a helyüket egy olyan környezetben, amelyre alkalmazhatók, és amelyre javasolják őket. Ha ragaszkodunk az elődeink által kidolgozott stílushoz, az szinte biztosan hatékony, világos, jól karbantartható kódot fog eredményezni.

Profi programozóként mindig ügyelnünk kell arra, hogy egy adott feladat megvalósítása során kezdetben feltétlenül ragaszkodjunk a környezetnek megfelelő stílushoz és tervezési módszerhez. Később aztán eldönthetjük, hogy végig ezen a keskeny ösvényen haladunk, vagy megfontoltan letérünk róla, igényeinknek megfelelően. Azt fogjuk tapasztalni, hogy általában hasznosabb rajta maradni, sőt ha nagyon letértünk a kijelölt csapásról, akár elveszettnek is érezhetjük magunkat.

Természetesen egy pillanatig sem akarom azt állítani, hogy az általam vagy kollégáim által ismert vagy kidolgozott tervezési módszerek a tervezési folyamat minden mozzanatát lefedik, és egyszerű receptként használhatók. A megfelelő stílus megkönnyíti a tervezési folyamatot, a kollégákkal való együttműködést, de soha nem helyettesíti a programozó kreativitását. Ez utóbbira mindenképpen szükség lesz, de csak a megfelelő helyeken. Vannak aztán olyan helyzetek és környezetek is, amikor egyetlen bevált tervezési módszer sem segít. Ilyenkor a programozónak nincs más választása, mint utat vágni magának.

A létező C++ stílusok közül az egyik leghasznosabb a másoló műveletekkel kapcsolatban kialakult szokásrendszer. Minden elvont objektummal kapcsolatban el kell döntenünk, hogyan működjön a rá vonatkozó másoló és értékadó (hozzárendelő) művelet. A legegyszerűbb esetben hagyhatjuk, hogy a fordító maga állítsa elő ezeket. Megírhatjuk őket mi magunk is, ha tudjuk, hogyan kell, és végül meg is tilthatjuk az adott objektummal kapcsolatban ezeknek a műveleteknek a használatát (lásd a 49. hibát).

Ha a programozó maga írja meg ezeket a műveleteket, akkor természetes, hogy pontosan tudja, mit csinál. Az a „szabványos mód" azonban, ahogyan ezeket a kódrészleteket megírjuk, az idők során némiképp változott. Éppen ez a stílus előnye az előírással szemben: az előbbi idővel változhat, alkalmazkodhat az új igényekhez vagy környezetekhez.

```
class X {
 public:
   X( const X & );
   X &operator =( const X & );
   // . . .
};
```

Bár a C++ nyelv a másoló műveletek megadása tekintetében meglehetősen sok szabadosságot megtűr, csaknem mindig célszerű ezeket a fenti módon programozni. Mindkét művelet egy-egy állandóra vonatkozó hivatkozást igényel, a másolási művelet nem virtuális, visszatérési értéke pedig egy nem állandó értékre való hivatkozás. Világos módon egyik művelet sem módosítja a tényezőt, hiszen ennek semmi értelme nem lenne.

```
X a;
X b( a ); // a tartalma nem fog változni
a = b; // b tartalma nem változik
```

Persze azért néha mégis... A C++-ban szabványos `auto_ptr` sablonnak van néhány fura elvárása. Ez egy olyan erőforrás-kezelő, amelynek az a feladata, hogy felszabadítsa a lefoglalt tárat, ha arra már nincs szükség:

```
void f() {
    auto_ptr<Blob> blob( new Blob );
    // . . .
    // A Blob objektum számára lefoglalt terület automatikus
➥ felszabadítása
}
```

Igen ám, de mi van akkor, ha néhány slendrián diák „rászabadul" erre a kódra?

```
void g( auto_ptr<Blob> arg ) {
    // . . .
    // A Blob objektum számára lefoglalt terület automatikus
➥ felszabadítása
}
void f() {
    auto_ptr<Blob> blob( new Blob );
    g( blob );
    // A Blob objektum egy újabb törlése!!!
}
```

Az egyik védekezési mód az lehetne, hogy megtiltjuk a másolási műveletet az `auto_ptr` objektumra. Ezzel azonban sajnos erősen korlátoznánk annak ésszerű használatát, és eleve megakadályoznánk a programozókat abban, hogy bizonyos – amúgy igen hasznos – tervezési módszereket alkalmazzanak. Egy másik megoldás az lenne, ha egy hivatkozásszámlálót építenénk be az `auto_ptr` objektumba. Ez viszont ésszerűtlenül megnövelné az erőforrás-kezelés teljesítményigényét. Ezért a szabványos megoldás meglepő módon az, hogy az `auto_ptr` megvalósításában a programozók szándékosan eltértek a másolási műveletekkel kapcsolatos szokásoktól:

```
template <class T>
class auto_ptr {
    public:
    auto_ptr( auto_ptr & );
    auto_ptr &operator =( auto_ptr & );
    // . . .
```

```
  private:
    T *object_;
};
```

(A szabványos `auto_ptr` objektum tartalmaz az itt látható, nem sablonként adott másolási műveleteknek megfelelő sablonokat is, de ezek viselkedése azonos. Lásd még a 88. hibát.) Mint látható, itt egyik művelet jobb oldala sem állandó! Ha egy `auto_ptr` objektumhoz egy másikat rendelünk (kezdeti beállításnál vagy későbbi értékadásnál), akkor a forrásobjektum átadja a kérdéses memóriaterület tulajdonjogát a másiknak, mégpedig úgy, hogy belső objektum mutatóját nullára állítja.

Amint az már lenni szokott, ha elhagyunk egy megszokott írásmódot, kezdetben volt némi zavar akörül, hogyan is kell pontosan használni egy `auto_ptr` objektumot. Ugyanakkor ez az új megközelítés lehetővé tette néhány, az objektumok tulajdonjogával kapcsolatos igen hasznos új stílus kifejlődését. Az `auto_ptr` „adatforrásként" és „adatnyelőként" való használata például kifejezetten gyümölcsöző ötletnek tűnik. Összefoglalva tehát, ennek az esetnek a fő tanulsága az, hogy ha jól átgondoltan, szakértő módjára térünk le a kitaposott ösvényről, az akár forradalmian új megközelítések megszületését is eredményezheti.

11. hiba: Az „ügyes" megoldások túlzott használata

Úgy tűnik, a C és a C++ elképesztő mennyiségű magamutogató embert vonz. (Talán az Olvasó is hallott már az „ámokfutó Eiffel" versenyről.) Ezek a programozók valahogy úgy képzelik, hogy két pont között a legrövidebb út az euklideszi tér egy gömbi torzulásának főkörén át vezet.

Mármost a C++ körökben (euklideszi meg minden egyéb fajtában) közismert, hogy a forráskód formázása kizárólag az azt olvasó ember érdekeit szolgálja. A fordítás folyamatára és eredményére elvileg semmiféle hatása nincs, feltéve persze, hogy ugyanazok a jelek ugyanabban a serrendben követik egymást. Ez az utóbbi kitétel persze nagyon fontos, ugyanis a következő két sor például nagyon hasonló, mégis mást jelentenek (amúgy nézzük meg a 87. hiba leírását):

```
a+++++b; // Hiba!
a+++ ++b; // Rendben
```

Akárcsak a következő kettő (lásd a 17. hibát):

```
ptr->*m; // Rendben
ptr-> *m; // Hiba!
```

Mint mondottuk – és ezzel a legtöbb C++ programozó is egyetért majd – a program szövegének formázása nem befolyásolja annak jelentését. (Eltekintve talán a bemenő karaktersorozatok formázásától és értelmezésétől.) Így például lehetőségünk van arra, hogy egy változó bevezetését több sorban végezzük el. Az eredmény ugyanaz lesz. (Egyes fejlesztőkörnyezetek nyomkövetői és egyéb eszközei egyszerűen sorszámokkal azonosítják a programkód helyeit, ahelyett, hogy valamilyen egzaktabb módon kezelnék a nyelvi elemeket. Ez aztán arra készteti a programozókat, hogy efféle furcsa, kényelmetlen, sőt egyenesen természetellenes formázást alkalmazzanak. Csak így tudnak ugyanis értelmes hibaüzeneteket kicsikarni a rendszerből, és egyedül ez ad nekik lehetőséget arra, hogy a töréspontokat a megfelelő helyekre illesszék be. Mindennek persze semmi köze magához a C++ nyelvhez. Itt a programozási környezetek fejlesztőinek kellene jobban dolgozni.)

```
long curLine = __LINE__; // Aktuális sorszám
long       curLine
     =                    __LINE__
  ; // Ugyanaz a deklaráció
```

Ami a formázás veszélytelenségét illeti, sajnos a legtöbb C++ programozó téved. Nézzük például a következő metaprogramozási sablont, amely fordítási időben határoz meg egy típust:

➡ 11. hiba/select.h

```
template <bool cond, typename A, typename B>
struct Select {
   typedef A Result;
};
template <typename A, typename B>
struct Select<false, A, B> {
   typedef B Result;
};
```

Ha létrehozunk egy, a `Select` sablonnak megfelelő objektumot, az fordításkor egy logikai feltétel kiértékelését váltja ki, majd az eredménytől függően a sablon két lehetséges változata közül érvénybe lép az egyik. Ez tehát egy fordítási időre vonatkozó `if` utasítás, ami azt mondja, hogy ha a feltétel igaz, akkor a beágyazott `Result` típusa `A`, ellenkező esetben `B`.

➡ 11. hiba/lineno.cpp

```
Select< sizeof(int)==sizeof(long), int, long >::Result temp = 0;
```

Ez a programsor `int`-ként vezeti be a `temp` nevű változót, ha az `int` és a `long` azonos számú bájton ábrázolható, ellenkező esetben a `temp` típusa `long` lesz.

Nézzük most a `curLine` korábbi deklarációját. Felmerülhet bennünk a kérdés: miért vesztegetjük az értékes memóriát egy `long` típusra, ha nem is használjuk ki? Legyünk tehát a példa kedvéért kissé bonyolultabbak (bár tulajdonképpen nem sok okunk van rá):

➡ 11. hiba/lineno.cpp

```
const char CM = CHAR_MAX;
const Select<__LINE__<=CM,char,long>::Result curLine = __LINE__;
```

Ez működik, sőt még helyes is, csak ez a sor egyeseknek túl hosszú. Így aztán a program karbantartója, aki utánunk érkezik, átalakítja egy kissé:

➡ 11. hiba/lineno.cpp

```
const Select<__LINE__<=CM,char,long>::Result
   curLine = __LINE__;
```

Na, most már tényleg rossz. Ugye látszik, miért?

Mi történik, ha a bevezetés a `CHAR_MAX` által jelzett sorban történik (ami most egyébként mindössze a 127-es)? A `curLine` típusa ilyenkor `char` lesz, alapértelmezett értéke pedig az ebben a típusban tárolható legnagyobb szám. Amint a beállítás átcsúszik a következő sorba, a `char` típusú változóban eggyel nagyobb számot próbálunk meg elhelyezni, mint ez a bizonyos maximum. Az eredmény valószínűleg egy negatív sorszám lesz (például -128). Ügyes.

A szükségtelen okoskodás a C++ programozók egyik leggyakoribb hibája. Soha ne feledjük, hogy egy kevésbé „ügyes", kevésbé hatékony, de teljesen áttekinthető és könnyedén karbantartható kód sokkal hasznosabb, mint egy elvetemülten trükkös, áttekinthetetlen és „karbantarthatatlan".

12. hiba: Az „ifjonti hév"

Mi programozók általában élen járunk a jótanácsok osztogatásában, de mi magunk csak ritkán tartjuk be, amit mi magunk mondtunk. Prédikálunk például a globális változók, az érthetetlen változónevek meg a mágikus számok ellen, de amikor leülünk programozni, bizony használjuk őket rendesen. Bevallom, emiatt a jelenség miatt én sokáig szégyenkeztem, míg aztán egy magazinban ugyanennek a jelenség-

nek a leírását olvastam, csak a serdülők viselkedésével kapcsolatban. Egészen gyakori jelenség, hogy a serdülők kritizálnak valakit annak kockázatos cselekedetei miatt, de belül végül is arra a következtetésre jutnak, hogy őket semmi baj nem érheti, ha ugyanezt teszik. A programozók, mint sajátos társadalmi osztály, úgy tűnik, visszamaradtak az érzelmi fejlődésben.

Dolgoztam olyan helyeken, ahol a programozók nemcsak hogy megtagadták a kódolási szabványok betartását, hanem egyenesen felmondással fenyegetőztek pusztán azért, mert arra kértem őket, hogy kettő helyett négy szóközt használjanak a szintek jelölésére. Láttam olyan esetet, amikor a programozók egyik csoportja nem volt hajlandó elmenni arra az értekezletre, amelyen egy másik csoport tagjai is jelen voltak. Láttam, amint programozók dokumentálatlan és átláthatatlan kódot állítottak elő, talán pont azért, hogy rajtuk kívül senki emberfia ne legyen képes megérteni. Láttam aztán amúgy tehetséges embereket, akik nem voltak hajlandók tanácsot elfogadni a másiktól, csak azért, mert az öregebb/fiatalabb/egyenesebb ember volt, vagy agyon volt lyuggatva testékszerekkel. Az eredmény aztán az esetek többségében kisebb-nagyobb katasztrófa lett.

Akár serdülők vagyunk lélekben, akár nem, mindannyiunknak vannak „felnőttes vonásai", és felelősségei. (Ezzel kapcsolatban nem árt elolvasni például az Association for Computing Machinery e témával kapcsolatos állásfoglalásait, melyek az *ACM Code of Ethics and Professional Conduct* és a *Software Engineering Code of Ethics and Professional Practice* című lapokban jelentek meg.)

Először is tartozunk annyival a szakmánknak, hogy képességeinkhez mérten a lehető legjobbat nyújtjuk minden helyzetben.

Másodszor tartozunk ezzel a társadalomnak, amelyben, és a Földnek, amelyen élünk. Választott szakmánk egyenlő arányban tudomány, és szakmai szolgáltatás. Ha munkánkkal nem járulunk hozzá ahhoz, hogy a világnak ez a szeglete szebb, jobb és élhetőbb legyen, akkor elvesztegettük a tehetségünket, az időnket és végsősoron az életünket.

Harmadszor tartozunk szűkebb közösségünknek azzal, hogy rendesen végezzük a munkánkat, ez az egyetlen módja ugyanis annak, hogy a politikusokra befolyást gyakorolhassunk. Bár társadalmunk egyre inkább egy technikai civilizáció képét ölti, hivatalnokaink többsége jogász vagy közgazdász. Ez sajnos egyben azt is jelenti, hogy műszaki szempontból ezek az emberek jószerével írástudatlanok, sőt a számokkal is hadilábon állnak. Az egyik amerikai államban például hoztak egy rendeletet, miszerint a pi értéke legyen 3. Ez persze röhejes. (Ami azt illeti, a kerekeken gördülő szállítójárművek kissé akadozva haladtak, amíg a törvényt vissza nem von-

ták). Van azonban számos olyan kormányzati döntés, aminek a téves volta nem ennyire nyilvánvaló. Nekünk, írástudóknak, kötelességünk tájékoztatni a kormányzati szerveket döntéseik hatásairól, érveinket pedig számokkal és logikával alátámasztani.

Negyedszer tartozunk a kollégáinknak azzal, hogy kollegiálisak vagyunk. Ez magában foglalja a helyi kódolási szabályok tiszteletben tartását, az átlátható és karbantartható kód írását, no meg azt, hogy odafigyelünk a másikra, ha közölni akar velünk valamit. (Lehet persze, hogy a helyi kódolási előírások nem jók. Ebben az esetben sem az a megoldás, hogy figyelmen kívül hagyjuk, hanem az, hogy megváltoztatjuk őket.)

Természetesen mindezzel nem azt akarom mondani, hogy ész nélkül játsszuk a „csapatjátékost", vagy lelkesedjünk a vállalat erőltetett belső vagy külső uniformizálásáért. Legtiszteletreméltóbb munkatársaim közül néhány meglehetősen furcsán öltözött, magának való ember volt, amihez rendszerint szokatlan politikai beállítottság és egyedi szokások társultak. Ezek az amúgy furcsa emberek mindig meghallgatták, sőt kifejezetten respektálták az elképzeléseimet, feltéve persze, hogy rászolgáltam. Ha pedig hibáztam, azt is bátran megmondták, és mindig igazuk volt. Együtt dolgoztunk, és mindig pontosan azt és úgy valósítottuk meg, amiben megegyeztünk.

Ötödször tartozunk azzal a szakmabelieknek, hogy megosztjuk velük a tudásunkat és tapasztalatainkat.

Hatodszor tartozunk önmagunknak is. Munkánk és gondolataink örömet kell, hogy okozzanak nekünk, hiszen ez az, amiért ezt a szakmát választottuk. És talán ez a legfontosabb mind közül. Ha élvezzük, amit csinálunk, ha az, amit csinálunk, személyiségünk része, akkor a felsorolt kötelezettségeket nem tehernek, hanem szórakozásnak érezzük majd.

2

Nyelvtan

A C++ nyelvnek mind a nyelvtana, mind a nyelvi elemei meglehetősen összetettek. Ezt részben a C nyelvtől örökölte, részben pedig bizonyos nyelvi szolgáltatások végett van rá szükség.

Ebben a fejezetben néhány, a nyelvtannal kapcsolatos problémát tekintünk át. Ezek egy része egyszerűen elírásnak minősíthető, amelyek azonban meglepő módon „sikeresen" lefordíthatók. Az eredmény persze meglepő lesz. Mások végső soron abból erednek, hogy a nyelvtan és a működés között néha meglehetősen laza a kapcsolat, így könnyű hibázni. Megint mások arra vezethetők vissza, hogy a C++ programok és programozók stílárisan sokfélék lehetnek. Így ha két programozónak megmutatjuk ugyanazt a kódrészletet, és megkérjük őket, mondják el, mit látnak, nem biztos, hogy ugyanaz lesz a végeredmény.

13. hiba: A tömbök és a kezdeti beállítások összekeverése

Foglaljunk tárterületet 12 egész típusú változó számára! Ugye ezzel még semmi gond nincs?

```
int *ip = new int(12);
```

Minden nagyon szép, minden nagyon jó, mindennel meg vagyunk elégedve. Aztán ha végeztünk, szépen eltakarítjuk magunk után a szemetet:

```
for( int i = 0; i < 12; ++i )
   ip[i] = i;
delete [] ip;
```

Figyeljük meg az üres zárójelek használatát. Innen tudja a fordító, hogy `ip` nem egyetlen egészre, hanem egészek tömbjére mutat. Vagy hogy is van ez?!

Nos, ami azt illeti, az `ip` valójában mégis csak egyetlen egészre mutat, aminek a `12` kezdőértéket adtuk. Elkövettük ugyanis azt a gyakori gépelési hibát, hogy szögletes zárójelek helyett kerek zárójeleket használtunk. Ebből természetesen az is következik, hogy mind a ciklusban történő értékadások (legalábbis a nulladik elem utániak), mind a későbbi törlő utasítás hibás. Ugyanakkor kicsi annak a valószínűsége, hogy a fordító ezt a hibát felfedezze, hiszen egy mutató tényleg címezhet egyetlen

egészet és egy egész tömböt is. Ami a nyelvtant illeti, a két eset között semmi különbség nincs, így a fordító az érvénytelen értékadásokat és a törlést is elhiszi nekünk. Aztán futásidőben persze minden kiderül, de ezek alapján a hibát nem biztos, hogy könnyen megtaláljuk.

Sőt, az is lehet, hogy még futásidőben sem lesz semmi gond. Természetesen nem megengedett egy mutatóval a megfelelő objektumon túlra címezni (bár maga a nyelv megengedi egyetlen elem objektumon túli címzését), és egyetlen elemet tömbként törölni sem lehet. Sajnos azonban az, hogy egyszer valami rosszaságot csinálunk, nem feltétlenül jelenti azt, hogy le is bukunk. (Gondoljunk csak a tőzsdére!) Az ilyen kód esetleg hibátlanul fut az egyik vagy néhány rendszeren, és következetesen hibázik a másikon. De az is lehet, hogy kiszámíthatatlanul fog viselkedni. Attól függően megy vagy nem, hogy hogyan használjuk előtte a memóriát. A helyes tárfoglalás természetesen a következőképpen fest:

```
int *ip = new int[12];
```

Mindazonáltal a legbiztosabb megoldás, ha egyáltalán nem foglalkozunk a tár kezelésével, hanem a szabványos könyvtárat használjuk:

```
std::vector<int> iv( 12 );
for( int i = 0; i < iv.size(); ++i )
    iv[i] = i;
// Nincs kifejezett törlés...
```

A szabványos `vector` sablon csaknem ugyanolyan hatékony, mint a „kézzel" lefoglalt tár, de sokkal biztonságosabb, gyorsabb és áttekinthetőbb. Általánosságban tehát célszerűbb a `vector` sablont használni az alacsonyszintű tömbök helyett. Ami azt illeti, ugyanez a hiba előjöhet egy egyszerű deklaráció során is, de ezt valószínűleg könnyebb felderíteni:

```
int a[12]; // 12 egészet tartalmazó tömb
int b(12); // Egyetlen egész, ami a 12 kezdőértéket kapja
```

14. hiba: Határozatlan kiértékelési sorrend

A C++ és a hagyományos C nyelv rokonsága talán az utasítások kiértékelési sorrendjével kapcsolatban a legszembetűnőbb. Bizton állíthatjuk, ez az a csapda, ami minden figyelmetlen kezdőre leselkedik. Mindkét nyelv számos lehetőséget ad

a programozó számára a kiértékelési sorrend meghatározására. És éppen ez okozza a problémákat, amelyek számos különböző formában jelenhetnek meg ugyan, de a gyökerük, a tényleges hiba tulajdonképpen mindig ugyanaz. A nyelv e kiértékelési sorrenddel kapcsolatos rugalmassága egyrészt hasznos, mivel nagyon hatékony kód előállítását teszi lehetővé, másrészt megköveteli a programozótól, hogy nagyon pontosan tudja, mit csinál. Aki a kiértékelési sorrendet megpróbálja megkerülni, és egyszerűen csak logikusnak tűnő feltételezésekre alapozza a munkáját, az biztosan hibázni fog.

Függvényparaméterek kiértékelési sorrendje

```
int i = 12;
int &ri = i;
int f( int, int );
//  . . .
int result1 = f( i, i *= 2 ); // Nem hordozható
```

A függvények paramétereinek kiértékelési sorrendjét alapvetően semmi sem rögzíti. Így aztán a fenti példában az `f` függvény vagy a `12` és a `24` számokat kapja meg bemenő paraméterként, vagy kétszer a `24`-et. Az óvatos programozó persze eleve nem módosít egy paramétert, ha az egynél többször fordul elő egy paraméterlistában, itt azonban ez sem segít:

```
int result2 = f( i, ri *= 2 ); // Nem hordozható
int result3 = f( p(), q() ); // Kockázatos...
```

Az első esetben `ri` az `i` változó álneve, így a `result2` értéke ugyanolyan bizonytalan, mint a `result1` változóé. A második esetben hallgatólagosan feltételezzük, hogy a `p()` és `q()` függvények kiértékelési sorrendje lényegtelen. Mármost az lehetséges, hogy ez a feltételezés ebben a pillanatban még igaz is, de semmi biztosíték nincs rá, hogy a jövőben is az lesz. Ráadásul ez a `p`-vel és `q`-val kapcsolatos feltevés sehol sincs dokumentálva.

Összefoglalva tehát az a legjobb, ha minimálisra csökkentjük a függvényhívásokkal kapcsolatos lehetséges félreértések számát:

```
result1 = f( i, i*2 );
result2 = f( i, ri*2 );
int a = p();
result3 = f( a, q() );
```

Részkifejezések kiértékelési sorrendje

A részkifejezések kiértékelési sorrendjével kapcsolatban szintén semmiféle megkötés nincs:

```
a = p() + q();
```

Lehet, hogy a `p` függvény a `q` előtt fut le, de az sem kizárt, hogy fordítva lesz. A műveletek rangja és kötése (asszociativitása) szintén nem tisztázza a helyzetet:

```
a = p() + q() * r();
```

A `p`, `q` és `r` függvények már hat különböző sorrendben értékelődhetnek ki, és a hat közül bármelyik lehetséges. A szorzás magasabb rangja mindössze azt biztosítja, hogy a `q` és az `r` eredménye azelőtt lesz összeszorozva, mielőtt a program összeadná ezt a `p` eredményével. Ugyanígy a következő példában sem garantálja a hívási sorrendet az összeadás balról kötő tulajdonsága. Mindössze abban lehetünk biztosak, hogy a program az eredményeket balról jobbra fogja összeadni:

```
a = p() + q() + r();
```

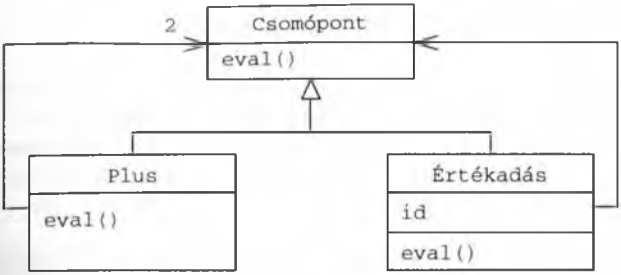
Az sem segít, ha szándékunknak megfelelően zárójelezzük a kifejezést:

```
a = (p() + q()) * r();
```

Az biztos, hogy programunk előbb össze fogja adni `p` és `q` eredményét, mielőtt megszorozná azt az `r` visszatérési értékével, de ennek az állításnak semmi köze a három függvény kiértékelési sorrendjéhez. Egyáltalán nem biztos, hogy az `r` lesz az először kiértékelt függvény. A részkifejezések kiértékelési sorrendjének megkötésére egyetlen valóban hatásos módszer létezik: ideiglenes változókat kell használnunk, és ezekben a nekünk tetsző sorrendben kell elhelyeznünk a részkifejezések eredményeit:

```
a = p();
int b = q();
a = (a + b) * r();
```

Hogy ez a probléma milyen gyakran fordul elő? Elég gyakran ahhoz, hogy évente tönkretegyen egy-két hétvégét. Vessünk például egy pillantást a 2.1. ábrára. Ez egy egyszerű számológép logikai kiértékelési összefüggéseit mutatja.



2.1. ábra

Egy egyszerű számológép elvont szerkezete, (egyszerűsített) fagráfként ábrázolva. A plusz csomópontnak van egy jobb és egy bal oldali részfája. Az értékadás (egyenlőség) műveletnek ugyanakkor csak egyetlen részfája van, ami a művelet jobb oldalának felel meg.

A logikai fa következő megvalósítása nem hordozható:

➡ 14. hiba/e.cpp

```
int Plus::eval() const
    { return l_->eval() + r_->eval(); }
int Assign::eval() const
    { return id->set( e_->eval() ); }
```

A gondot a `Plus::eval` megvalósítása okozza, mivel itt a jobb és bal részfa kiértékelési sorrendje bizonytalan. No de van ennek jelentősége az összeadással kapcsolatban? Végül is az összeadás tényezői felcserélhetők! Nos igen, de nézzük csak, mi történik a következő kifejezés kiértékelése során:

```
(a = 12) + a
```

Attól függően, hogy a `Plus::eval()` jobb és bal részfája milyen sorrendben értékelődik ki, a művelet eredménye vagy `24` lesz, vagy az `a+12` művelet előző eredménye. Ha azt akarjuk, hogy számológépünk az értékadást az összeadás előtt hajtsa végre, a `Plus::eval` megvalósítása során kénytelenek leszünk bevezetni egy ideiglenes változót:

➡ 14. hiba/e.cpp

```
int Plus::eval() const {
    int lft = l_->eval();
    return lft + r_->eval();
}
```

A new művelettel kapcsolatos kiértékelési sorrend

A következőkben tárgyalt eset nem valami gyakran bukkan fel. A new művelet utasításformája nemcsak azt teszi lehetővé, hogy az új objektum lefoglalásakor az alapbeállítást végző kódnak (általában egy konstruktornak) adjunk át paramétereket, hanem azt is, hogy magával a lefoglalást végző operator new függvénnyel kommunikáljunk ilyen módon.

```
Thing *pThing =
   new (getHeap(), getConstraint()) Thing( initval() );
```

A fenti paraméterlistát megkapja az operator new függvény, ami elfogadhatja azokat, de megkapja a Thing konstruktora is. Erre a listára is igaz azonban a korábban a függvények kiértékelési sorrendjével kapcsolatban felvetett aggály: nem tudhatjuk, hogy a getHeap vagy a getConstraint fut le előbb. Sőt, azt sem tudhatjuk, hogy az operator new vagy a Thing konstruktorának paraméterei értékelődnek ki előbb. Egyedül abban lehetünk biztosak, hogy az operator new előbb fog végrehajtódni, mint a konstruktor, hiszen amíg nem foglaltunk tárterületet egy objektumnak, addig nem is hozhatjuk létre.

A kiértékelési sorrendet meghatározó műveletek

Egyes műveletek kiértékelési sorrendet meghatározó képességében sokkal inkább bízhatunk, mint másokéban. Feltéve persze, hogy békén hagyjuk őket. A vessző például egyértelműen meghatározza a vele elválasztott kifejezések kiértékelési sorrendjét:

```
result = expr1, expr2;
```

Ez a sor előbb kiértékeli az expr1-et, aztán az expr2-t, majd ez utóbbi eredményét helyezi el a result változóban. Persze ezt is – mint megannyi más nyelvi szolgáltatást – felhasználhatjuk arra, hogy valami szokatlant hozzunk össze vele:

```
return f(), g(), h();
```

Nos, aki ezt ebben a formában képes leírni, annak szüksége lenne még némi szocializációra. Ha egy mód van rá, ne írjunk ilyeneket, hacsak nem kifejezetten az a célunk, hogy összezavarjunk másokat. Mennyivel jobban néz ki ugyanennek a következő megfogalmazása!

```
f();
g();
return h();
```

A vessző egyetlen valóban gyakori felhasználása az olyan `for` ciklus, amelyben több ciklusváltozót használunk:

```
for( int i = 0, j = MAX; i <= j; ++i, --j ) // . . .
```

Érdemes megjegyezni, hogy itt a legelső vessző nem a vessző művelet (operátor), hanem az `i` és `j` változók bevezetésének része.

Az `&&` és `||` műveletek segítségével egészen összetett logikai hálózatokat tervezhetünk szabványos és áttekinthető módon:

```
if( f() && g() ) // . . .
if( p() || q() || r() ) // . . .
```

Az első kifejezés a következőt jelenti: „hívd meg az `f` függvényt; ha a visszatérési értéke igaz, akkor a feltétel igaz; ha a visszatérési érték hamis, hívd meg a `g` függvényt és a logikai kifejezés értékének tekintsd ennek a visszatérési értékét". Ehhez hasonlóan a második sor a következőt jelenti: „hívd meg a `p`, `q` és `r` függvényeket pontosan ebben a sorrendben, de ha közben bármelyik visszatérési érték igaz, rögtön állj meg; ha mindhárom visszatérési érték hamis, akkor a végeredmény is hamis, ellenkező esetben igaz." Elnézve ezt a két meglehetősen összetett, de tömör és mégis áttekinthető kódrészletet, rögtön világos, miért használják ezt a két műveletet olyan gyakran a C és C++ programozók.

A háromtényezős feltételes művelet, a „`?:`", szintén kötött kiértékelési sorrendet biztosít:

```
expr1 ? expr2 : expr3
```

Először mindig az első kifejezés értékelődik ki, aztán ennek eredményétől függően vagy csak a második, vagy csak a harmadik. Az egész feltételes kifejezés végeredménye a két utóbbi közül ténylegesen kiértékelt kifejezés visszatérési értéke.

```
a = f()+g() ? p() : q();
```

Ebben az esetben a függvények kiértékelési sorrendje meghatározott. Abban biztosak lehetünk, hogy az `f` és a `g` előbb fut le, mint a `p` vagy a `q`, illetve azt is tudjuk,

hogy a két utóbbi közül csak az egyik fog lefutni. Az azonban, hogy az `f` és a `g` milyen sorrendben értékelődik ki, továbbra is bizonytalan. Az olvashatóságot talán javítja, ha e kettőt kerek zárójelek közé tesszük, ez azonban nem kötelező:

```
a = (f()+g()) ? p() : q();
```

Ha nem használjuk ezt a zárójelezést, előfordulhat, hogy a kódunkat karbantartó programozó később lázas sietségében feltételezi, hogy az összeadás csak a logikai vizsgálat után hajtódik végre, ami persze merő tévedés:

```
a = f()+(g() ? p() : q());
```

Műveletek téves túlterhelése

A műveletek nyelvbe beépített változatai rendkívül hasznosak, túlterhelni őket azonban már nem jó ötlet. A C++-ban a túlterhelés csak amolyan nyelvi fűszer (úgynevezett „szintaktikus cukor"). Azt teszi lehetővé, hogy függvényhívás helyett valami jobban követhető formát használjunk, de semmi több. Ha például azt akarjuk elérni, hogy az `&&` művelet elfogadjon két `Thing` nevű paramétert, akkor megpróbálhatjuk túlterhelni:

```
bool operator &&( const Thing &, const Thing & );
```

Ha a műveleti jelet közbevetett jelként (infix forma) használjuk, a kódot értelmező programozók azt gondolhatják, hogy a szabványos beépített műveletről van szó, és a visszafejtés során ennek viselkedésmódját tételezik fel – persze tévesen:

```
Thing &tf1();
Thing &tf2();
// . . .
if( tf1() && tf2() ) // . . .
```

Ez a kód, tényleges jelentését tekintve, azonos a következő függvényhívással:

```
if( operator &&( tf1(), tf2() ) ) // . . .
```

Amint azt korábban láttuk, mind a `tf1`, mind a `tf2` függvény lefut, de azt nem tudjuk, hogy milyen sorrendben. Ugyanez a probléma természetesen a `||` és a `,` (vessző) műveletek túlterhelése esetén is felléphet. Szerencsére a nyelvi szabvány a „`?:`" művelet túlterhelését megtiltja.

15. hiba: Problémák az elsőbbség körül

Ez a rész nem arról fog szólni, hogy a grófnőnek vagy a bárónőnek kell-e a nagykövet mellett ülnie a vacsorán (ennek a problémának köztudottan nincs megoldása). Nem kérem! Itt arról lesz szó, hogy a C++ kifejezéseiben előforduló műveletek különböző rangja miféle bosszantó helyzeteket teremthet.

Kötés és elsőbbség

Általában hasznos, ha egy programozási nyelvben a különböző műveleteknek kötött elsőbbségi sorrendje van, mivel ez lehetőséget teremt az összetett kifejezések egyszerűsítésére anélkül, hogy irgalmatlan mennyiségű zárójelet kellene használnunk. (Ugyanakkor nem haszontalan, ha az erősen összetett kifejezéseket megfelelően zárójelezzük, ez ugyanis javítja az olvashatóságot. Természetesen ellenkező hatást érünk el, ha a mindenki számára érthető kifejezéseket is „agyonzárójelezzük".)

```
a = a + b * c;
```

Pontosan tudjuk, hogy a fenti kifejezésben a `*` műveletnek a legmagasabb a rangja, úgy is fogalmazhatunk, hogy ennek a legnagyobb a kötési ereje. Ugyanígy az is világos, hogy az értékadás művelete az utolsó a ranglétrán, tehát ez hajtódik végre legutoljára.

```
b = a = a + b + c;
```

Azt ebben az esetben is pontosan tudjuk, hogy az összeadások végrehajtása meg fogja előzni az értékadást, hiszen a rangsor ezt a sorrendet diktálja. Az azonban a rangsorból már nem derül ki, hogy a két összeadás és a két értékadás közül melyik lesz az első. Ehhez a műveletek kötését (asszociativitását) kell megvizsgálnunk. A C++-ban a műveletek jobbról kötők vagy balról kötők lehetnek. Az összeadás történetesen balról kötő, ami azt jelenti, hogy a és `b` előbb összeadódik, és csak aztán adódik hozzá ez az eredmény a `c` tartalmához.

Az értékadás ezzel szemben jobbról köt, vagyis az `a+b+c` művelet eredményét a program előbb az `a`-ban fogja elhelyezni, majd a tartalmát másolja át `b`-be. Egyes nyelvekben léteznek nem asszociatív műveletek is. Ezekben egy `a@b@c` jellegű művelet nem megengedett, ha `@` nem asszociatív. Ezer szerencse, hogy a C++ egyáltalán nem tartalmaz nem asszociatív műveleteket.

Elsőbbségi problémák

Az `iostream` könyvtárat úgy tervezték meg, hogy a lehető legkevesebb zárójel használatát igényelje:

```
cout << "a+b = " << a+b << endl;
```

A `+` művelet rangsorban elfoglalt helye magasabb, mint a balra tolásé, így a fenti kifejezés eredménye éppen az, amit elvárunk tőle: a program előbb összeadja a két változót, csak aztán küldi az eredményt a `cout`-ra.

```
cout << a ? f() : g();
```

Itt a C++ egyetlen háromtényezős műveletének használata bajt okoz, de az ok nem a három tényezőben keresendő, hanem abban, hogy a „`?:`" rangja alacsonyabb, mint a `<<` műveleté. Ez a sor tehát azt jelenti, hogy kiküldjük az a változó tartalmát a `cout`-ra, majd ugyanezt a tartalmat egy logikai kifejezés feltételeként is felhasználjuk. A legtragikusabb, hogy ez a műveletsor nyelvileg teljesen rendben van. (A kimeneti objektumoknak, mint amilyen a `cout` is, van egy `operator void *` tagja, amely közvetve felhasználható `void *` típusú érték előállítására, ami viszont ismét felhasználható arra, hogy segítségével logikai igaz vagy hamis értéket állítsunk elő, attól függően, hogy a mutató értéke nulla vagy sem.) A kívánt célt itt csak zárójelek segítségével érhetjük el:

```
cout << (a ? f() : g());
```

Ha még áttekinthetőbb kódot akarunk, így is eljárhatunk:

```
if( a )
   cout << f();
else
   cout << g();
```

Ebből a megoldásból persze már hiányzik az a hanyag elegancia, mint az előzőből, cserébe azonban bárki számára könnyen érthető és karbantartható.

Ritka az a C++ programozó, aki képes belekeveredni az objektumok mutatóival kapcsolatos elsőbbségi viszonyokba, ugyanis köztudott, hogy a `->` és a `.` (pont) műveletek rangja igen magas. Így aztán, ha egy olyan kifejezést olvasunk valahol, mint például `a=++ptr->mem`, akkor minden további nélkül tisztában vagyunk vele, hogy a `ptr` által címzett `mem` nevű tag tartalmát kell növelni. Ha az lett volna a szán-

dékunk, hogy előbb a mutatót toljuk el eggyel, akkor ezt írtuk volna: `a=(++ptr)->mem`. Vagy legfeljebb ezt: `++ptr; a=ptr->mem`. Vagy ha tényleg nagyon rossz napunk van, akkor ezt: `a=(++ptr , ptr->mem)`.

Ha a mutató nem egy osztályt, hanem annak egy elemét címzi, már egészen más a helyzet. Ezeket a mutatókat a kérdéses objektumnak megfelelő módon kell használni. Erre a célra két külön művelet létezik. Az egyik a `->*`, amit akkor alkalmazunk, ha az osztályelemet címző mutatót úgy akarjuk használni, hogy magát az osztályt is egy másik mutatón keresztül érjük el. A másik a `.*`, ami arra szolgál, hogy vele az osztályelemet az osztály ismeretében érjük el.

A tagfüggvényeket címző mutatók ugyan gyakran okoznak fejfájást, de utasításformájuk nem különösebben problémás:

```
class C {
    // . . .
    void f( int );
    int mem;
};
void (C::*pfmem)(int) = &C::f;
int C::*pdmem = &C::mem;
C *cp = new C;
// . . .
cp->*pfmem(12); // Hiba!
```

Itt fordítási hibát kapunk, mivel a függvényhívás műveletének magasabb a rangja, mint a `->*` műveleté. Sajnos azonban egy mutatóval címzett függvényt addig nem hívhatunk meg, amíg a mutatót vissza nem követtük. A megoldás természetesen a megfelelő zárójelezés:

```
(cp->*pfmem)(12);
```

Az adattagokat címző mutatók már problémásabbak. Nézzük például a következő kifejezést:

```
a = ++cp->*pdmem
```

A `cp` változó itt egy ugyanolyan, osztálypéldányt címző mutató, mint az előbb, a `pdmem` viszont nem adattag, hanem egy azt címző mutató. Mivel a `->*` műveletnek alacsonyabb a rangja, mint a `++`-nak, a program a `cp`-t fogja eggyel léptetni az adattagot címző mutató követése előtt. Hacsak a `cp` nem osztálypéldányok egy egész tömbjére mutat, az eredmény valami téves művelet lesz.

Az osztálytagokat címző mutatókkal számos C++ programozónak vannak gondjai. Ezért ha azt akarjuk, hogy programunk a jövőben is könnyen karbantartható legyen, igyekezzünk a lehető legegyszerűbb és legvilágosabb szerkezeteket alkalmazni:

```
++cp;
a = cp->*pdmem;
```

Kötési problémák

A C++ nyelv legtöbb művelete balról kötő, nem asszociatív műveleteket pedig egyáltalán nem használ a nyelv. Mindez számos programozót nem akadályoz meg abban, hogy a következő megoldással próbálkozzon:

```
int a = 3, b = 2, c = 1;
// . . .
if( a > b > c ) // Megengedett, de valószínűleg hibás...
```

Ez a kódrészlet formailag helyes, de minden valószínűség szerint rossz. A `3>2>1` művelet eredménye természetesen `false`. A „nagyobb mint" művelet, mint a C++ legtöbb művelete, balról kötő, így először a `3>2` kifejezés fog kiértékelődni. Ezek után marad a `true>1` kifejezés. Itt a `true` átalakul egésszé, és a program az `1>1` összehasonlítást végzi el, aminek az eredménye `false`.

A dolog tehát látszólag rendben is van, csak annyi a gond, hogy a programozó minden valószínűség szerint a következő összetett logikai kifejezésre gondolt, amikor ezt írta: `a>b && b>c`. Ha mégsem így lenne, és tényleg az volt a szándéka, amit a kifejezés ténylegesen jelent, akkor is jobban nézne ki az `a>b?1>c:0>c` vagy a `(c-(a>b))<0` kifejezés. Mindkettő elég furán néz ki ahhoz, hogy a programot karbantartó programozó kétszer is elgondolkodjon rajta, mit is lát tulajdonképpen. Ilyen esetekben természetesen illik őt kisegítenünk egy kellően világos megjegyzéssel (lásd az 1. hibát).

16. hiba: A for utasítás okozta zavar

A C++ lehetőséget ad arra, hogy bizonyos helyeken korlátozott hatókörű változókat vezessünk be. Lehetőségünk van például arra, hogy egy `if` szerkezet logikai feltételében hozzunk létre egy új változót. Ez a teljes `if` blokkban – beleértve annak hamis ágát is – használható lesz, de sehol másutt:

```
if( char *theName = lookup( name ) ) {
   // Csinálunk valamit a name változóval...
}
// A theName változó itt már nem látható
```

Korábban ezt a változót valahol e blokkon kívül kellett bevezetnünk, és a változó sajnos akkor is megmaradt, ha már semmi szükség nem volt rá. Így aztán tévedésből fel is használhattuk valamilyen hibás művelet során:

```
char *theName = lookup( name );
if( theName ) {
   // Csinálunk valamit a name változóval...
}
// A theName itt még mindig hozzáférhető...
```

Általában jó ötlet a változók érvényességét kizárólag arra a programrészre korlátozni, ahol ténylegesen szükség van rájuk. Ellenkező esetben ugyanis a karbantartást végzők – valamilyen, számomra felfoghatatlan okból kifolyólag – előszeretettel fogják azokat eredeti rendeltetésüktől teljesen eltérő célra használni. Ennek pedig enyhén szólva negatív hatása van a karbantartásra. (Ezzel kapcsolatban lásd még a 48. hibát.)

```
theName = new char[ ISBN_LEN ]; // Tárterület az ISBN számára
```

Hasonló a helyzet a `for` ciklusokkal. A ciklusváltozót bevezethetjük a ciklusvezérlő kód első részében:

```
for( int i = 0; i < bufSize; ++i ) {
   if( !buffer[i] )
      break;
}
if( i == bufSize ) // Korábban helyes volt, most már nem az.
➥ Az i nem hozzáférhető.
```

Ez a lehetőség már évek óta rendelkezésre áll a C++ nyelvben, az így bevezetett változó hatóköre azonban idővel változott. Kezdetben a hatókör a bevezetéstől a `for` ciklust tartalmazó blokk végéig terjedt (de lásd még a 21. hibát), újabban azonban csak magára a ciklusra terjed ki. Bár a legtöbb C++ programozó úgy gondolja, hogy ez a váltás a legtöbb tekintetben hasznos volt – nagyobb az összhang a nyelv többi tulajdonságával, könnyebb a ciklusokat finomhangolni –, az senkinek sem hiányzott igazán, hogy átírja az összes korábbi forráskódot az új szabványnak megfelelően.

Néha ráadásul ez a váltás még több is, mint púp az ember hátán. Nézzük például a következő kódot, és képzeljük el, hogy a szabványban bekövetkezett „csendes váltásnak" miféle hatása lesz a működésére:

```
int i = 0;
void f() {
    for( int i = 0; i <bufSize; i++ ) {
        if( !buffer[i] )
        break;
    }
    if( i == bufSize ) // Ez már egy másik i!
```

Szerencsére az ilyen jellegű hiba ritka, és a jobb fordítóprogramok figyelmeztetnek a lehetőségére. Vegyük komolyan az ilyen figyelmeztető üzeneteket (véletlenül se kapcsoljuk ki őket fordítás közben), és próbáljuk elkerülni azt, hogy a belső deklarációk elfedjék a külső változókat. Ami pedig a globális változók használatának mellőzését illeti, nos erről már esett szó a 3. hiba kapcsán.

Furcsa módon a változók hatókörében bekövetkezett változás legnegatívabb hatása éppen az az utasításforma, amire a C++ programozók többsége átállt a `for` ciklusokkal kapcsolatban:

```
int i;
for( i = 0; i < bufSize; ++i ) {
    if( isprint( buffer[i] ) )
        massage( buffer[i] );
    // . . .
    if( some_condition )
        continue;
    // . . .
}
```

Ez nem C++, hanem egyszerű C kód. Megvan az az előnye, hogy érzéketlen a hatókörben bekövetkezett változással szemben, de fontoljuk meg, mit veszítünk vele. Először is, a változó érvényben marad azután is, hogy a `for` ciklus befejeződött. Másodszor, a példában szereplő `i` változónak nem adtunk kezdőértéket. Ezeknek persze semmi jelentősége nincs a program írója számára, de annál több a karbantartást végzőknek. Ők ugyanis esetleg megpróbálhatják használni az `i` változó tartalmát a ciklus előtt, vagy művelhetnek vele valami egészen furcsa dolgot annak befejeződése után is, annak ellenére, hogy az alkotó elképzelése szerint az `i` érvényessége csak a ciklusra korlátozódik, utána csendben el kellene tűnnie.

Megint más probléma, hogy mindez egyes programozókat arra késztethet, hogy egyáltalán ne használjanak `for` ciklust:

```
int i = 0;
while(  i < bufSize ) {
    if( isprint( buffer[i] ) )
        massage( buffer[i] );
    // . . .
    if( some_condition )
        continue; // Hoppá!
    // . . .
    ++i;
}
```

Itt az a gond, hogy a `while` ciklus nem teljesen azonos a `for` ciklussal. Ha a ciklusmagban például van egy `continue` utasítás, annak a két esetben más és más lesz a hatása, az ilyen jellegű hibát pedig igen nehéz felderíteni. A fenti példában a program végtelen ciklusba fog kerülni, ami egyértelműen jelzi, hogy baj van. Nincs azonban mindig ekkora szerencsénk.

Ha akkora szerencsénk van, hogy kizárólag olyan rendszerekre fejlesztünk, amelyek a `for` ciklusok új értelmezését támogatják, a legjobb, ha mielőbb hozzászokunk ennek használatához.

Sajnos azonban az esetek többségében az általunk írt kódot e tekintetben eltérő viselkedésű rendszereken is használnunk kell majd. Ilyenkor persze logikusnak tűnhet, hogy olyan `for` ciklusokat használjunk, amelyek mindkét esetben jól működnek:

```
int i;
for( i = 0; i < bufSize; ++i ) {
    if( isprint( buffer[i] ) )
        massage( buffer[i] );
    // . . .
}
```

Én mégis azt javaslom, hogy következetesen ragaszkodjunk ilyenkor is az új értelmezéshez. Az ezzel kapcsolatos gondokat például úgy kerülhetjük meg, hogy a `for` ciklusokat néhány kapcsos zárójel segítségével önálló programblokkokba helyezzük:

```
{for( int i = 0; i < bufSize; ++i ) {
    if( isprint( buffer[i] ) )
        massage( buffer[i] );
    // . . .
}}
```

Ez elég feltűnő ahhoz, hogy később észrevegyük és eltávolítsuk a zárójeleket, ha már nincs rájuk szükség. E módszernek az az előnye is megvan, hogy nem kell a későbbi karbantartás során a `for` ciklusok nagyobb arányú átszerkesztésével foglalkoznunk.

17. hiba: A legtágabb értelem elvével kapcsolatos gondok

Mit tesz a Kedves Olvasó, ha ilyen programsorral találkozik?

```
++++p->*mp
```

Avagy volt már dolga valaha a „Testőr művelettel"?

```
template <typename T>
class R {
   // . . .
   friend ostream &operator <<< // Ez egy testőr művelet lenne?
       T >( ostream &, const R & );
};

//
```

Töprengett valaha azon, hogy egy ilyen programsornak egyáltalán van-e értelme?

```
a+++++b
```

Ha igen, Isten hozta a legtágabb értelem világában. A C++ kód lefordításának kezdetén a fordítóprogram egy része (a „nyelvi" vagy „lexikális elemző") a forráskódot értelmezhető egységekre (alapelem, token) tördeli, és ellenőrzi, hogy azok formailag helyesek-e. Ha a program a `->*` karaktersorral találkozik, több lehetősége is van annak értelmezésére. Gondolhatja, hogy három különböző értelmes jelet látott (`-`, `>` és `*`). Aztán hiheti azt, hogy amit lát, az két dolog (`->` és `*`), de dönthet úgy is, hogy a karaktersor a `->*` műveleti jel. A három esetnek három teljesen különböző jelentése van, a helyzet tehát nem egyértelmű. Ennek elkerülésére vezették be azt a szabályt, hogy a nyelvi elemzőnek mindig a lehető legtöbb karaktert kell egyszerre értelmeznie. Ez a legtágabb értelem elve.

Ennek megfelelően az `a++++++b` kifejezés nem megengedett, ugyanis értelmezése és tördelése során az `a ++ ++ + b` karaktersorozat áll elő, a nyelv szabályai pedig tiltják, hogy egy jobbértéket, mint amilyen az `a++`, újból növeljünk egy `++` művelettel. Ha netán az a célunk, hogy a utólagosan megnövelt értékéhez `b` előzetesen megnövelt értékét adjuk, legalább egy szóközt ki kell tennünk a megfelelő helyre: `a+++ ++b`. Persze ha tekintettel akarunk lenni programunk későbbi olvasóira, illene kitenni még egy szóközt: `a++ + ++b`. No nem azért, mert kötelező, inkább azért, hogy a kód olvashatóbb legyen. És persze azért sem kapunk senkitől bírálatot, ha a fenti szerkezetet néhány – amúgy szintén teljesen felesleges – zárójellel még áttekinthetőbbé tesszük: `(a++) + (++b)`.

A legtágabb értelem elve sokkal több gondot old meg, mint amennyit okoz, de van két olyan helyzet, amikor kifejezetten zavaró. Az első, amikor paraméterekkel rendelkező sablonoknak megfelelő objektumpéldányokat állítunk elő, de úgy, hogy a paraméterek maguk is sablonok alapján elállított objektumok. Ha a szabványos könyvtárnál maradunk, előfordulhat például, hogy olyan listát (`list`) szeretnénk alkotni, amelynek elemei vektorok (`vector`) vagy karakterláncok (`string`):

```
list<vector<string>> lovos; // Hiba!
```

Sajnos a két, egymást követő csúcsos zárójelet a fordító eltolási műveletnek fogja nézni, és nyelvtani hibát jelez. A megoldás persze egyszerű, hiszen csak egy szóközt kell kitennünk a megfelelő helyre:

```
list< vector<string> > lovos;
```

Egy másik zavaró helyzet az, amikor alapértelmezett beállítást adunk meg mutató típusú formális paraméterre:

```
void process( const char *= 0 ); // Hiba!
```

Itt a `*=` értékadó műveleti jel okozza a gondot, amit formális paraméter bevezetésére nem használhatunk. Az eredmény ismét nyelvtani hiba. Ez amúgy a „bűn és bűnhődés" filozófiai kategóriájába esik, mivel ha a programozó nevet adott volna a formális paraméternek, az egész ügy fel sem merül. Ha egy paraméternek neve van, az egyrészt önmagában is jelzi, miről van az adott környezetben szó, másrészt semmi gondunk nem támadhat a legtágabb értelem elvével:

```
void process( const char *processId = 0 );
```

18. hiba: A deklaráció-módosítók kreatív elrendezése

Ha csupán a nyelvi szabványt tekintjük, az lehetőséget ad a deklarációkban használt módosítók (declaration-specifier) tetszőleges elrendezésére:

```
int const extern size = 1024; // Megengedett, de furcsa
```

Hacsak nincs rá igen jó okunk, célszerű a de facto szabvánnyá vált sorrendhez ragaszkodni. Első a sorban az összeszerkesztést (linkage) vezérlő módosító, ezt követi a típusminősítő (type qualifier), majd utolsó a sorban maga a típus:

```
extern const int size = 1024; // Rendben
```

Nézzük például, vajon mi lehet itt a `ptr` típusa:

```
int const *ptr = &size;
```

Igen, ez egy mutató egy állandóként bevezetett egészre. Nem is hinnénk, hány programozó értelmezné ez a sort úgy, hogy `ptr` egy egészet címző állandó mutató. Ez az értelmezés viszont valójában a következő sornak felelne meg:

```
int * const ptr2 = &size; // Hiba!
```

Ez természetesen két különböző mutatótípus, hiszen az egyikkel címezhetünk egész állandót, a másikkal pedig nem. És ez még csak az egyik baj. A másik az, hogy a köznyelvben mindkét bemutatott mutatótípust egyszerűen „állandó mutatónak" (const pointer) titulálják, ami az esetek egy részében fedi ugyan a valóságot, a gyakorlott C++ programozók számára azonban, akik szó szerint értik, amit mondanak nekik, bizonyos helyzetekben kifejezetten félrevezető lehet.

Kétségtelen, hogy maga a szabványos könyvtár is tartalmaz egy `const_iterator` típust, ami azonban – megbocsáthatatlan módon – maga nem állandó, csak állandó értékekre vonatkozik. (Csak azért, mert a C++ szabványt kidolgozó bizottságnak volt egy rossz napja, még nem kell utánozni őket.) Saját jól felfogott érdekünkben élesen különböztessük meg az „állandót címző mutatót" (pointer to const), a nem feltétlenül állandót címző „állandó mutatótól". (Ezzel kapcsolatban olvassuk el a 31. hibát is.)

Mivel a C++ nyelvben a deklarációkban szereplő módosítók sorrendje tulajdonképpen lényegtelen, egy állandót címző mutatót kétféleképpen is bevezethetünk:

```
const int *pci1;
int const *pci2;
```

Egyes C++ szakértők a második formát javasolják, mivel szerintük ez könnyebben áttekinthető a bonyolultabb helyzetekben:

```
int const * const *pp1;
```

Ha a `const` minősítőt mindig a sor végén szerepeltetjük, az lehetővé teszi, hogy a deklarációt értelmezéskor visszafelé, vagyis jobbról balra olvassuk: `pp1` egy mutató (`*`) egy olyan állandó mutatóra (`const *`), ami egy egész állandót (`const int`) címez. A szokásos elrendezés ugyanezt az olvasási sorrendet nem teszi lehetővé:

```
const int * const *pp2; // Ugyanolyan típus, mint a pp1
```

Ez a forma sem sokkal bonyolultabb, mint az előző, és szinte biztos, hogy a kódok karbantartásával gyakran foglalkozó C++ programozók ezzel is boldogulni fognak. Ráadásul a mutatókat címző mutatók, és az ehhez hasonlóan bonyolult szerkezetek viszonylag ritkák, különösen a programozási felületekben, amelyekkel a gyakorlatlanabb programozók „érintkeznek". A bonyolultabb hivatkozások általában megmaradnak a szolgáltatásokat nyújtó könyvtárak forráskódjának mélyén. Az állandókat címző mutatók már sokkal gyakoribbak, így a fenti szakértői javaslat ellenére én mégis úgy gondolom, hogy a félreértések elkerülése végett a nyelvtan tekintetében célszerűbb ragaszkodni a szokásokhoz:

```
const int *pci1; // Helyes: állandót címző mutató
```

19. hiba: Függvény vagy objektum?

Egy objektum alapértelmezett beállítását nem célszerű üres paraméterlistával megadni, mivel ezt a legtöbben függvénynek fogják nézni:

```
String s( "Semantics, not Syntax!" ); // Közvetlen beállítás
String t;  // Alapértelmezett beállítás
String x(); // Függvény bevezetése
```

Ez a C++ nyelv egy sajátságos kétértelműsége. Itt valószínűleg az történt, hogy a szabványosítással foglalkozók feldobtak egy érmét, és úgy döntöttek, hogy a fenti

példában x jelentse egy függvény deklarációját. Érdemes persze megjegyezni, hogy a new művelettel létrehozott objektumok esetében ez a félreértelmezhetőség már nem áll fenn:

```
String *sp1 = new String();  // Itt nincs kétértelműség...
String *sp2 = new String;  // Ugyanaz, mint a this
```

Ugyanakkor itt is inkább a második forma használata a célszerűbb, mivel a programozók többsége ezt használja, és jobban megfelel az objektumok bevezetésének.

20. hiba: Típusminősítők vándorlása

A C++-ban nem léteznek const vagy volatile típusú tömbök, így ha tömbök bevezetésekor alkalmazzuk ezeket a típusminősítőket, automatikusan a típus megfelelő helyére vándorolnak:

```
typedef int A[12];
extern const A ca; // 12 állandó egész tömbje
typedef int *AP[12][12];
volatile AP vm; // Egészeket címző volatile mutatók kétdimenziós
➥ tömbje
volatile int *vm2[12][12]; // Volatile egészeket címző mutatók
➥ kétdimenziós tömbje
```

Ez rendben is van így, hiszen egy tömb végül is nem egyéb, mint literális mutató a tömbelemekre. Ennek a mutatónak pedig nincs olyan saját tárterülete, ami állandó (const) vagy eltűnő (volatile) lehetne, így hát természetes, hogy a fordító a minősítőt automatikusan a tömbelemekre alkalmazza. Nem árt azonban észben tartani, hogy ez a szolgáltatás számos fordítóprogramban hibásan működik a bonyolultabb helyzetekben. Gyakori például, hogy a fenti példában szereplő vm-nek végül (tévesen) ugyanaz lesz a típusa, mint a vm2-nek.

Kicsit zavarosabb a helyzet a függvények bevezetésével kapcsolatban. Ezekre korábban ugyanaz a vándorlási szabály érvényesült, mint a tömbökre:

```
typedef int FUN( char * );
typedef const FUN PF; // Korábban ez egy állandó egészet
➥ visszaadó függvény volt. Most már nem megengedett.
```

A ma érvényes szabvány azonban kimondja, hogy típusminősítőt függvény bevezetésével kapcsolatban csak globális szintű típusmeghatározásban (`typedef`) lehet használni, valamint hogy ilyen típusmeghatározással csak nem statikus tagfüggvényeket vezethetünk be:

```
typedef int MF() const;
MF nonmemfunc; // Hiba!
class C {
   MF memfunc; // Rendben.
};
```

Valószínűleg az a legjobb, ha egyszerűen elkerüljük az ilyen helyzeteket. Ezt a formát számos mai fordítóprogram hibásan értelmezi, sőt a programozók is nehezen boldogulnak vele.

21. hiba: Önbeállítás

Vajon mi lehet a belső `var` változó értéke ebben a programban?

```
int var = 12;
{
   double var = var;
   // . . .
```

Nem meghatározott (undefined). A C++ szabályai szerint a változók neve még azelőtt lép érvénybe, mielőtt a beállításukra vonatkozó információt a fordító értelmezni próbálná. Persze nem sok olyan programozó van, aki ezt a feladatot a fent látható módon próbálná megoldani, de azért lehetséges, hogy ha vakon másolni próbáljuk a saját kódunkat, egyszer csak ilyesfajta bajba kerülünk:

```
int copy = 12; // Mélyen elásott változó
// . . .
int y = (3*x+2*copy+5)/z; // Ezt kivágjuk...
// . . .
void f() {
   // Szükségünk van y eredeti értékének másolatára
   int copy = (3*x+2*copy+5)/z; // … és beillesztjük ide
   // . . .
```

Ugyanezt a hibát persze nem csak esztelen másolással, hanem az előfeldolgozó „segítségével" is előállíthatjuk, hiszen az sem tesz egyebet, csak esztelenül másol (lásd még a 26. hibát):

```
int copy = 12;
#define Expr ((3*x+2*copy+5)/z)
// . . .
void g() {
    int copy = Expr; // Mintha már láttam volna valahol...
```

Hasonló hiba állhat elő, ha az általunk használt elnevezési szabályokból nem derül ki világosan, hogy mi jelöl típust, és mi mást:

```
struct buf {
    char a, b, c, d;
};
// . . .
void aFunc() {
    char *buf = new char[ sizeof( buf ) ];
    // . . .
```

A helyi `buf` nevű változó (valószínűleg) egy négy bájt hosszúságú átmeneti tár akart lenni, ami elég nagy egy karaktervátozót címző mutató (`char *`) tárolásához. Ez ilyen hiba aztán sokáig felderítetlen maradhat, különösen ha a `struct buf` mérete pont akkora, mint egy mutatóé. Persze ha olyan neveket használtunk volna, amelyekből azonnal látszik, hogy mi típus és mi nem az, az egész probléma fel sem merül (lásd még a 12. hibát):

```
struct Buf {
    char a, b, c, d;
};
// . . .
void aFunc() {
    char *buf = new char[ sizeof( Buf ) ]; // Rendben.
    // . . .
```

Nos, akkor rendben is volnánk. Most már pontosan ismerjük azt a szabványos módszert, amivel az ilyen hibákat elkerülhetjük:

```
int var = 12;
{
    double var = var;
    // . . .
```

De mi a helyzet a téma alábbi változatával?

```
const int val = 12;
{
   enum { val = val };
   // . . .
```

Vajon mi itt a `val` nevű felsoroló típusú objektum értéke? Ez is meghatározatlan, mint az előbb? Nem nyert! Most `12` az érték. Ennek pedig az az oka, hogy a felsoroló típusok (enumerator) nevének érvénybe lépte – eltérően a változókétól – a beállítás (értékadás) utánra esik. Ez pedig azt jelenti, hogy a fenti példában az egyenlőségjel utáni `val` a külső `val` változót jelenti, még nem magát a felsoroló típust. Ez pedig rögtön elvezet bennünket egy még ennél is szövevényesebb helyzet tárgyalásához:

```
const int val = val;
{
   enum { val = val };
   // . . .
```

Hála a mindenhatónak, ez a deklaráció – már ami a felsoroló típust illeti – érvénytelen. A beállításhoz használni kívánt érték ugyanis a látszat ellenére nem állandó egész kifejezés, mivel a fordítóprogram fordítási időben képtelen meghatározni a külső `val` változó értékét.

22. hiba: Statikus és külső típusok

Ez egy igen érdekes témakör, ugyanis a címben említett dolgok egyszerűen nem léteznek. Mindazonáltal a gyakorlott C++ programozók gyakran kergetik őrületbe kevésbé gyakorlott társaikat azzal, hogy a típusok bevezetésekor szerkesztésmódosítókat adnak meg (lásd még a 11. hibát):

```
static class Repository {
   // . . .
} repository; // statikus
Repository backUp;  // nem statikus
```

Mármost a típusoknak tényleg van valami közük az összeszerkesztéshez, szerkesztésmódosítókat alkalmazni azonban csak objektumokra és függvényekre lehet, típusokra nem. Jobb tehát, ha tisztábban fogalmazzuk meg, amit akarunk:

```
class Repository {
  // . . .
};
static Repository repository;
static Repository backUp;
```

Szintén érdemes megjegyezni, hogy egy névtelen névtér használata ilyen esetekben célszerűbb, mint a `static` szerkesztésmódosító alkalmazása:

```
namespace {
Repository repository;
Repository backUp;
}
```

A `repository` és `backUp` neveknek ezzel megvan a szükséges mértékű külső kapcsolata, így sokkal többféleképpen használhatók, mint a statikus nevek. (Használhatók például sablonok alapján történő objektumalkotás során.) Ugyanakkor akárcsak a statikus változatok, nem hozzáférhetők az adott fordítási egységen kívül.

23. hiba: A műveletfüggvények keresésével kapcsolatos gondok

A túlterhelt műveletek (operátorok) lényegüket tekintve olyan szabványos tagfüggvények vagy külső függvények, amelyek közbevetett (infix) jelölésmóddal hívhatók. Egyszerűen a C++ egy kiegészítő szolgáltatásának tekintendők:

```
class String {
 public:
   String &operator =( const String & );
   friend String operator +( const String &, const String & );
   String operator -();
   operator const char *() const;
   // . . .
};
```

```
String a, b, c;
// . . .
a = b;
a.operator =( b ); // Ugyanaz.
a + b;
operator +( a, b ); // Ugyanaz.
a = -b;
a.operator =( b.operator -() ); // Ugyanaz.
const char *cp = a;
cp = a.operator const char *(); // Ugyanaz.
```

Azt hiszem, a közbevetett jelölés ellen egyes esetekben vádat emelhetünk a kód tisztaságának védelmében. Ezt a jelölésmódot általában túlterhelt műveletekkel kapcsolatban használjuk. Végül is éppen ez az, amiért túlterheltük őket.

Ugyanakkor vannak olyan helyzetek, amikor a hagyományos jelölésmód sokkal áttekinthetőbb kódot eredményez. Ilyen például a függvényhívás, vagy az, amikor az alaposztály értékadó műveletét hívjuk meg egy származtatott osztály hasonló műveletén keresztül:

```
class A {
 protected:
   A &operator =( const A & );
   // . . .
};
class B : public A {
 public:
   B &operator =( const B & );
   // . . .
};

B &B::operator =( const B &b ) {
   if( &b != this ) {
       A::operator =( b ); // Világosabb, mint ha azt írtuk
➥ volna, hogy
                           // (*static_cast<A*const>(this))=b
       // Értékadás a helyi tagoknak…
   }
   return *this;
}
```

Akkor is a hagyományos függvényhívási formát használjuk, amikor a közbevetett jelölés ugyan teljesen szabályos, „visszafejtése" azonban akár percekig is eltarthat:

```
value_type *Iter::operator ->() const
   { return &operator *(); } // Inkább, mint: &*(*this)
```

Vannak aztán olyan „kétes esetek" is, amikor sem a közbevetett, sem a hagyományos jelölés nem világos teljesen, bár mindkettő helyes:

```
bool operator !=( const Iter &that ) const
    { return !(*this == that); } // Vagy: !operator ==(that)
```

Érdemes azt is észben tartani, hogy a közbevetett jelölésnél más keresési sorrend érvényes, mint a függvényhívásnál, ami megint csak váratlan dolgokat eredményezhet:

```
class X {
 public:
    X &operator %( const X & ) const;
    void f();
    // . . .
};
X &operator %( const X &, int );
void X::f() {
    X &anX = *this;
    anX % 12; // Rendben; nem tag
    operator %( anX, 12 ); // Hiba!
```

A függvényhívásos formánál a fordító a hagyományos sorrendet követi a hívott függvény azonosítása során. Ennek megfelelően az `X::f` függvény esetében először az `X` osztályban fog keresni egy `operator %` nevű függvényt. Ha talál ilyet, azonnal meg is elégszik vele, és nem fog a program többi részében ugyanilyen nevű függvény után kutatni.

Sajnos a fenti példában három paramétert próbálunk átadni egy kéttényezős műveletnek. Mivel az `operator %` függvénynek van egy közvetve megadott `this` paramétere is, a fordítónak az lesz az érzése, hogy a bináris `%` műveletet akarjuk háromtényezőssé tenni. A helyes hívásban vagy fel kellene tüntetni, hogy nem tagfüggvényről van szó (`::operator %( anX, 12 )`), vagy a megfelelő számú paramétert kellene átadni a tagfüggvénynek (`::operator %( anX )`).

A közbevetett jelölés ugyanakkor arra készteti a fordítót, hogy először a bal oldali tényező által jelzett tartományban (jelen esetben az `X` osztályban, hiszen az `anX` ilyen típusú) keresse az `operator %` nevű tagfüggvényt, majd ha ott nem találja, akkor a tágabb környezetben tegye ugyanezt, persze már egy nem tagfüggvényt keresve. Ha tehát azt írjuk hogy `anX % 12`, akkor a fordító két jelöltet talál, és helyesen a nem tagfüggvényre fog „szavazni".

24. hiba: A -> művelettel kapcsolatos ravaszságok

A beépített -> művelet kéttényezős: a bal oldal egy mutató, a jobb pedig a megfelelő osztálytag neve. A túlterhelt operator -> azonban már egytényezős!

➡ 24. hiba/ptr.h

```
class Ptr {
 public:
   Ptr( T *init );
   T *operator ->();
   // . . .
 private:
   T *tp_;
};
```

A túlterhelt -> művelet meghívásával olyan visszatérési értékhez jutunk, amit aztán a -> művelettel együtt használva férhetünk hozzá a kérdéses taghoz:

➡ 24. hiba/ptr.cpp

```
Ptr p( new T );
p->f(); // p.operator ->()->f()!
```

Ezt akár úgy is értelmezhetjük, hogy a -> jelet az operator -> túlterhelés nem használja el, hanem meghagyja további felhasználásra annak eredeti értelmében. Ez egyben azt is jelenti, hogy általában némi nyelvi kiegészítés még szükséges ahhoz, hogy a túlterhelt -> művelettel egy „okos mutatótípust" alkothassunk:

➡ 24. hiba/ptr.cpp

```
T *Ptr::operator ->() {
    if( today() == TUESDAY )
        abort();
    else
        return tp_;
}
```

Amint az előbb említettem, a túlterhelt operator -> olyasvalamit ad vissza, amit a taghoz való hozzáférés során használhatunk. Ennek a valaminek nem kell előre megadott mutatónak lennie. Lehet olyan osztályelem is, amely önmaga terheli túl az operator -> függvényt:

➡ 24. hiba/ptr.h

```
class AugPtr {
 public:
   AugPtr( T *init ) : p_( init ) {}
   Ptr &operator ->();
   // . . .
```

```
 private:
   Ptr p_;
};
```

➡ 24. hiba/ptr.cpp

```
Ptr &AugPtr::operator ->() {
   if( today() == FRIDAY )
     cout << '\a' << flush;
   return p_;
}
```

Ez lehetővé teszi, hogy az „okos mutatókból" láncot alkossunk:

➡ 24. hiba/ptr.cpp

```
AugPtr ap( new T );
ap->f(); // ap.operator ->().operator ->()->f()!
```

Jegyezzük meg, hogy ilyen esetekben az `operator ->` hívások sorrendjét mindig statikusan meghatározza azoknak az objektumoknak a típusa, amelyek az `operator ->` meghatározását tartalmazzák, maga a függvényhívási sorozat pedig mindig egy olyan hívással végződik, amely egy osztályt címző beépített mutatót ad vissza. Ha például a `->` műveletet az `AugPtr` objektumra alkalmazzuk, a hívási sorrend a következő lesz. Először az `AugPtr::operator ->` fut le, ez meghívja a `Ptr::operator ->` függvényt, az viszont a beépített `->` műveletet alkalmazza egy `T *` mutatóra. (A `->` művelet egy sokkal valószerűbb alkalmazását a 83. hibánál fogjuk bemutatni.)

3

Az előfeldolgozó

Az előfeldolgozás a C++ kód fordításának talán legveszélyesebb lépése. Az előfeldolgozó ugyanis „szavakkal" (alapelemekkel, token) dolgozik, de egyáltalán nem törődik a C++ nyelv nyelvtanával (szintaxis) és a jelentésréteggel (szemantika). Ez gyakorlatilag azt jelenti, hogy az előfeldolgozó olyan eszköz, amely nincs tudatában saját erejének, s így sok probléma forrása lehet.

Ha tehát egyetlen mondatban akarnám összefoglalni az ebben a fejezetben megfogalmazott tanácsokat, azt mondanám, hogy az előfeldolgozót csak olyasmi megvalósítására használjuk, amihez csak „erő" kell, de nem igényli a C++ logikájának ismeretét. Minden olyan feladatot, amihez intelligencia is szükségeltetik, valósítsunk meg mi magunk.

25. hiba: A #define literálok

A C++ programozók nem használják a `#define` utasítást literálok megadására, mivel ez számos hibát okozhat és a hordozhatóságot is csökkenti. Vegyünk egy szokványos, C stílusú `#define` szerkezetet:

```
#define MAX 1<<16
```

Az előfeldolgozónak átadott szimbólumokkal az a gond, hogy azok az előfeldolgozás során még azelőtt kifejtődnek, hogy a C++ fordító megvizsgálhatná őket. Az előfeldolgozó azonban semmit sem tud a C++ hatókörökkel vagy típusokkal kapcsolatos szabályairól.

```
void f( int );
void f( long );
// . . .
f( MAX ); // Melyik f?
```

Ebben a példában a `MAX` előfeldozó szimbólum nem egyéb, mint az `1<<16` utasításnak megfelelő egész szám. A fordító csak erre az egyetlen információra támaszkodhat, amikor feloldja a túlterhelt függvényre való hivatkozást. Az előfeldolgozás fázisában azonban a kérdéses értéknek még nincs típusa. Lehet, hogy `int`, de az is lehet, hogy `long` típus lesz belőle, attól függően, milyen rendszerre fordítjuk le a programot. Ez pedig azt jelenti, hogy különböző rendszereken különböző függvények hívódnak majd meg a program adott pontján.

A `#define` utasítás a hatókörökkel sem foglalkozik. A legtöbb C++ szolgáltatásnak határozott, névterekhez kötődő hatóköre van. Ebből pedig a nyelvnek számos előnyös tulajdonsága következik. Hogy csak a leglényegesebbet és legnyilvánvalóbbat említsük, minimálisra csökken annak az esélye, hogy a különböző szolgáltatások zavarják egymás hatását. A `#define` azonban mit sem tud a névterekről:

```
namespace Influential {
#  define MAX 1<<16
// . . .
}
namespace Facility {
const int max = 512;
// . . .
}
// . . .
int a[MAX]; // Hoppá!
```

Ebben a példában a programozó elfelejtette a megfelelő helyre berakni a `max` nevű változót, ráadásul el is gépelte `MAX`-nak. A kód persze lefordult, mert az előfeldolgozó behelyettesítette az `1<<16` értéket a `MAX` szimbólum helyére. Programozónk aztán csodálkozhat, miért használ olyan sok memóriát a programja.

A megoldás mindezekre persze egyszerű. A `#define` helyett típussal rendelkező állandókat kell használnunk:

```
const int max = 1<<9;
```

A `max` típusa immár valamennyi rendszeren azonos lesz, ráadásul ugyanazok a hatókörökkel kapcsolatos megkötések érvényesek rá, mint minden más C++ objektumra. Azt is érdemes megemlíteni, hogy az így megadott `max` állandó használata valószínűleg ugyanolyan hatékony lesz, mint a `#define` utasításé, mivel a fordító szabadon végezhet vele kapcsolatban finomhangolást, és ha úgy látja jónak, érték szerint behelyettesítheti azokon a pontokon, ahol jobbértékként használjuk. Ugyanakkor, mivel a `max` ebben a formában megadva balérték (módosítani ugyan nem lehet, de attól még balérték; lásd a 6. hibát), rendelkezik címmel, s így megcímezhetjük mutatón keresztül is. Ugyanezt egy `#define` segítségével megadott értékkel nem tehetjük meg:

```
const int *pmax = &Facility::max;
const int *pMAX = &MAX; // Hiba!
```

A `#define` literálokkal kapcsolatos másik probléma behelyettesítésük módjával kapcsolatos. Mint említettük, a behelyettesítést az előfeldolgozó vakon és süketen, vagyis tisztán lexikális módon végzi el, egyáltalán nem foglalkozik a kialakuló szerkezet jelentésével. A mi korábbi példánkban ezzel kapcsolatban nem volt ugyan semmi baj, de nézzük csak ezt az ártatlannak tűnő megoldást:

```
int b[MAX*2];
```

Ez már valami! Mivel sajnálatos módon elfelejtettük kitenni a megfelelő helyre a zárójeleket, itt bizony egy igen nagy tömböt fogunk lefoglalni, hiszen a behelyettesítés után a fordító a következő sort kapja meg:

```
int b[ 1<<16*2 ];
```

Megengedem persze, hogy ez a hiba tulajdonképpen a `#define` szerkezet nem megfelelő megfogalmazásából ered (kitehettük volna ott is a zárójeleket), viszont az egész dolog fel sem merül, ha `#define` helyett típusos állandókat használunk.

Ugyanez a gond merül fel az osztályokkal kapcsolatos hatóköröknél. Itt azt akarjuk elérni, hogy egy érték egy osztály hatókörén belül mindenütt hozzáférhető legyen, de sehol másutt. Erre a szabványos C++ megoldás egy felsoroló típus alkalmazása:

```
class Name {
    // . . .
    void capitalize();
    enum { nameLen = 32 };
    char name_[nameLen];
};
```

A felsoroló típusú `nameLen` változó nem foglal tárterületet, határozott típussal rendelkezik, és kívánalmainknak megfelelően az osztály teljes érvényességi területén – beleértve a tagfüggvényeket is – hozzáférhető:

```
void Name::capitalize() {
    for( int i = 0; i < nameLen; ++i )
        if( name_[i] )
            name_[i] = toupper( name_[i] );
        else
            break;
}
```

Szintén szabályos, de egyelőre nem általánosan támogatott az a megoldás, amikor egy statikus egész állandó adattagot vezetünk be és látunk el kezdőértékkel magán az osztálymeghatározáson belül (ezzel kapcsolatban lásd még az 59. hibát):

```
class Name {
    // . . .
    static const int nameLen_ = 32;
};
// . . .
const int Name::nameLen_; // Itt nincs semmi, ami beállítaná az
➥ értéket!
```

Ugyanakkor az is lehetséges, hogy az ilyen módon megadott egész állandót a fordító nem tudja finomhangolni, így összességében talán mégis inkább a hagyományos módszert érdemesebb használni.

26. hiba: A #define álfüggvények

A C nyelvben a `#define` utasítást gyakran használják álfüggvények (pszeudofüggvények) megadására olyan helyeken, ahol a tényleges függvényhívás elkerülése fontosabb szempont, mint a biztonság:

```
#define repeated(b, m) (b & m & (b & m)-1)
```

Természetesen ezekre a megoldásokra is érvényesek mindazok a megállapítások, amelyeket az előfeldolgozóval kapcsolatban korábban említettünk. Ami azt illeti, a fenti meghatározás is hibás:

```
typedef unsigned short Bits;
enum { bit01 = 1<<0, bit02 = 1<<1, bit03 = 1<<2, // . . .
Bits a = 0;
const Bits mask = bit02 | bit03 | bit06;
// . . .
if( repeated( a+bit02, mask ) ) // Hoppá!
```

Azt az általános hibát követtük el, hogy nem használtunk megfelelő zárójelezést. Az igazi megoldás nem bíz semmit a véletlenre:

```
#define repeated(b, m) ((b) & (m) & ((b) & (m))-1)
```

Persze a mellékhatásokkal szemben még ez sem véd. Ha valaki nem szokványos módon akarja használni az általunk megadott álfüggvényt, az eredmény egyszerre lesz téves és kétséges:

```
if( repeated( a+=bit02, mask ) ) // Kétszer is hoppá!
   // . . .
```

Ha ebben a példában a `repeated` egy valódi függvény lenne, akkor a használatával kapcsolatos mellékhatás is csak egyszer jelentkezne, közvetlenül a függvényhívás előtt. Így azonban az álfüggvényekre érvényes eltérő szabályok miatt a mellékhatás kétszer jelentkezik, ráadásul az sem tisztázott, milyen sorrendben (lásd a 14. hibát). Az álfüggvények használata azért különösen veszélyes, mert „szemre" azonosnak tűnnek egy ugyanazt a szerepet betöltő valódi függvénnyel. Így még a leggyakorlottabb C++ programozókat is félrevezethetik, mert a kód kellős közepén semmi sem fog arra utalni, hogy nem valódi függvényről van szó, és nem ugyanazokat a szabályokat kell szem előtt tartani a használata során.

A C++-ban a fordító által helyben kifejtett függvények (inline függények) használata csaknem mindig megfelelőbb, mint az álfüggvényeké, mivel meghívásuk során ugyanazt az értelmezést kell követnünk, mint a közönséges függvények esetében. Jelentésük és működésük is teljesen azonos a hagyományos függvényekével:

```
inline Bits repeated( Bits b, Bits m )
   { return b & m & (b & m)-1; }
```

Az álfüggvényként használt makrók alkalmazásakor a hatókörök tekintetében ugyanazokkal a gondokkal kell számolnunk, mint az ilyen módon megadott állandóknál (lásd a 25. hibát):

➡ 26. hiba/execbump.cpp

```
int kount = 0;
#define execBump( func ) (func(), ++kount)
// . . .
void aFunc() {
   extern void g();
   int kount;
   while( kount++ < 10 )
       execBump( g ); // A helyi kount változó tartalmát
➥ növeljük!
}
```

Az `execBump` függvény felhasználójának valószínűleg fogalma sem lesz róla, hogy a nevezett álfüggvény egy `kount` nevű változóra hivatkozik, az adott környezetben

azonban ez nem az ilyen nevű globális változó tartalmát módosítja, hanem a helyiét. A helyes megoldás persze egy valódi függvény használata lenne:

➡ 26. hiba/execbump.cpp

```
int kount = 0;
inline void execBump( void (*func)() )
   { func(); ++kount; }
```

Ha a fenti módon egy helyben kifejtett függvényt alkalmazunk a cél érdekében, akkor ez fordítási időben örökre összekapcsolódik a globális `kount` változóval, és nem hagy semmi kétséget afelől, mikor melyik változó tartalma módosul. Hiába bukkan fel egy szűkebb környezetben egy ugyanilyen nevű helyi változó, a függvény nem ezzel fog dolgozni. (No persze itt most egy globális változót használunk, amelynek káros mellékhatásairól a 3. hibánál már esett szó.)

Még tökéletesebb megoldás, ha függvényobjektumot használunk, amely még egyértelműbben elhatárolja környezetétől a számlálóként használt változót:

➡ 26. hiba/execbump.cpp

```
class ExecBump { // Monostate tervezési módszer. Lásd a 69. hibát.
 public:
   void operator ()( void (*func)() )
        { func(); ++count_; }
   int get_count() const
         { return count_; }
 private:
   static int count_;
};
// . . .
int ExecBump::count_ = 0;
// . . .
void aFunc() {
   extern void g();
   ExecBump exec;
   int count = 0;
   while( count++ < 10 )
        exec( g );
}
```

Azok a helyzetek, amikor az álfüggvények használata tényleg hasznosabb, meglehetősen ritkák, és általában a `__LINE__`, `__FILE__`, `__DATE__` vagy a `__TIME__` előfeldolgozó szimbólumokkal kapcsolatosak:

```
#define myAssert( e ) ((!(e))?void(std::cerr << "Failed: " \
   << #e << " line " << __LINE__ << std::endl): void())
```

Ezzel kapcsolatban olvassuk el még a 28. hibát.

27. hiba: Az #if túlzott használata

Hibakeresésre használt #if

Hogyan szúrunk be hibakereséshez (debugging) használatos kódot a programjainkba? Mindenki tudja a választ – az előfeldolgozó segítségével:

```
void buggy() {
#ifndef NDEBUG
   // Hibakereső kód...
#endif
   // Valódi kód...
#ifndef NDEBUG
   // További hibakereső kód...
#endif
}
```

Ami azt illeti, mindenki téved. A veterán programozók hosszú és unalmas történeteket tudnak arról mesélni, hogy programjuk hibakereséssel kiegészített változata tökéletesen működött, az `NDEBUG` szimbólum megadása után viszont rejtélyes módon működésképtelenné vált.

Amúgy ebben a dologban tulajdonképpen nincs is semmi rejtélyes. A két esetben ugyanis két teljesen különböző programról van szó, amelyek történetesen ugyanabból a forrásfájlból állnak elő. Még ugyanazt a forráskódot is legalább kétszer kell lefordítanunk, ha biztosak akarunk lenni benne, hogy nyelvileg helyes. A helyes szemlélet az, ha eleve nem is gondolkodunk hibakereső és „éles" változatban, hanem egyetlen programot írunk:

```
void buggy() {
   if( debug ) {
      // Hibakereső kód...
   }
   // Valódi kód...
   if( debug ) {
      // További hibakereső kód...
   }
}
```

No de mi lesz a hibakereső kóddal, ami benne marad a program kész változatában? Nem pocsékolunk ezzel helyet? Nem fogja a sok lehetséges elágazás lassítani a program futását? Nem, ha a hibakereső kód nincs benne a kész változatban! A mai fordítóprogramok elképesztően okosak, amikor a használaton kívüli kód eltávolításáról

van szó. Ami azt illeti, sokkal okosabbak, mint mi magunk a szánalmas `#ifndef` feltételeinkkel. Csak annyit kell tennünk, hogy program megírásakor egyértelművé tesszük a dolgokat:

```
const bool debug = false;
```

Az itt látható `debug` kifejezést hívja a szabvány egészre kiértékelhető állandó kifejezésnek. Minden C++ fordítónak képesnek kell lennie kiértékelni az ilyen állandó kifejezéseket, hiszen másként nem tudná fordítási időben meghatározni a tömbök határát, a többszörös elágazások (`switch`) címkéit vagy a bitmezők hosszát. Minden kicsit is értelmes fordító képes ezen kívül eltávolítani az elérhetetlen kódrészleteket:

```
if( false ) {
   // Elérhetetlen kód…
}
```

Bizony, bizony, az a fordító, amit az elmúlt öt évben annyit szidtunk a főnöknek, még az is képes erre. Persze az, hogy egy kódrészletet a fordító végül is eltávolít, nem jelenti azt, hogy nem ellenőrzi végig, és nem értelmezi azt.

Mivel az állandó kifejezések (constant-expression) a szabvány részét képezik, a fordító még akkor is sikeresen megtalálja a szükségtelen kódrészeket, ha azokat összetett feltételrendszer határolja:

```
if( debug && debuglvl > 5 && debugopts&debugmask ) {
   // Esetleg elérhetetlen kód…
}
```

A fenti példa még mindig egyszerűnek számít. A fordítók az ennél lényegesen összetettebb esetekkel is vidáman elboldogulnak. Próbáljuk például beszúrni a feltételes kifejezésbe a kedvenc helyben kifejtett függvényemet:

```
typedef unsigned short Bits;
inline Bits repeated( Bits b, Bits m )
   { return b & m & (b & m)-1; }
// . . .
if( debug && repeated( debugopts, debugmask ) ) {
   // Esetleg elérhetetlen kód…
   error( "One option only" );
}
```

Persze ha már függvényhívás is van a feltételek között – és ebből a szempontból most mindegy, hogy helyben kifejtett vagy közönséges függvényről van szó –, a fordító már nem feltétlenül lesz képes a kifejezést fordítási időben kiértékelni, így a kódot sem távolítja el. Ha tehát kifejezetten a kód szűrése a célunk, akkor a fenti megoldás nem hordozható. Egyes programozók, akik korábban sok évet töltöttek hagyományos C programok írásával, esetleg a következő megoldást javasolnák:

```
#define repeated(b, m) ((b) & (m) & ((b) & (m))-1)
```

Nos, nem. Ezt ne csináljuk. (Lásd a 26. hibát.)

Megjegyzendő, hogy egyes esetekben azért mégis célszerű lehet feltételesen fordítandó szakaszokat szerepeltetni a forráskódban. Ez teremt például lehetőséget arra, hogy bizonyos állandók értékét a fordítónak átadott parancssorban határozzuk meg:

```
const bool debug =
#ifndef NDEBUG
   false
#else
   true
#endif
;
```

Ugyanakkor még ennek a kicsi, feltételesen fordítandó kódnak a jelenléte sem elengedhetetlenül szükséges. Használhatunk például egy szerkesztési fájlt (makefile) vagy más hasonló szolgáltatást arra, hogy a hibakereső és a végleges változat között váltogassunk.

A hordozhatóság érdekében használt #if

Mármost valaki azzal hozakodhat elő, hogy az ő programja bizony rendszerfüggetlen, de ezt csak úgy tudja elérni, ha a különböző rendszerek eltérő jellemzőihez fordítási feltételeken keresztül alkalmazkodik. Szavainak alátámasztására pedig a következő kódot tárja elénk:

```
void operation() {
   // Hordozható kód...
#ifdef PLATFORM_A
   // Csinálunk valamit...
   a(); b(); c();
#endif
#ifdef PLATFORM_B
```

```
    // Itt ugyanazt csináljuk...
    d(); e();
#endif
}
```

Ez a kód nem rendszerfüggetlen. Csak több rendszeren is jó. De ha ezeken bármilyen változás következik be, az nemcsak a forráskód újrafordítását vonja maga után, hanem át is kell írnunk az egészet az új helyzetnek megfelelően. A fenti megoldással természetesen a lehető legnagyobb átjárhatóságot biztosítottuk az érintett rendszerek között, ami figyelemreméltó előrelépés, de nem feltétlenül praktikus.

No de mindez eltörpül az `operation` függvény konkrét megvalósításával kapcsolatos lehetséges problémák mellett. A függvény mindig egyfajta elvonatkoztatás („absztrakció"): az `operation` is egy olyan műveletsor elvont ábrázolása, ami a különböző rendszereken esetleg eltérő alacsonyszintű műveleteknek felel meg, vagyis eltérő módon valósítható meg rajtuk. Ha magasszintű programnyelvet használunk, nagyon gyakran ugyanazt a forráskódot fordítjuk le a különböző operációs rendszereken ugyazon elvont ábrázolás megvalósítása végett. Az például, hogy `a=b+c` – ahol `a`, `b`, és `c` egész számok –, egészen más módon valósítható meg a különböző processzorokon, de maga a művelet elég egyszerű, általános, és közel áll a processzorok működéséhez, így minden rendszeren ugyanaz a forráskód feleltethető meg neki. A helyzet persze nem mindig ennyire egyszerű, különösen, ha függvényünk az operációs rendszer részét képező könyvtárakra, vagy rendszerszolgáltatásokra támaszkodik.

A fenti példában az `operation` függvény megvalósítása azt sugallja, hogy minden rendszeren ugyanannak kell történni, ami talán még igaz is. A karbantartás során azonban kapunk majd egy csomó hibajelentést, amelyek egy része az egyes rendszerek belső működésével függ majd össze. Az `operation` eltérő jelentéseinek megfelelően eltérő módon kell majd javítani a függvény különböző rendszerekre vonatkozó részeit, aminek rövid időn belül az lesz az eredménye, hogy tulajdonképpen nem egy függvényt tartunk karban, hanem rendszerenként egyet-egyet. Az `operation` megvalósítása akkor tökéletes, ha a rendszerfüggő részeket a felhasználó elől elrejti, és a szolgáltatásokhoz mindig ugyanazon a rendszerfüggetlen programozási felületen keresztül férhetünk hozzá:

```
void operation() {
    // Hordozható kód…
    doSomething(); // Hordozható felület...
}
```

Ami az elvont forma tényleges megvalósítását illeti, sokkal valószínűbb, hogy a karbantartás során magának a függvény által képviselt műveletnek a jelentése változatlan marad, csak lesznek olyan részműveletek, amelyeket máshogy kell programozni az egyes rendszereken. A fenti példában a `doSomething()` függvény bevezetése a megvalósítás rendszerfüggetlen része. Ennek a függvénynek aztán többféle, az egyes rendszerekhez alkalmazkodó megvalósítása létezhet. (Ha helyben kifejtett függvényről van szó, a probléma rendszerfüggő fejállományokkal kezelhető.)
A rendszerek közti választást a szerkesztési fájlon (makefile) belül oldhatjuk meg, vagyis sehol egy `#if`. Ezzel a módszerrel ráadásul akkor sem kell a teljes forráskódot átszerkesztenünk, ha új rendszert akarunk felvenni a támogatottak közé, vagy el akarunk egyet távolítani.

És mi a helyzet az osztályokkal?

Akár a függvények, az osztályok is általánosítások. Az elvont, általános ábrázolás megvalósítása aztán fordítási vagy futási időben a különböző rendszereknek megfelelően módosulhat. Akár a függvények esetében, az osztályok rendszerfüggő megvalósítása során is életveszélyes az `#if` használata:

```
class Doer {
#  if ONSERVER
   ServerData x;
#  else
   ClientData x;
#  endif
   void doit();
   //  . . .
};
void Doer::doit() {
#  if ONSERVER
   // A kiszolgáló ügyeit intézzük...
#  else
   // Az ügyfél ügyeit intézzük...
#  endif
}
```

Szigorúan fogalmazva ez a kód tulajdonképpen nem helytelen, hacsak a `Doer` osztályban az `ONSERVER` szimbólum beállított vagy be nem állított volta el nem tér a különböző fordítási egységekben. Néha azonban kifejezetten örömteli volna, ha az efféle megvalósítást a fordító nem engedné meg. Általános ugyanis az a jelenség, hogy a `Doer` osztály különböző változatait a programozók különböző fordítási

egységekben helyezik el, a programot gond nélkül lefordítják és összeszerkesztik, majd csodálkoznak az eredményen. A hiba ilyenkor nyilván csak futásidőben jelentkezik, felderíteni pedig kifejezetten kalandos és rettenetesen nehéz.

Szerencsére az efféle hibák megjelenése ma már nem annyira általános, mint egykoron. Az osztályok efféle változatosságának kifejezésére a leghatékonyabb módszer a többalakúság (polimorfizmus) használata:

```
class Doer { // Rendszerfüggetlen
 public:
   virtual ~Doer();
   virtual void doit() = 0;
};
class ServerDoer : public Doer { // Rendszerfüggő
   void doit();
   ServerData x;
};
class ClientDoer : public Doer { // Rendszerfüggő
   void doit();
   ClientData x;
};
```

Az elmélet és a gyakorlat

Az előzőekben viszonylag egyszerű példákon keresztül igyekeztem bemutatni azokat a hibákat, amelyeket akkor követhetünk el, amikor egyetlen forráskóddal akarunk elkészíteni több különböző programot. Megmutattam néhány olyan programozási eljárást is, amelyek segíthetnek e hibák elkerülésében. Bár ezek alkalmazása ezen a ponton egészen egyszerűnek tűnhet, ez elsősorban a felhozott példák egyszerűségével magyarázható.

A valóság azonban sajnos gyakran sokkal bonyolultabb, a problémák sokkal összetettebbek. Teljesen általános például, hogy a forráskód fordítását nem egyetlen kapcsoló (például az NDEBUG) határozza meg, hanem több, amelyek egyenként is többféle értéket vehetnek fel, sőt bizonyos helyeken a kombinációik fordulnak elő. Amint azt korábban is említettem, e szimbólumok minden értéke és kombinációja gyakorlatilag más és más programnak felel meg, eltérő jelentéssel és eltérő belső viselkedéssel. Még ha műszakilag lehetséges is a változatok szimbólumok segítségével történő együtt tartása és egyértelmű megkülönböztetése, akkor is szinte elkerülhetetlen, hogy a karbantartás során ne módosuljon legalább egy rendszeren a kód működése.

A kód teljes átírása csak akkor válik szükségessé, ha a program jelentését már hosszas elemzéssel is nehéz visszafejteni, vagy ha több száz, különböző kapcsolókkal végrehajtott fordításra van szükség ahhoz, hogy legalább a kód formai helyességét ellenőrizhessük. Összességében tehát célszerűbb elkerülni az `#if` utasítás programváltozatok összetartására való használatát.

28. hiba: A feltételes töréspontok mellékhatásai

Míg a `#define` felhasználási módjait többségükben ki nem állhatom, a `<cassert>`-ben meghatározott szabványos `assert` utasítás alkalmazását elviselem. Sőt, még bátorítok is mindenkit a használatára, feltéve persze, hogy helyesen teszi. A gondot ugyanis rendszerint éppen ez, vagyis a használat módja jelenti.

Bár többféle változata létezik, az `assert` makró megvalósítása általában valahogy így fest:

➡ 28. hiba/myassert.h

```
#ifndef NDEBUG
#define assert(e) ((e) \
   ? ((void)0) \
   :__assert_failed(#e,__FILE__,__LINE__) )
#else
#define assert(e) ((void)0)
#endif
```

Ha az `NDEBUG` szimbólumot beállítottuk, akkor nem alkalmazunk hibakeresést, így az `assert` üres utasításnak felel meg. Ellenkező esetben hibakeresést végzünk, s ilyenkor az `assert` makrót az előfeldolgozó kibontja egy feltételes kifejezéssé, amely jelen megvalósításában egy feltételt vizsgál meg. Ha a feltétel hamis, hibaüzenetet küldünk a kimenetre, a program pedig leáll.

Az `assert` használata általában sokkal megfelelőbb, mint ha hosszadalmas megjegyzésekben ecseteljük a végrehajtandó művelet előfeltételeit, hatásait és lehetséges ellenőrzési módjait. Az `assert` futásidejű ellenőrzést végez, így nem kerülhető meg olyan egyszerűen, mint egy megjegyzés (lásd az 1. hibát). Ha az `assert` segítségével derül fény egy hiba meglétére, azt általában ki is javítja a programozó, hiszen a program azonnali leállítása kellően jól látható jele a hibajavítás szükségességének:

➡ 28. hiba/myassert.cpp

```
template <class Cont>
void doit( Cont &c, int index ) {
```

```
    assert( index >= 0 && index < c.size() ); // #1
    assert( process( c[index] ) ); // #2
    // . . .
}
```

Persze nem mindegy, hogyan használjuk ezt az eszközt. A fenti példában a `#2` jelű sor például egy nyilvánvalóan téves felhasználási mód, hiszen az `assert`-en belülről hívunk meg egy függvényt, amelynek önmagában is nem várt mellékhatásai lehetnek. A program viselkedése ennek megfelelően gyökeresen eltérhet, attól függően, hogy az `NDEBUG`-ot beállítottuk vagy sem. Ez az a jellegzetes eset, amikor bekapcsolt hibakereséssel minden tökéletesen működik, kikapcsolva pedig nem. Érzékeljük a hibát, bekapcsoljuk a hibakeresést, mire a hiba eltűnik. Kikapcsoljuk, és a hiba visszatér. Öngól!

Az `#1` jelű sor még ennél is rosszabb. A `Cont` osztály `size` tagfüggvényéről mindenki úgy gondolná, hogy állandó tagfüggvény, tehát mellékhatás kizárva. Hát nem! Attól eltekintve, hogy a `size`-nak van egy szokványos jelentése, semmi a világon nem biztosít állandó jelentést. Még ha a `size` tagfüggvény állandó is, akkor sincs semmi garancia arra, hogy nem lesznek mellékhatásai. Még ha a `c` logikai állapotát nem is változtatja meg a függvényhívás, a fizikai állapota akkor is megváltozhat (lásd a 82. hibát). Végezetül soha nem felejtsük el, hogy az `assert` hibák felderítésére való. Még ha bizton állíthatjuk, hogy a `size` függvény végrehajtásának nem lehet semmiféle hatása a program későbbi mozzanataira, akkor is lehetséges, hogy `size` megvalósításába hiba csúszott. Márpedig mi nem elrejteni akarjuk az `assert` segítségével az ilyen hibákat, hanem felderíteni. Ez pedig csak úgy lehetséges, ha az `assert` használatával kapcsolatban mindennemű mellékhatás kizárható:

```
template <class Cont>
void doit( Cont &c, int index ) {
    const int size = c.size();
    assert( index >= 0 && index < size ); // Helyes.
    // . . .
```

Az `assert` makró természetesen nem általános gyógymód minden helyzetre és hibára. Ugyanakkor megvan a létjogosultsága a helytelen viselkedés felderítése terén, a megjegyzések és a kivételkezelés mellett. Az egyetlen gond vele az, hogy ez is egy álfüggvény, így természetesen érvényes rá mindaz, amit korábban ezzel a nyelvi szolgáltatással kapcsolatban említettünk (lásd a 26. hibát). Ugyanakkor annyi előnye azért van, hogy szabványos, beépített álfüggvény, így lehetséges hibái előre ismertnek tekinthetők. Ha ügyesen használjuk, nagyon hasznos segédeszköz.

4

Típusátalakítás

A C++ nyelv típusrendszere kellően összetett ahhoz, hogy a nyelvnek óriási kifejezőerőt biztosítson. Ezt az eleve összetett alaprendszert tovább bonyolítja az a lehetőség, hogy a felhasználó saját típusátalakításokat adhat meg, amelyeket a fordítás során a fordítóprogram közvetve felhasználhat. Az a lehetőség, hogy a C++-ban a felhasználó saját elvont adattípusokat hozhat létre, egyben azt a felelősséget is ráruházza, hogy gondoskodjon ennek a rendszernek a hatékonyságáról, biztonságáról, és belső ellentmondásoktól mentes voltáról. Mivel a C++ típusrendszere javarészt statikus, lehetőségünk van arra, hogy ezt az erősen összetett rendszert hatékony tervezéssel „megszelídítsük".

Sajnos azonban bizonyos hibás kódolási módszerek a legkörültekintőbben megtervezett rendszert is tönkretehetik. Ebben a fejezetben azokról a hibákról esik szó, amelyek alááshatják a nyelv statikus típusaival kapcsolatos biztonságot. Ugyancsak bemutatjuk a C++ néhány olyan, igen gyakran félreértett területét, amelyek szintén a típusokkal kapcsolatos biztonságot veszélyeztethetik.

29. hiba: Típusátalakítás a void * segítségével

Azt még a C programozók is tudják, hogy a `void *` a típusátalakítás (típuskényszerítés, cast) másod-unokatestvére, és amennyire lehet, el kell kerülni. Ami a típusátalakítást illeti, egy mutatónak `void *` típussá alakításával elvész az összes, a mutató által címzett dolog típusára vonatkozó információ. Erre az egyetlen megoldás általában az, hogy „emlékeznünk" kell a mutató eredeti típusára, és vissza kell állítanunk azt, amikor ténylegesen használni akarjuk. Ha ilyenkor a megfelelő típust állítjuk vissza (sikerült jól visszaemlékezni), minden jól működik. (Ehhez persze az is kell, hogy legyen tervezési stratégiánk, és az működjön is.)

```
void *vp = new int(12);
// . . .
int *ip = static_cast<int *>(vp); // Működni fog
```

Sajnos a `void *` még eme igen egyszerű használata mellett is sérülhet a hordozhatóság. Emlékezzünk vissza, hogy ha biztonságos és viszonylag hordozható típusátalakítást akarunk megvalósítani (avagy kell megvalósítanunk), a `static_cast` műveletet használjuk. A `static_cast`-ra van szükség például akkor, ha egy alaposztály mutatóját egy nyilvános származtatott osztály mutatójává akarjuk átalakítani. A `reinterpret_cast` művelet használata ezzel szemben kevésbé biztonságos, rá-

adásul rendszerfüggő. Ezt használjuk, ha például egészet akarunk mutatóvá alakítani, vagy olyan típusok között végzünk átalakítást, amelyeknek egymáshoz eredetileg semmi közük sincs:

```
char *cp = static_cast<char *>(ip); // Hiba!
char *cp = reinterpret_cast<char *>(ip); // Működik.
```

Ha a `reinterpret_cast` művelet felbukkan a kódban, az számunkra, és persze a programunkat később karbantartó programozó számára is egyértelmű jele annak, hogy nemcsak típusátalakításba bonyolódtunk, hanem feltehetőleg nem hordozható módon oldottuk meg. A `void *` közbeiktatásával persze megkerülhetjük ezt a problémát, és elterelhetjük róla az olvasó figyelmét:

```
char *cp = static_cast<char *>(vp); // Egy egész címét helyezzük
➥ el egy char * típusban.
```

Sajnos ez még rosszabb. Vegyünk például egy olyan programozási felületet, ami lehetővé teszi, hogy egy grafikus elem (`widget`) címét tároljuk, és később lekérdezzük:

```
typedef void *Widget;
void setWidget( Widget );
Widget getWidget();
```

Ez a felület persze eleve feltételezi, hogy a felhasználó emlékezni fog a kérdéses elem típusára, és azt a lekérdezés után vissza is állítja:

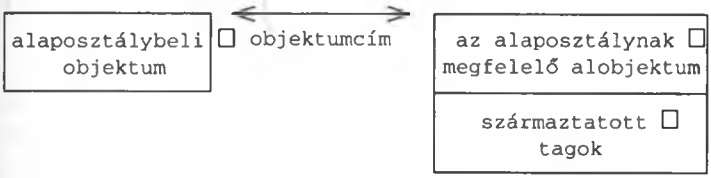
```
// Valahol egy fejállományban...
class Button {
   // . . .
};
class MyButton : public Button {
   // . . .
};
// Máshol...
MyButton *mb = new MyButton;
setWidget( mb );

// Egészen máshol...
Button *b = static_cast<Button *>(getWidget());
```

Bár a `Widget` visszanyerése során elveszítünk némi információt annak típusával kapcsolatban, ez a kód azért általában működni fog. A `Widget` típusa eredetileg

`MyButton` volt, de amikor visszakapjuk, `Button` típusúvá alakítjuk. Ez pedig azt jelenti, hogy e kód gyakori helyes működése inkább csak az objektumok memóriában való elhelyezési módjának köszönhető.

Általános ugyanis az a megoldás, hogy egy származtatott osztályobjektum saját alaposztályát a `0` eltolási címnek megfelelő tárterületen tartalmazza, mintha ez lenne az első adattagja. Az összes többi adattag ezután következik a tárban, ahogy azt a 4.1. ábra mutatja. Ebből pedig az következik, hogy a származtatott osztály címe gyakran megegyezik saját alaposztályának címével. (Ugyanakkor érdemes megjegyezni, hogy maga a szabvány csak akkor biztosítja e módszer helyes működését, ha a típus, amire a `void *` által meghatározott címet alakítjuk, pontosan megegyezik azzal, amit a `void *` beállításakor használtunk. A 70. hibánál bemutatunk egy olyan helyzetet, amikor ez a kód még egyszeres öröklődés esetén is hibásan működik.)



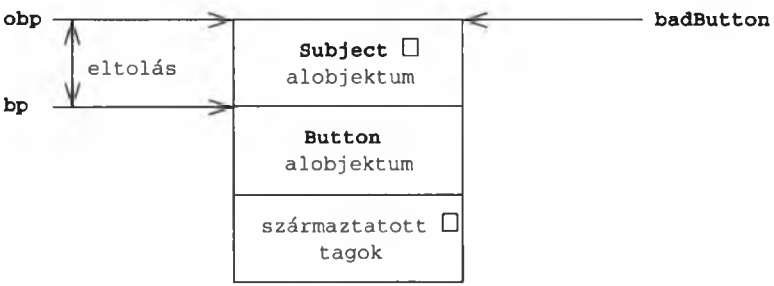
4.1. ábra

Származtatott osztály várható tárfoglalási szerkezete egyszeres öröklődés esetén.

Bár ez a kód, mint említettük, működőképes, erősen növeli a hibák lehetőségét, hiszen egy későbbi, a karbantartás során végzett változtatás is járhat olyan mellékhatással, ami működésképtelenné teheti. A többszörös öröklődés egyszerű és helyes használata például azonnal hibát okoz:

```
// Valahol egy fejállományban...
class Subject {
   // . . .
};
class ObservedButton : public Subject, public Button {
   // . . .
};
// Máshol...
ObservedButton *ob = new ObservedButton;
setWidget( ob );
// . . .
Button *badButton = static_cast<Button *>(getWidget()); //
Katasztrófa!
```

A problémát a többszörösen örökölt osztályok összetettebb memóriaképe okozza. Egy `ObservedButton` osztálynak ugyanis már két alapobjektum része van, és ezek közül értelemszerűen csak az egyik címe egyezhet meg magának az objektumnak a címével. Általában a teljes öröklési lánc alapját képező alaposztály (jelen esetben a `Subject`) kezdődik a `0` eltolási címen, ezt követi a tárban a második alaposztály (jelen esetben a `Button`), majd csak ezután következnek a származtatott osztály új tagjai (lásd a 4.2. ábrát). Többszörös öröklés esetén tehát egy objektummal kapcsolatban több érvényes cím is szóba jöhet.



4.2. ábra

Objektum várható memóriaképe többszörös öröklés esetén. Az ObservedButton objektum mind a Subject, mind a Button alaposztály számára tartalmaz elkülönített helyet. A badButton mutatón keresztüli hivatkozással azonban elveszítjük a szükséges típusinformációt, így a mutató valójában nem a Button osztályt fogja címezni.

Közönséges esetekben ez nem jelent gondot, mert a fordítóprogram elég okos ahhoz, hogy figyeljen a többféle lehetséges eltolási értékre, és ha kell, fordítási időben módosítsa azokat:

```
Button *bp = new ObservedButton;
ObservedButton *obp = static_cast<ObservedButton *>(bp);
```

Ebben a kódban a `bp` helyesen az `ObservedButton` objektum `Button` osztályt tartalmazó részére mutat, nem az objektum elejére. Ha a típusát `Button`-ról `ObservedButton`-ra módosítjuk, a fordító egyszerűen az `ObservedButton` kezdőcímére fogja változtatni a mutatóban tárolt címet, mert tudja, melyik típushoz melyik cím tartozik. A fordító minden esetben ismeri az összes alaposztály eltolási címét a származtatott osztályokon belül, és ezzel az információval egészen addig rendelkezik, amíg tudatában lehet az alap- és származtatott osztályok típusának.

És itt jön a probléma. Ha a `setWidget` függvényt használjuk, eldobunk minden használható típusinformációt. Így amikor a `getWidget` segítségével visszakérjük a tárolt címet, és az objektumot `Button` típusúvá alakítjuk, a fordító már nem képes beállítani a megfelelő eltolást, mert nem ismeri, így mutatónk valójában a `Subject` osztályt fogja címezni, nem a `Button`-t.

Összefoglalva tehát, a típus nélküli mutatóknak megvan a maga felhasználási területe, de ha lehet, ne használjuk őket túl gyakran. Az pedig egyértelműen rossz ötlet, ha `void *` mutatót használunk egy olyan felület megvalósítása során, amely később felhasználójától azt igényli, hogy visszaállítsa a megfelelő – a fordító számára immár elveszett – típusinformációt.

30. hiba: Kivágás

Kivágásról (slicing) akkor beszélünk, ha egy származtatott osztálynak megfelelő objektumot egy, az alaposztálynak megfelelő objektumra másolunk. Ilyenkor a származtatás során hozzáadott tagok és viselkedésformák elvesznek (a fordító kivágja ezeket), ami rendszerint a program hibás, vagy legalábbis igen meglepő működését okozza:

```
class Employee {
 public:
   virtual ~Employee();
   virtual void pay() const;
   // . . .
 protected:
   void setType( int type )
        { myType_ = type; }
 private:
   int myType_; // Rossz ötlet! Lásd a 69. hibát.
};
class Salaried : public Employee {
   // . . .
};
Employee employee;
Salaried salaried;
employee = salaried; // Kivágás!
```

A fenti példában a `salaried` objektum tartalmának hozzárendelése az `empolyee`-hez formailag teljesen helyénvaló, hiszen a `Salaried` típus egyben `Employee` is. Ugyanakkor a hatás nagy valószínűséggel nem az lesz, mint amit vártunk. Az értékadás után az `employee` viselkedése mind virtuális, mind nem virtuális függvényeit tekintve `Employee`-szerű lesz. A `Salaried` osztályban bevezetett tagok és tulajdonságok ugyanis nem másolódnak át, hanem elvesznek.

És ami még borzasztóbb, az `employee` állapota a `salaried` objektumpéldány `Salaried` részének másolata lesz. Hogy mi ebben olyan rettenetes? Nos csupán az, hogy egy származtatott `Salaried` típusú objektum az `Employee` alaposztálytól örökölt változókban azt tárol, amit csak akar. Megteheti, hogy a saját adatait teszi belé-jük, amelyeknek az `Employee` típussal kapcsolatban semmi értelme (ezzel kapcsolatban lásd még a 91. hibát).

Példaképpen tételezzük fel, hogy az `Employee` osztályból származtatott osztályok valamiféle típusazonosításra szolgáló információt tárolnak az `Employee` alaposztálynak megfelelő területükön. (Jegyezzük meg rögtön, hogy ez egyáltalán nem jó tervezési módszer, amint azt a 69. hibánál tárgyalni is fogjuk. Itt is csak szemléltetési célzattal használjuk.) A kivágás (másolás) után az `employee` `Employee` típusúként fog viselkedni, viszont a `Salaried` típusnak megfelelő adatokat fogja tartalmazni. A gyakorlatban az objektum tartalma és viselkedése közti összhang felborulásának általában igen súlyos következményei vannak.

A kivágás leggyakoribb felbukkanási helye, amikor egy származtatott osztálynak megfelelő objektumot érték szerint adunk át, egy alaposztály kezdeti beállítása végett formális paraméterként:

```
void fire( Employee victim );
// . . .
fire( salaried ); // Kivágás!
```

A problémát persze elkerülhetjük, ha hivatkozást (vagy mutatót) használunk az érték szerinti átadás helyett. Ebben az esetben nem jelentkezik a kivágási hatás, mivel a származtatott osztálynak megfelelő objektum ténylegesen nem másolódik át sehová, a formális paraméter pedig csupán álnév a beállításhoz használt értékre (lásd az 5. hibát):

```
void rightSize( Employee &asset );
// . . .
rightSize( salaried ); // Nincs kivágás!
```

Más kivágási problémák is jelentkezhetnek, bár felbukkanásuk kevésbé általános. Megeshet például, hogy egy származtatott osztályból átmásolunk egy alaposztálynak megfelelő objektumot egy másik, de eltérő típusú származtatott osztályba tartozó objektumba:

```
Employee *getNextEmployee(); // Az Employee osztályból
➥ származtatott osztály egy objektumának megszerzése
// . . .
Employee *ep = getNextEmployee();
*ep = salaried; // Kivágás!
```

A kivágási gondok jelentkezése egy hierarchiában általában nem elsősorban programozási, hanem alapvető tervezési hibákra utal. Ezt elkerülni legegyszerűbben úgy lehet, ha tartózkodunk a konkrét alaposztályok használatától (lásd még a 93. hibát):

```
class Employee {
 public:
   virtual ~Employee();
   virtual void pay() const = 0;
   // . . .
};

void fire( Employee ); // Szerencsére ez hibás.
void rightSize( Employee & ); // Rendben.
Employee *getNextEmployee(); // Rendben.
Employee *ep = getNextEmployee(); // Rendben.
*ep = salaried; // Szerencsére ez hibás.
Employee e2( salaried ); // Szerencsére ez hibás.
```

Elvont alaposztályba tartozó objektumokat nem adhatunk meg közvetlenül, így a legtöbb olyan helyzetre, ami a kivágás megjelenésével fenyegethet, még fordítási időben fény derül.

Zárásként érdemes megjegyezni, hogy bizonyos meglehetősen ritka esetekben a kivágást szándékosan is használhatjuk arra, hogy általa megváltoztassuk egy származtatott osztályba tartozó objektum típusát vagy viselkedését. Általában azonban ilyenkor sem használjuk a módszert adatok kivágására, a módszer értelme inkább az alaposztály adatainak egy másik származtatott osztálynak megfelelő „átértelmezése". Az ilyen módszerek néha hasznosak lehetnek ugyan, de valóban csak ritkán, attól pedig kifejezetten óvakodjunk, hogy általános célú felület részeként használjuk őket.

31. hiba: Az állandót címző mutatóvá történő átalakítás félreértelmezése

Mindenekelőtt tisztázzuk a kifejezések jelentését. Az „állandó mutató" (const pointer) olyan mutató, ami maga állandó, vagyis nem módosíthatjuk. Ebből nem következik, hogy amit a mutató címez, az maga is állandó volna. Igaz ugyan, hogy a szabványos C++ könyvtár tartalmaz egy `const_iterator` nevű elemet, ami valójában nem állandó, hanem állandók sorozatát címzi, de ennek felbukkanása csak annak köszönhető, hogy a C++ nyelvet is egy bizottság tervezte.

```
const char *pci; // Állandót címző mutató
char * const cpi = 0; // Állandó mutató
char const *pci2; // Állandót címző mutató, lásd a 18. hibát!
const char * const cpci = 0; // Állandót címző állandó mutató
char *ip; // Mutató
```

A szabvány megengedi az „állandóságot növelő" típusátalakításokat. Átmásolhatunk például egy nem állandó értéket címző mutatót egy állandót címzőbe. Ez – többek között – lehetővé teszi számunkra, hogy nem állandó karakterváltozó címét adjuk át paraméterként az `strcmp` vagy az `strlen` függvényeknek, holott ezek állandó karakterváltozót várnak. Nincs is ezzel semmi baj, hiszen megérzésünk is azt súgja, hogy azzal, hogy megengedjük egy állandót címző mutatónak, hogy egy nem állandóként bevezetett adatra mutasson, nem sértünk meg semmiféle, az adatok felhasználási módjával kapcsolatos szabályt. Az is világos, hogy a fordított eset nem megengedett, mert nagyobb szabadságot adna az adatok kezelésében, mint ami azok deklarációjában szerepel:

```
size_t strlen( const char * );
// . . .
int i = strlen( cpi ); // Rendben.
pci = ip; // Rendben.
ip = pci; // Hiba!
```

Érdemes ugyanakkor megjegyezni, hogy a nyelvi szabályok e tekintetben meglehetősen konzervatív viselkedést is előírhatnak. Általában nem okoz azonnali leállást, ha egy állandót címző mutatón keresztül egyszer csak módosítunk egy adatot, ami egyébként nem állandó, vagy állandó, de a rendszer az állandóként bevezetett adatokat nem a csak olvasható memóriaterületen tárolja. Ugyanakkor, ha valamit állan-

dóként adunk meg, azzal általában azt fejezzük ki, hogy ennek a dolognak sérthetetlen területre kell kerülnie. És bizony vannak fordítóprogramok és rendszerek, amelyek nem a tényeket, hanem a programozó feltételezett szándékát veszik alapul.

32. hiba: Állandót címző mutatót címző mutató átalakítása

Az az örvendetesen egyszerű szabály, amely egy mutatónak állandót címző mutatóvá való átalakítására érvényes, sajnos nem áll az állandót címző mutatót címző mutató átalakítására. Vegyük például azt az esetet, amikor egy karaktert címző mutatót címző mutatót akarunk állandó karakterváltozót címző mutatót címző mutatóvá alakítani. (Egyszóval egy `char **` típusú mutatót akarunk `const char **` típusúvá alakítani):

```
char **ppc;
const char **ppcc = ppc; // Hiba!
```

Ez veszélytelennek tűnik, de mint annyi más veszélytelennek látszó átalakítás, ez is számos hiba forrása lehet:

```
const T t = init;
T *pt;
const T **ppt = &pt; // Szerencsére hibás.
*ppt = &t;  // Egy T * állandó tartalmát helyezzük egy T *
➥ változóba
*pt = value; // t megsemmisítése!
```

Ezt az impozáns témát maga a nyelvi szabvány is érinti a 4.4. szakaszában Qualification Conversions (Minősítők átalakítása) cím alatt. (A `const` és `volatile` kulcsszavakat a C-ben típusminősítőnek szokás hívni, a C++ szabvány azonban cv-minősítőként említi őket. Én személy szerint hajlok a típusminősítő szó használatára.) A szabványban a következő egyszerű szabályokat találjuk a típusminősítők átalakíthatóságával kapcsolatban:

A többszintű mutatókat érintő átalakítások során az első szinttől eltekintve a következő szabályoknak megfelelő típusminősítőket vezethetünk be:

A T1 és T2 mutatótípusokat akkor nevezzük hasonlónak, ha létezik hozzájuk egy olyan T típus és egy olyan n>0 egész, amelyekre teljesülnek a következő feltételek:

T1 egy Cv1,0 mutatót címez, ami maga egy Cv1,1 mutatót címez, ami ... Cv1,n-1 típusú mutatót címez, ami maga egy Cv1,nT típusú mutatót címez

és

T2 egy Cv2,0 mutatót címez, ami maga egy Cv2,1 mutatót címez, ami ... Cv2,n-1 típusú mutatót címez, ami maga egy Cv2,nT típusú mutatót címez

ahol minden Cvi,j const, volatile, const volatile, vagy semmi.

Másképp fogalmazva két mutató hasonló, ha azonos az alaptípusuk, és azonos számú *-ot tartalmaznak. Így például a char * const ** és a const char ***const típusok hasonlók, de az int * const * és az int *** típusok nem.

Egy mutatótípus első utáni n darab típusminősítőjét, például a T1 mutatótípus Cv1,1 , Cv1,2 , ... Cv1,n minősítőit az adott típus típusminősítő aláírásának (cv-qualification signature) nevezzük. Egy T1 típusú kifejezés csak akkor alakítható T2 típusúvá, ha az alábbiak teljesülnek:

- A két típus hasonló.
- Valamennyi j>0 esetén, ha Cv1,j const, akkor Cv2,j is tartalmazza a const kulcsszót, és ugyanez áll a volatile típusminősítőre is.
- Ha Cv1,j és Cv2,j eltérő, akkor a const típusminősítő szerepel minden olyan Cv2,k szinten, melyekre 0<k<j.

Ezzel a tudással – no meg kellő mennyiségű türelemmel – felfegyverkezve már könnyen megállapíthatjuk, hogy a következő típusátalakítások megengedettek-e:

```
int * * * const cnnn = 0;
   // n==3, signature == none, none, none
int * * const * ncnn = 0;
   // n==3, signature == const, none, none
int * const * * nncn = 0;
   // signature == none, const, none
int * const * const * nccn = 0;
   // signature == const, const, none
```

```
const int * * * nnnc = 0;
   // signature == none, none, const

// Példák a szabályok alkalmazására
ncnn = cnnn; // Rendben.
nncn = cnnn; // Hiba!
nccn = cnnn; // Rendben.
ncnn = cnnn; // Rendben.
nnnc = cnnn; // Hiba!
```

Bár ezek a szabályok furcsának tűnhetnek, gyakran alkalmaznunk kell őket. Nézzük például a következő, különlegesnek egyáltalán nem számító helyzetet:

```
extern char *namesOfPeople[];
for( const char **currentName = namesOfPeople; // Hiba!
   *currentName; currentName++ ) // . . .
```

Tapasztalataim szerint erre a hibára a programozók többsége azzal reagál, hogy küld egy hibabejelentést a fordítóprogram gyártójának, megkerüli a hibát valamilyen típusátalakítással, majd a programja futás közben szépen leáll. Persze amint az lenni szokott, a fordítónak van igaza, nem a programozónak:

Nézzük előző példánk egy „kiélezettebb" változatát:

```
typedef int T;
const T t = 12345;
T *pt;
const T **ppt = (const T **)&pt; // Gonosz típusátalakítás!
*ppt = &t;  // Egy T * állandó tartalmát helyezzük egy T *
➥ változóba
*pt = 54321; // t megsemmisítése!
```

A kódban az az igazán tragikus, hogy a hiba évekig csendben lapulhat, míg aztán egy közönséges karbantartás során váratlanul előjön. A `t` értékét használhatjuk például a következő módon:

```
cout << t; // A kimenet valószínűleg 12345 lesz.
```

Mivel a fordítónak bárhol jogában áll egy állandót annak kezdeti értékével helyettesíteni, ez a művelet nagy valószínűséggel akkor is az `12345` értéket fogja kiírni, ha az állandót már régen `54321`-re változtattuk. Egy kissé más típusú felhasználás azonban leleplezi a bűnöst:

```
const T *pct = &t;
```

```
// . . .
cout << t; // A kimenet 12345
cout << *pct; // A kimenet 54321
```

Általában célszerűbb hivatkozások vagy a szabványos könyvtár használatával elkerülni a mutatókat címző mutatók okozta gondokat. A C nyelvben például általános, hogy egy mutató értékének módosításakor a mutató címét, vagyis egy mutatót címző mutatót adunk át a megfelelő függvénynek:

➡ 32. hiba/gettoken.cpp

```
// A get_token a következő olyan karakterláncot címző mutatót
➥ adja vissza,
// amelyet a ws-ben megadott karakterek határolnak.
// A paraméterként megadott mutató értékét úgy állítja be, hogy
// az a visszaadott elem utánra mutasson.
char *get_token( char **s, char *ws = " \t\n" ) {
    char *p;
    do
        for( p = ws; *p && **s != *p; p++ );
    while( *p ? *(*s)++ : 0 );
    char *ret = *s;
    do
        for( p = ws; *p && **s != *p; p++ );
    while( *p ? 0 : **s ? (*s)++ : 0 );
    if( **s ) {
        **s = '\0';
        ++*s;
    }
    return ret;
}

extern char *getInputBuffer();
char *tokens = getInputBuffer();
// . . .
while( *tokens )
    cout << get_token( &tokens ) << endl;
```

A C++-ban ezzel szemben általában nem állandóra vonatkozó hivatkozásként adjuk át a mutatónak megfelelő paramétert. Ez tisztábbá teszi a mutatót kezelő függvény megvalósítását, és kevésbé nehézkessé teszi annak kezelését:

➡ 32. hiba/gettoken.cpp

```
char *get_token( char *&s, char *ws = " \t\n" ) {
    char *p;
    do
        for( p = ws; *p && *s != *p; p++ );
    while( *p ? *s++ : 0 );
    char *ret = s;
```

```
    do
        for( p = ws; *p && *s != *p; p++ );
    while( *p ? 0 : *s ? s++ : 0 );
    if( *s ) *s++ = '\0';
    return ret;
}
// . . .
while( *tokens )
    cout << get_token( tokens ) << endl;
```

Persze ami az eredeti problémánkat illeti, azt a szabványos könyvtár használatával még egyszerűbben megoldhatjuk:

```
extern vector<string> namesOfPeople;
```

33. hiba: Mutató átalakítása alapmutatóvá

Hasonló problémával szembesülünk a származtatott osztályok mutatóit címző mutatók esetében:

```
D1 d1;
D1 *d1p = &d1; // Rendben!
B **ppb1 = &d1p; // Szerencsére hibás
D2 *d2p;
B **ppb2 = &d2p; // Szerencsére hibás
*ppb2 = *ppb1; // d2p most már D1-re mutat
```

Ismerős? Ahogy az állandóságra vonatkozó tulajdonságok sem őrződnek meg, ha újabb hivatkozási szintet vezetünk be, ugyanez a helyzet az osztályelemeket címző mutatókkal is. Míg a származtatott osztályok tagjait címző mutatók alapja nyilvános (`public`), az ilyen mutatókat címző mutatóké már nem. Akárcsak az állandók bevezetésével kapcsolatos példáknál, a helyzet elsőre most is veszélytelennek tűnik. Számos helyzetben azonban egy felület óvatlan megtervezése, majd annak téves használata együtt rejtélyes hibához vezet:

```
void doBs( B *bs[], B *pb ) {
   for( int i = 0; bs[i]; ++i )
         if( somecondition( bs[i], pb ) )
               bs[i] = pb; // Hoppá!
}
// . . .
extern D1 *array[];
```

```
D2 *aD2 = getMeAD2();
doBs( (B **)array, aD2 ); // Egy másik halált megvető
➥ típusátalakítás...
```

A fejlesztő itt ismét a fordítóra fog gyanakodni, pedig megint téved, a hibát pedig valószínűleg megint megkerüli egy típusátalakítással. Persze ebben a helyzetben a felület tervezőjének is lesz mit megmagyaráznia. Ha kicsit körültekintőbben járt volna el, olyan tároló objektumot használhatott volna, amit a felhasználó nem tud egy ilyen egyszerű módszerrel megkerülni.

34. hiba: A többdimenziós tömböket címző mutatók körüli gondok

A C és a C++ tömbtípusainak megvalósítása enyhén szólva minimalista. A tömb itt nem egyéb, mint a tömb első elemét címző mutatóliterál:

```
int a[5];
int * const pa = a;
int * const *ppa = &pa;
const int alen = sizeof(a)/sizeof(a[0]); // alen == 5
```

Gyakorlati szempontból egy állandó mutató és egy tömbnév között elenyésző a különbség: a tömb nevével meghívva a `sizeof` a tömb és nem a kérdéses mutató méretét adja vissza, maga a tömbnév pedig – bár bizonyos értelemben mutató – nem foglal tárhelyet, és így címe sincs. Egyszerűen fogalmazva tehát a tömbnek van címe, és ezt maga a tömbnév jelzi, a tömbnévnek magának ugyanakkor nincs címe:

```
int *ip = a; // „a" egy tömb első elemét címző mutató
int (*ap)[5] = &a; // „&a" egy tömb és nem az „a" változó címe
int (*ap2)[sizeof(a)/sizeof(a[0])] = &a; // Ugyanaz.
int **pip = &ip; // „&ip" egy mutató címe, nem a tömbé
```

Ugyanez a helyzet a többdimenziós tömbökkel, vagy hogy még precízebben fogalmazzunk, a tömbök tömbjeivel. Ugyanakkor ne feledjük, hogy a többdimenziós tömbök első elemének típusa tömb, és nem az alaptípus:

```
int aa[2][3];
const int aalen = sizeof(aa)/sizeof(aa[0]); // aalen = 2
```

Így példánkban az `aa` lényegét tekintve egy három egész elemet tartalmazó tömb első elemét címző mutatóliterál, nem pedig egy egészet címző mutató. ennek pedig lehet néhány meglepő következménye még akkor is, ha a megvalósítás technikai szempontból helyes:

```
void processElems( int *, size_t );
void processElems( void *, size_t );
// . . .
processElems( a, alen );
processElems( aa, aalen ); // Hoppá!
```

A túlterhelt `processElems` függvény első hívása azt a változatot jelöli, amelynek `int *` típusú a paramétere. Az a tömbnév ugyanis tulajdonképpen egy `int *`, csak álcázza magát. A második hívás ugyanakkor már azt a változatot jelenti, amelyiknek `void *` típusú a paramétere. A programozó persze valószínűleg nem ezt akarta elérni. A többdimenziós tömb neve az első elemét címző mutató, ami azonban most maga is a megadott méretű tömb, és nem a megadott alaptípusú tömbelem. Nincs olyan beleértett típusátalakítás, ami az `int(*)[3]` típust (vagyis a három egészből álló tömböt címző mutatót) az `int *` típusra alakítaná, viszont létezik olyan átalakítás, aminek a „célja" `void *`.

```
int (* const paa)[3] = aa;
int (* const *ppaa)[3] = &paa;
void processElems( int (*)[3], size_t );
// . . .
processElems( aa, aalen ); // Rendben.
```

A többdimenziós tömbök használata mindig gondot okoz. Célszerűbb helyettük a szabványos könyvtár tárolóit vagy olyan különleges tároló osztályokat használni, amelyek elvont módon valósítanak meg többdimenziós tömböket. Ha a helyzet valóban elkerülhetetlenné teszi a „nyers" többdimenziós tömbök kezelését, akkor is célszerű őket valamiféleképpen elszigetelni. Persze ez sem fog visszatartani egy igazán naiv felhasználót attól, hogy a következővel próbálkozzon:

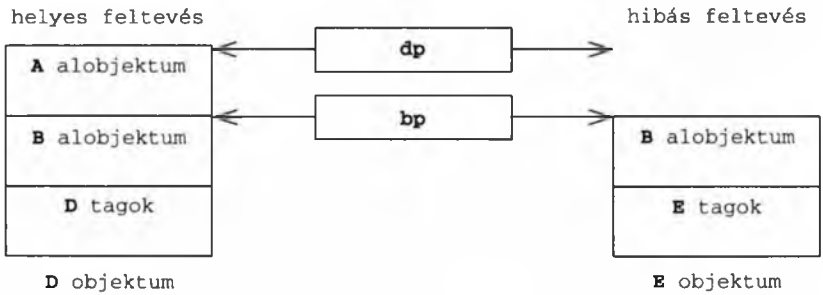
```
int *(*(*aryCallback)(int *(*)[n]))[n];
```

Ez (természetesen) egy olyan függvényt címző mutató, amelynek paramétere egy `n` darab egészet címző mutatót tartalmazó tömb mutatója, és egy ugyanilyen típusú értéket ad vissza. Nos rendben, ezt csak a példa kedvéért mutattam be (lásd a 11. hibát). Egy `typedef` persze nagyban egyszerűsítheti a helyzetet:

```
typedef int *(*PA)[n];
PA (*aryCallback)(PA); // Sokkal emberibb.
```

35. hiba: Nem ellenőrzött típusátalakítás származtatott típussá

Egy alaposztály mutatójának egy származtatott osztály mutatójává való átalakítása („downcasting") hibás címmeghatározáshoz vezethet, amint azt a következő példa és a 4.3. ábra is szemlélteti. A fordító által az átalakított mutatón végrehajtott delta aritmetika feltételezi, hogy az alaposztály címe a származtatott osztály alaposztálynak megfelelő területére esik:



4.3. ábra

Egy hibás statikus típusátalakítás hatása: egy eredetileg az E objektum B alobjektumát címző mutatót alakítunk D típusú mutatóvá.

```
class A { public: virtual ~A(); };
class B { public: virtual ~B(); };
class D : public A, public B {};
class E : public B {};
B *bp = getMeAB(); // Egy B-ből származtatott objektum
➥ előállítása
D *dp = static_cast<D*>(bp); // Biztonságos???
```

A legjobb megoldás, ha a tervezés során eleve elkerüljük az ilyen típusátalakításokat. Ezek rendszeres használata általában tervezési hibára utal. Ha mégis elkerülhetetlen egy ilyen átalakítás alkalmazása, akkor is használjuk inkább a `dynamic_cast` műveletet, mivel ez futásidejű átalakítást végez és ellenőrzi a címzés helyességét:

```
if( D *dp = dynamic_cast<D *>(bp) ) {
    // sikeres átalakítás
}
```

```
else {
   // sikertelen átalakítás
}
```

36. hiba: A típusátalakító műveletek helytelen használata

A típusátalakítások túlzott használata erősen növelheti a forráskód összetettségét. Mivel a típusátalakító műveleteket a fordító rejtve végzi, kétes esetek állhatnak elő, ha sok van belőlük egy osztály megvalósításában:

```
class Cell {
 public:
   // . . .
   operator int() const;
   operator double() const;
   operator const char *() const;
   typedef char **PPC;
   operator PPC() const;
   // És így tovább...
};
```

Az itt látható `Cell` osztály például annyi különböző igényt próbál egyszerre kielégíteni, hogy a fordítási félreértések miatt felhasználói gyakran mást kapnak majd tőle, mint amit vártak. És ami még rosszabb, még ha nincs is sehol félreértés vagy fordítási hiba, akkor is rettentően nehéz megállapítani, hogy a fordítóprogram miféle rejtett átalakításokat használt. Célszerűbb tehát, ha önmérsékletet tanúsítunk a típusátalakító műveletek használatával kapcsolatban, ahol pedig tényleg sok ilyenre van szükség, ott használjunk inkább közvetlenül megadott függvényeket:

```
class Cell {
 public:
   // . . .
   int toInt() const;
   double toDouble() const;
   const char *toPtrConstChar() const;
   char **toPtrPtrChar() const;
   // És így tovább...
};
```

Normális esetben elvárható, hogy osztályonként legfeljebb egy típusátalakító művelet forduljon elő, vagy ha lehet, annyi sem. Ha már kettő van belőlük, célszerű még egy pillantást vetni a kódra, a feladot ugyanis biztosan meg lehet egyszerűbben is oldani. Ha ennél is több fordul elő, ideje az egész felépítést újra átgondolni.

Még ha csupán egyetlen típusátalakító művelet van is egy osztályon belül, de az a konstruktorral kapcsolatos, akkor is akadhatnak gondjaink a nem egyértelmű megfogalmazással:

```
class B;

class A {
 public:
   A( const B & );
   // . . .
};

class B {
 public:
   operator A() const;
   // . . .
};
```

Mint látható, itt két módja is van annak, hogy egy `B` objektumot `A` típusúvá alakítsunk: használhatjuk `A` konstruktorát, vagy `B` megfelelő átalakító függvényét. Az eredmény fordítási bizonytalanság:

```
extern A a;
extern B b;
a = b; // Hiba! Kétértelmű!
a = b.operator A(); // Rendben, de furcsa.
a = A(b); // Hiba! Kétértelmű!
```

Vegyük észre, hogy nem tudjuk közvetlenül meghívni a konstruktort, vagy előállítani annak címét. Az `A(b)` kifejezés nem a konstruktort hívja, bár az efféle kifejezések normális esetben ezt szokták jelenteni. Ez a parancs csupán azt kéri a fordítótól, hogy alakítsa `b`-t `A` típusúvá, de hogy ezt hogyan csinálja, az nem derül ki. (Sajnálatos módon a legtöbb fordító ilyenkor nem jelez hibát, hanem csendben megoldja a dolgot az `A` konstruktorával.)

Általánosságban tehát jobb, ha feladataink megoldásához nem használjuk a típusátalakító műveleteket, és az egyetlen paramétert váró konstruktorokat `explicit`-ként vezetjük be, kivéve persze azokat az eseteket, amikor ez egyszerűen

nem megoldható. Ha a nem `explicit` konstruktorok és a típusátalakító műveletek használata elkerülhetetlen, az általános szabály az, hogy a konstruktorokkal a felhasználói típusok átalakítását célszerű megoldani, míg a típusátalakító műveletek alapvetően a beépített típusok átalakítására szolgálnak.

A típusátalakító műveletek igazi célja az, hogy az elvont adattípusokat beillesszék egy már létező típusrendszerbe azzal, hogy lehetővé teszik velük kapcsolatban ugyanazoknak a beleértett típusátalakításoknak a használatát, amelyek a beépített típusokkal kapcsolatban alkalmazhatók. Ennek megfelelően hiba ezeket az eszközöket valamiféle „értéknövelt" átalakítás megvalósítására használni:

```
class Complex {
   //  . . .
   operator double() const;
};
Complex velocity = x + y;
double speed = velocity;

class Container {
   // . . .
   virtual operator Iterator *() const = 0;
};
Container &c = getNewContainer();
Iterator *i = c;
```

Ebben a példában a `Complex` osztály tervezője azt akarta megvalósítani, hogy meghatározható legyen egy komplex számmal megadott vektor hossza. Ugyanakkor a felület felhasználója azt is gondolhatja, hogy a `double` típussá való átalakítás a komplex szám valós részét adja vissza, vagy a képzetest, vagy az irányszöget, vagy bármi más, az adott helyzetben logikusnak tűnő dolgot. A program tehát helyes, csak a programozó szándéka nem világlik ki belőle.

Az elvont `Container` felület tervezője nyilván egy „gyári eljárást" akart megvalósítani, amely elhelyezi a megfelelő bejáró (iterátor) mutatóját egy, a `Container` osztályból származtatott objektumban. Ugyanakkor nem alakítjuk át a `Container` objektumot `Iterator` típusúvá, így a „gyári eljárás" átalakításként való megvalósítása nem megfelelő, sőt félrevezető. Ezenkívül ez a megoldás még karbantartási problémákat is okozhat, ha a jövőben az eljárásnak egy paramétert is át kell vennie. Mivel a típusátalakító művelet nem tud paramétert kezelni, az egész megoldást helyettesítenünk kell majd egy nem műveletként működő függvénnyel. Ez pedig végül a `Container` osztály összes felhasználóját arra késztetheti, hogy megkeresse és átírja az átalakító művelet minden előfordulását.

Sokkal jobb tehát, ha a típusátalakító műveleteket arra használjuk, amire tervezőjük szánta őket. Ami pedig a felületeket illeti, azokat inkább közönséges függvényekkel, és ne műveletekkel oldjuk meg:

```
class Complex {
   // . . .
   double magnitude() const;
};
Complex velocity = x + y;
double speed = velocity.magnitude();

class Container {
   // . . .
   virtual Iterator *genIterator() const = 0;
};
Container &c = getNewContainer();
Iterator *i = c.genIterator();
```

Ez a tanács – magam tanúsíthatom – még akkor is érvényes, ha mindössze arról van szó, hogy `bool` (vagy esetenként `void *`) típusra akarunk alakítani, mondjuk azt jelzendő, hogy egy objektum érvényes vagy használható állapotban van:

```
class X {
 public:
   virtual operator bool() const = 0;
   // . . .
};
// . . .
extern X &a;
if( a ) {
   // „a" továbbra is használható
```

Itt megint valami „értéknövelt" átalakítási szolgáltatást akarunk nyújtani a típusátalakító műveletek segítségével, de az eredmény ismét bizonytalanság. A jövőben például meg akarhatjuk különböztetni azokat az `X` objektumokat, amelyek érvénytelenek, nem használhatók, illetve hibásak. Ezért jobb, ha precízebben fogalmazunk:

```
class X {
 public:
   virtual bool isValid() const = 0;
   virtual bool isUsable() const = 0;
   // . . .
};
```

```
// . . .
if( a.isValid() ) {
   // . . .
```

A szabványos `iostream` könyvtár típusátalakító műveletek használatával teszi lehetővé egy adatfolyam állapotának ellenőrzését:

```
if( cout ) // A cout megfelelő állapotban van?
   // . . .
```

Egy `operator void *` biztosítja ezt a lehetőséget, így a fenti kifejezés valahogy így fordítható le:

```
if( static_cast<bool>(cout.operator void *()) ) // . . .
```

Ha az `iostream` nem a megfelelő állapotban van, az átalakító művelet null mutatót ad vissza, ellenkező esetben ettől különbözőt. Mivel a mutatónak `bool` típussá való átalakítása a nyelvben előre meghatározott, ezt az értéket felhasználhatjuk az adatfolyam állapotának ellenőrzésére. Sajnos ugyanez a módszer felhasználható egy `void` típusú mutató értékének beállítására is:

```
void *coutp = cout; // Furcsa, és csaknem használhatatlan.
cout << cout << cin << cerr; // Bizonyos void *-ok kiírása.
```

Mindazonáltal az, hogy létezik az `iostream` `void *` típussá való átalakításának lehetősége, nem akkora baj, mint ha létezne ugyanez az átalakítás `bool` típusra is:

```
cout >> 12; // Ezt szerencsére nem lehet lefordítani.
```

Itt azt a gyakori hibát követtük el, hogy a balra tolás művelete helyett a jobbra tolás műveletét használtuk egy kimeneti folyammal kapcsolatban. Ha a fenti, `bool` típussá történő átalakítás létezne, ezt a kódot le lehetne fordítani. A `cout`-ot a fordító `bool` típussá alakítaná, azt rögtön átalakítaná egésszé (`int`), majd az eredményt 12 bittel eltolná jobbra. Világos tehát, hogy a `cout` `void *` típussá való átalakítása egyáltalán nem olyan veszélyes, mint ugyanez `bool` céltípussal. Ugyanakkor talán körültekintőbb tervezésre vallana, ha a nyelv alkotói beérték volna egy megfelelő és félreérthetetlen tagfüggvénnyel, valahogy így:

```
if( !cout.fail() )
   // . . .
```

37. hiba: A konstruktor szándékolatlan típusátalakítása

Egy egyparaméteres konstruktor mind alapbeállítást, mind típusátalakítást meghatározhat. Akár a műveletekkel megadott típusátalakítás esetében, a fordító a konstruktornál is rejtve (beleértve) kezeli a megadott átalakítást. Ez néha igen kényelmes lehet:

```
class String {
 public:
   String( const char * );
   operator const char *() const;
   // . . .
};
String name1( "Fred" ); // Közvetlen beállítás
name1 = "Joe"; // Beleértett átalakítás
const char *cname = name1; // Beleértett átalakítás
String name2 = cname; // Beleértett átalakítás, másoló beállítás
String name3 = String( cname ); // Meghatározott átalakítás,
➥ másoló beállítás
```

(Lásd még az 56. hibát.) A konstruktorok által végzett rejtett típusátalakítások persze nehezen érthetővé tehetik a kódot, sőt nehezen megfejthető hibák is előállhatnak. Vegyük például egy rögzített legnagyobb méretű verem tároló sablonját:

```
template <class T>
class BoundedStack {
 public:
   BoundedStack( int maxSize );
   ~BoundedStack();
   bool operator ==( const BoundedStack & ) const;
   void push( const T & );
   void pop();
   const T &top() const;
   // . . .
};
```

Az itt megvalósított `BoundedStack` típus rendelkezik az olyan szokásos veremműveletekkel, mint a `push` vagy a `pop`, sőt két vermet össze is lehet hasonlítani, hogy megállapítsuk, azonos-e a tartalmuk. Amikor létrehozzuk a `BoundedStack<T>` sablont, meg kell adnunk a verem legnagyobb méretét.

```
BoundedStack<double> s( 128 );
s.push( 37.0 );
s.push( 232.78 );
// . . .
```

Sajnos az egyparaméteres konstruktor értelmezhető típusátalakításként is, pedig számunkra talán hasznosabb lenne egy fordítási hiba:

```
if( s == 37 ) { // Hoppá!
   // . . .
```

A fenti esetben nagy az esélye annak, hogy valódi szándékunk egy ilyen feltétel megfogalmazása volt: `s.top() == 37`. Sajnos azonban a kód hiba nélkül lefordítható, mivel a fordítóprogram a megadott egész számot (`37`) képes `BoundedStack<double>` típusúvá alakítani, és a `BoundedStack<double>::operator ==` műveletnek paraméterként átadni. A fordító tehát tulajdonképpen a következő kódot állítja elő:

```
BoundedStack<double> stackTemp( 37 );
bool resultTemp( s.operator ==( stackTemp ) );
stackTemp.~BoundedStack<double>();
if( resultTemp ) {
   // . . .
```

Ez a kód nyelvi szempontból helyes, de működését tekintve hibás, és esetenként igen sokba kerülhet a kijavítása. Biztonságosabb megoldás lett volna a `BoundedStack` konstruktorát `explicit`-ként megadni. Az `explicit` kulcsszó megtiltja a fordítónak, hogy a konstruktort rejtett típusátalakításra használja, ugyanakkor a közvetlenül meghatározott (explicit) típusátalakítás továbbra is lehetséges:

```
template <class T>
class BoundedStack {
 public:
   explicit BoundedStack( int maxSize );
   // . . .
};
// . . .
if( s == 37 ) { // Szerencsére hibás
   // . . .
if( s.top() == 37 ) { // Helyes, nincs típusátalakítás.
   // . . .
if( s == static_cast< BoundedStack<double> >(37) ) { // Helyes...
   // . . .
```

Egy alattomos rejtett típusátalakítás sokkal veszélyesebb, mint az a kényelmetlenség, amit egy esetleges meghatározott típusátalakítás megadása okozhat. Ennek megfelelően célszerű az összes egyparaméteres konstruktort `explicit`-ként megadni.

Ha így teszünk, az azt is befolyásolja, hogy milyen forma megengedett az osztályok objektumainak bevezetésénél. Módosítsuk korábbi `String` osztályunkat, hogy lássuk a hatást:

```
class String {
 public:
   explicit String( const char * );
   operator const char *() const;
   // . . .
};
String name1( "Fred" ); // Rendben.
name1 = "Joe"; // Hiba!
const char *cname = name1; // Rejtett típusátalakítás, rendben
String name2 = cname; // Hiba!
String name3 = String( cname ); // Meghatározott típusátalakítás,
➥ rendben
```

A `name2` másolva történő beállításának részét képező rejtett ideiglenesváltozó-előállítás, valamint a `String::operator =` paramétere immár nem megengedett. A `name3` kezdeti beállítása még mindig helyes, mivel meghatározott típusátalakítást tartalmaz. (Bár helyesebb lett volna ezt is a `static_cast` művelettel végrehajtani; lásd a 40. hibát.) Mint általában, itt is célszerűbb közvetlen kezdeti beállítást használni a másolva történő helyett. (Ezzel kapcsolatban lásd az 56. hibát.)

Mielőtt magunk mögött hagynánk az `explicit` kulcsszóval kapcsolatos történeteket, érdemes megnézni egy gondolatébresztő, ám ma már elavult módszert arra, hogyan valósíthatjuk meg az e kulcsszó nyújtotta nyelvi szolgáltatást, magának az `explicit` szónak a leírása nélkül:

```
class StackInit {
 public:
   StackInit( size_t s ) : size_( s ) {}
   int getSize() const { return size_; }
 private:
   int size_;
};
template <class T>
class BoundedStack {
```

```
  public:
    BoundedStack( const StackInit &init );
    // . . .
};
```

Mivel a `BoundedStack` konstruktorának megadásában most nem szerepel az `explicit` kulcsszó, a fordítóprogram kísérletet tesz arra, hogy minden `StackInit` objektumot `BoundedStack` objektummá alakítson. Ugyanakkor egy egészet már nem fog előbb `StackInit` típusúvá alakítani, majd azt egy másik, szintén rejtett átalakítással `BoundedStack` típusúvá. A szabvány ugyanis előírja, hogy a fordító egyszerre csak egy rejtett felhasználói típusátalakítást hajthat végre:

```
BoundedStack<double> s( 128 ); // Rendben.
BoundedStack<double> t = 128; // Rendben.
if( s == 37 ) { // Hiba!
    // . . .
```

Ez a módszer csaknem ugyanazt nyújtja, mint korábban az `explicit` kulcsszó. A fenti példában `s` és `t` deklarációja helyes, ugyanis a `128` egyetlen rejtett átalakítással átvihető a `StackInit` típusba a konstruktornak való átadás során. Ugyanakkor a `37` `BoundedStack<double>` típussá való rejtett átalakítását a fordító már nem végzi el, mivel ez két rejtett felhasználói típusátalakítást igényelne: előbb egy `int` típust kellene `StackInit`-té, majd ezt `BoundedStack<double>`-lá alakítani.

38. hiba: Típusátalakítás többszörös öröklés esetén

Többszörös öröklés esetén egy objektumnak több érvényes címe is lehet. Az öröklő objektumokban jelen levő alaposztálybeli alobjektumoknak ugyanis egyedi címük van, és ezek bármelyike érvényes címe lehet az objektumnak. (Egy gyengén megtervezett egyszeres öröklésre alapozott rendszerben minden objektumnak két érvényes címe lehet. Ezzel kapcsolatban lásd még a 70. hibát.)

```
class A { /* . . .*/ };
class B { /* . . .*/ };
class C : public A, public B { /* . . .*/ };
// . . .
C *cp = new C;
A *ap = cp; // Rendben.
B *bp = cp; // Rendben.
```

A fenti példában a C teljes objektum B alobjektumának kezdőcíme feltehetőleg adott értékkel („delta") a teljes objektum kezdete utánra mutat. Ennek megfelelően ha cp-t B * típusúvá alakítjuk, akkor a mutató ezzel az eltolással módosítja annak tartalmát s így egy érvényes, a B osztályba tartozó objektumot címző mutató keletkezik. Ez az átalakítás megőrzi a típust, hatékony, és a fordító automatikusan képes végrehajtani.

Hasonlóan, az a tény, hogy egy objektumnak több érvényes címe is lehet, arra készteti a C++ fordítót, hogy pontosan járjon el a mutatók összehasonlítása során:

```
if( bp == cp ) // . . .
```

Ebben a sorban nem azt a kérdést tesszük fel, hogy „Ugyanazt a bitmintát tartalmazza ez a két mutató?", hanem azt, hogy „Azonos objektumot címez ez a két mutató?". Az összehasonlítás ennél természetesen sokkal összetettebb is lehet, ez a fordítót nem zavarja. Hatékonyan, automatikusan és megbízhatóan fogja elvégezni az összehasonlítást. A fordítóprogram egyébként valószínűleg a következőhöz hasonló módon valósítja meg ezt a szolgáltatást:

```
if( bp ? (char *)bp-delta==(char *)cp : cp==0 )
```

A régi és az új stílusú típusátalakító műveleteket egyaránt használhatjuk, mivel mind helyesen kezelik az osztályobjektumok eltolási címeivel kapcsolatos számításokat. Ugyanakkor a fenti típusátalakítástól eltérően arra már nincs garancia, hogy a típusátalakítás végeredményeként érvényes objektumcímet kapunk. (A dynamic_cast művelet használata ezt is garantálja, viszont egyéb problémákat vet fel. Ezzel kapcsolatban lásd a 97. , 98. és 99. hibákat.)

```
B *gimmeAB();
bp = gimmeAB();
cp = static_cast<C *>(bp); cp = (C *) bp;
typedef C *CP;
cp = CP( bp );
```

A fenti példában a bp mutatón végrehajtott mindhárom típusátalakítás helyesen fogja kezelni a „delta aritmetikát", de az eredmény csak akkor lesz érvényes objektumcím, ha a bp által címzett B objektum része a befoglaló C objektumnak. Ha ez a feltétel nem teljesül, a kapott cím érvénytelen lesz, viszont megfeleltethető valamiféle „kreatív" C stílusú kód eredményének:

```
cp = (C *)((char *)bp-delta)
```

A `reinterpret_cast` művelet pontosan azt teszi, amit a neve ígér: átértelmezi egy mutató által hordozott bitminta jelentését anélkül, hogy magukat a biteket bárhogyan módosítaná. Ez a megoldás gyakorlatilag kikapcsolja az előbb említett eltolási címeket kezelő „delta aritmetikát". (Hogy egészen pontos legyek, maga a szabvány úgy fogalmaz, hogy ennek a típusátalakításnak a viselkedése a konkrét megvalósítástól függ, a gyakorlatban azonban ezalatt mindig az említett viselkedést értik. Ugyanakkor erre a szabvány garanciát nem ad, és egyes esetekben a `reinterpret_cast` akár módosíthatja is a bitmintát.)

```
cp = reinterpret_cast<C *>(bp);  // Igen, tényleg azt akarom,
➥ hogy ez a program vészleállást hajtson végre...
```

A típusátalakítások valamennyi felsorolt típusával tulajdonképpen azt várjuk el egy objektumtól, hogy többet nyújtson számunkra, mint amennyit a felülete biztosít, ami egyértelműen rossz tervezésre utal. Túl keveset tudunk az objektumról, és egy statikus típusátalakítással olyan szerepre kényszerítjük, amit nem is biztos, hogy képes lesz betölteni. Hasznosabb tehát, ha az osztályobjektumokkal kapcsolatban igyekszünk elkerülni a statikus típusátalakításokat. Valamivel később azt is meg fogom mutatni, hogy még a dinamikus típusátalakítás sem igazán elfogadható ilyen helyzetekben.

39. hiba: Nem teljes típusok átalakítása

A nem teljes típusoknak nincs meghatározása, de ettől még megcímezhetjük őket mutatók vagy hivatkozások segítségével, és olyan függvényeket is bevezethetünk, amelyek ilyen típusú paramétereket várnak, illetve adnak vissza. Ez általánosan használt és bevált programozási módszer:

```
class Y;
class Z;
Y *convert( Z * );
```

Gond akkor lehet, ha a programozó elkezdi erőltetni a dolgot. A tudatlanság csak egy bizonyos mértékig boldogít:

```
Y *convert( Z *zp )
   { return reinterpret_cast<Y *>(zp); }
```

A `reinterpret_cast` műveletre itt feltétlenül szükség van, mivel a fordító semmilyen információval nem rendelkezik az `Y` és `Z` típusok viszonyáról. Így a legtöbb, amit ez az művelet nyújtani tud számunkra, az, hogy nevének megfelelően „átértelmezi" a `Z` mutatóban kapott bitmintát `Y` típusúvá. Ez egy ideig akár működhet is:

```
class Y { /* . . .*/ };
class Z : public Y { /* . . .*/ };
```

Az valószínű, hogy a `Z` objektumon belül található `Y` alobjektumnak ugyanaz a címe, mint a teljes objektumé. Ezt a szívességet azonban nem várhatjuk el mindig tőlük, és egy távoli időpontban, mondjuk egy karbantartás alkalmával a helyzet egyszer csak megváltozhat, érvénytelenítve ezzel a típusátalakítást is. (Ezzel kapcsolatban lásd még a 38. és a 70. hibákat.)

```
class X { /* . . .*/ };
class Z : public X, public Y { /* . . .*/ };
```

A `reinterpret_cast` művelet használata feltehetőleg ki fogja kapcsolni az eltolási címeket kezelő „delta aritmetikát" és ezzel el is rontja `Y` címét.

Ami azt illeti, a `reinterpret_cast` művelet nem az egyetlen lehetséges megoldás, használhatunk régi stílusú típusátalakítást is. Ez elsőre jó megoldásnak tűnhet, hiszen egy régi vágású típusátalakítás biztosan helyesen kezeli az eltolási címeket, ha elég információ áll a fordító rendelkezésére. Ugyanakkor éppen ez a rugalmasság okozhatja a vesztünket, hiszen ugyanannak az átalakításnak más és más lehet a végeredménye, attól függően, hogy éppen milyen információval rendelkezik a fordító abban a pillanatban, amikor az átalakítást kezeli:

```
Y *convert( Z *zp )
   { return (Y *)zp; }
// . . .
class Z : public X, public Y { // . . .
// . . .
Z *zp = new Z;
Y *yp1 = convert( zp );
Y *yp2 = (Y *)zp;
cout << zp << ' ' << yp1 << ' ' << yp2 << endl;
```

Ebben a példában `yp1` értéke vagy `zp`, vagy `yp2` értékével lesz azonos, attól függően, hogy az átalakítást a `Z` osztály meghatározása előtt vagy után végezzük.

A helyzet aztán még ennél is bonyolultabbá válhat, ha a `convert` egy sablonfüggvény, amelynek számos különböző példánya létezik számos különböző objektumfájlban. Ebben az esetben a típusátalakítás végkifejlete az összeszerkesztő egyéni hóbortjaitól függ (lásd a 11. hibát).

Ha már választani kell, használjuk inkább a `reinterpret_cast` műveletet, annak eredménye legalább egyértelműen hibás. Ami engem illet, én az Olvasó helyében mindkettőt messze elkerülném.

40. hiba: Régi stílusú típusátalakítás

Ne használjunk régi stílusú típusátalakításokat. Túl sokat tudnak, de ehhez képest túl egyszerű a használatuk. Vegyünk például egy fejállományt:

```
// emp.h
// . . .
const Person *getNextEmployee();
// . . .
```

Ezt az állományt számos helyen használhatjuk programunk forráskódjában, beleértve a következő szakaszt is:

```
#include "emp.h"
// . . .
Person *victim = (Person *)getNextEmployee();
dealWith( victim );
// . . .
```

Innen bármilyen típusátalakítás, ami az adott típus állandóságát sérti, életveszélyes lehet, de legalábbis nem hordozható. Most mégis tételezzük fel egy pillanatra, hogy aki ezt a kódot írta, az sokkal tisztábban lát az ilyen kérdésekben, mint mi, átlagemberek, így azt is kiderítette, hogy ez a bizonyos átalakítás helyes is, meg hordozható is. Én állítom, hogy a kód mindezek ellenére mégis hibás, sőt kétszeresen is az. Először is az adott típusátalakítás sokkal erősebb annál, mint ami az adott helyzetben szükséges lenne. Másodszor a szerző ugyanabba a hibába esett, mint megannyi kezdő programozó: programjában egyfajta „másodlagos jelentésre" épít, amely feltételezi, hogy az az általa felderített titkos, elvont viselkedésforma, ami a `getNextEmployee` függvényben ölt testet, a jövőben is támogatott lesz.

Lényegében ez a megvalósítás azt tételezi fel, hogy a `getNextEmployee` függvény soha több nem fog változni. Persze ez a feltevés hamarosan megdől. Az `emp.h` fejállomány szerzője ráeszmél, hogy az alkalmazottak nem emberek, és ennek megfelelően kijavítja a kódot:

```
// emp.h
// . . .
const Employee *getNextEmployee();
// . . .
```

Sajnos a típusátalakítás továbbra is érvényes, de jelentése már egészen más. Ebben a helyzetben már nem az objektum állandóságát változtatja meg, hanem az adott objektumra alkalmazható műveletek halmazát módosítja. Amikor ilyen típusátalakítást használunk, ezzel azt mondjuk a fordítónak, hogy mi többet tudunk az érvényben lévő típusrendszerről, mint ő. Kezdetben persze lehet, de az már nem valószínű, hogy ha módosult a fejállomány, akkor minden egyes helyet végignéztünk az egész kódban, ahol hivatkoztunk rá. A fordítónak tett dölyfös kijelentésünk így a vesztünket okozza, mert senki nincs, aki kijavíthatna bennünket. Ha új stílusú típusátalakítást használtunk volna, a fordító észrevette volna a változást, és hibaüzenettel figyelmeztetett volna bennünket annak mellékhatásaira:

```
#include "emp.h"
// . . .
Person *victim = const_cast<Person *>(getNextEmployee());
dealWith( victim );
```

Érdemes megjegyezni, hogy a `const_cast` művelet használata a régi stílusú típusátalakítások helyett még mindig veszélyes. Ebben az esetben is arra a ki nem mondott feltételezésre támaszkodunk ugyanis, hogy a `const_cast` műveletet megengedő `dealWith` és `getNextEmployee` függvények kapcsolata a jövőben is változatlan marad.

41. hiba: Statikus típusátalakítás

Statikus típusátalakítás alatt – minő meglepetés – a nem dinamikus típusátalakításokat értjük. Ez a fantasztikus meghatározás magában foglalja nemcsak a `static_cast`, hanem a `reinterpret_cast`, a `const_cast` műveleteket, és a régi stílusú típusátalakításokat is.

A statikus típusátalakítások legnagyobb baja az, hogy statikusak. Ha ilyesmit alkalmazunk, azzal arra kényszerítjük a fordítót, hogy az objektum tulajdonságaival kapcsolatban a mi elképzelésünket fogadja el, és ne azt, amit maga az objektum állít magáról. Bár a legtöbb statikus típusátalakításokra épített kód kezdetben helyes, az ilyen program elveszti azt a képességét, hogy önmagát hozzáigazítsa az objektumok típusrendszerében később bekövetkező változásokhoz. Mivel ezek a változások általában időben távol esnek a statikus típusátalakítás „elkövetésétől", a programozók nem szokták megfelelően módosítani az ilyen kódokat. Ezek után igazán sajnálatos, hogy a statikus típusátalakítással az összes olyan ellenőrzési műveletet is kikapcsoljuk, amelyek segítségével a fordító az esetlegesen felmerülő hibákat kiszűrhetné.

A típusátalakítások persze nem eredendően gonoszak, de csínján kell bánni velük, és olyan rendszert kell kidolgozni, ami lehetővé teszi az időben távoli karbantartást és azt, hogy a módosítások ne érvénytelenítsék a korábbi típusátalakításokat. Mindez tisztán gyakorlati szempontból azt jelenti, hogy határozottan kerülendő az elvont adattípusok átalakítása, különösen abban az esetben, ha azok egy hierarchia részét képezik.

Vegyünk egy egyszerű hierarchiát:

```
class B {
 public:
   virtual ~B();
   virtual void op1() = 0;
};

class D1 : public B {
 public:
   void op1();
   void op2();
   virtual int thisop();
};
```

Ehhez a hierarchiához tartozzon egy függvény, amely arra való, hogy segítségével bizonyos `B` típusú objektumokat állítsunk elő. Kezdetben esetleg csak egyetlen származtatott osztályunk van, tehát a megvalósítás pofonegyszerű:

```
B *getAB() { return new D1; }
```

Sajnos később kiderülhet, hogy a fejlesztőnek vagy a karbantartást végző programozónak szüksége van a `getAB` által visszaadott objektum `D1` típussal kapcsolatos műveleteire is. Ebben az esetben a helyes eljárás az, ha újratervezzük a hierarchiát úgy, hogy az objektum típusa statikusan ismert legyen. Ha ez nem lehetséges vagy valamiért nem célszerű, akkor használhatjuk a `dynamic_cast` műveletet is (persze csak megfelelő önvizsgálat után). A statikus típusátalakítás használata – mint ahogy mi is tettük ebben a példában – szinte soha nem jó ötlet:

```
B *bp = getAB();
D1 *d1p = static_cast<D1 *>(bp);
d1p->op1();
d1p->op2();
int a = d1p->thisop();
```

Ez a kód kizárólag azért működőképes, mert a visszaadott objektum történetesen `D1` típusú. Nem valószínű azonban, hogy ez a rendkívül egyszerű helyzet a jövőben is megmarad. Bármikor felvehetünk a hierarchiába egy új származtatott osztályt a megfelelő előállító függvénnyel együtt:

```
class D2 : public B {
 public:
   void op1();
   void op2();
   virtual char thatop();
};
// . . .
B *getAB() {
   if( rand() & 1 )
        return new D1;
   else
        return new D2;
}
```

Ráadásul, ahogy az lenni szokott, ez a változás a kód egy távoli zugában fog bekövetkezni, amit a statikus típusátalakítást végrehajtó kódrészlet karbantartója esetleg nem is ismer. Joggal remélhetnénk persze, hogy a `getAB` függvény megváltozása legalább az érintett kódrészlet újrafordítását kiváltja majd, de a gyakorlatban még ez sem biztos. Még ha meg is történik az újrafordítás, a statikus típusátalakítás akkor is gondoskodik a fordító által végrehajtható ellenőrzések kikapcsolásáról. Meglehetősen kevés dolog garantálható a program működésével kapcsolatban akkor, ha a `getAB` függvény `D2` típusú objektumot ad vissza. Mindazonáltal nagyon valószínű, hogy futni azért fog, csak a működése követi mindig a legújabb divatot. Az alábbi kódban szereplő megjegyzések a leggyakoribb viselkedésformákat próbálják meg leírni:

```
B *bp = getAB(); // Egy D2 objektum megszerzése
D1 *d1p = static_cast<D1 *>(bp); // Csináljunk úgy, mintha D2
➥ valójában D1 lenne
d1p->op1(); // #1: D2::op1 hívása!
d1p->op2(); // #2: D1::op2 hívása!!
int a = d1p->thisop(); // #3: D2::thatop hívása!!!
```

A garancia teljes hiánya ellenére a `#1`-gyel jelzett sor nagy valószínűséggel azt csinálja, amit kell. Persze az azért itt is jobb megoldás lett volna, ha az `op1` függvényt az alaposztály nyújtotta felületen keresztül hívjuk meg, ez ugyanis biztosítja a helyes viselkedést.

A `#2`-vel jelzett sor már problémásabb. Ez egy nem virtuális tagfüggvény hívása a `D1` taggal. Sajnos ez a függvény egy `D2` típusú objektumot kap meg, ami közelebbről meg nem határozható viselkedést eredményez majd futás közben. Akár még működhet is.

Valószínűleg a `#3`-mal jelzett sorral lesz a legtöbb gond . Statikusan hívjuk meg a `D1` `thisop` nevű virtuális függvényét, aminek `int` a visszatérési típusa. Ugyanakkor dinamikusan hívjuk meg a `D2` `thatop` nevű virtuális függvényét, ami `char` típust ad vissza. Ha ez a kód egyáltalán lefut anélkül, hogy a program leállna tőle, megpróbáljuk bemásolni a `char` típusú visszatérési értéket egy `int` típusú változóba.

A statikus típusátalakítások használata – amint azt Scott Meyers bölcsen megállapította – rendszerint annak a jele, hogy „a köztünk és a fordítóprogram közti tárgyalások sajnálatos módon megszakadtak". A statikus típusátalakítással azt mondjuk ugyanis a fordítónak, hogy „csináld ezt, azért mert én ezt mondom". Mármost egy ilyen fordulat emberi beszélgetőpartnerrel folytatott társalgás esetén is a téma és az értelmes eszmecsere statáriális lezárását szokta jelenteni. Az ilyen módszer alkalmazásával teljesen figyelmen kívül hagyjuk a típusátalakításra ítélt elvont adattípus nyilvános felülete által felkínált szolgáltatásokat. A szakmailag megalapozott, átgondolt, az adattípusok nyilvános tulajdonságait figyelembe vevő megoldások megvalósítása persze rendszerint nagyobb szellemi erőfeszítést igényel, mint egy „ellentmondást nem tűrő" típusátalakítás, viszont cserébe hibatűrőbb, hordozhatóbb és legfőképpen használhatóbb programot kapunk.

42. hiba: Formális paraméterek ideiglenes kezdeti beállítása

Vegyünk egy `String` nevű osztályt, amely egyenlőségi műveleteket is tartalmaz:

```
class String {
 public:
   String( const char * = "" );
   ~String();
   friend bool operator ==( const String &, const String & );
   friend bool operator !=( const String &, const String & );
   // . . .
 private:
   char *s_;
};
inline bool
operator ==( const String &a, const String &b )
   { return strcmp( a.s_, b.s_ ) == 0; }

inline bool
operator !=(const String &a, const String &b )
   { return !(a == b); }
```

Vegyük észre, hogy ez a megoldás egy nem közvetlenül meghatározott („nem explicit") egyparaméteres konstruktort és nem tag egyenlőségi műveleteket tartalmaz. Ezzel pedig arra bátorítjuk az osztály felhasználóját, hogy rejtett típusátalakításokkal egyszerűsítse az általa írt kódot:

```
String s( "Hello, World!" );
String t( "Yo!" );
if( s == t ) {
   //  . . .
}
else if( s == "Howdy!" ) { // Rejtett típusátalakítás
   //  . . .
}
```

Az itt látható első feltétel (`s == t`) valóban hatékony. Az `operator ==` két hivatkozás típusú formális paraméterének kezdőértékként az `s`-t és a `t`-t adjuk meg, majd az összehasonlítást az `strcmp` függvénnyel végezzük el. Ha a fordító úgy dönt, hogy helyben (inline) beszúrja az `operator ==` hívását (valószínűleg ezt fogja tenni, hacsak valamilyen hibakereső kapcsolót be nem állítunk), az eredmény az `strcmp` közvetlen hívása lesz.

A második hívás (`s == "Howdy!"`) már nem annyira hatékony, de még mindig helyes. Az `operator ==` második formális paraméterének beállításához a fordítónak létre kell hoznia egy ideiglenes `String` típusú objektumot, és abban a megadott szöveget kell elhelyeznie. Ezt az ideiglenes adatot fogja aztán a formális paraméter beállítására használni. A függvény visszatérése után ezt az ideiglenes objektumot meg kell semmisíteni. A művelet hívása tehát valahogy így fest:

```
String temp( "Howdy!" );
bool result = operator ==( s, temp );
temp.~String();
if( result ) {
   // . . .
}
```

Ebben az esetben a beleértett típusátalakítás nyújtotta kényelem megéri az árát, mivel mind a `String` osztály megvalósítását, mind a felhasználói kódot röviddé és világossá teszi.

Ugyanakkor létezik két olyan eset, amikor az ilyen rejtett típusátalakítás biztosan nem elfogadható. Az egyik természetesen az, amikor a kód igen gyakran használ rejtett átalakításokat, és ez sebességgel vagy tárfoglalással kapcsolatos gondokat okoz. A másik, ha a `const char *` típusról `String` típusra való rejtett átalakítás a `String` osztály más helyen történő használatával kapcsolatban félreérthető helyzetet teremt, így a programozó kénytelen úgy dönteni, hogy a konstruktort `explicit`-ként adja meg.

A `String` osztály egyenlőségi műveleteinek túlterhelésével persze ez a probléma könnyen megoldható:

```
class String {
 public:
   explicit String( const char * = "" );
   ~String();
   friend bool operator ==( const String &, const String & );
   friend bool operator !=( const String &, const String & );
   friend bool operator ==( const String &, const char * );
   friend bool operator !=( const String &, const char * );
   friend bool operator ==( const char *, const String & );
   friend bool operator !=( const char *, const String & );
   // . . .
};
```

Így a fordító a paraméterek bármilyen érvényes kombinációjához képes kiválasztani a megfelelő függvényt, s így nem lesz szükség ideiglenes `String` objektumokra. Persze ilyen módon a `String` osztály megvalósítása bonyolultabb, nehezebben érthető, amiből az következik, hogy ezt a megközelítési módot inkább csak akkor tanácsos használni, ha a vele kapcsolatos igény egyértelmű.

A kezdő C++ programozók egyik leggyakoribb hibája az, hogy akkor is érték szerint adnak át egy objektumot, amikor egy hivatkozás használata célszerűbb lenne. Vegyük például a következő függvényt, amely egy `String` típusú objektumot vár bemenő paraméterként:

```
String munge( String s ) {
   // munge s...
   return s;
}
// . . .
String t( "Munge Me" );
t = munge( t );
```

Őszintén szólva nehéz erről a kódról bármi jót mondani, annak ellenére, hogy hasonlót gyakran találhatunk a kezdők programozóknál. A `munge` függvény meghívásakor szükség van mind az `s` formális paraméter, mind a visszatérési érték másoló műveletének meghívására, majd visszatérés után az ideiglenes `s` objektumot el is kell takarítani. Mivel a visszatérési értéket ugyanabba az objektumba akarjuk visszatenni, ahonnan a bemenetet vettük, esetleg abban reménykedhetünk, hogy a fordító majd felismeri ezt a helyzetet, és egy üres utasítást hajt végre a sok másolás helyett. Sajnos nincs ilyen szerencsénk. A fordító kénytelen a visszatérési értéket egy ideiglenes változóban elhelyezni (amit aztán később törölnie is kell), így az értékadás nem optimalizálható. Ami tehát egyetlen függvénynek látszik, az valójában hat különböző függvényhívás.

Jobb megoldást eredményez, ha a `munge` függvény átírjuk úgy, hogy az általa kezelt adathoz egy álnéven keresztül férjen hozzá:

```
void munge( String &s ) {
   // munge s...
}
// . . .
munge( t );
```

Ez pontosan egy függvényhívás. A két eltérő módon megvalósított `munge` függvénynek persze kissé eltérő jelentése van, hiszen az `s` paraméteren végrehajtott

művelet eredménye most azonnal megjelenik a bemenetként használt objektumban, nem pedig visszatérési értékként jutunk hozzá. (Ez az eltérés akkor válik lényegessé, ha a függvény végrehajtása során kivétel keletkezik, vagy megszakítás érkezik, illetve ha a `munge` meghív egy másik függvényt, ami ugyanarra a változóra hivatkozik.) A program összetettsége viszont csökkent, a kód rövidebb és hatékonyabb lett.

A hivatkozásokon keresztül megvalósított paraméterátadás sablonok esetén különösen nagy jelentőséggel bír, mivel itt senki nem tudja előre megjósolni, hogy egy tényleges megvalósítás során mekkora számításigényt jelentene a paraméterek érték szerinti átadása:

```
template <typename T>
bool operator >( const T &a, const T &b )
    { return b < a; }
```

A hivatkozással történő paraméterátadásnak ugyanakkor előre adott, és viszonylag alacsony teljesítményigénye van, ami nem változik a paraméter típusával. Előfordulhat persze, hogy egyes egészen kis osztálytípusok vagy beépített típusok esetében az érték szerinti átadás hatékonyabb. Ilyenkor, ha ezek a különleges esetek számunkra valóban fontosak, túlterhelhetjük a sablont (függvénysablon esetén), vagy „kihegyezhetjük" az adott helyzetre (osztálysablon esetén).

Egyes esetekben a hagyomány diktálhatja az érték szerinti átadás használatát. A C++ szabványos sablonkönyvtárában például „hagyományosan" érték szerint történik a függvényobjektumok átadása. (A függvényobjektum egy osztály olyan eleme, amely a függvényhívás műveletét terheli túl. Ez is ugyanolyan osztályelem, mint az összes többi, de lehetővé teszi számunkra, hogy a függvényhívások utasításformáját használjuk vele kapcsolatban.)

Ezt a lehetőséget felhasználhatjuk például egy „állító" függvény (predikátum) írására, amely a paramétereivel kapcsolatban valamilyen állítást fogalmaz meg:

```
struct IsEven : public std::unary_function<int,bool> {
    bool operator ()( int a )
        { return !(a & 1); }
};
```

Az `IsEven` objektumoknak nincsenek adattagjaik, virtuális függvényeik, nincs konstruktoruk vagy destruktoruk. Ha egy ilyen objektumot adunk át érték szerint, az nem igényel szinte semmiféle számítási teljesítményt (gyakran ténylegesen se-

mennyit), ezért a szabványos sablonkönyvtár (STL) használatakor ez az általánosan elfogadott módszer a függvényparaméterek névvel nem rendelkező ideiglenes objektumként történő átadására:

```
extern int a[n];
int *thatsOdd = partition( a, a+n, IsEven() );
```

Az `IsEven()` kifejezés létrehoz egy `IsEven` típusú névtelen ideiglenes objektumot, amit a program később érték szerint fog átadni a `partition` algoritmusnak (lásd a 43. hibát). Természetesen ez a szokványos megoldás hallgatólagosan feltételezi, hogy a szabványos sablonkönyvtár segítségével használt függvényobjektumok viszonylag kicsik, s így érték szerinti átadásuk hatékony.

43. hiba: Ideiglenes élettartam

Bizonyos helyzetekben a fordító kénytelen ideiglenes objektumokat létrehozni. Az ilyen ideiglenes objektumok élettartamával kapcsolatban a szabvány úgy fogalmaz, hogy az az objektum létrehozásától a legtágabb befoglaló kifejezés végrehajtásáig terjed. (A szabvány a legtágabb befoglaló kifejezést „teljes kifejezésként" említi.) A legáltalánosabb problémát ezzel kapcsolatban az jelenti, hogy a kód egyes elemei esetleg akkor is támaszkodni próbálnak ezen ideiglenes elemek létezésére, amikor azok már megszűntek létezni:

```
class String {
 public:
   // . . .
   ~String()
        { delete [] s_; }
   friend String operator +( const String &, const String & );
   operator const char *() const
        { return s_; }
 private:
   char *s_;
};
// . . .
String s1, s2;
printf( "%s", (const char *)(s1+s2) ); // #1
const char* p = s1+s2; // #2
printf( "%s", p ); // #3
```

A `String` osztály kéttényezős `+` műveletének fenti megvalósítása általában azt igényli, hogy a visszatérési érték számára a fordító egy ideiglenes objektumot hozzon létre. Ez a művelet mindkét fent látható használatára érvényes. A `#1` jelű sorban az `s1+s2` művelet eredménye előbb egy névtelen `String` típusú ideiglenes objektumba kerül, majd ezt a fordító `const char *` típusúvá alakítja, mielőtt átadná a `printf` függvénynek. A `printf` visszatérése után aztán ez az ideiglenes objektum megszűnik létezni. A rendszer tökéletesen működik, hiszen ideiglenes változónk pontosan addig élt, amíg valakinek szüksége volt rá.

A `#2`-vel jelzett sorban kezdetben pontosan ugyanez történik: az `s1+s2` művelet eredménye bekerül egy ideiglenes változóba, majd a fordító azt `const char *` típusúvá alakítja. A különbség a két eset között az, hogy a `p` mutató beállítása után ez a `String` típusú ideiglenes objektum eltűnik. Amikor aztán a `printf` függvény használni próbálja a mutató által címzett tartalmat, az már talán nem is létezik. A dolog végkifejlete megjósolhatatlan.

Az igazán szerencsétlen dolog ebben az esetben az, hogy ez a kód gond nélkül lefordítható, sőt talán hosszú ideig helyesen fog működni (legalábbis a tesztelés alatt). Ennek pedig az az oka, hogy az ideiglenes `String` objektum törlésekor a törlő művelet esetleg nem változtatja meg ténylegesen a korábbi tartalmat, csak bejegyzi, hogy az adott tárterület felszabadult. Ha ezt a területet a `#2` és `#3` sorok között semmi sem használja, a tartalom megússza a törlést, a program pedig működni látszik. Ha aztán ezt a kódot később beépítjük például egy többszálú alkalmazásba, az rejtélyes módon szórványosan hibázni fog.

Jobb megoldás tehát, ha valamilyen összetettebb kifejezést használunk, vagy közvetlenül létrehozunk egy valamivel hosszabb élettartamú ideiglenes változót:

```
String temp = s1+s2;
const char *p = temp;
printf( "%s", p );
```

A korlátozott élettartamú ideiglenes objektumoknak időnként nagy hasznát vehetjük. A szabványos könyvtár használata során például általános az eljárás, hogy a könyvtár egyes összetevőit függvényobjektumok segítségével idomítjuk saját igényeinkhez:

```
class StrLenLess
   : public binary_function<const char *, const char *, bool> {
 public:
   bool operator() ( const char *a, const char *b ) const
        { return strlen(a) < strlen(b); }
};
```

```
// . . .
sort( start, end, StrLenLess() );
```

Az `StrLenLess()` kifejezés láttán a fordító létrehoz egy névtelen ideiglenes objektumot, amelynek érvényessége a `sort` algoritmus befejeződéséig tart. Egy közvetlenül megadott ideiglenes változó használata ebben e helyzetben szükségtelenül bonyolítaná a kódot, és akkor is érvényben maradna, ha már nincs is rá szükség (lásd még a 48. hibát):

```
StrLenLess comp;
sort( start, end, comp );
// A comp még mindig elérhető...
```

Egy másik lehetséges gond a régi, a szabvány megjelenése előtt készült fordítók számára írt kódokkal kapcsolatos. A szabvány megjelenése előtt az ideiglenes objektumok élettartamára ötletszerű szabályokat alkalmaztak a fordítók gyártói. Volt olyan fordítóprogram, amely annak a blokknak a végén törölte az ideiglenes változókat, amelyben létrejöttek. Volt aztán olyan, ami a létrehozó kifejezés végén végezte el a törlést, és így tovább. Ha tehát ilyen régi keletű kódot próbálunk meg karbantartani, ügyeljünk a szabvány okozta jelentésmódosulásokra.

44. hiba: Hivatkozások és ideiglenes változók

A hivatkozás (reference, referencia) nem egyéb, mint az általa hivatkozott érték egyfajta álneve (lásd a 7. hibát). A fordítónak jogában áll ezt az álnevet a megfelelő értékkel helyettesíteni, ez semmiféle jelentésmódosulással nem járhat. Nos... a legtöbbször nem.

```
int a = 12;
int &r = a;
++a; // Ugyanaz, mint a ++r
int *ip = &r; // Ugyanaz, mint az &a
int &r2 = 12; // Hiba! A „12" literális!
```

Hivatkozást csak balértékre adhatunk meg, vagyis olyan objektumra, amelynek van címe is és értéke is (lásd a 6. hibát).A dolog kissé bonyolultabb, ha a hivatkozás állandóra (`const`) vonatkozik. A hivatkozott értéknek most is muszáj balértéknek lennie, így a fordító kénytelen az állandóból balértéket előállítani:

```
const int &r3 = 12; // Rendben.
```

Ebben a példában tehát az `r3` hivatkozás egy, a fordító által létrehozott `int` típusú ideiglenes változóra mutat. Normális esetben a fordító által létrehozott ideiglenes objektumok élettartama az őket használó legtágabb kifejezés élettartamára korlátozódik. Ugyanakkor a szabvány ebben az esetben azt garantálja, hogy az említett ideiglenes változó egészen addig érvényben marad, amíg az általa beállított hivatkozás létezik. Az ideiglenes változónak mindezek ellenére semmiféle kapcsolata nincs az őt beállító értékkel, így az alább látható meglehetősen idétlen kód nem képes elrontani a literális `12` értéket:

```
const_cast<int &>(r3) = 11; // hozzárendelés az ideiglenes
➥ változóhoz vagy kilépés...
```

A fordító az olyan ideiglenes változó létrehozását sem tagadja meg tőlünk, amelynél a szükséges balérték típusa nem egyezik meg az azt címző hivatkozás típusával:

```
const string &name = "Fred"; // Rendben.
short s = 123;
const int &r4 = s; // Rendben.
```

Itt persze nyelvi nehézségekbe ütközhetünk, mivel a hivatkozás és a hivatkozott érték közti szokványos kapcsolat most kissé ködössé vált. Könnyű ugyanis megfeledkezni arról, hogy a hivatkozás beállítására használt változó valójában ideiglenes változó, és nem az, amelyik a forráskódban szerepel. Ez pedig azt jelenti, hogy várakozásaink ellenére az `s` nevű `short` változón elvégzett bármilyen módosítás az `r4` hivatkozás által visszaadott értékben nem fog jelentkezni:

```
s = 321; // r4 még mindig 123
const int *ip = &r4; // nem &s
```

Baj ez? Nos, azzá válhat, ha még egy kicsit segítünk neki. Vegyük például azt az esetet, amikor a hordozhatóság érdekében `typedef`-ekkel próbálkozunk. Például egy globális fejállományban megkísérelhetünk szabványos rendszerfüggetlen neveket bevezetni a különböző méretű egészekre:

```
// fejállomány big/sizes.h
typedef short Int16;
typedef int Int32;
// . . .

// fejállomány small/sizes.h
typedef int Int16;
typedef long Int32;
// . . .
```

(Vegyük észre, hogy most nem használtunk `#if` utasításokat az egyes rendszerekre ahhoz, hogy a különböző típusmeghatározásokat egyetlen fájlba sűríthessük. Ez – mint azt már megtárgyaltuk – meglehetősen veszélyes dolog, ami tönkreteheti hétvégénket, szakmai becsületünket és az életünket. Lásd még a 27. hibát.) Ezzel semmi gond nincs, amíg az összes fejlesztő következetesen használja a neveket. Sajnos azonban nem mindig cselekszenek így:

```
#include <sizes.h>
// . . .
Int32 val = 123;
const int &theVal = val;
val = 321;
cout << theVal;
```

Ha történetesen a „nagy" („large") rendszerre fejlesztünk, a `theVal` a `val` egy álneve, ami azt jelenti, hogy a `cout`-ra `321` kerül. Ha aztán később ki akarjuk használni kódunk állítólagos rendszerfüggetlenségét, és le akarjuk fordítani a „kicsi" („small") rendszerre is, akkor a `theVal` már egy ideiglenes változóra mutat, és a kimenetre `123` kerül. Ez a változás persze a legnagyobb csendben zajlik majd, és sokkal kevésbé lesz feltűnő, mint az, ha mondjuk a kimenetben egy teljes sor megváltozik.

Egy másik lehetséges gond az, hogy ha állandóra mutató hivatkozást használunk, azzal belekeveredhetünk egy, az ideiglenes változók élettartamával kapcsolatos problémába. Amint azt az előző részben láttuk, a fordító gondoskodik róla, hogy egy ilyen ideiglenes változó egészen addig hozzáférhető legyen, amíg él a rá vonatkozó hivatkozás. Ez elsőre biztonságos megoldásnak tűnik, de nézzük csak a következő egyszerű függvényt:

```
const string &
select( bool cond, const string &a, const string &b ) {
   if( cond )
        return a;
   else
        return b;
}
// . . .
string first, second;
bool useFirst = false;
// . . .
const string &name = select( useFirst, first, second );
// Rendben.
```

Első látásra a függvény tökéletesnek tűnik. Végül is semmi egyebet nem tesz, mint visszaadja a két paramétere közül az egyiket. A probléma meglepő módon éppen a `return` környékén van. Nézzünk egy másik egyszerű függvényt, ahol a gond ennél kicsit nyilvánvalóbb:

```
const string &crashAndBurn() {
   string temp( "Fred" );
   return temp;
}
// . . .
const string &fred = crashAndBurn();
cout << fred; // Hoppá!
```

Itt egy olyan hivatkozást adunk vissza, ami egy helyi változóra vonatkozik. Visszatéréskor azonban a helyi változó megszűnik létezni, a függvény használója pedig egy immár nem létező objektumhoz férhet hozzá. Szerencsére a legtöbb fordító felfedezi az ilyen helyzeteket és figyelmeztet. Ugyanakkor a következő hibát már egyik fordító sem fogja észrevenni, mert nem is teheti:

```
const string &name = select( useFirst, "Joe", "Besser" );
cout << name; // Hoppá!
```

Itt az a gond, hogy a `select` függvény második és harmadik paramétere állandóra való hivatkozás, így ezek ideiglenes karakterlánc objektumokra fognak mutatni. És bár ezek az ideiglenes változók nem helyiek a `select` függvényre nézve, csak addig lesznek hozzáférhetők, amíg az őket befoglaló legtágabb kifejezésnek szüksége van rájuk. Ez a „vég" pedig a `select` függvény visszatérése utánra esik, de megelőzi a visszatérési érték felhasználását. Az áthidaló megoldás az lehetne, hogy a függvényhívást egy nagyobb kifejezésbe ágyazzuk be:

```
cout << select( useFirst, "Joe", "Besser" ); // Működik, de
➥ érzékeny a hibákra.
```

Nos, ez az a bizonyos kódtípus, ami működik, ha egy szakértő megírja, és ami rögtön elromlik, ha egy kevésbé szakértő próbálja meg karbantartani.

A legbiztosabb, ha megpróbáljuk teljesen elkerülni az állandóra mutató hivatkozások visszaadását. A mi `select` függvényünk esetében két értelmes lehetőség is kínálkozik. A szabványos `string` típus nem többalakú, (vagyis nincsenek virtuális függvényei), így nyugodtan élhetünk azzal a feltételezéssel, hogy az ilyen típusú hivatkozások a `string`-hez mint típushoz kötődnek, és nem a belőle származtatott objektumokhoz. Ez pedig azt jelenti, hogy a „kivágási hatás" veszélye nélkül hasz-

nálhatunk érték szerinti átadást. Csupán a teljesítményigény némi növekedésével kell számolnunk, hiszen a visszatérési érték beállításához meg kell hívni a `string` másoló konstruktorát:

```
string
select( bool cond, const string &a, const string &b ) {
    if( cond )
        return a;
    else
        return b;
}
```

Egy másik lehetséges megoldás, ha a formális paramétereket nem állandóra mutató hivatkozásként adjuk meg. Ez teljes bizonyossággal fordítási hibát vált ki, ha véletlenül ideiglenes változót akarunk a beállításukhoz használni. Ez a megoldás nem javítja ki a fenti hibát, de azonnal tudjuk, hogy valami nincs rendben vele:

```
string &
select( bool cond, string &a, string &b ) {
    if( cond )
        return a;
    else
        return b;
}
```

Tulajdonképpen egyik említett megoldás sem különösebben vonzó, de bármelyik jobb, mint benne hagyni a kódban egy nehezen felderíthető hibát.

45. hiba: A dynamic_cast művelet használata körüli bizonytalanságok

Szinte biztos, hogy bűntudatunk lesz miatta. Valószínűleg soha nem fogjuk megosztani a kollégákkal. Talán még az emberi kapcsolatainkra is rátelepszik. De ha falhoz szorítottak bennünket, ha kaptunk egy eszetlenül megtervezett programmodult javításra, ha a főnök szorongat, ha tegnapra kellene készen lennünk, akkor bizony ideje használni a `dynamic_cast` műveletet.

Legyen a feladat mondjuk az, hogy el kell döntenünk, hogy egy adott képernyőobjektum adatbevitelre szolgál, vagy valami egészen más van rajta. A gond az, hogy éppen egy általános célú program kellős közepén vagyunk, és a képernyőt úgy általában kellene kezelnünk. Első ötletünk valószínűleg az lesz, hogy valamennyi képernyőtípus felületét kiegészítjük olyan szolgáltatásokkal, amelyek rendelkezésünkre bocsátják a szükséges információt:

```
class Screen {
 public:
   // . . .
   virtual bool isEntryScreen() const
       { return false; }
};
class EntryScreen : public Screen {
 public:
   bool isEntryScreen() const
       { return true; }
};
//  . . .
Screen *getCurrent();
//  . . .
if( getCurrent()->isEntryScreen() )
   //  . . .
```

Ezzel a megközelítéssel csak az a gond, hogy legitimáljuk vele a felhasználó által a képernyőobjektumoknak feltett kíváncsiskodó kérdéseket. A `Screen` alaposztály ilyen kialakítás mellett szinte ingerli a fejlesztőt a „Te beviteli képernyő (`EntryScreen`) vagy?" jellegű kérdések feltételére. Ezzel aztán megnyitottuk a kaput, mert a későbbi fejlesztők még ennél is illetlenebb kérdések megválaszolására kényszerítik majd szegény objektumot (lásd a 98. hibát):

```
class Screen {
 public:
   // . . .
   virtual bool isEntryScreen() const
       { return false; }
   virtual bool isPricingScreen() const
       { return false; }
   virtual bool isSwapScreen() const
       { return false; }
   // ad infinitum...
};
```

Egy ilyen felület megléte természetesen garantálja, hogy a fejlesztők élni is fognak vele:

```
// . . .
if( getCurrent()->isEntryScreen() )
   // . . .
else if( getCurrent()->isPricingScreen() )
   // . . .
else if( getCurrent()->isSwapScreen() )
   // . . .
```

Ez itt olyan, mint egy `switch` szerkezet, csak sokkal lassabb, és nehezebben karbantartható. Kevésbé ronda megoldás, ha egyszerűen csak leütjük a feldobott labdát, és a `dynamic_cast` művelethez nyúlunk. Ilyenkor a legtöbb programozó azt reméli, hogy egy ilyen apró művelet észrevétlenül megbújik a kód sűrűjében, senkit nem ösztönöz arra, hogy kövesse a példánkat, mi meg majd egyszer, ha lesz elég időnk rendesen átírni a kódot, gyorsan eltávolítjuk:

```
if( EntryScreen *es = dynamic_cast<EntryScreen *>(sp) ) {
   // Műveleteket végzünk a beviteli képernyővel...
}
```

Ha a típusátalakítás sikeres, az `es` egy `EntryScreen` objektumra fog mutatni, ami lehet ennek az objektumnak a tényleges típusa, de az is előfordulhat, hogy egy kifinomultabb képernyőobjektum `EntryScreen` alobjektumát címezzük meg vele. Na de mi itt a gond?

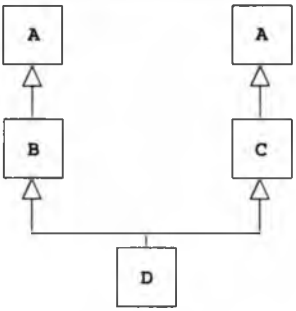
A `dynamic_cast` művelet eredménye négy különböző ok bármelyike miatt lehet nulla. Először is lehet, hogy maga a típusátalakítás érvénytelen. Ha az `sp` nem `EntryScreen` objektumra vagy abból származtatott másik objektumra mutat, a típusátalakítás nem végezhető el. Másodszor, ha az `sp` tartalma történetesen maga is null, akkor természetesen a típusátalakítás után is ezt kapjuk eredményül. Harmadszor, a típusátalakítás nem lehetséges, ha kiindulási vagy céltípusa egy éppen nem hozzáférhető alaposztály. Negyedszer, a típusátalakítást a nem egyértelmű típusviszonyok is meggátolhatják.

A típusátalakítással kapcsolatos efféle gondok egy jól megtervezett hierarchiában ritkák, gyenge tervezés esetén azonban bármikor bajba kerülhetünk általuk, ha a hierarchia egyes elemeinek megvalósítása vagy hozzáférhetősége nem megfelelő.

A 4.4. ábra egy egyszerű, többszörös öröklésen alapuló hierarchiát mutat. Tételezzük fel, hogy `A` többalakú (van virtuális függvénye), valamint hogy kizárólag nyil-

vános öröklést alkalmazunk. Ebben az esetben egy D objektumnak két A típusú alobjektuma is van, ami azt jelenti, hogy legalább az egyik közülük nem virtuális alaposztály:

```
D *dp = new D;
A *ap = dp; // Hiba! Kétértelmű!
ap = dynamic_cast<A *>(dp); // Hiba! Kétértelmű!
```



4.4. ábra

Virtuális alaposztályokat nem tartalmazó többszörös öröklésen alapuló hierarchia. A D teljes objektum két A típusú alobjektumot tartalmaz.

Itt az ap mutató beállítása kétséges, mivel két szóba jöhető A objektum is van, amelyiknek a címe belekerülhet. Ugyanakkor, ha már rendelkezünk az egyik A alobjektum címével, a hierarchia összes többi eleméhez való hozzáférés egyértelműen megoldható:

```
B *bp = dynamic_cast<B *>(ap); // Működik.
C *cp = dynamic_cast<C *>(ap); // Működik.
```

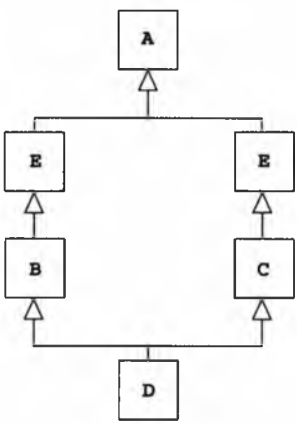
Teljesen mindegy, hogy az ap a kettő közül melyik A alobjektum címét tartalmazza, ha átállítjuk úgy, hogy a B, a C, vagy a teljes D objektumra mutasson, a művelet végrehajtása egyértelmű, hiszen a teljes objektum mindegyik alobjektumnak pontosan egy példányát tartalmazza:

Ha mindkét A alobjektum virtuális alaposztály lenne, akkor a rendszerben egyáltalán nem lenne semmiféle félreérthető momentum, mivel a D objektum ebben az esetben csak egy A alobjektumot foglalna magában:

```
D *dp = new D;
A *ap = dp; // Rendben, egyértelmű
ap = dynamic_cast<A *>(dp); // Rendben, egyértelmű
```

A félreértelmezhetőség visszatér, ha a 4.5. ábrán bemutatott módon egy kissé tovább bonyolítjuk a hierarchia belső viszonyait. Az előző probléma ebben az esetben nem merül fel, mivel a `D` teljes objektum most csak egy `A` alobjektumot tartalmaz:

```
A *ap = new D; // Nem érthető félre.
```



4.5. ábra

Többszörös öröklési hierarchia azonos típusú alobjektumok virtuális és nem virtuális öröklésével. A D teljes objektum egyetlen A, de két E típusú alobjektumot tartalmaz.

Ugyanakkor van egy új gond, amit a következő példa szemléltet:

```
E *ep = dynamic_cast<E *>(ap); // Hibás!
```

Az `ap` mutatót két különböző `E` alobjektum címével is fel lehet tölteni. Megoldhatjuk persze a helyzetet, ha pontosabban körülírjuk, mit szeretnénk elérni:

```
E *ep = dynamic_cast<B *>(ap); // Működik.
```

A `D` típus csak egy `B` alobjektumot tartalmaz, így egy `A *` mutatónak `B *` típusúvá való átalakítása egyértelműen megoldható, a `B *` ezt követő, saját nyilvános alaptípusává való átalakítása pedig nem kíván típusátalakítást. Ugyanakkor ez a megoldás nyilvánvalóan feltételezi a programozóról, hogy ismeri a hierarchia `A` és `E` alatti viszonyait is. Ezért a „dinamikus félreérthetőség" elkerülése végett talán egyszerűbb magának a hierarchiának az egyszerűsítése.

Mivel éppen a `dynamic_cast` művelet használatát tárgyaljuk, érdemes rámutatni a vele kapcsolatos jelentésbeli problémákra is. Mindenekelőtt, a `dynamic_cast` neve ellenére nem feltétlenül dinamikus, mivel nem mindig biztosít futásidejű ellenőrzést. Ha például egy származtatott osztály mutatóját (vagy egy ilyen hivatkozást) alakítjuk át a `dynamic_cast` segítségével annak nyilvános alaptípusává, nincs szükség futásidejű ellenőrzésre, hiszen a fordító – ismerve az osztályviszonyokat – statikusan képes megállapítani, hogy a típusátalakítás sikeres lesz-e. Ebben az esetben természetesen semmiféle átalakításra nincs szükség, hiszen egy származtatott osztálynak saját nyilvános alaposztályává való átalakítása eleve adott. (Az efféle nyelvi szabályok elsőre értelmetlennek tűnhetnek, de nagyban megkönnyítik a sablonok segítségével történő programozást azokban az esetekben, amikor a kezelendő típusokat nem lehet előre meghatározni.)

Arra is lehetőségünk van, hogy egy többalakú (polimorf) típusra vonatkozó mutatót vagy hivatkozást `void *` típusúvá alakítsunk. Ebben az esetben az eredményként kapott mutató a „leginkább származtatott" vagy legteljesebb objektum kezdetére mutat. Természetesen azt még így sem tudjuk megmondani, pontosan mit is címeztünk meg, de azt legalább tudjuk, hogy hol van az a valami...

46. hiba: A kontravariancia félreértéséből eredő hibák

A tagmutatók (pointer to member) átalakítására vonatkozó szabályok logikusak ugyan, mégis gyakran félrevezetnek. De ha közelebbről megvizsgáljuk e mutatók megvalósítását, az gyakran segít a dolgokat tisztázni.

A mutató egy memóriaterületet címez, vagyis tartalmaz egy címet, amelyen keresztül az adott területet elérhetjük. (Az alább bemutatott kódrészlet két kifogásolható megoldást is tartalmaz: nyilvános adatokat használ, és elrejti egy alaposztály egy nem virtuális függvényét. Ezeket csak a példa kedvéért mutatom be, nem azért, mert követendő példaként akarom őket az Olvasó elé állítani. Ezzel kapcsolatban amúgy olvassuk el a 8. és a 71. hibát.)

```
class Employee {
 public:
   double level_;
   virtual void fire() = 0;
   bool validate() const;
};
class Hourly : public Employee {
 public:
```

```
    double rate_;
    void fire();
    bool validate() const;
};
// . . .
Hourly *hp = new Hourly, h;
// . . .
*hp = h;
```

Jegyezzük meg, hogy egy adott objektum egy adattagjának címe nem tagmutató. Ez csupán egy egyszerű mutató, amely egy bizonyos objektum egy bizonyos tagjára mutat:

```
double *ratep = &hp->rate_;
```

A tagmutató valójában nem mutató. Nem tartalmazza semminek a címét, és nem is mutat egy bizonyos objektum egy bizonyos pontjára, vagy bármilyen konkrét helyre. A tagmutató egy közelebbről meg nem határozott objektum egy bizonyos tagjára mutat. Ennek megfelelően ahhoz, hogy ezt a mutatót egyáltalán használhassuk valamire, meg kell adnunk egy konkrét objektumot is:

```
double Hourly::*hvalue = &Hourly::rate_;
hp->*hvalue = 1.85;
h.*hvalue = hp->*hvalue;
```

A `.*` és a `->*` műveletek olyan kétényezős hivatkozó műveletek, amelyek egy objektum vagy objektummutató segítségével követnek vissza egy tagmutatót. (Ezzel kapcsolatban lásd még a 15. és 17. hibát.) Példánkban a `hvalue` egy olyan tagmutató, amely bármilyen, az `Hourly` osztályba tartozó objektum `rate_` nevű elemére mutat. Ezt a tagmutatót először a `h Hourly` típusú objektumpéldánnyal, majd az ugyanilyen típusú objektumot címző `hp` objektummutatóval együtt használtuk a nevezett taghoz való hozzáférésre.

Az adattagmutatókat a fordító általában eltolási címként valósítja meg. Ez azt jelenti, hogy ha vesszük az adott tag címét, – példánkban ezt az `&Hourly::rate_` sorral tettük – akkor azt kapjuk meg, hogy az adott tag az adott osztály kezdetétől hány bájtnyi távolságra található. Ezt az eltolási értéket a fordító általában eggyel növeli, mert így a `null` érték is használható. Egy ilyen tagmutató használata során a fordító előbb eggyel csökkenti a benne tárolt értéket, majd hozzáadja azt a megadott objektum kezdőcíméhez. Az így keletkező valódi mutatón keresztül aztán már hozzáférhetünk az adattag tartalmához. Ezt a kifejezést például

```
h.*hvalue = 1.85
```

ezek alapján a következővel helyettesíthetnénk:

```
*(double *)((char *)&h+(hvalue-1)) = 1.85
```

Nézzünk egy másik adattagot címző tagmutatót:

```
double Employee::*evalue = &Employee::level_;
Employee *ep = hp;
```

Mivel az `Hourly` típusba tartozó objektum egyben `Employee` is, az `evalue` mutatónak bármelyik típust megadhatjuk használat közben. Ilyenkor a jól ismert rejtett típusátalakítás megy végbe a származtatott típus és annak nyilvános alaposztálya között:

```
ep->*evalue = hp->*evalue;
```

Bár az `Hourly` típust ebben az értelemben használhatjuk, mint egy `Employee` helyettesítőjét, ha ugyanezt a megfelelő tagmutatókkal próbáljuk meg, az már hiba:

```
evalue = hvalue; // Hiba!
```

Nincs olyan típusátalakítás, amellyel egy származtatott osztály tagmutatóját a megfelelő nyilvános alaposztály tagmutatójává alakíthatnánk. A művelet fordítottja viszont lehetséges:

```
hvalue = evalue; // Rendben.
```

Ez a jelenség a kontravariancia. A tagmutatók átalakítására vonatkozó szabályok pontosan az osztálymutatók átalakítására vonatkozó előírások ellentétei. (Mindazonáltal ne keverjük össze a kontravarianciát a kovariáns visszatérési típusokkal. Ezzel kapcsolatban olvassuk el a 77. hibát.) Némi elmélkedés után világossá válik ennek az elsőre nem túl értelmes szabályrendszernek a logikája. Mivel egy `Hourly` objektum egyben `Employee` is, tartalmaznia kell egy `Employee` típusú alobjektumot. Ennek megfelelően bármilyen, az `Employee` típuson belül érvényes eltolási cím értelmezhető az `Hourly` típuson belül is. Ugyanakkor az `Hourly` típuson belül vannak olyan eltolási címek, amelyek az `Employee` típuson belül nem értelmesek. Mindebből már egyenesen következik, hogy egy nyilvános alaposztály tagmutatói minden további nélkül átalakíthatók egy származtatott osztály tagmutatóivá, de fordítva a művelet már nem biztonságos:

```
T SomeClass::*mptr;
. . . ptr->*mptr  . . .
```

A fenti kódrészletben a `ptr` mutató érvényes tagmutatója lehet bármely `SomeClass` objektumnak vagy bármely ebből az osztályból származtatott típusnak. Az `mptr` nevű elemet címző tagmutató tartalmazhatja a `SomeClass` típus megfelelő elemének eltolási címét, de tárolhatja `SomeClass` bármely hozzáférhető, nyilvános alaposztályának hasonló adatát is.

A kontravariancia a függvénytagokat címző tagmutatókra is érvényesül. Ez elsőre ugyanolyan érthetetlennek tűnik, de némi töprengés árán megint minden világossá válik:

```
void (Employee::*action1)() = &Employee::fire;
(hp->*action1)(); // Hourly::fire
bool (Employee::*action2)() const = &Employee::validate;
(hp->*action2)(); // Employee::validate
```

A függvénytagokat címző tagmutatók megvalósítása rendszerként más és más, de általában kis szerkezetek. Ez a szerkezet tartalmazza azt az információt, amely alapján megkülönböztethetők a virtuális és nem virtuális tagfüggvények, valamint mindazt a rendszerfüggő információt, ami ahhoz szükséges, hogy az öröklési rendszer működhessen. A fenti példában az első, az `action1`-en keresztül megvalósított függvényhívás során egy közvetett virtuális hívást hajtunk végre, amelynek célja az `Hourly::fire` függvény. Az `&Employee::fire` ugyanis egy virtuális tagfüggvényt címző tagmutató. Az `action2`-n keresztül megvalósított második hívásnál az `Employee::validate` függvény fut le, ugyanis az `&Employee::validate` egy nem virtuális tagfüggvény tagmutatója:

```
action2 = &Hourly::validate; // Hiba!
bool (Hourly::*action3)() = &Employee::validate; // Rendben.
```

Ismét itt a kontravariancia. Nem rendelhetjük hozzá egy származtatott osztály `validate` függvényének címét a megfelelő alaposztály tagmutatójához, de minden további nélkül beállíthatjuk egy származtatott osztály függvényt címző tagmutatóját az alaposztály tagfüggvényének címével. Ugyanúgy, ahogy az adattagoknál, a megszorítás oka itt is a tagokhoz való hozzáférés biztonságának megőrzése. A `Hourly::validate` függvény esetleg megpróbálhat hozzáférni olyan függvényekhez vagy adattagokhoz, amelyek az `Employee` típusban nem is szerepelnek. Ugyanakkor minden, az `Employee:validate` által hozzáférhető elem biztosan megtalálható az `Hourly` típusban.

5

Kezdeti beállítás

A C++ objektumoknál a kezdeti beállítás („inicializálás") jelentése szövevényes és összetett, de a téma nagyon fontos.

Az összetettség természetesen nem öncélú. A C++-ban a programozás javarészt az elvont adattípusok osztályok segítségével történő megtervezését jelenti. Ezzel tulajdonképpen a nyelvbe eleve beépített típusrendszert egészítjük ki saját igényeinknek megfelelően. Ez a tevékenység aztán arra késztet bennünket, hogy egyszerre két dologra összpontosítsunk. Egyrészt figyelembe kell vennünk a nyelv lehetőségeit, ha használható típusokat akarunk létrehozni, másrészt meg kell győznünk a fordítóprogramot, hogy amit írtunk, az értelmes és hatékonyan lefordítható. A kezdeti beállítások és a másoló műveletek megvalósítása elsődleges fontossággal bír az objektumok hatékony használatának szempontjából.

A hatékonysághoz persze épp ennyire fontos a kód helyessége is. Ha nem vagyunk kellőképpen tisztában a nyelv szükségszerűen bonyolult jelentésrétegével („szemantika"), akkor tévesen fogjuk használni a nyelv szolgáltatásait is.

Ebben a fejezetben a kezdeti beállítások megvalósításával kapcsolatos problémákat tárgyaljuk, bemutatjuk, miként vehetjük rá a fordítót a felhasználó által meghatározott beállító és másoló műveletek hatékonyságának növelésére, valamint tanulmányozni fogunk néhány, a beállító műveletek értelmezésével kapcsolatos általános félreértést is.

47. hiba: Az értékadás és a kezdeti beállítás összekeverése

Az egyszerű értékadás (assignment, hozzárendelés) és a kezdeti beállítás (kezdőérték-adás) két teljesen különböző dolog. Ezek különböző műveletek, amelyeket eltérő helyzetekben használhatunk. A beállítás az a művelet, amikor egy „nyers" tárterületet objektummá alakítunk. Egy osztályobjektum esetében ez a virtuális függvények és virtuális alaposztályok, a futásidejű típusinformáció, valamint egyéb, szintén a típusokkal kapcsolatos információk kezeléséhez szükséges belső eljárások beállítását jelenti. (Ezzel kapcsolatban olvassuk el az 53. és a 78. hibát is.) Az értékadás során ezzel szemben egy már eleve létező és jól meghatározott objektum állapotát állítjuk át új értékek megadásával. Az értékadás nem módosítja az objektum belső műveleti rendszerét, viselkedését. Mindebből az is nyilvánvaló, hogy az értékadás célja soha nem lehet nyers tárterület.

Kicsit árnyaltabban fogalmazva azt mondhatjuk, hogy ha az egyik területen a másoló létrehozást tekintjük lényegesnek, a másikon ugyanezt állíthatjuk a másolva értékadásról, és megfordítva. Ha az értékadás és a kezdeti beállítás közül bármelyiket elhanyagoljuk, az óhatatlanul programozási hibákhoz vezet:

```
class SloppyCopy {
 public:
   SloppyCopy &operator =( const SloppyCopy & );
   // Megjegyzés: a fordító alapértelmezése a SloppyCopy(const
➥ SloppyCopy &)...
 private:
   T *ptr;
};

void f( SloppyCopy ); // Érték szerinti átadás

SloppyCopy sc;
f( sc ); // Álnév arra, amire a ptr mutat; valószínűleg hiba
```

A paraméterek átadása kezdeti beállításon és nem értékadáson keresztül valósul meg. Példánkban az `f` függvény formális paraméterét az `sc` tényleges paraméterrel állítjuk be. Ezt a beállítást a `SloppyCopy` objektum másoló konstruktora fogja végrehajtani. Ha nem adunk meg kifejezetten ilyen másoló műveletet, akkor azt maga a fordítóprogram „írja meg". Esetünkben azonban ez a fordító által előállított művelet helytelen lesz. (Lásd még a 49. és 53. hibát.)

A hiba mögött megbúvó feltételezés az, hogy bár a másoló konstruktor nem azonos a másoló értékadással, elvileg hasonló, vagy legalábbis alakilag megfelelő jelentéssel kellene bírniuk:

```
extern T a, b;
b = a;
T c( a );
```

A fenti kódot látva a `T` típus minden felhasználója azt várná, hogy a `b` és a `c` objektum alakilag egymásnak megfelelő, vagyis a további végrehajtás szempontjából elvileg lényegtelennek kellene lennie annak, hogy jelenlegi tartalmukat értékadással vagy kezdeti beállítással szerezték. Az alakhűség e feltételezése annyira beleivódott a C++ közösség gondolkodásmódjába, hogy még a szabványos könyvtár is támaszkodik rá:

➡ 47. hiba/rawstorage.h

```
template <class Out, class T>
class raw_storage_iterator
```

```
    : public iterator<output_iterator_tag,void,void,void,void> {
  public:
    raw_storage_iterator& operator =( const T& element );
    // . . .
  protected:
    Out cur_;
};
template <class Out, class T>
raw_storage_iterator<Out, T> &
raw_storage_iterator<Out,T>::operator =( const T &val ) {
    T *elem = &*cur_; // Mutató előállítása az elemhez
    new ( elem ) T(val); // Elhelyezés és másoló konstruktor
    return *this;
}
```

A „szűz tárterülethez" való hozzárendelésre a `raw_storage_iterator` típust használhatjuk. Normális esetben egy értékadó/hozzárendelő művelet megköveteli, hogy mindkét oldalán szabályosan beállított objektumok legyenek. Ellenkező esetben ugyanis gondok merülhetnek fel, amikor a művelet a bal oldalán álló tényezőt „kitakarítja" az értékadás előtt. Ha egy felülírandó objektum például tartalmaz egy dinamikusan lefoglalt tárterületet címző mutatót, akkor az új érték beállítása előtt a program rendszerint törli ezt a memóriaterületet. Ha az objektum azonban nem tartalmaz érvényes beállításokat („nincs inicializálva"), akkor ennek a törlésnek beláthatatlan következményei lehetnek:

➡ 47. hiba/rawstorage.cpp

```
struct X {
    T *t_;
    X &operator =( const X &rhs ) {
        if( this != &rhs )
            { delete t_; t_ = new T(*rhs.t_); }
        return *this;
    }
    // . . .
};
// . . .
X x;
X *buf = (X *)malloc( sizeof(X) ); // Nyers tárterület
X &rx = *buf; // Piszkos trükk...
rx = x; // Valószínűleg hibás
```

A szabványos könyvtár `copy` algoritmusa egy bemenő adatsort alakít egy kimenő adatsorrá, mégpedig úgy, hogy az értékadó művelet segítségével elemenként végzi el a másolást:

```
template <class In, class Out>
Out std::copy( In b, In e, Out r ) {
```

```
    while( b != e )
        *r++ = *b++; // A forrás elemét a cél megfelelő eleméhez
➥ rendeli
    return r;
}
```

Ha a `copy` függvényt `X` objektumok beállítatlan tömbjével próbáljuk használni, az valószínűleg hibát okoz:

➥ 47. hiba/rawstorage.cpp

```
X a[N];
X *ary = (X *)malloc( N*sizeof(X) );
copy( a, a+N, ary ); // Nyers tárterülethez rendel hozzá
```

Az értékadás tehát csaknem (de nem pontosan!) olyan, mintha egy törlő műveletet egy létrehozó követne. A `raw_storage_iterator` ezzel szemben lehetővé teszi, hogy nem beállított (előkészítetlen) területre másoljunk, mégpedig úgy, hogy a művelet átértelmezéséből kihagyja a problémás törlő lépést. Ez persze csak azzal a feltételezéssel működik, hogy a másoló értékadás és a másolva létrehozás elfogadható mértékig hasonló eredménnyel jár.

➥ 47. hiba/rawstorage.cpp

```
raw_storage_iterator<X *, X> ri( ary );
copy( a, a+N, ri ); // Működik!
```

Nem tételezhetjük fel persze minden esetben, hogy a példánkban szereplő `X` osztályt létrehozó programozó teljes mértékig tisztában van a szabványos könyvtár belső működésével. Ugyanakkor azzal tisztában kell lennie, hogy a másoló értékadás és a másolva létrehozás csak akkor feleltethető meg egymásnak, ha maga a felhasználó által megadott elvont adattípus támogatja ezt. Az olyan adattípusok, amelyek nem képesek megfelelni ennek az elvárásnak, nem kezelhetők hatékonyan a szabványos könyvtárral, s így kevésbé használhatók lesznek, mint egy megfelelő típus.

Egy másik gyakori félreértés annak feltételezése, hogy az itt látható kezdeti beállításnak valamilyen módon része egy értékadás:

```
T d = a; // Ez nem értékadás.
```

Az itt látható = jel nem az értékadás jele, a `d` objektumnak itt valójában kezdeti beállítást adunk `a`-val. Ez amúgy szerencse is, hiszen ellenkező esetben az értékadás egyik oldalán előkészítetlen tárterületre hivatkoznánk. (De ezzel kapcsolatban olvassuk el az 56. hibát!)

48. hiba: A változók hatókörének helytelen megválasztása

A C és a C++ nyelvben egyaránt sok programozási hiba vezethető vissza a nem megfelelően beállított változók használatára. Ez pedig éppen olyan probléma, aminek nem szabadna léteznie. Egy változó bevezetésének (deklarációjának) és kezdeti beállításának elválasztása általában semmiféle előnnyel nem jár:

```
int a;
a = 12;
string s;
s = "Joe";
```

Ez itt egyszerűen butaság. Az egész változónak határozatlan lesz az értéke egészen a következő értékadó utasításig. A karakterláncot ezzel szemben az alapértelmezett konstruktora megfelelően be fogja állítani, de ezt a beállítást rögtön a következő sorban felülírjuk egy értékadással. (Ezzel kapcsolatban olvassuk el az 51. hibát is.) Mindkét esetben jobb lett volna egyértelmű kezdeti beállítást használni a deklaráció részeként:

```
int a = 12;
string s( "Joe" );
```

A deklaráció és a kezdő értékadás szétválasztásának igazi veszélye az, hogy a későbbi karbantartás során valamilyen új kód befurakodhat közéjük. A jellemző helyzet általában valamivel körmönfontabb, mint az előző kódrészlet:

```
bool f( const char *s ) {
   size_t length;
   if( !s ) return false;
   length = strlen( s );
   char *buffer = (char *)malloc( length+1 );
   // . . .
}
```

A `length` változó itt nemcsak hogy nincs beállítva, állandónak (`const`) is kellene lennie. E kód szerzője megfeledkezett róla, hogy a C-vel ellentétben a C++-ban a deklaráció utasítás, egészen pontosan fogalmazva egy bevezető (deklaráló) utasítás, amely ennek megfelelően bárhol előfordulhat, ahol utasítás állhat:

```
bool f( const char *s ) {
```

```
    if( !s ) return false;
    const size_t length = strlen( s );
    char *buffer = (char *)malloc( length+1 );
    // . . .
}
```

Nézzünk most egy másik, a karbantartással kapcsolatban általánosnak tekinthető problémát. Ebben a kódrészletben semmi különleges nincs:

```
void process( const char *id ) {
    Name *function = lookupFunction( id );
    if( function ) {
        // . . .
    }
}
```

A `function` deklarációja most még rendben van, de a karbantartás során hibák forrása lehet. Amint azt korábban már említettem, a karbantartást végző programozók gyakran eredeti rendeltetésüktől eltérő dolgokra használják fel a helyi változókat. Hogy miért? Mert ott vannak.

```
void process( const char *id ) {
    Name *function = lookupFunction( id );
    if( function ) {
        // A function paraméter feldolgozása
    }
    else if( function = lookupArgument( id ) ) {
        // A paraméter feldolgozása
    }
}
```

Egyelőre még nincs hiba, de attól tartok, a paraméter feldolgozási módjának megadása kissé bonyolult a felkészületlen olvasó számára. („A kód e részében valahányszor azt mondom, hogy *függvény*, azt kell érteni, hogy *paraméter*.") De mi történik, amikor programozónk később visszatér, hogy karbantartsa a kódot, és kissé módosítja a függvény kezelésének módját?

```
void process( const char *id ) {
    Name *function = lookupFunction( id );
    if( function ) {
        // A function paraméter feldolgozása
    }
    else if( function = lookupArgument( id ) ) {
        // A paraméter feldolgozása
    }
```

```
    // . . .
    if( function ) {
        // A function paraméter utófeldolgozása
    }
}
```

Most már tényleg egy paramétert akarunk az utófeldolgozás keretében függvényként kezelni.

Általában célszerű egy változó hatókörét arra a kódrészletre korlátozni, ahol a szerző azt az eredeti célra használja. Az érvényes, de már használaton kívüli változók olyanok, mint az unatkozó tizenévesek: csak ténferegnek napestig és várják, mikor kerülhetnek bele valami balhéba. Példánkban megtehettük volna, hogy a `function` változó hatókörét az őt kezelő függvényre korlátozzuk:

```
void process( const char *id ) {
    if( Name *function = lookupFunction( id ) ) {
        // . . .
    }
}
```

A változók hatókörét korlátozva egyrészt megszüntetjük azt a kísértést, hogy azokat a később jövő programozók újra felhasználják, másrész a példánkban szereplő `function` változó kezelése is sokkal ésszerűbb lesz:

```
void process( const char *id ) {
    if( Name *function = lookupFunction( id ) ) {
        // . . .
        postprocess( function );
    }
    else if( Name *argument = lookupArgument( id ) ) {
        // . . .
    }
}
```

A változók kezdeti beállításának, valamint helyesen beállított hatókörének fontosságát a C++ tervezői is átlátták, így a nyelv számos szolgáltatást nyújt ehhez.

49. hiba: A C++ másoló műveletekkel kapcsolatos megkötéseinek figyelmen kívül hagyása

A C++ igen komolyan veszi a másoló műveleteket. Ezek a nyelv igen fontos területét képviselik, különösen ami az osztályobjektumok másolását illeti. E műveletek fontosságát az is jelzi, hogy ha kifejezetten nem adjuk meg őket, maga a fordító teszi meg ezt helyettünk. Sőt, ha meg is adunk ilyen műveleteket, a fordító néha akkor is félrelök bennünket, és maga is felkínálja őket. A fordító néha helyesen valósítja meg a másoló műveleteket, néha nem, ezért jobb, ha pontosan tudjuk, mit vár tőlünk egy C++ fordító a másoló műveletekkel kapcsolatban.

Jegyezzük meg, hogy a másoló konstruktor és a másoló értékadás (egyéb konstruktorokkal és destruktorokkal egyetemben) nem örökölhető az alaposztályoktól. Ez pedig azt jelenti, hogy minden osztálynak tartalmaznia kell a saját másoló műveleteit.

A másoló konstruktor alapértelmezett megvalósítása az, hogy tagról tagra haladva átmásolja a teljes objektumot az új helyre. A tagról tagra haladó beállításnak persze semmi köze a C stílusú bitenkénti másoló művelethez. Vegyünk egy egyszerű osztályt:

```
template <int maxlen>
class NBString {
 public:
   explicit NBString( const char *name );
   // . . .
 private:
   std::string name_;
   size_t len_;
   char s_[maxlen];
};
```

Feltéve, hogy mi magunk nem írtuk meg a másoló műveleteket, a fordító fogja előállítani azokat. Ezek az automatikusan előállított függvények nyilvános helyben kifejtett tagfüggvények lesznek.

```
NBString<32> a( "String 1" );
// . . .
NBString<32> b( a );
```

Ez a rejtett („beleértett") másoló konstruktor tagonkénti beállítást fog végrehajtani, mégpedig úgy, hogy a `string` másoló konstruktorát meghívva az `a.name` alapján beállítja a `b.name` tagot, az `a.len_` alapján a `b.len_` elemet, valamint az `a.s_` elemei alapján a `b.s_` megfelelő elemeit. (Ami azt illeti, a nyelvi szabvány valószínűleg egy hirtelen jött furcsa ötletroham hatására úgy rendelkezik, hogy az olyan „skalár" típusokat, mint amilyenek a beépített és a felsoroló típusok, valamint a mutatók, a rejtett másoló konstruktornak értékadással kell beállítania. Bár az az agyműködés, ami e szabály megfogalmazásához vezethetett, túlmutat szerény ismereteimen, bizton állíthatom, hogy – legalábbis ezekre a típusokra – az eredmény ugyanaz lesz, mintha kezdeti beállítást használna a fordító.)

```
b = a;
```

A rejtett értékadó művelet, hasonlóan a konstruktorhoz, tagonkénti értékadást valósít meg. Meghívja a `string` értékadó műveletét, hogy aztán azzal rendelje az `a.name` értékét a neki megfelelő `b.name` taghoz, az `a.len_` tartalmát a `b.len_` elemhez, valamint az `a.s_` minden egyes elemét a `b.s_` megfelelő elemeihez.

A másoló műveletek ezen alapértelmezett meghatározása a másolás szokványos értelmezését követi. (Ezzel kapcsolatban lásd még a 81. hibát.) Ugyanakkor vegyük a névvel ellátott, korlátos karakterláncok osztályának egy kissé eltérő megvalósítását:

```
class NBString {
 public:
   explicit NBString( const char *name, int maxlen = 32 )
       : name_(name), len_(0), maxlen_(maxlen),
         s_(new char[maxlen] )
       { s_[0] = '\0'; }
   ~NBString()
       { delete [] s_; }
   // . . .
 private:
   std::string name_;
   size_t len_;
   size_t maxlen_;
   char *s_;
};
```

Most a konstruktor állítja be a karakterlánc legnagyobb méretét, a karakterek által ténylegesen elfoglalt tárterület pedig már nem tartozik fizikailag az `NBString` objektumhoz. A fordító által előállított másoló műveletek most már hibásak lesznek:

```
NBString c( "String 2" );
NBString d( c );
```

```
NBString e( "String 3" );
e = c;
```

A rejtett másoló konstruktor a c és d objektum s_ tagjában ugyanannak a memóriaterületnek a címét helyezi el, ez a terület így – tévesen – kétszer törlődik, hiszen a törlést a c és a d destruktora (megsemmisítő függvénye) is el fogja végezni. A c-nek e-hez való hozzárendelésével ugyanilyen eredményre jutunk: a két objektum s_ tagja ugyanoda fog mutatni, és mindkét objektum törölni próbálja majd ezt a területet. (A virtuális alaposztályok fordító által előállított másoló műveleteivel kapcsolatban olvassuk el az 53. hibát is.)

A fenti osztály helyes megvalósításának tehát tartalmaznia kell a másoló műveleteket, nem szabad azok megírását teljes egészében a fordítóra hagynia:

```
class NBString {
 public:
   // . . .
   NBString( const NBString & );
   NBString &operator =( const NBString & );
 private:
   std::string name_;
   size_t len_;
   size_t maxlen_;
   char *s_;
};
// . . .
NBString::NBString( const NBString &that )
   : name_(that.name_), len_(that.len_), maxlen_(that.maxlen_),
     s_(strcpy(new char[that.maxlen_], that.s_))
   {}
NBString &NBString::operator =( const NBString &rhs ) {
   if( this != &rhs ) {
        name_ = rhs.name_;
        char *temp = new char[rhs.maxlen_];
        len_ = rhs.len_;
        maxlen_ = rhs.maxlen_;
        delete [] s_;
        s_ = strcpy( temp, rhs.s_ );
   }
   return *this;
}
```

Minden osztályokat tervező programozónak kínosan ügyelnie kell tehát a másoló műveletekre. Ezeket vagy kifejezetten meg kell adni (és persze megfelelően módosítani az osztály karbantartása során), vagy a fordítóra lehet bízni a megírásukat

(ebben az esetben is felül kell vizsgálni ennek a hatását minden egyes karbantartáskor), vagy a következő fordulat segítségével egyszerűen meg is lehet tagadni a másoló műveletek végzését az adott osztállyal kapcsolatban:

```
class NBString {
 public:
   // . . .
 private:
   NBString( const NBString &);
   NBString &operator =( const NBString & );
   // . . .
};
```

Ha a másoló műveleteket privátként vezetjük be, de ténylegesen nem valósítjuk meg, azzal gyakorlatilag kizárjuk azok végrehajtását. A fordító nem fog függvényeket előállítani az ilyen osztályhoz, a kód javarésze pedig hozzá sem fér a kérdéses tagokhoz. Ha pedig az osztály tagjai vagy barátai (friend függvények) hajtanak végre véletlenül egy másoló műveletet, az az összeszerkesztés során kiderül, hiszen a hívott függvény nem létezik.

Gyakorlatilag lehetetlen rábeszélni a fordítót arra, hogy a nekünk tetsző módon valósítsa meg a másoló műveleteket. Az alábbi kód három ilyen erőtlen próbálkozást mutat be:

```
class Derived;
class Base {
 public:
   Derived &operator =( const Derived & );
   virtual Base &operator =( const Base & );
};
class Derived : public Base {
 public:
   using Base::operator =; // Rejtett
   template <class T>
   Derived &operator =( const T & ); // Ez nem másoló értékadás
   Derived &operator =( const Base & ); // Ez nem másoló
➥ értékadás
};
```

Azt már tudjuk, hogy a másoló műveletek nem öröklődnek. Az viszont talán elsőre meglepő, hogy a `using` kulcsszóval a nem virtuális alaposztályból beemelt értékadó művelet jelenléte egyáltalán nem akadályozza meg a fordítót abban, hogy a saját változatát előállítsa. Ez a változat pedig elrejti az általunk beemelt változatot. (Amúgy az, hogy az alaposztály kifejezetten említi a származtatottat, eleve hibás tervezésre utal. Ezzel kapcsolatban olvassuk el a 69. hibát.)

Az sem segít, ha sablonként megadott értékadó tagfüggvényt használunk, a sablontagok ugyanis soha nem használhatók másoló műveletek megvalósítására. (Lásd bővebben a 88. hibánál.) Az alaposztály virtuális értékadó műveletét a származtatott osztály felülbírálja, a felülbíráló származtatott osztály értékadó művelete azonban nem másoló értékadás (lásd a 76. hiba leírását). A C++ nyelv tehát meglehetősen makacs. Vagy mi írjuk meg a másoló műveleteket, vagy a fordító teszi meg helyettünk. A szabályt kijátszani nem lehet.

50. hiba: Osztályobjektumok bitenkénti másolása

Nincs abban semmi rossz, ha a másoló műveletek megvalósítását a fordítóprogramra bízzuk, bár ez inkább csak az egyszerűbb osztályok, pontosabban fogalmazva az egyszerű szerkezettel bíró osztályok esetében célszerű. Ami azt illeti, az egyszerű osztályok esetében már csak a hatékonyság végett is célszerű ezt a feladatot a fordítóra bízni. Vegyünk például egy osztályt, ami csupán egy adatgyűjtemény:

```
struct Record {
    char name[maxname];
    long id;
    size_t seq;
};
```

Kifejezetten hasznos, ha ezen osztály másoló műveleteit a fordítóra hagyjuk. Az ilyen osztályokat – amelyek tulajdonképpen teljes egészében megfelelnek a C stílusú szerkezeteknek (`struct`) – az angol szakirodalomban POD-nek (Plain Old Data) nevezik. Ebben az esetben a szabvány nagyon pontosan megadja az alkalmazható rejtett másoló műveletek leírását, sőt arról is gondoskodik, hogy ezek megfeleljenek a régi C szerkezetek másolásának. Ezek a régi másoló műveletek pedig bitenkénti másolást jelentettek.

Ha egy adott rendszeren létezik egy „másolj n bájtot, de nagyon gyorsan" jellegű processzorutasítás, akkor a fordítónak módjában áll a bitenkénti másoló művelet megvalósítása során ezt alkalmazni. Ez a hatékonyságnövelés egyébként a nem POD jellegű osztályoknál is megfelelőnek bizonyulhat. A 49. hibánál bemutatott `NBString` osztály első, sablonokra épülő változatának másolása például egy ilyen rendszeren megoldható lenne úgy, hogy a `name_` tagnak a `string` osztály másoló műveletével történő átvitele után a fennmaradó elemekre a gyors bitenkénti másolást alkalmazzuk.

Időnként előfordul, hogy bátor programozók úgy döntenek, inkább maguk valósítják meg a bitenkénti másolást, ezzel gyorsítva a kódot. Ez azonban rendszerint hiba, mivel a fordítóprogram sokkal mélyebben átlátja mind az adott rendszer képességeit, mind az adott osztály belső felépítésének sajátosságait. Egy kézzel kivitelezett bitenkénti másolás tehát általában lassabb is, mint a fordító változata, és elhibázni is elég könnyű:

```
class Record {
 public:
   Record( const Record &that )
       { *this = that; }
   Record &operator =( const Record &that )
       { memcpy( this, &that, sizeof(Record) ); return *this; }
   // . . .
 private:
   char name[maxname];
   long id;
   size_t seq;
};
```

Ebben a példában a `Record` típus immár igazi osztállyá nőtte ki magát, ezért gyors és hatékony másoló műveletet írtunk számára, ami szükségtelen volt, mivel a fordító minden további nélkül előállított számunkra egy hatékony és biztosan hibamentes kódot. Az igazi hiba azonban akkor jön csak elő, amikor a `Record` osztály folytatja növekedését:

```
class Record {
 public:
   virtual ~Record();
   Record( const Record &that )
       { *this = that; }
   Record &operator =( const Record &that )
       { memcpy( this, &that, sizeof(Record) ); return *this; }
   // . . .
 private:
   char name[maxname];
   long id;
   size_t seq;
};
```

Ez bizony nem jó. A bitenkénti másolás ennek az objektumszerkezetnek már nem felel meg, mivel a virtuális függvény megjelenésével bekerültek az osztályba az osztályt kezelő szerkezetek is. Ez utóbbi egyébként rendszerint egy mutató a virtuálisfüggvény-táblára (lásd a 78. hibát).

A fordító által előállított másoló műveletek minden esetben gondoskodnak az objektumok belső szerkezetének sértetlenségéről. A másoló konstruktor megfelelően állítja be a mutatókat, a másoló értékadás művelete pedig ügyel rá, hogy nem módosítsa ezt a tartalmat. Az általunk a `memcpy` függvény segítségével megvalósított változat azonban azonnal felülírja a virtuális függvények tábláját címző mutatót, rögtön miután a másoló konstruktor létrehozta azt, majd a felülírást is felülírja a másoló értékadás művelete. Az osztály tartalmának számos más módosítása is okozhat ilyen jellegű hibákat. Ha az osztályt egy virtuális alaposztályból származtatjuk, ha felveszünk egy olyan új adattagot, ami valamilyen nem nyilvánvaló másoló műveletet igényel, vagy ha egy mutatón keresztül csatolunk külső tárterületet az osztályhoz, az mind hasonló eredményre vezethet.

Általában nem jó „kézzel" megvalósítani egy osztály bitenkénti másoló műveleteit, hacsak nincsenek megcáfolhatatlan érveink arra, hogy ez a megoldás egyrészt szükséges, másrészt hatékonyabb lesz, mint a fordító másoló műveletei. Ha mégis ilyen megoldás mellett döntünk, az osztály minden módosításakor kötelező azt átvizsgálni.

Természetesen az sem nevezhető jó ötletnek, ha egy osztály megvalósításán kívül használjuk a bitenkénti másolást osztályokkal kapcsolatban. Bármilyen másoló műveletnek a `memcpy` függvény segítségével való megvalósítása igen merész vállalkozás. Titokban biteket fésülgetni egyszerűen életveszélyes:

```
extern Record *exemplaryRecord;
char buffer[sizeof(Record)];
memcpy( buffer, exemplaryRecord, sizeof(buffer) );
```

Akárki írta ezt a kódot, egyrészt nem lehetett teljesen magánál, másrészt valószínűleg jól el is rejtette azt a forráskódnak egy olyan zugába, ami kellően távol esik a `Record` osztály megvalósításától. Ennek aztán az lesz az eredménye, hogy a `Record` típus bármilyen, a bitenkénti másolást nem támogató módosítása, és az általa okozott probléma rejtve marad egészen addig, amíg futási hibát nem okoz. Ha már mindenképpen olyan kódot kell írnunk, mint ez itt, akkor is úgy kell azt megvalósítani, hogy az érintett osztály saját másoló műveleteit hívjuk meg (ezzel kapcsolatban lásd még a 62. hibát):

```
(void) new (buffer) Record( *exemplaryRecord );
```

51. hiba: A kezdeti beállítás és az értékadás összekeverése a konstruktorokban

Egy konstruktor garantáltan be fog állítani minden olyan osztálytagot, amely ezt igényli. Fontos kihangsúlyozni, hogy nem értéket ad nekik, hanem kezdeti beállítást végez. Az ezt igénylő elemek az állandók, a hivatkozások, a konstruktorral rendelkező osztályok, valamint az alaposztályoknak megfelelő alobjektumok. (Az állandókkal és hivatkozásokkal kapcsolatban olvassuk el a 81. hiba leírását is.)

```
class Thing {
 public:
   Thing( int );
};
class Melange : public Thing {
 public:
   Melange( int );
 private:
   const int aConst;
   const int &aRef;
   Thing aClassObj;
};
// . . .
Melange::Melange( int )
   {} // Hibák!
```

A fordító ennél a kódnál négy hibát is jelezni fog. A `Melange` konstruktorában nem fogja tudni beállítani az alaposztályt, valamit három adattagot. Az ilyen fordítás közben napvilágra kerülő hibák tulajdonképpen nem súlyosak, hiszen még azelőtt kijavíthatók, hogy bárkinek az életét negatívan befolyásolnák:

```
Melange::Melange( int arg )
   : Thing( arg ), aConst( arg ), aRef( aConst ), aClassObj(
➥ arg )
   {}
```

Az már sokkal súlyosabb, ha a programozó megfeledkezik a kezdeti beállításról, de a kód helyes marad:

```
class Person {
 public:
   Person( const string &name );
   // . . .
```

```
  private:
    string name_;
};
// . . .
Person::Person( const string &name )
    { name_ = name; }
```

Ez az egyébként helyes kód körülbelül kétszeresére növeli a `Person` típus konstruktorának futásidejét és még a lefordított kód is nagyobb lesz tőle. A `string` osztálynak van konstruktora, tehát kezdeti beállítást igényel. Természetesen van neki alapértelmezett konstruktora is, ami akkor fut le, ha nincs kifejezett kezdeti beállítás a kódban. A `Person` konstruktora ennek megfelelően kénytelen meghívni a `string` alapértelmezett konstruktorát, hogy aztán rögtön a következő lépésben felül is írja az objektum tartalmát egy értékadással. A helyes megoldásban természetesen elég egyszerűen előkészíteni a `string` objektumot, és kész:

```
Person::Person( const string &name )
    : name_( name )
    {}
```

Általában is célszerűbb a kezdeti beállításokat a tagbeállító lista segítségével megvalósítani, nem pedig a konstruktor törzsében elvégzett értékadásokkal.

Persze ezt a tanácsot sem szabad túlságosan komolyan venni, és egyedül üdvözítő megoldásnak tekinteni. A lényeg a mértéktartás. Vegyük például egy nem szabványos `String` osztály konstruktorát:

```
class String {
  public:
    String( const char *init = "" );
    // . . .
  private:
    char *s_;
};
// . . .
String::String( const char *init )
    : s_(strcpy(new char[strlen(init?init:"")+1],init?init:"") )
    {}
```

Ez már egy fokkal bonyolultabb eset, mint az előbbi, és a neki megfelelő konstruktor a törzsében kell, hogy tartalmazza az értékadásokat:

```
String::String( const char *init ) {
```

```
    if( !init ) init = "";
    s_ = strcpy( new char[strlen(init)+1], init );
}
```

Két adattípus nem állítható be a tagbeállító lista segítségével: a statikus adatok és a tömbök. A statikus adatokkal az 59. hibánál foglalkozunk részletesen. A tömbök elemeit a konstruktor törzsében egyenként kell beállítani. A tömb helyett talán éppen ezért célszerűbb a szabványos `vector` típust használni.

52. hiba: A tagbeállító lista nem következetes elrendezése

Az osztályobjektumok különböző tagjainak beállítási sorrendjét a C++ nyelv szigorúan rögzíti. (Ezzel kapcsolatban olvassuk el a 49. hibát is.) Ez a rögzített sorrend a következő:

- A virtuális alaposztályok alobjektumai, függetlenül attól, hogy azok a hierarchia mely pontján találhatók.
- A nem virtuális közvetlen alaposztályok, abban a sorrendben, ahogy az alaposztályok listájában felbukkannak.
- Az osztály adattagjai, bevezetésük sorrendjében.

Mindez azt jelenti, hogy egy osztály konstruktorának valamennyi kezdeti beállítást ebben a sorrendben kell elvégeznie. Ebből pedig az is következik, hogy a fordítót „nem érdekli", hogy a tagbeállító listában az egyes elemek milyen sorrendben jelennek meg:

```
class C {
 public:
   C( const char *name );
 private:
   const int len_;
   string n_;
};
// . . .
C::C( const char *name )
   : n_( name ), len_( n_.length() ) // Hiba!!!
   {}
```

A fenti osztály megadása során a `len_` tagot előbb vezettük be, mint az `n_` nevűt, ezért a listában elfoglalt helyétől függetlenül a konstruktor ezt fogja előbb beállítani. Ebből pedig példánkban az következik, hogy egy tagfüggvényt előkészítetlenül akarunk meghívni, aminek a kimenetele nem határozható meg.

Mindezek után kifejezetten jó ötlet, ha a tagbeállító listában az elemeket ugyanabban a sorrendben adjuk meg, mint az alaposztály, illetve úgy, ahogy bevezettük azokat. Ez a sorrend ugyanis biztosan megegyezik a tényleges beállítási sorrenddel. Szintén helyes eljárás, ha a lehetőségekhez mérten kerüljük a beállító listákban a sorrendiségtől való függést:

```
C::C( const char *name )
    : len_( strlen(name) ), n_( name )
    {}
```

Hogy a tagbeállítás tényleges sorrendje és a forráskódban megadott lista elrendezése között miért nincs semmiféle kapcsolat, arra a 67. hiba kapcsán derül fény.

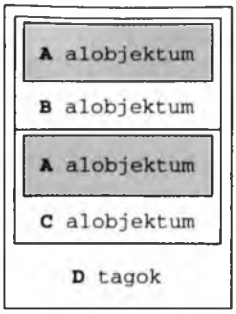
53. hiba: Virtuális alap alapértelmezett beállítása

A virtuális alap alobjektumok máshogy épülnek be az objektumokba, mint egy nem virtuális alap. Amint az 5.1. ábra is mutatja, a nem virtuális alaposztály elhelyezkedése olyan, mintha az adott osztály egyik adattagja lenne. Ez azt is jelenti, hogy egy objektumban egynél többször is előfordulhat:

```
class A { tagok };
class B : public A { tagok };
class C : public A { tagok };
class D : public B, public C { tagok };
```

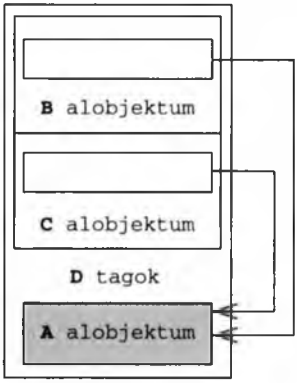
Egy virtuális alaposztály ezzel szemben mindig csak egyszer jelenik meg az objektumokban, még akkor is, ha többször is felbukkan az osztályhierarchiában (lásd az 5.2. ábrát):

```
class A { tagok };
class B : public virtual A { tagok };
class C : public virtual A { tagok };
class D : public B, public C { tagok };
```



5.1. ábra

Az objektumok legvalószínűbb elhelyezkedése többszörös nem virtuális öröklés esetén. A D objektum két A alobjektumot is tartalmaz.



5.2. ábra

Az objektumok valószínű elhelyezkedése a tárban virtuális többszörös öröklődés esetén. A D objektum egyetlen A alobjektumot tartalmaz.

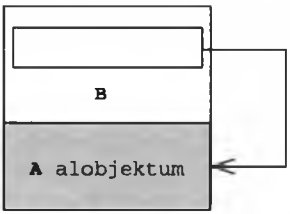
A szemléltetés kedvéért a virtuális alaposztályok egy kissé divatjamúlt, mutatókat használó megvalósítását mutatom be. Azon a helyen, ahol egy nem virtuális A alaposztálynak kellene megjelennie a teljes objektumban, helyette egy, a megosztott tárat címző mutatót találunk, amely egyetlen A alobjektumra mutat. Ennél a megoldásnál sokkal gyakoribb az, amikor a virtuális alaposztályhoz való hozzáférés módját egy eltolási cím, vagy a virtuálisfüggvény-táblában tárolt információ segítségével valósítjuk meg. Ugyanakkor a most következő tárgyalás bármely megvalósításra érvényes.

A megosztott virtuális alapobjektumot tartalmazó tár általában közvetlenül a teljes objektum memóriaterülete után következik. A fenti példában a teljes objektum D típusú, így az A alobjektum területe D területe után következik a tárban. Egy olyan objektumnak azonban, amelynek „legalsó származtatott osztálya" B, más a memóriaképe.

Egy pillanatnyi gondolkodással rögtön beláthatjuk, hogy csupán a legalsó származtatott osztály (most derived class) tudja pontosan, hol található a virtuális alaposztály alobjektuma. Egy B típusú objektum lehet egy teljes osztály, de lehet egy nagyobb objektumba ágyazott alobjektum is. Ennek megfelelően a legalsó származtatott osztály feladata a hierarchiában szereplő összes virtuális alobjektum kezdeti beállítása, valamint az ezek kezeléséhez szükséges eljárások biztosítása.

Egy olyan objektum esetében, amelynek származtatott típusa B – amint azt az 5.3. ábra is mutatja – a B konstruktorok fogják beállítani az A alobjektumot, és az azt címző mutatót:

```
B::B( int arg )
   : A( arg ) {}
```



5.3. ábra

Egy objektum valószínű memóriaképe egyszeres öröklés és virtuális alaposztály esetén. A B objektum egyetlen A alobjektumot tartalmaz, de arra továbbra is csak közvetett módon lehet hivatkozni.

Ha viszont egy objektum legalsó származtatott típusa D (lásd az 5.2. ábrát), akkor a D objektumok konstruktorai fogják beállítani az A alobjektumot, és az azt címző mutatókat is a D, a C és a B közvetlen alaposztályokban is:

```
D::D( int arg )
   : A( arg ), B( arg ), C( arg+1 ) {}
```

Ha a D konstruktora beállította az A alobjektumot, a B és C konsruktorai már nem fogják ezt újra megtenni. (Ezt a fordító többek között úgy valósíthatja meg, hogy a D konstruktora beállít egy jelzőt a B és a C konstruktorai számára, vagy átadja nekik az A címét, jelezve ezzel, hogy az A objektumot már nem kell beállítani. Szóval nincs ebben az egészben semmi rejtélyes.) Nézzük most a D egy másik lehetséges konstruktorát:

```
D::D()
    : B( 11 ), C( 12 ) {}
```

Ez általános, félreértésen alapuló hibaforrás a virtuális alaposztályokkal kapcsolatban. A D objektum konstruktora most is beállítja az A alobjektumot, de most rejtve, az A alapértelmezett konstruktorának meghívásával teszi. Amikor aztán a D konstruktora meghívja a B alobjektum konstruktorát, az már nem fogja újra beállítani az A objektumot, így az A nem alapértelmezett konstruktorának általunk megadott meghívása sem megy végbe.

Az egyszerűség kedvéért virtuális alaposztályokat csak akkor célszerű használni, ha valamilyen tervezési szempont határozottan megköveteli. (Ugyanezt a tanácsot persze fordítva is meg lehet fogalmazni: ne kerüljük a virtuális alaposztályok használatát, ha ezt bizonyos tervezési szempontok egyértelműen igénylik.) Az is általános tervezési szempont, hogy a virtuális alaposztályokat általában „felületosztályként" (interface class) célszerű használni. A felületosztályoknak nincsenek adattagjaik, tagfüggvényeik tisztán virtuálisak (kivéve talán a destruktort), és általában nem rendelkeznek felhasználói konstruktorral sem, csak az egyszerű alapértelmezett változattal:

```
class A {
 public:
    virtual ~A();
    virtual void op1() = 0;
    virtual int op2( int src, int dest ) = 0;
    // . . .
};
inline A::~A()
    {}
```

Ha követjük ezeket az általános tanácsokat, az rendszerint segít elkerülni a konstruktorokkal és értékadásokkal kapcsolatos programozási hibákat. Ami azt illeti, a szabvány kimondja, hogy a fordító által előállított másoló értékadás művelete lefuthat ugyan többször egy virtuális alaposztállyal kapcsolatban, de ennek nem kell feltétlenül így lennie. Ha viszont minden virtuális alaposztály felületosztály, az ér-

tékadás művelete egyetlen üres utasítás, a többszöri értékadás tehát nem jelenthet gondot. (Emlékezzünk vissza, hogy az osztályok belső működése szerint a virtuális függvények mutatóit az értékadás nem, csak a kezdeti beállítás befolyásolja.)

Az értékadás műveletének megvalósítása virtuális alaposztályokat tartalmazó hierarchiákban általában azt jelenti, hogy – bizonyos értelemben – utánoznunk kell azt a működést, ahogy a virtuális alobjektumokat tartalmazó osztályok létrejönnek.

Vegyük például az 5.1. ábrán bemutatott `D` objektum megvalósítását, amely két nem virtuális `A` alobjektumot tartalmaz. Ebben az esetben, akárcsak a `D` konstruktora, egy, a közvetlen alaposztályokra vonatkozó, programozó által írt értékadó művelet is teljes egészében megvalósítható:

➡ 53. hiba/virtassign.cpp

```
D &D::operator =( const D &rhs ) {
    if( this != &rhs ) {
        B::operator =( *this ); // A B alobjektumhoz rendelünk
➥ hozzá
        C::operator =( *this ); // A C alobjektumhoz rendelünk
➥ hozzá
        // Minden D-hez tartozó tagnak értéket adunk
    }
    return *this;
}
```

Ez a megvalósítás azzal a logikus feltételezéssel él, hogy a `B` és `C` alobjektumok megfelelően be fogják állítani saját (nem virtuális) `A` alobjektumaikat. Ugyanúgy, mint a konstruktorok esetében, ez a rétegzett megoldás sem valósítható meg virtuális öröklés esetén. Itt is a legalsó származtatott osztálynak kell a virtuális alapobjektumok értékadásait elvégezni, és azt is meg kell valahogy gátolnia, hogy a közbenső alaposztályok alobjektumai ezt még egyszer megtegyék:

➡ 53. hiba/virtassign.cpp

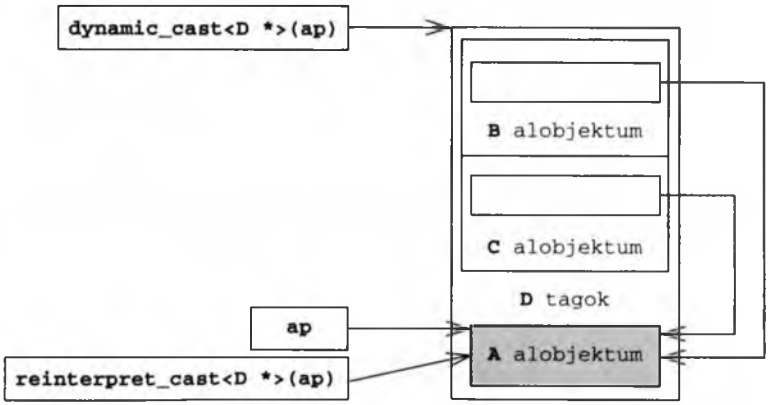
```
D &D::operator =( const D &rhs ) {
    if( this != &rhs ) {
        A::operator =( *this ); // A virtuális A objektumhoz
➥ rendelünk hozzá
        B::nonvirtAssign( *this ); // B-nek adunk értéket az
➥ A rész kivételével
        C::nonvirtAssign( *this ); // C-nek adunk értéket az
➥ A rész kivételével
        // Minden D-hez tartozó tagnak értéket adunk
    }
    return *this;
}
```

Ebben a példában a `B` és a `C` osztályokban egyedi, értékadás jellegű függvényeket vezettünk be. Ezek ugyanúgy működnek, mint a másoló értékadó műveletek, de nem adnak értéket a virtuális alaposztályok alobjektumainak. Ez a megvalósítás – bár hatékony – meglehetősen összetett, és azt is elvárja a `D` osztálytól, hogy „ismerje" az osztályhierarchia saját közvetlen alaposztályain túli felépítését is. A hierarchia bármilyen módosulása a `D` osztály teljes átírását igényli. Ezért, amint azt korábban is említettük, hasznosabb, ha a virtuális alaposztályokat felületosztályként használjuk.

A virtuális alaposztályok alobjektumainak különleges felépítése részben onnan ered, hogy kifejezetten tiltott egy virtuális alaposztályt statikusan annak származtatott osztályára alakítani (downcasting):

```
A *ap = gimmeanA();
D *dp = static_cast<D *>(ap); // Hiba!
dp = (D *)ap; // Hiba!
```

Ugyanakkor a `reinterpret_cast` műveletet használhatjuk a virtuális alaposztály származtatott osztályra való átalakítására. Persze amint az 5.4. ábrából kiderül, e művelet eredménye valószínűleg egy hibás cím lesz, tehát az egésznek nem sok haszna van. Az egyetlen működő módszer arra, hogy egy virtuális alaposztályból annak származtatott osztályát megcímezzük, a `dynamic_cast` művelet használata (lásd a 45. hibát):



5.4. ábra

A statikus és dinamikus típusátalakítás várható hatása virtuális alaposztályok és többszörös öröklés esetén. Ebben a megvalósításban a `D` objektumnak három különböző érvényes címe van, az átalakítás helyessége pedig attól függ, hogy a fordító ismeri-e a teljes objektumon belül a megfelelő eltolási címet.

```
if( D *dp = dynamic_cast<D *>(ap) ) {
 // Csinálunk valamit a dp-vel
}
```

Ugyanakkor a `dynamic_cast` rendszeres használata meglehetősen gyenge tervezésre vall. (Ezzel kapcsolatban lásd a 98. és 99. hibát.)

54. hiba: Az alap másoló konstruktor beállítása

Íme néhány egyszerű programelem:

```
class M {
 public:
   M();
   M( const M & );
   ~M();
   M &operator =( const M & );
   // . . .
};
class B {
 public:
   virtual ~B();
 protected:
   B();
   B( const B & );
   B &operator =( const B & );
   // . . .
};
```

Hozzunk létre ezekből az összetevőkből egy új osztályt, és a lehetőségekhez mérten hagyjuk, hogy a fordító megtegye számunkra, amit megtehet:

```
class D : public B {
   M m_;
};
```

A `D` osztály ugyan nem fogja örökölni a konstruktorokat, a destruktort és a másoló műveleteket az alaposztályától, a fordító azonban készségesen előállítja ezeket, betöltve az űrt (lásd a 49. hibát). A `D` alapértelmezett konstruktora például egy nyilvános helyben kifejtett tagfüggvény lesz. Ez a konstruktor előbb a `B` alaposztály alap-

értelmezett konstruktorát hívja meg, majd az M tagra vonatkozó alapértelmezett konstruktort. A destruktor ugyanezt fogja tenni, de fordított sorrendben: előbb megsemmisíti az elemet, majd meghívja az alaposztály alapértelmezett destruktorát.

A másoló műveletek már érdekesebbek. A fordító által előállított alapértelmezett másoló értékadás művelet tagról tagra történő másolást fog megvalósítani, mintha mi magunk írtuk volna meg, valahogy így:

```
D::D( const D &init )
    : B( init ), m_( init.m_ )
    {}
```

A szintén a fordító által előállított egyszerű értékadó művelet működése a következő kódnak feleltethető meg:

```
D &D::operator =( const D &that ) {
    B::operator =( that );
    m_ = that.m_;
    return *this;
}
```

Tegyük most fel, hogy felveszünk az osztályba egy olyan tagot, amely nem adja meg saját, ezeknek megfelelő műveleteit. Lehet ez például egy olyan adattag, amely dinamikusan foglalt tárterületet csatol az objektumhoz:

```
class D : public B {
 public:
    D();
    ~D();
    D( const D & );
    D &operator =( const D & );
 private:
    M m_;
    X *xp_; // Új adattag
};
```

A megfelelő műveleteket most magunknak kell megírni. Az alapértelmezett konstruktor és destruktor még egyszerű, hiszen a munka javát most is a fordítóra bízhatjuk:

```
D::D()
    : xp_( new X )
```

```
    {}
D::~D()
    { delete xp_; }
```

A fordító az alaposztály és az m_ tag alapértelmezett konstruktorát és destruktorát „beleértve" hívja meg. Ezek után csábító a gondolat, hogy ezzel a módszerrel élhetünk a másoló és értékadó műveletek megvalósítása során is, de ez sajnos nem így van:

```
D::D( const D &init )
    : xp_( new X(*init.xp_) )
    {}
D &D::operator =( const D &rhs ) {
    delete xp_;
    xp_ = new X(*rhs.xp_);
    return *this;
}
```

Mindkét megoldás hibátlanul lefordítható, a velük kapcsolatos hibára csak futás közben derül fény. Másoló konstruktorunk helyesen jár el az xp_ tag beállítása során, mivel abban a megfelelő beállító xp_ tagjának tartalmát helyezi el. Az alaposztály és az m_ tag esetében azonban a fordító már a B és az M alapértelmezett konstruktorát használja a beállításra, nem pedig azok másoló konstruktorait. Az értékadó műveletnél az alaposztály és az m_ tag tartalma változatlan marad.

Ha ezekkel a függvényekkel kapcsolatban levesszük a terhet a fordító válláról, a teljes megvalósításért már mi felelünk:

```
D::D( const D &init )
    : B( init ), m_( init.m_ ), xp_( new X(*init.xp_) )
    {}
D &D::operator =( const D &rhs ) {
    if( this != &rhs ) {
        B::operator =( rhs );
        m_ = rhs.m_;
        delete xp_;
        xp_ = new X(*rhs.xp_);
    }
    return *this;
}
```

Az alapértelmezett konstruktor és destruktor esetében nincs semmi változás, az alaposztály és az m_ tag alapértelmezett konstruktorának rejtett meghívása azonban

most helyes működést eredményez. Én személy szerint az alább látható megoldást kedvelem, mert kevesebbet kell hozzá gépelni, de aki akar, lehet ennél konkrétabb is a kód megfogalmazásában:

```
D::D()
    : B(), m_(), xp_( new X )
    {}
```

55. hiba: A futásidejű statikus beállítások sorrendje

A C++ programok valamennyi statikus adatát még az első használat előtt be kell állítani. E beállítások többsége a programkód betöltése során, még az indítás előtt végbemegy. Ha nincs kifejezetten megadott kezdőérték, akkor a változók tárterülete nullákkal töltődik fel:

```
static int question; // 0
extern int answer = 42;
const char *terminalType; // null
bool isVT100; // hamis
const char **ptt = &terminalType;
```

Ezek a beállítások „egyszerre" hajtódnak végbe, függetlenül a megadási sorrendtől.

Alkalmazhatunk futásidejű statikus beállítást is. Ilyenkor az egyes fordítási egységek közti beállítási sorrend nem meghatározott. (A fordítási egység alapvetően egy előfeldolgozott fájl.) Ez a kötetlenség gyakori forrása a hibáknak, mivel a beállítási sorrend anélkül is megváltozhat, hogy a kódot módosítanánk:

```
// A term.cpp fálban
const char *terminalType = getenv( "TERM" );

// A vt100.cpp fájlban
extern const char *terminalType;
bool isVT100 = strcmp( terminalType, "vt100" )==0; // Hiba?
```

Ebben a kódban rejtett összefüggés fedezhető fel a `terminal-Type` és az `isVT100` változók között, a C++ nyelv azonban nem tudja garantálni ezek beállítási sorrendjét. Ez a hiba általában akkor jön elő, amikor programunkat egy másik rendszeren akarjuk lefordítani, de az más statikus beállítási sorrendet használ az egyes fordítási

egységekre. Megtörténhet azonban, hogy maga a forráskód nem, csak a fordítási eljárás változik meg, vagy hogy egy eddig statikusan csatolt könyvtári szolgáltatást dinamikusan szerkesztünk hozzá a kódhoz.

Tartsuk észben, hogy a statikus objektumok beállítása szintén futásidejű statikus beállításnak számít:

```
class TermInfo {
 public:
   TermInfo()
       : type_( ::terminalType )
       {}
 private:
   std::string type_;
};
// . . .
TermInfo myTerm; // Futásidejű statikus beállítás
```

A futásidejű statikus beállítással kapcsolatos gondokat legegyszerűbben úgy kerülhetjük el, ha a lehető legkisebbre szorítjuk a külső változók, illetve a statikus osztálytagok használatát (lásd még a 3. hibát).

Ha ezeket az eshetőségeket kiküszöböltük, akkor is előállhat olyan helyzet, hogy programunk egy adott fordítási egységen belül a statikus beállítások sorrendjére támaszkodik. Ez a sorrend szerencsére pontosan meghatározott, ugyanis megegyezik a változók megadásának sorrendjével. Ha például a `terminalType` és az `isVT100` változók ebben a sorrendben bukkannak fel a kódban egyetlen fájlon belül, akkor a hordozhatósággal kapcsolatban semmiféle gond nem lehet. Ugyanakkor még ezzel a felállással is előfordulhat sorrendiséggel kapcsolatos probléma, például ha egy külső függvény – beleértve a tagfüggvényeket is – akarja használni valamelyik statikus változónkat. Ezt a függvényt ugyanis – közvetve vagy közvetlenül – még azelőtt meghívhatjuk, hogy az adott változót a program másik fordítási egysége beállította volna:

```
extern const char *termType()
   { return terminalType; }
```

Ha nem ez a gond, akkor alkalmazhatunk valamilyen más megközelítést az értékadás lusta kiértékelésének (lazy evaluation) helyettesítésére. Erre a célra rendszerint a Singleton tervezési módszert használják (lásd a 3. hibát).

Végső megoldásként meg is határozhatjuk a beállítási sorrendet, bizonyos szabványos módszerek alkalmazásával. Az egyik ezek közül a Schwarz számláló használata. A módszer a nevét Jerry Schwarz-ról kapta, aki javasolta és először alkalmazta az `iostream` könyvtár megvalósítása során:

➡ 55. hiba/term.h

```
extern const char *terminalType;
//Egyéb beállítandó dolgok...
class InitMgr { // Schwarz számláló
 public:
   InitMgr()
        { if( !count_++ ) init(); }
   ~InitMgr()
        { if( !--count_ ) cleanup(); }
   void init();
   void cleanup();
 private:
   static long count_; // Folyamatonként egy
};
namespace { InitMgr initMgr; } // Fájlbeszúrásonként egy
```

➡ 55. hiba/term.cpp

```
extern const char *terminalType = 0;
long InitMgr::count_ = 0;
void InitMgr::init() {
   if( !(terminalType = getenv( "TERM" )) )
        terminalType = "VT100";
   // Egyéb beállítandó dolgok...
}
void InitMgr::cleanup() {
   // Az esetleg szükséges takarítás
}
```

A Schwarz számláló azt számolja, hogy azt a fejállományt, amelyben szerepel, hányszor emelték be az `#include` utasítással. Az `InitMgr` `count_` nevű statikus tagjából folyamatonként csak egy van. Ugyanakkor valahányszor a `term.h` fejállományt beszerkesztjük valahová, egy új `InitMgr` objektumnak foglalunk helyet, és minden ilyen művelet futásidejű statikus beállítást igényel. Az `InitMgr` konstruktora megvizsgálja a `count_` tartalmát, hogy megtudja, ez az első beállítása-e az adott folyamat `InitMgr` objektumának. Ha igen, a beállítást végrehajtja.

Visszafelé, amikor a folyamat normálisan befejeződik, a destruktorral rendelkező statikus objektumokat a program törli. Minden `InitMgr` objektum törlésekor az `InitMgr` saját destruktora eggyel csökkenti a `count_` értékét. Amikor a `count_` nullára csökken, az egyértelműen jelzi, hogy az összes törlési folyamat megfelelően végbement.

Bár ez igen hatásos módszer, az igazán begyöpösödött programozók még a Schwarz számlálókon is képesek kifogni. Összességében tehát akkor vagyunk a legnagyobb biztonságban, ha eleve elkerüljük a statikus változókat és futásidejű beállításukat.

56. hiba: Közvetlen és másoló beállítás

Bizton állíthatom, hosszú életem során láttam már néhány igazán hanyagul végrehajtott kezdeti beállítást. Nézzük például a következő egyszerű osztályt:

```
class Y {
 public:
   Y( int );
   ~Y();
};
```

Egyáltalán nem nevezhető egyedinek, ha egy kódban azt látjuk, hogy az `Y` objektum beállítására a programozó a következő három módszert keverve használja, mintha azok egyenértékűek lennének:

```
Y a( 1066 );
Y b = Y(1066);
Y c = 1066;
```

A három beállítás valószínűleg ugyanazt a tárgykódot eredményezi, mégsem egyenértékűek. Az `a` objektum beállítását nevezzük közvetlen beállításnak (direct initialization), és ez tényleg pontosan azt jelenti, amit elvárunk tőle. A beállítás itt az `Y::Y(init)` közvetlen meghívásával megy végbe.

A `b` és a `c` beállítása már sokkal összetettebb. Ami azt illeti, túlságosan is összetett. A `b` objektum esetében azt kérjük a fordítótól, hogy állítson elő nekünk egy névtelen `Y` típusú objektumot, és értékül adja neki az `1066`-ot. Ezek után ezt a névtelen ideiglenes objektumot az `Y` osztály másoló konstruktorának paramétereként használjuk, ami a kapott érték alapján végre beállítja a `b` tartalmát. Ezután persze még meg kell hívni a névtelen objektum destruktorát is annak törlése során. Összességében azt kértük ezzel az utasítással a fordítótól, hogy a következő kódot állítsa elő:

```
Y temp( 1066 ); // Ideiglenes objektum beállítása
Y b( temp ); // Másoló létrehozás
temp.~Y(); // A destruktor indítása
```

A `c` objektum beállítása ugyanígy értelmezett, de itt a névtelen ideiglenes objektum létrehozása rejtve történik.

Módosítsuk most az `Y` osztály megvalósítását például úgy, hogy létrehozunk benne egy saját másoló konstruktort, és megnézzük, mi történik:

```
class Y {
 public:
   Y( int );
   Y( const Y & )
        { abort(); }
   ~Y();
};
```

Ebből a kódból világosan látszik, hogy az objektum a másoló műveletekre már rá sem hederít, helyette leáll. Ha viszont lefordítjuk az új kódot, legnagyobb megdöbbenésünkre mindhárom értékadás gond nélkül lefut, vagyis a programnak esze ágában sincs leállni. Hát ez meg mi?

Nos, mindössze arról van szó, hogy a szabvány szerint a fordítóprogramnak jogában áll a programot úgy módosítani, hogy az ideiglenes objektumok értelmetlen előállítására ne legyen szükség. Ilyenkor akármit is írtunk, pontosan ugyanaz a bináris kód keletkezik, mint a közvetlen kezdeti beállítás esetén. Vegyük azonban észre, hogy ez nem egyszerű hatékonyságnövelés, hiszen a program viselkedése is megváltozott (nem állt le). A legtöbb C++ fordító képes végrehajtani az ilyen kódátalakításokat, de arra ügyeljünk, hogy a szabvány ezt nem teszi kötelezővé. Ismerve e bizonytalanságot, a legjobb, ha mindig pontosan megmondjuk a fordítónak, mit szeretnénk, és mindig közvetlen beállítást használunk az objektumokkal kapcsolatban:

```
Y a(1066), b(1066), c(1066);
```

Persze valakinek az a perverz ötlete is támadhat, hogy kifejezetten megtiltja a fordítónak az ilyesféle kódátalakítást, például azért, mert valamilyen, a másoló műveletekkel kapcsolatos mellékhatásra akarja építeni programja működését, vagy egy igazán nagy és lomha programot akar írni.

Sajnos az efféle működést egy normális C++ fordítóból nem könnyű kicsikarni, mivel annak általában jogában áll az említett átalakításokat szó nélkül elvégezni. Aztán pedig az a helyzet, hogy egy rendszerfüggetlen (vagyis rendszerfüggő fordítási kapcsolókat és `#pragma` utasításokat nem használó) „kódátalakítás-elkerülő" kód annyira rémes látvány, hogy itt és most csak egy gyors pillantást szabad vetnünk rá:

```
struct {
   char b_[sizeof(Y)];
} aY; // A tárolót Y-nal azonos méretűre állítjuk
new (&aY) Y(1066); // Ideiglenes objektum létrehozása
Y d( reinterpret_cast<Y &>(aY) ); // Másoló konstruktor
reinterpret_cast<Y &>(aY).~Y(); // Az ideiglenes objektum törlése
```

Ez a programocska majdnem pontosan tükrözi a fordító által át nem alakított kód jelentését. (Az aY számára lefoglalt területet a program később valószínűleg nem fogja tudni újra felhasználni, ellentétben a fordító által automatikusan létrehozott névtelen változók memóriaterületével. Ezzel kapcsolatban olvassuk el a 66. hibát.) Persze létezik ennél sokkal egyszerűbb módja is annak, hogy nagy és lomha programot írjunk.

Lényeges a fenti programmal kapcsolatban, hogy a kódátalakítást a fordító csak az eredeti kód ellenőrzése után hajtja végre. Ha tehát az eredeti kód hibás, csak egy hibaüzenetet kapunk, még akkor is, ha az átalakítással a hiba „kijavítható" lett volna. Vegyük például a következő X osztályt:

```
class X {
 public:
   X( int );
   ~X();
   // . . .
 private:
   X( const X & );
};

X a( 1066 ); // Rendben.
X b = 1066; // Hiba!
X c = X(1066); // Hiba!
```

A b és c objektumok eredeti kódnak megfelelő beállítása az X osztály másoló konstruktorához való hozzáférést igényli. Sajnos azonban az X osztály tervezője úgy döntött, hogy a másoló konstruktor priváttá tételével megtiltja e szolgáltatás használatát. Bár a kódátalakítás ezt megkerülhetné, nem teszi. A megfelelő pontokon fordítási hibát kapunk.

A közvetlen és másoló beállítással kapcsolatos megkülönböztetés a nem osztály jellegű típusokra is érvényes, az eredmény azonban itt megjósolható, a kód pedig hordozható:

```
int i(12); // Közvetlen.
int j = 12; // Másolás, de az eredmény ugyanaz.
```

Ezekben az esetekben használjuk nyugodtan azt a formát, amelyik a világosabb leírást eredményezi. Ugyanakkor egy sablonban, ahol a változók típusa előre nem ismert, célszerűbb a közvetlen beállításhoz ragaszkodni. Vegyünk például egy egyszerűsített sorozathossz-algoritmust, ami nemcsak a sorozat bejárójának típusát (`In`), hanem a számláló (`N`) típusát is megkapja paraméterként:

➡ 56. hiba/seqlenght.cpp

```
template <typename N, typename In>
void seqLength( N &len, In b, In e ) {
   N n( 0 ); // Így nem "N n = 0;"
   while( b != e ) {
        ++n;
        ++b;
   }
   len = n;
}
```

Ez a megvalósítás a közvetlen kezdeti beállítás alkalmazásával lehetővé teszi számunkra, hogy egy (kétségtelenül szokatlan) saját számtípust használjunk, amely megtiltja a másoló konstruktor használatát. A `seqLength` egy olyan megvalósítása, amely az `N` objektum értékét másoló beállítással határozná meg, nem tenné ugyanezt lehetővé.

Összefoglalva tehát, az egyszerű és világos kódolás érdekében célszerű minden esetben közvetlen beállítást használni az osztály jellegű típusokkal kapcsolatban, illetve azokon a helyeken, ahol ilyen típusok felbukkanására esély van.

57. hiba: Paraméterek közvetlen kezdeti beállítása

Azt mindannyian tudjuk, hogy a formális paraméterek értéke a tényleges paraméterek tartalma alapján áll be, de vajon közvetlen vagy másoló beállítással? Ezt egy kísérlettel egész könnyen megmondhatjuk:

```
class Y {
 public:
   Y( int );
   ~Y();
 private:
   Y( const Y & );
   // . . .
};
```

```
void f( Y yFormalArg ) {
    // . . .
}
// . . .
f( 1337 );
```

Ha a paraméterek átadása közvetlen beállítással történik, az `f` függvény hívása helyes kell legyen. Ha viszont másoló beállítás valósul meg, a fordítónak hibát kell jeleznie, hiszen az `Y` osztály privát másoló konstruktorához akarunk rejtve hozzáférni. A legtöbb fordító nem fog hibát jelezni, tehát boldogan levonjuk a következtetést, hogy a paraméterek átadása közvetlen beállítással történik. Igen ám, csakhogy a legtöbb fordító rosszul teszi, amikor nem jelez hibát, minden valószínűség szerint azért, mert működését még nem igazították a szabványhoz. A szabvány ugyanis határozottan úgy foglal állást a kérdésben, hogy a paraméterek átadásának másoló beállítással kell történnie, vagyis a fenti kódban az `f` hívása hibás. Az `yFormalArg` beállítása teljesen megfelel az alábbi kódnak:

```
Y yFormalArg = 1337; // Hiba!
```

Ha szabványos, hordozható kódot akarunk írni, ami minden körülmények között helyes marad, még akkor is, ha egy korábban a szabványnak meg nem felelő fordító egyszerre szabványos lesz, kénytelenek leszünk elkerülni az olyan megvalósításokat, mint amilyen a fenti kódban az `f` függvény hívása.

Vannak aztán a teljesítmény körül is gondok. Ha az `f`-et hívó függvény hozzáférhet az `Y` osztály amúgy privát másoló konstruktorához, a program ugyan helyesen működik, de a keletkező futtatható állomány körülbelül a következő kódnak felel meg:

```
Y temp( 1337 );
yFormalArg( temp );
// Az f törzse
yFormalArg.~Y();
temp.~Y();
```

Ez pedig azt jelenti, hogy a formális paraméter beállítása során létre kell hozni egy ideiglenes objektumot, be kell állítani ezt a másoló konstruktorral, a függvény visszatérésekor a formális paramétert meg kell semmisíteni, és ráadásul még az ideiglenes objektum destruktorát is meg kell hívni. Ez tehát négy függvényhívás, illetve magának az `f`-nek a hívása. Szerencsére a legtöbb fordító észreveszi az egyszerűsí-

tés lehetőségét, nem hoz létre ideiglenes objektumot, és nem használja a másoló konstruktort. Helyette a következő, közvetlen beállítást használó forráskódnak megfelelő tárgykódot állítja elő:

```
yFormalArg( 1337 );
// Az f törzse
yFormalArg.~Y();
```

Néha azonban még ez a megoldás sem a legjobb. Mi van például, ha az `yFormalArg`-ot egy `Y` objektummal akarjuk beállítani?

```
Y aY( 1453 );
f( aY );
```

Itt az `yFormalArg` létrehozásához az `aY` alapján le kell futnia egy másoló konstruktornak, majd a függvény visszatérésekor meg kell semmisíteni a formális paramétert a megfelelő destruktor meghívásával. Az adott helyzetben és általában is sokkal hatékonyabb megoldás, ha kerüljük az objektumok érték szerinti átadását. Helyette állandóra mutató hivatkozást célszerű használni:

```
void fprime( const Y &yFormalArg );
// . . .
fprime( 1337 ); // Működik! Nem másoló konstruktor.
fprime( aY ); // Működik, hatékony.
```

Az első esetben a fordító létrehoz egy `Y` típusú ideiglenes objektumot, majd annak tartalmát `1337`-re állítja. Ezt fogja aztán használni a hivatkozás típusú formális paraméter beállítására. Az ideiglenes objektum az `fprime` visszatérése után azonnal megsemmisül. (A témával kapcsolatban olvassuk el a 44. hibát, ahol az ilyen paraméterek visszatérési értékként való használatával kapcsolatos veszélyeket tárgyalom.) Ez a megoldás teljesítmény szempontjából egyenértékű a fenti átalakított kóddal, de megvan az az előnye is, hogy teljesen szabványos C++ kód. Az `fprime` második hívása egyáltalán nem igényli ideiglenes objektum létrehozását, és az ezzel kapcsolatos teljesítménytöbbletet, sőt egy destruktorhívással is kevesebb a keletkező futtatható állomány.

58. hiba: A visszatérési érték finomhangolásának figyelmen kívül hagyása

Gyakran van szükség arra, hogy egy függvény érték szerint adjon vissza valamilyen eredményt. Az alább látható `String` osztálynak például van egy kéttényezős összefűző művelete, amelynek a két összefűzött `String` objektumot kell visszaadnia érték szerint:

```
class String {
 public:
   String( const char * );
   String( const String & );
   String &operator =( const String &rhs );
   String &operator +=( const String &rhs );
   friend String
        operator +( const String &lhs, const String &rhs );
   // . . .
 private:
   char *s_;
};
```

Ugyanúgy, ahogy a formális paraméterek beállítása, a visszatérési érték megadása is másoló beállítás segítségével történik:

```
String operator +( const String &lhs, const String &rhs ) {
   String temp( lhs );
   temp += rhs;
   return temp;
}
```

Logikus módon a fordító a `temp` értéke alapján a másoló konstruktorral beállítja a hívó számára kijelölt visszatérési területet, majd magát a `temp` objektumot megsemmisíti. Általában a fordító az érték szerinti átadást úgy valósítja meg, hogy a célobjektumot a függvény beleértett paraméterének tekinti, vagyis mintha a következő kódot fordítanánk az eredeti helyett:

```
void
operator +( String &dest, const String &lhs, const String &rhs )
{
   String temp( lhs );
   temp += rhs;
```

```
    dest.String::String( temp ); // Másoló konstruktor
    temp.~String();
}
```

Jegyezzük meg, hogy bár a fordító meghívja a másoló konstruktrort, mi magunk ezt nem tehetjük meg. Az igazán ravasz programozóknak persze erre is van megoldása:

```
new (&dest) String( temp ); // A new művelettel kapcsolatos
➥ trükk. Lásd a 62. hibát.
```

A fenti átalakítás egyik értelme az, hogy egy osztályobjektumot általában hatékonyabb visszatérési érték alapján beállítani, mint értékadással átvinni az eredményt:

```
String ab( a+b ); // Hatékony.
ab = a + b; // Valószínűleg nem hatékony.
```

Az `ab` bevezetésekor a fordító megteheti, hogy közvetlenül másolja bele az `a+b` művelet eredményét, értékadásnál ez már nem lehetséges. Az értékadó művelet a `String` osztály tagfüggvénye, amely úgy működik, hogy a destruktorhoz hasonlóan előbb törli az `ab` tartalmát, majd újra beállítja azt. Ez egyben azt is jelenti, hogy beállítatlan (előkészítetlen) területtel soha nem szabad értékadó műveletet végezni (lásd a 47. hibát):

```
String &String::operator =( const String &rhs );
```

A `String` osztály `operator =` tagfüggvényének `rhs` változóját csak úgy lehet beállítani, ha a fordító az `a+b` művelet eredményét előbb egy ideiglenes objektumba másolja, azzal beállítja az `rhs`-t, majd megsemmisíti az ideiglenes objektumot a műveletfüggvény visszatérésekor. A hatékonyság tehát mindenképpen a kezdeti beállítás és nem az értékadás mellett szól.

Nézzük, mit jelent a másoló beállítás abban az esetben, ha a visszaadni kívánt érték típusa nem azonos a függvény visszatérési típusával:

```
String promote( const char *str )
    { return str; }
```

Itt a másoló beállítás értelme megköveteli a fordítótól, hogy a függvény visszatérése előtt létrehozzon egy `String` típusú helyi ideiglenes objektumot, ebben helyezze el az `str` tartalmát, majd a másoló beállítással ezt vigye át a visszatérési érték számára kijelölt területre. Visszatéréskor az ideiglenes objektumot természetesen még meg

kell semmisíteni. Ugyanakkor a fordítónak itt is, mint a formális paraméterekkel kapcsolatos beállításoknál, lehetősége van arra, hogy átalakítsa a kódot, és kikerülje a másoló beállítást és az ideiglenes objektum létrehozását. Nagyon valószínű, hogy a legtöbb fordítóprogram ebben az esetben az `str` segítségével közvetlenül állítja be a visszatérési érték tárterületét, mégpedig a `String` osztály nem másoló konstruktorának meghívásával. Ez a módszer, vagyis amikor a fordító a kód átalakításával a függvény visszatérésekor kiküszöböli az ideiglenes objektum használatát, a „visszatérési érték finomhangolása" (Return Value Optimization, RVO).

A programozók persze gyakran próbálnak elérni nagyobb hatékonyságot valamilyen alacsonyabb szintű megközelítéssel:

```
String operator +( const String &lhs, const String &rhs ) {
   char *buf = new char[ strlen(lhs.s_)+strlen(rhs.s_)+1];
   String temp( strcat( strcpy( buf, lhs.s_ ), rhs.s_ ) );
   delete [] buf;
   return temp;
}
```

Sajnos ez a kód általában sokkal lassabb, mint az `operator +` előbbi megvalósítása. Lefoglalunk ugyanis egy tárterületet csupán azért, hogy abban összefűzhessük a két, paraméterként kapott karakterláncot. A karaktertömb tartalma alapján aztán létrehozunk egy ideiglenes `String` típusú objektumot, magát a tömböt pedig eldobjuk.

Az olyan esetekben, mint ez is, néha célszerű „számoló konstruktort" használni az osztály megvalósítása során. A számoló konstruktor olyan konstruktor, amely ténylegesen – általában privát – tagfüggvénye az osztálynak. Ez tulajdonképpen egy kisegítő függvény, amely rendelkezik azokkal a különleges tulajdonságokkal, amelyekkel a konstruktorok bírnak, de az egyszerű tagfüggvények nem. A számunkra érdekes tulajdonság általában annak garantáltsága, hogy a konstruktor nem beállított memóriaterületekkel és nem objektumokkal dolgozik. Ez röviden azt jelenti, hogy a műveletek során nincs mit eltakarítani:

```
class String {
   // . . .
 private:
   String( const char *a, const char *b ) { // Számított
        s_ = new char[ strlen(a)+strlen(b)+1];
        strcat( strcpy( s_, a ), b );
   }
   char *s_;
};
```

Ezt a „számoló konstruktort" aztán az osztály egyéb tagfüggvényeiben arra használhatjuk, hogy azok a segítségével hatékonyan adhassanak vissza eredményeket érték szerint:

```
inline String operator +( const String &a, const String &b )
   { return String( a.s_, b.s_ ); }
```

Emlékezzünk rá, hogy a visszatérési értékkel végzett másoló beállítás megegyezik egy deklarációval:

```
String retval = String( a.s_, b.s_ );
```

Ha tehát a fordító alkalmazza a visszatérési értékkel kapcsolatos finomhangolást, akkor egy, a közvetlen kezdeti beállítással azonos függvényhívást valósítottunk meg:

```
String retval( a.s_, b.s_ );
```

A számoló konstruktorok gyakran egészen egyszerű függvények, így használhatók helyben kifejtett kódrészletként. Ezek után persze már maga az `operator +` is esélyes lehet erre, így végül igen hatékony kódot kapunk. Csaknem olyan hatékonyat, mintha az egész műveletet magunk kódoltuk volna. Ugyanakkor fontos szem előtt tartani, hogy a számoló műveleteken keresztül rendszerint semmilyen módon nem egyszerűsíthetjük egy osztály felületét. Ez pedig azt jelenti, hogy ezeket a különleges függvényeket kizárólag az osztály megvalósításához kapcsolódónak kell tekintenünk, és a privát részben kell szerepeltetnünk. Emellett minden egyparaméteres konstruktort kivétel nélkül `explicit`-ként célszerű bevezetni, mert így elkerülhetjük az osztályok beállításával kapcsolatos rejtett átalakítások körüli gondokat. (Ezzel kapcsolatban olvassuk el a 37. hibát.)

A C++ fordítók általában tartalmaznak egy másik, a „nevesített visszatérési értékkel" kapcsolatos finomhangolást is (Named Return Value Optimization, NVO). Ez tulajdonképpen ugyanúgy működik, mint az előbb tárgyalt módszer, de lehetővé teszi, hogy a visszatérési értéket egy helyi változóban tároljuk. Vegyük a korábban említett `operator +` eredeti megvalósítását:

```
String operator +( const String &lhs, const String &rhs ) {
   String temp( lhs );
   temp += rhs;
   return temp;
}
```

Ha a fordító alkalmazza az NRV finomhangolást, akkor a `temp` helyi változó helyett ténylegesen egy, a hívó által a visszatérési érték fogadására megadott objektumot címző hivatkozás kerül a kódba, mintha a ténylegesen begépelt helyett a következő kódot fordítanánk:

```
void
operator +( String &dest, const String &lhs, const String &rhs )
{
   dest.String::String( lhs ); // Másoló konstruktor
   dest += rhs;
}
```

Az NRV eljárást a fordító csak akkor alkalmazza, ha képes megállapítani, hogy az adott függvény visszatérési értékét tartalmazó valamennyi kifejezés azonos, és ugyanarra a helyi változóra hivatkozik. Célszerű tehát minden függvényben csak egy visszatérési utasítást megadni, ha pedig mégis többet használunk, akkor mindenütt ugyanazt a helyi változót visszaadni. Minél egyszerűbb a megvalósítás, annál jobb. Jegyezzük meg, hogy az NRV eljárás tulajdonképpen nem hatékonyságnövelés, hanem kódátalakítás, hiszen az ideiglenes objektumok beállításával és eltávolításával kapcsolatos valamennyi mellékhatás megszűnik.

Az a teljesítménytöbblet, amit az említett finomhangoló eljárások által nyerhetünk, esetenként igen jelentős lehet, így célszerű alkalmazhatóságukat számoló konstruktorok vagy visszatérési értéket tároló helyi változók használatával elősegíteni.

59. hiba: Statikus tag beállítása a konstruktorban

A statikus adattagok általában az adott osztály elemeit képező objektumoktól függetlenül léteznek, és még azelőtt létrejönnek, hogy egyetlen objektum is keletkezne. (Persze azért ügyeljünk az ezen a területen általánosságban érvényes korlátokra.) Akár a tagfüggvények (statikusak és nem statikusak egyaránt) a statikus adatok is csak egy külső kapcsolaton keresztül kötődnek az osztályukhoz, és csak az osztály számára elérhetők:

```
class Account {
   // . . .
 private:
   static const int idLen = 20;
   static const int prefixLen;
   static long numAccounts;
```

```
};
// . . .
const int Account::idLen;
const int Account::prefixLen = 4;
long Account::numAccounts = 0;
```

Az egész állandók és a felsoroló típusú statikus adatok beállítása történhet az osztályon belül vagy azon kívül is, de szigorúan csak egyszer. Az állandó egészek és azok beállítása helyett amúgy általában célszerűbb felsoroló típust használni:

```
class Account {
   // . . .
 private:
   enum {
        idLen = 20,
        prefixLen = 4
   };
   static long numAccounts;
};
// . . .
long Account::numAccounts = 0;
```

Bár a felsoroló típusok csaknem minden helyzetben helyettesíthetik az egész állandókat, az a hátulütőjük azért megmarad, hogy mivel ténylegesen nem foglalnak tárterületet, nem adhatunk meg rájuk mutatót. Emellett önálló típust képviselnek, ami nem azonos az `int` típussal, így gondot okozhatnak a túlterhelt függvények azonosítása során, ha azok paramétereként használjuk őket. Jegyezzük meg azt is, hogy míg példánkban a `numAccounts` osztályon kívüli megadása szükséges volt, addig kifejezett kezdeti beállítása nem. Ha mégis élünk a beállítás lehetőségével, az értéket feltétlenül nullára, vagyis „teljes nullázásra" („all zeros") kell állítanunk, mivel ezzel megakadályozzuk a későbbi karbantartókat abban, hogy valami más értéket használjanak. (Az emberek különböző okok miatt az 1 és a -1 értékeket nagyon szokták szeretni.) Ezzel kapcsolatban olvassuk el a 25. hibát is.

A statikus osztályelemek futás közben történő beállítása elképesztően rossz ötlet. Előfordulhat, hogy egy statikus tag beállítása esetleg még meg sem történt, amikor egy statikus objektumé már igen:

```
class Account {
 public:
   Account() {
        . . .  calculateCount()  . . .
   }
   // . . .
```

```
    static long numAccounts;
    static const int fudgeFactor;
    int calculateCount()
        { return numAccounts+fudgeFactor; }
);
// . . .
static Account myAcct; // Hoppá!
// . . .
long Account::numAccounts = 0;
const int Account::fudgeFactor = atoi(getenv("FUDGE"));
```

Példánkban a `myAcct` nevű, `Account` típusú objektumot előbb állítjuk be, mint az osztály `fudgeFactor` nevű statikus elemét. Ennek pedig az a következménye, hogy az `Account` osztály konstruktora az objektum létrehozása során egy nem beállított memóriaterületet használ a `calculateCount` meghívásakor. A `fudgeFactor` tartalma amúgy csupa nulla lesz, mivel ez az alapértelmezett a statikus adattagoknál. Ha ez érvényes érték is lehet az adott összefüggésben, a hiba okát sokáig kereshetjük.

Egyes programozók ezt úgy próbálják kiküszöbölni, hogy a konstruktort használják a statikus adattagok beállítására. Ez több szempontból is használhatatlan módszer, hiszen egyrészt a tagbeállító listán statikus adattagok nem szerepeltethetők, másrészt amint a végrehajtás a konstruktor törzséhez érkezik, kezdeti beállításra már nincs lehetőségünk, csak értékadásra:

```
Account::Account() {
    // . . .
    fudgeFactor = atoi( getenv( "FUDGE" ) ); // Hiba!
}
```

Az egyetlen járható út, ha a `fudgeFactor`-t nem állandóként vezetjük be, hanem az osztály valamennyi konstruktorában megírjuk hozzá a késői beállításhoz szükséges kódot, aztán reménykedünk benne, hogy karbantartáskor a programozók odafigyelnek majd rá, hogy az összes ilyen kódrészletet megfelelően módosítsák.

Ezek után talán egyszerűbb, ha a statikus adattagokat ugyanúgy kezeljük, mint az összes többi statikus információt: próbáljuk elkerülni a használatukat, ha lehetséges. Ha mindenképpen használnunk kell ilyen adatokat, legalább ne futásidőben állítsuk be azokat.

6

A memória és az erőforrások kezelése

A C++ nyelv ugyan hatalmas rugalmasságot mutat a memória kezelése terén, de olyan programozó viszonylag kevés akad, aki valóban érti és átlátja a lehetőségeket. A nyelvnek ezen a területén egyszerre találkozik a túlterhelés, a névrejtés, a konstruktorok és destruktorok, a kivételkezelés, a statikus és virtuális függvények, a műveleti és nem műveleti függvények. Ezek együttesen biztosítják a memóriakezelés rugalmasságát és testreszabhatóságát. Mindeközben a dolgok sajnos kénytelen-kelletlen egy kissé bonyolulttá válhatnak.

Ebben a fejezetben azt fogjuk megvizsgálni, hogy az említett nyelvi elemek miként használhatók a memóriakezelés területén. Bemutatjuk lehetséges – és néha egészen meglepő – kölcsönhatásaikat, és azt, hogy miként egyszerűsíthetjük használatukat az említett összetettség ellenére.

Minthogy a memória a programok által kezelhető erőforrások közül csupán egy, azt is megvizsgáljuk, miként köthetünk egyéb erőforrásokat a memóriához, és hogyan használhatjuk a C++ kifinomult memóriakezelő eljárásait az ezekhez való hozzáférés során.

60. hiba: A skalár- és tömbtípusok tárfoglalásának megkülönböztetése

Azonos-e egyetlen `Widget` (grafikus objektum) az ilyen objektumok tömbjével? Természetesen nem. Akkor viszont miért lepődik meg a C++ programozók többsége azon, hogy a skalár- és tömbtípusokra eltérő műveleteket kell alkalmazni?

Azt tudjuk, hogyan foglalhatunk tárat egy `Widget`-nek, és hogyan szabadíthatjuk fel azt. A `new` és a `delete` műveleteket használjuk:

```
Widget *w = new Widget( arg );
// . . .
delete w;
```

A C++ számos más műveletétől eltérően a `new` művelet működését nem módosíthatjuk túlterheléssel. A `new` művelet mindig egy `operator new` nevű függvényt hív meg, ami (valószínűleg) lefoglalja a szükséges tárterületet és beállítja annak tartalmát. A `Widget` objektum esetében a `new` művelet az objektumtípusnak megfelelő

operator new függvényt hívja meg, amely egyetlen size_t típusú paramétert vár. Ezután azonnal lefut a Widget konstruktora, ami beállítja az operator new által visszaadott területet, és ezzel el is készült az új Widget típusú objektum.

A delete művelet a Widget típusú objektumon előbb lefuttatja az osztály destruktorát, majd az operator delete nevű függvénnyel (valószínűleg) felszabadítja a korábban lefoglalt memóriaterületet.

A memóriakezelés módjával kapcsolatos változtatásokat az operator new és az operator delete függvények túlterhelésével, rejtésével, vagy helyettesítésével kell megoldani, vagyis a módosítás soha nem a new és a delete műveleteket érinti.

Azt is tudjuk persze, hogyan foglalhatunk le, illetve szabadíthatunk fel objektumok egy egész tömbjét tartalmazó területet. Itt azonban nem használjuk a new és delete műveleteket:

```
w = new Widget[n];
// . . .
delete [] w;
```

Helyettük most a new [] és a delete [] műveletekhez kell fordulnunk. (Ezek kiejtése angolul „array new" és „array delete".) Akár az egyszerű new és delete, tömbökre vonatkozó megfelelőik jelentése sem módosítható. A tömbökre vonatkozó new [] művelet először a típusnak megfelelő operator new [] függvényt futtatja le, majd ha szükséges, minden elemen elvégzi az alapértelmezett beállítást az elsőtől az utolsó felé haladva. A tömbökre vonatkozó delete [] először a létrehozás fordított sorrendjében egyenként megsemmisíti a tömbelemeket, majd a típusnak megfelelő operator delete [] függvény meghívásával felszabadítja az általuk lefoglalt memóriaterületet.

Itt érdemes megjegyezni, hogy a tömbök helyett az esetek többségében célszerű a szabványos könyvtár vector típusát használni. A vector csaknem olyan hatékony, mint a tömb, így általában egyfajta „okos tömbnek" tekinthető, aminek jelentése is hasonló a tömbökéhez. Ugyanakkor egy vector megsemmisítésekor az elemek az elsőtől az utolsó elé haladva törlődnek, vagyis épp fordított sorrendben, mint a tömböknél.

A memóriakezelő függvényeket megfelelő módon párosítva kell használnunk. Ha a new művelettel foglaltunk tárat egy objektumnak, akkor azt a delete segítségével kell felszabadítanunk. Ha a malloc függvényt használtuk erre a célra, akkor

a `free` a megfelelő pár. Néha akkor is működik egy program bizonyos rendszereken és bizonyos típusokkal, ha a `new` párjaként a `free`-t, a `malloc` párjaként pedig a `delete` műveletet használjuk, de ilyenkor semmi garancia nincs arra, hogy a kód később is működőképes lesz:

```
int *ip = new int(12);
// . . .
free( ip ); // Hibás!
ip = static_cast<int *>(malloc( sizeof(int) ));
*ip = 12;
// . . .
delete ip; // Hibás!
```

Ugyanezek a megkötések érvényesek a tömbök tárfoglalására is. Az egyik leggyakoribb hiba, hogy lefoglaláskor még a tömbökre vonatkozó `new` műveletet használjuk, a felszabadításra azonban már a skalárokra vonatkozó `delete` utasítást. Mint a `new` és a `free` párosításakor, itt is van rá esély, hogy a program bizonyos helyzetekben működni fog, de hogy később is működőképes marad-e, az már kétséges:

```
double *dp = new double[1];
// . . .
delete dp; // Hibás!
```

Vegyük figyelembe, hogy a fordító nem képes észrevenni azt a hibát, ha egy tömböt a skalárokra vonatkozó `delete` művelettel semmisítünk meg, ugyanis nem tud különbséget tenni az egyetlen elemet és a tömböt címző mutatók között. A tömbökre vonatkozó `new` művelet viszont általában elhelyez valahol a tömb számára lefoglalt memóriaterület mellett valamilyen információt, ami a tömb méretén kívül a benne található elemek számára is utal. A tömbtörlő `delete` művelet azután ezt elolvassa és fel is használja működése közben.

E kiegészítő információ formája feltehetőleg minden rendszeren eltér attól, amit a skalárokra vonatkozó `new` művelet hoz létre. Ebből pedig már egyenesen következik, hogy ha egy tömb területét a skalárokra vonatkozó `delete` művelettel akarjuk felszabadítani, akkor az félre fogja értelmezni a kapott információt, és megjósolhatatlan dolgokat fog művelni. Az is előfordulhat, hogy a skalárokra és tömbökre vonatkozó műveletek nem is ugyanazzal a memóriaterülettel gazdálkodnak, hanem mindegyiknek megvan a maga elkülönített területe. Ilyen esetben a skaláris törlő művelet tömbre való alkalmazása csaknem biztosan katasztrófához vezet.

```
delete [] dp; // Helyes
```

A skalárokkal és tömbökkel kapcsolatos pontatlanságok a memóriát kezelő tagfüggvényeknél is gyakran felbukkannak:

```
class Widget {
 public:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   // . . .
};
```

Példánkban a `Widget` osztály szerzője elhatározta, hogy a grafikus elemekkel kapcsolatos memóriakezelést maga veszi kézbe. Arról azonban megfeledkezett, hogy a tömbökre vonatkozó `new` és `delete` műveletekkel kapcsolatos függvényeknek nem ugyanaz a neve, mint a skalárokra vonatkozóknak. Az általa írt tagfüggvények így valójában nem rejtik el a megfelelő alapértelmezett műveletfüggvényeket:

```
Widget *w = new Widget( arg ); // Helyes.
// . . .
delete w; // Helyes.
w = new Widget[n];  // Hoppá!
// . . .
delete [] w; // Hoppá!
```

Mivel a `Widget` osztály nem tartalmaz `operator new []` és `operator delete []` nevű függvényeket, a fordító ezek globális megfelelőit fogja használni, ami feltehetőleg helytelen működéshez vezet. A hiba természetesen a megfelelő műveletfüggvények megadásával javítható.

Ha az említett működés mégis helyes, a `Widget` osztály szerzőjének illik ezt a dokumentációban megemlítenie, ellenkező esetben ugyanis a későbbi karbantartók pótolni fogják a – ténylegesen nem létező – hiányt. A dokumentálás legjobb módja ebben az esetben nem is egy figyelemfelkeltő megjegyzés, hanem magának a kódnak a megfelelő megfogalmazása:

```
class Widget {
 public:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   void *operator new[]( size_t n )
        { return ::operator new[](n); }
   void operator delete[]( void *p, size_t )
        { ::operator delete[](p); }
   // . . .
};
```

A függvények itt látható helyben kifejtett (inline) változatai futásidőben semmiféle többletterhelést nem jelentenek, viszont a legszórakozottabb programozó figyelmét is felhívják arra, hogy ne kísérletezzen a szerző gondolatainak kifürkészésével, jó ez a kód úgy, ahogy van. A `Widget` osztály igenis a globális `new` és `delete` műveleteket kívánja használni.

61. hiba: A tárfoglalási hiba ellenőrzése

Vannak kérdések, amiket egyszerűen nem kellene feltennünk. És az, hogy egy adott memóriafoglalási művelet sikeres volt-e, ebbe a kategóriába tartozik.

Nézzük, hogy valósul meg mindez a való életben, a C++ programok írása során. Íme egy kódrészlet, amely körültekintően ellenőrzi, hogy valamennyi tárfoglalás sikeres volt-e:

```
bool error = false;
String **array = new String *[n];
if( array ) {
   for( String **p = array; p < array+n; ++p ) {
        String *tmp = new String;
        if( tmp )
            *p = tmp;
        else {
            error = true;
            break;
        }
   }
}
else
   error = true;
if( error )
   handleError();
```

Az ilyen kódolási stílus használata bizony sok bajjal jár, de megérné az erőfeszítést, ha valóban képes lenne felderíteni minden tárfoglalási hibát. De nem képes. Sajnos a `String` osztály konstruktora maga is „belekeveredhet" a tárfoglalással kapcsolatos problémákba, és nincs egyszerű módszer arra, hogy ezt a hibát a konstruktor a külvilággal is tudassa. Járható ugyan, de nem túl megnyerő módszer, ha ilyenkor a konstruktor a teljes `String` objektumot valamilyen kezelhető hibaállapotba hozza, és ezt egy jelző (flag) beállításával tudatja az osztály felhasználójával. Persze,

még ha történetesen hozzá is férünk a `String` osztály forráskódjához e szolgáltatást megvalósítandó, akkor is újabb terheket rovunk mind az eredeti szerzőre, mind a későbbi karbantartókra, hiszen egy újabb feltételt kell ellenőrizniük.

A másik megoldás, hogy egyáltalán nem ellenőrizzük a hibák esetleges bekövetkeztét. Az összetett hibaellenőrző kód egyrészt ritkán old meg minden gondot, másrészt csaknem biztosan elrontják a később jövő karbantartók. Sokkal jobb megközelítés, ha egyáltalán nem ellenőrizzük a hibákat:

```
String **array = new String *[n];
for( String **p = array; p < array+n; ++p )
   *p = new String;
```

Ez a kód rövidebb, világosabb, gyorsabb, és még helyesen is működik. A `new` művelet szabványos viselkedése tárfoglalási hiba esetén az, hogy `bad_alloc` kivételt vált ki. Ez pedig lehetőséget teremt arra, hogy a hibaellenőrzést elszigeteljük a kód többi részétől, ami világosabb, és általában hatékonyabbá is teszi a programot.

Ha szabványos módon használjuk a `new` műveletet, akkor az eredmény vizsgálata soha nem fog hibát jelezni, hiszen a `new` vagy sikeresen lefoglalja a szükséges tárat, vagy kivételt jelez:

```
int *ip = new int;
if( ip ) { // Mindig igaz feltétel
   // . . .
}
else {
   // Soha nem fut le
}
```

Persze lehetőségünk van rá, hogy előállítsuk az `operator new` függvény egy „szabványosan" működő változatát, amely hiba esetén nullmutatót ad vissza:

```
int *ip = new (nothrow) int;
if( ip ) { // Mindig igaz feltétel
   // . . .
}
else {
   // Csaknem biztosan soha nem fut le
}
```

Ezzel azonban visszajutottunk az eredeti írásmódhoz, annak minden említett bajával együtt. Jobb tehát, ha igyekszünk elkerülni az ilyen „visszamenőlegesen megfelelő" megoldásokat, és a kód működését a „kivételkiváltó" `new` műveletre alapozzuk.

Amúgy maga a futásidőben működő rendszer is fel van készítve a tárfoglalási hiba által jelentett meglehetősen kellemetlen helyzetekre. Emlékezzünk rá, hogy a `new` művelet két függvényt hív meg: előbb az objektumnak megfelelő `operator new` függvény segítségével lefoglalja a szükséges tárat, majd a konstruktor meghívásával beállítja (előkészíti) azt:

```
Thing *tp = new Thing( arg );
```

Ha most `bad_alloc` kivételt érzékelünk, akkor biztosan tárfoglalási hiba keletkezett, de nem tudjuk, hogy hol. Keletkezhetett a `Thing` típusú objektum számára történő eredeti tárfoglaláskor, de felbukkanhatott a `Thing` konstruktorának futása közben is. Az első esetben nincs memória, amit felszabadíthatnánk, hiszen a `tp` objektumba soha semmi nem került. A második esetben a `tp` által lefoglalt (de be soha nem állított) memóriaterületet vissza kell juttatnunk a rendszernek. Azt azonban már nehéz, sőt esetleg lehetetlen megállapítani, melyik esettel állunk szemben.

Szerencsére a futásidőben működő rendszer megoldja ezt a gondot. Ha a `Thing` típusú objektum számára sikerül tárat foglalnia, de a konstruktor nem képes hibátlanul lefutni, a rendszer meghívja a megfelelő `operator delete` függvényt, és felszabadítja a tárat (lásd még a 62. hibát).

62. hiba: A globális new és delete műveletek helyettesítése

Általában rossz ötlet a szabványos, globális `operator new`, `operator delete`, vagy tömbökre vonatkozó megfelelőik helyettesítése, bár a szabvány lehetővé teszi ezt. A szabványos változatok általában optimálisan működnek, így valószínűtlen, hogy mi valami jobbat leszünk képesek előállítani helyettük. (Ugyanakkor egyes osztályok és hierarchiák esetében szükség lehet memóriakezelést finomhangoló tagfüggvényeket írnunk.)

Az is nagyon valószínű, hogy a szabványos `operator new` és `operator delete` függvények helyettesítésével számtalan hibába fogunk ütközni, ugyanis a szabvá-

nyos könyvtár egyéb részei, valamint nagyon sok, harmadik fél által szállított könyvtár hibátlan működése e műveletek szabványos viselkedésén alapul.

Sokkal biztonságosabb a globális `operator new` túlterhelése, semmint helyettesítése. Tegyük fel például, hogy egy újonnan lefoglalt memóriaterületet egy adott karaktermintázattal akarunk feltölteni:

```
void *operator new( size_t n, const string &pat ) {
    char *p = static_cast<char *>(::operator new( n ));
    const char *pattern = pat.c_str();
    if( !pattern || !pattern[0] )
        pattern = "\0"; // Megjegyzés: két nulla karakter
    const char *f = pattern;
    for( int i = 0; i < n; ++i ) {
        if( !*f )
            f = pattern;
        p[i] = *f++;
    }
   return p;
}
```

Az `operator new` függvénynek ez a változata egy `string` típusú paraméterben várja a kitöltéshez használandó karaktermintát, így a fordító egyértelműen meg tudja különböztetni a szabványos `operator new` függvénytől a túlterhelés feloldása során.

```
string fill( "<garbage>" );
string *string1 = new string( "Hello" ); // Szabványos változat
string *string2 =
    new (fill) string( "World!" ); // Túlterhelt változat
```

A szabvány maga is meghatározza az `operator new` egy túlterhelt változatát, amely a szokásos `size_t` paraméter mellett egy második, `void *` típusút is vár. A függvény egyszerűen visszaadja a második paraméterét. (A megvalósításban látható `throw()` csupán azt jelzi, hogy a függvény nem adhat tovább kivételeket. Ez a további tárgyalást nem befolyásolja, nyugodtan figyelmen kívül hagyhatjuk.)

```
void *operator new( size_t, void *p ) throw()
    { return p; }
```

Ez a szabványos „irányított `new`" művelet, amely egy objektumot a megadott címen hoz létre. (A szabványos, egyparaméteres `operator new` függvénytől eltérően ezt tilos túlterhelni.) Ezt a függvényt tulajdonképpen a fordító becsapására használjuk,

és arra kényszerítjük vele, hogy meghívjon egy konstruktort. Egy beágyazott rendszer számára készült programban például szükségünk lehet arra, hogy egy „állapotregiszter" objektumot egy adott hardvercímen hozzunk létre:

```
class StatusRegister {
   // . . .
};
void *regAddr = reinterpret_cast<void *>(0XFE0000);
// . . .
// place register object at regAddr
StatusRegister *sr = new (regAddr) StatusRegister;
```

Természetesen az irányított `new` művelettel létrehozott objektumokat is meg kell egyszer semmisíteni. Ugyanakkor ez a művelet ténylegesen nem foglal tárat, így gondoskodnunk kell róla, hogy tényleges tártörlés se történjen. Emlékezzünk rá, hogy a szabványos `delete` művelet előbb meghívja a kérdéses objektum destruktorát, majd az `operator delete` függvény meghívásával szabadítja fel a korábban lefoglalt tárat. Ez pedig azt jelenti, hogy ha egy objektumot az irányított `new` művelettel hoztunk létre, a destruktort kifejezetten meg kell hívnunk a tárfelszabadítás elkerülése végett:

```
sr->~StatusRegister(); // A destruktor kifejezett meghívása,
➥ nincs operator delete
```

Bár az irányított `new` művelet a destruktorhívással párosítva igen hasznos eszköz lehet, veszélyessé is válhat, ha nem bánunk vele takarékosan és óvatosan. (Egy szabványos könyvtárral kapcsolatos ilyen jellegű problémáról szól a 47. hiba.)

Jegyezzük meg, hogy bár az `operator new` függvényt túlterhelhetjük, ezeket a változatokat a szabványos törlő kifejezések soha nem hívják meg:

```
void *operator new( size_t n, Buffer &buffer ); // Túlterhelt new
void operator delete( void *p,
   Buffer &buffer ); // A neki megfelelő delete
// . . .
Thing *thing1 = new Thing; // A szabványos operator new
➥ használata
Buffer buf;
Thing *thing2 = new (buf) Thing; // A túlterhelt operator new
➥ használata
delete thing2; // Helytelen. Itt a túlterhelt delete műveletet
➥ kellett volna használni.
delete thing1; // Helyes. A szabványos operator delete fut le.
```

Ehelyett – akárcsak az irányított `new` művelettel létrehozott objektumok esetében – itt is meg kell hívnunk az objektum destruktorát, majd a megfelelő `operator delete` függvénnyel felszabadítani a megfelelő tárterületet:

```
thing2->~Thing(); // Helyes. Töröljük a Thing objektumot.
operator delete( thing2, buf ); // Helyes. A túlterhelt delete
➥ műveletet használjuk.
```

A gyakorlatban az egyik leggyakoribb hiba az, hogy a szabványos `operator new` egy túlterhelt változatával létrehozott objektumot a szabványos globális `operator delete` segítségével próbálunk megsemmisíteni. Az egyik lehetséges módszer az ilyen hibák elkerülésére az, ha gondoskodunk róla, hogy az `operator new` minden túlterhelt változata valójában a szabványos globális `operator new` segítségével kapja meg a kívánt memóriaterületet. Pontosan ez az, amit a fenti megvalósítás során az első túlterhelt változatnál tettünk. Ez a változat hiba nélkül képes együttműködni a szabványos globális `operator delete` függvénnyel:

```
string fill( "<garbage>" );
string *string2 = new (fill) string( "World!" );
// . . .
delete string2; // Működik.
```

A globális `operator new` túlterhelt változataival tehát lehetőség szerint nem szabad ténylegesen tárfoglaló műveletet végeztetni. Ezeket a függvényeket inkább a szabványos globális változat „burkolóiként" kell használni.

Gyakran az a legjobb megoldás, ha teljesen elkerüljük a globális érvényű memóriakezelő függvények használatát. Helyettük a memóriakezelés testreszabásához az osztályok vagy az osztályhierarchiák szintjén használhatjuk a `new` és `delete` tagműveleteket, illetve ezek tömbökkel kapcsolatos változatait.

A 61. hibánál említettük, hogy a futásidőben működő rendszer mindig a „megfelelő" `operator delete` függvényt hívja meg, ha a `new` művelettel kapcsolatos kifejezések kiértékelésekor bármilyen kivétel keletkezik:

```
Thing *tp = new Thing( arg );
```

Ha a `Thing` számára történő tárfoglalás sikeres volt, de a konstruktor végrehajtása során kivétel keletkezik, a futásidejű rendszer automatikusan a megfelelő `operator delete` függvényt fogja meghívni a `tp` objektum által lefoglalt terület felszabadításához. A fenti példában azonban két „megfelelő" `operator delete` függvény is

van. Az egyik a globális `operator delete(void *)`, a másik az ugyanezzel a paraméterlistával rendelkező `operator delete` tagfüggvény. Ha egy másik `operator new` függvényt használnánk, az egy másik `operator delete` használatát írná elő:

```
Thing *tp = new (buf) Thing( arg );
```

Ebben az esetben a megfelelő törlő művelet a kétparaméteres túlterhelt változat, hiszen ez felel meg a `Thing` típusú objektum létrehozásakor használt túlterhelt `operator new` függvénynek. Most tehát az `operator delete( void *, Buffer &)` függvény a „megfelelő", és a futásidejű rendszer is ezt fogja meghívni.

Összefoglalva az elmondottakat, a C++ nagyfokú rugalmasságot mutat a memóriakezelő függvények terén, ez azonban óhatatlanul a programok összetettségét vonja maga után. A legtöbb igényt kielégítik a szabványos globális `operator new` és `operator delete` függvények; a bonyolultabb megoldásokat csak akkor célszerű használni, ha erre az adott feladat megoldásához egyértelműen szükség van.

63. hiba: A new és delete tagok hatókörének és környezetének összekeverése

Az osztályok tagjaiként megadott `operator new` és `operator delete` függvények az adott osztályba tartozó objektumok létrehozásakor és megsemmisítésekor futnak le. Az, hogy az ezen műveleteket tartalmazó kifejezések pontosan milyen környezetben bukkannak fel, lényegtelen:

```
class String {
 public:
    void *operator new( size_t ); // Tagként megadott operator new
    void operator delete( void * ); // Tagként megadott operator
➥ delete
    void *operator new[]( size_t ); // Tagként megadott operator
➥ new[]
    void operator delete [] ( void * ); // Tagként megadott
➥ operator delete[]
    String( const char * = "" );
    // . . .
};

void f() {
```

```
    String *sp = new String( "Heap" ); // A String::operator
➥ new-t használja
    int *ip = new int( 12 ); // Az ::operator new-t használja
    delete ip; // Az ::operator delete-et használja
    delete sp; // A String::operator delete-et használja
}
```

Megismétlem: lényegtelen, hogy a tárfoglalás milyen hatókörben meg végbe. A meghívandó függvényt egyedül a létrehozandó objektum típusa határozza meg:

```
String::String( const char *s )
    : s_( strcpy( new char[strlen(s)+1], s ) )
    {}
```

A fenti példában a karaktertömb számára ugyan a `String` osztály hatókörén belül foglalunk helyet, a lefoglaláshoz azonban a globális, tömbökre vonatkozó `new` műveletet használjuk, nem a `String` osztály ugyanilyen függvényét. Értelemszerűen egy `char` típusú változó nem lehet egyben `String` típusú is. A pontos meghatározás persze segíthet:

```
String::String( const char *s )
    : s_( strcpy( reinterpret_cast<char *>
        (String::operator new[](strlen(s)+1 )),s ) )
    {}
```

Nagyon szép lett volna a `String` osztály saját `operator new []` függvényére hivatkozni úgy, hogy valami ilyesmit írunk: `String::new char[strlen(s)+1]`. Sajnos azonban ez az írásmód nem használható. (Ugyanakkor használhatjuk a `::new` formát a globális `operator new` és `operator new []` függvényekhez, illetve a `::delete` írásmódot a globális `operator delete` és `operator delete []` függvényekhez való hozzáféréshez.)

64. hiba: Karakterlánc literálok kivételként való visszaadása

A C++ programozásról szóló tankönyvek szerzői között sok olyan van, akik a kivételkezelést a következő, karakterlánc literált visszaadó példával szemléltetik:

```
throw "Stack underflow!";
```

Persze ők is tudják, hogy ez elítélendő eljárás, de arra hivatkoznak, hogy csupán pedagógiai célzatú példáról van szó. Azt viszont már elfelejtik megemlíteni, hogy az a magatartásforma, amire itt közvetett módon a tanulókat csábítják, egyenes út a romlás felé.

Soha ne használjunk karakterlánc literált kivételt jelző objektumként! A keletkezett kivételobjektumokat a kód többi részének el is kell fognia, az elfogás alapja pedig a kivétel típusa és nem a tartalma:

```
try {
   // . . .
}
catch( const char *msg ) {
   string m( msg );
   if( m == "stack underflow" ) // . . .
   else if( m == "connection timeout" ) // . . .
   else if( m == "security violation" ) // . . .
   else throw;
}
```

Ha a kivételkezelő rendszert egyszerűen karakterláncok küldésére és elkapására alapozzuk, azzal gyakorlatilag semmiféle információt nem adunk át a kivétel jellegéről annak típusán keresztül. Az összes olyan ponton, ahol a kódnak ügyelnie kell a kivételekre, kénytelenek leszünk elfogni és megvizsgálni az összes lehetséges kivételt, hátha éppen arról van szó, amelyiket az adott ponton kell kezelni. És ami még ennél is rosszabb, az ilyen összehasonlítás az elgépelés melegágya. Még ha eredetileg helyes is a kód, a karbantartás során valakinek eszébe juthat, hogy az egyik ilyen „hibaüzenetben" valamit mégis inkább nagy kezdőbetűvel kellene írni, és máris hibás az összehasonlítás alapja. Fenti példánkban egy ilyen eset miatt például soha nem fogjuk észrevenni a verem alulcsordulását.

A fenti megállapítások természetesen nemcsak a karakterláncokra, hanem a nyelv által megadott minden szabványos típusra vonatkoznak. Egész (`int`) vagy lebegőpontos számok (`float`), karakterláncok (`string`), vagy – ha tényleg nagyon rossz napunk van – `float` típusokat tartalmazó vektorok (`vector`) vagy halmazok (`set`) kivételként való visszaadása ugyanilyen problémákat vet fel. Egyszerűen fogalmazva, a szabványos típusok kivételkezelésben való felhasználását az gátolja, hogy ha elkaptunk egy kivételt, nem tudjuk megmondani, hogy azt mi váltotta ki, és hogyan kell rá reagálnunk. Az egész helyzet olyan, mintha a kivételt kiváltó kódrészlet azt üzente volna: „Itt valami nagyon, nagyon rossz dolog történt. Találd ki, mi lehetett!" Ilyenkor aztán nincs más választásunk, mint belemenni egy kitalálósdiba, ahol általában veszítünk.

A kivételtípus olyan elvont adattípus, ami a kivétel jellegét adja meg. Megvalósítása semmiben sem különbözik a többi elvont adattípusétól: adj nevet egy fogalomnak, határozd meg azon elvont műveletek halmazát, amelyeket az adott típusra alkalmazni szeretnél, aztán valósítsd meg, amit kitaláltál. A megvalósítás során ügyelj a megfelelő beállításokra, a másoló műveletekre és a típusátalakításokra. Egyszerű. Karakterláncot kivételobjektumként használni körülbelül annyira értelmes, mintha ugyanezt a típust komplex számok leírására próbálnánk alkalmazni. Elméletileg működhet ugyan, de ami a gyakorlatot illeti, szinte biztos, hogy a megvalósítás hosszadalmas és unalmas lesz, és ráadásul tele lesz hibával.

Határozzuk meg, milyen elvont fogalmat takar a verem alulcsordulását jelző kivételobjektum! Hm.

```
class StackUnderflow {};
```

Egy ilyen objektum neve gyakran mindent elárul arról, hogy mire is vonatkozik, és az sem ritka a kivételkezelő típusokkal kapcsolatban, hogy egyetlen tagfüggvénnyel sem rendelkeznek. Ugyanakkor az a képesség, hogy az objektum maga valamiféle szöveges információval tudjon szolgálni a keletkezett kivételről, gyakran hasznos. Ugyanígy az is gyakori, hogy az objektum az eseménnyel kapcsolatos egyszerűbb információkat is hordoz:

```
class StackUnderflow {
 public:
   StackUnderflow( const char *msg = "stack underflow" );
   virtual ~StackUnderflow();
   virtual const char *what() const;
   // . . .
};
```

Ha megadunk ilyen, szöveges információt nyújtó függvényt, akkor célszerű azt `what`-nak nevezni, és fent látható paraméterezéssel virtuális tagfüggvényként bevezetni. Ez azért lényeges, mert az általunk megadott kivételtípusok így a nyelv szabványos kivételtípusaival azonos módon fognak működni, amelyek mind tartalmazzák ezt a függvényt. Tulajdonképpen az is célszerű, ha saját kivételtípusainkat valamelyik szabványos típusból vezetjük le:

```
class StackUnderflow : public std::runtime_error {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
        : std::runtime_error( msg ) {}
};
```

Ez a megadási mód lehetővé teszi, hogy a kivételt leíró objektumot akár `StackUnderflow`, akár a még általánosabb `runtime_error` formában adjuk vissza. A legáltalánosabb megadási mód pedig az `exception` (a `runtime_error` nyilvános alaposztálya). Általában jó ötlet, ha megadunk egy viszonylag általános, de nem szabványos kivételtípust. Ez a típus ugyanis később alapjául szolgálhat egy teljes modul vagy könyvtár összes kivételkezeléssel kapcsolatos típusának:

```
class ContainerFault {
 public:
   virtual ~ContainerFault();
   virtual const char *what() const = 0;
   // . . .
};
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
        : std::runtime_error( msg ) {}
   const char *what() const
        { return std::runtime_error::what(); }
};
```

Végezetül természetesen a kivételkezelő típusokhoz is meg kell adnunk a létrehozásukhoz, másolásukhoz és megsemmisítésükhöz szükséges függvényeket. A kivételkezelés módja mindenképpen szükségessé teszi a megfelelő objektumok másoló konstruktorainak létezését, hiszen a futásidejű kivételkezelő rendszer semmi egyebet nem tesz, mint másolással továbbítja a megfelelő objektumot. (Ezzel kapcsolatban olvassuk el a 65. hibát is.) Ha az objektum a másolások révén elérte a célját, végül nyilván meg kell semmisíteni. Gyakran az is elég, ha ezen műveletek megírását a fordítóra bízzuk (lásd a 49. hibát):

```
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
        : std::runtime_error( msg ) {}
   // StackUnderflow( const StackUnderflow & );
   // StackUnderflow &operator =( const StackUnderflow & );
   const char *what() const
        { return std::runtime_error::what(); }
};
```

Az általunk megadott veremtípus felhasználóinak immár többféle választása is van a veremműveletekkel kapcsolatos kivételek figyelésére és kezelésére. Használhat-

ják a `StackUnderflow` típust, ha viszonylag pontosan ismerik a veremtípus belső megvalósítását. Használhatják a valamivel általánosabb `ContainerFault` típust, ha tisztában vannak vele, hogy a mi tárolókönyvtárunkat használják, és ügyelnek az azzal kapcsolatos kivételekre. Használhatják a szabványos `runtime_error` formát, ha semmit sem tudnak a könyvtár felépítéséről, de kezelni akarják annak összes szabványos hibajelenségét. Végezetül használhatják az `exception` formát, ha egyszerűen csak minden szabványos kivételre fel akarnak készülni.

65. hiba: Nem megfelelő kivételkezelés

A C++ kivételkezelő eljárásai és a velük kapcsolatos rendszabályok máig vita tárgyát képezik. A kivételek kiváltásával és elfogásával kapcsolatos alapok jól átgondoltak, mégis általános jelenség, hogy a programozók figyelmen kívül hagyják őket. Ha valahol a programban kivétel keletkezik, a futásidejű kivételkezelő rendszer mindenekelőtt elteszi azt egy ideiglenes objektumba, ami valamilyen „biztos" helyen jön létre. Hogy ez a biztos hely pontosan hol is található, az erősen rendszerfüggő, azt azonban minden rendszer garantálja, hogy az említett objektum a kivétel kezeléséig megmarad. Ez azt jelenti, hogy az ideiglenes objektum egészen az utolsó kivételkezelő kifejezés lefutásáig megmarad, még akkor is, ha egymás után több különböző ponton kezeljük az adott eseményt. Ez igen fontos tulajdonság, hiszen – kereken kimondva – amikor egy programban hiba keletkezik, minden esélyünk megvan rá, hogy elszabadul a pokol. Ez a bizonyos ideiglenes objektum az utolsó mentsvárunk a kivételkezelés forgatagában.

És ezért nagyon rossz ötlet mutatót kivételkezelési célra használni.

```
throw new StackUnderflow( "operator stack" );
```

A kupacban levő `StackUnderflow` objektum címe ugyan biztos helyre kerül, azt a memóriaterületet azonban, amit ez a mutató címez, semmi sem védi. Ez a módszer annak a lehetőségét sem zárja ki, hogy a megcímzett terület a futásidejű veremben található:

```
StackUnderflow e( "arg stack" );
throw &e;
```

Ebben a példában az a terület, amelyre a kivételkezelő objektum hivatkozik (emlékezzünk rá, hogy mutatót adunk át kivételként, nem pedig azt, amire mutat) a ki-

vétel elfogásakor esetleg már nem is létezik. (Amúgy ha karakterláncot adunk át kivételkezelő objektumként, akkor annak teljes tartalma átmásolódik a biztonságos területen létrehozott ideiglenes objektumba, nemcsak az első karaktert címző mutató. Persze jobban meggondolva ennek az információnak nincs semmiféle gyakorlati haszna, hiszen amint azt a 64. hibánál kifejtettem, soha nem szabad karakterláncokat kivételkezelésre használni.) Ráadásul egy mutatónak jogában áll a null értéket tartalmazni. Végül is kinek jó ez a zűrzavar?! Használjunk objektumokat mutatók helyett, és egycsapásra minden megoldódik:

```
StackUnderflow e( "arg stack" );
throw e;
```

A kivételobjektum azonnal a biztonságos átmeneti tárba másolódik, így példánkban az `e` bevezetése szükségtelen. Általában névtelen ideiglenes objektumokat dobunk kivételként:

```
throw StackUnderflow( "arg stack" );
```

A névtelen objektum használata ebben a környezetben világosan jelzi, hogy a `StackUnderflow` objektumot kizárólag kivételkezelési célra akarjuk felhasználni, hiszen élettartama világos módon a kivételkezelés tartamára korlátozódik. Igaz ugyan, hogy az általunk bevezetett `e` objektumot is megsemmisíti a rendszer a kivételkezelés végén, az azonban érvényben marad és hozzáférhető egészen a bevezető blokk végéig. A névtelen ideglenes objektumok használata mellesleg gátat szab a „kreatív" kivételkezelési módszerek alkalmazhatóságának is:

```
static StackUnderflow e( "arg stack" );
extern StackUnderflow *argstackerr;
argstackerr = &e;
throw e;
```

Ebben a példában programozónk úgy döntött, hogy a kivételként dobott objektum címét elteszi valami biztos helyre, majd később – valószínűleg egy kivételkezelő eljárás során – felhasználja. Sajnos azonban az `argstackerr` nevű mutató a programozó várakozásával ellentétben nem a kivételobjektumra mutat (ami egy ideglenes objektum egy közelebbről meg nem határozott helyen), hanem az annak beállításához használt, immár nem létező másikra. Egy kivételkezelő eljárás minden valószínűség szerint nem a legmegfelelőbb hely arra, hogy efféle hibákkal tarkítsuk. Törekedjünk tehát az egyszerűségre.

Mi a legjobb módja egy kivételobjektum elfogásának? Nos, az érték szerinti vizsgálat nem jó:

```
try {
   // . . .
}
catch( ContainerFault fault ) {
   // . . .
}
```

Képzeljük csak el, mi történne, ha ez a függvény sikeresen elfogna egy kivételként dobott `StackUnderflow` objektumot. Kivágás! Mivel a `StackUnderflow` típus egyben `ContainerFault` is, nincs akadálya annak, hogy a `fault` változót az elfogott objektum alapján állítsuk be. Csakhogy ezzel a származtatott osztály összes adatát és viselkedésformáját kivágjuk belőle. (Ezzel kapcsolatban lásd a 30. hibát.)

A mi esetünkben a kivágás problémája nem jelentkezne, mivel a `ContainerFault` – amint az egy alaposztálytól el is várható – elvont (lásd a 93. hibát). A kivétel elfogását célzó megoldás viszont pontosan ezért hibás. Nem lehet érték szerint elfogni egy kivételobjektumot, mint `ContainerFault` típust.

Az érték szerinti elfogással amúgy még ennél is hajmeresztőbb dolgoknak tesszük ki magunkat:

```
catch( StackUnderflow fault ) {
   // Részleges helyreállítás...
   fault.modifyState(); // Az én hibám
   throw; // Az aktuális kivétel újbóli kiváltása
}
```

Nem ritka, hogy egy kivételkezelő eljárás csak részben oldja meg a problémát, majd az elfogott kivételobjektumot továbbadja további feldolgozásra. Sajnos a mi esetünkben nem ez történik. A mi függvényünk részlegesen kijavítja a hibát, egy helyi objektumban rögzíti a javítás eredményeként előálló állapotot, majd a változatlan objektumot adja tovább.

Mindezek elkerülése végett mindig névtelen objektumokat, és mindig hivatkozással adjunk tovább a kivételkezelés során.

Ügyeljünk arra is, hogy ne okozzunk a részleges kezelés során az értékek téves másolásával problémát. Ez a hibatípus általában akkor bukkan fel, ha a hibakezelő nem ugyanazt az objektumtípust adja tovább, mint amit elfogott:

```
catch( ContainerFault &fault ) {
    // Részleges helyreállítás...
    if( condition )
        throw; // Újbóli kiváltás
    else {
        ContainerFault myFault( fault );
        myFault.modifyState(); // Még mindig az én hibám
        throw myFault; // Új kivételobjektum
    }
}
```

Ebben az esetben az elvégzett javítások listája ugyan nem vész el, az eredetileg elfogott objektum típusa azonban igen. Tegyük fel, hogy az objektum típusa `StackUnderflow` volt. Ha `ContainerFault` típusú hivatkozáson keresztül fogjuk el, az objektum dinamikus típusa továbbra is `StackUnderflow` marad, így ha továbbadjuk, utólag egyaránt kezelheti a `StackUnderflow` típust, illetve a `ContainerFault` típust kezelő függvény is. Ugyanakkor a példánkban továbbadott `myFault` objektum `ContainerFault` típusú, így nem foghatja el egy `StackUnderflow` típusra várakozó kezelőfüggvény. Célszerűbb tehát új objektumok létrehozása helyett a már meglevőt továbbadni a többlépcsős kezelési eljárás fokozatai között.

```
catch( ContainerFault &fault ) {
    // Részleges helyreállítás...
    if( !condition )
        fault.modifyState();
    throw;
}
```

Szerencsénkre a `ContainerFault` alaposztály elvont, így az említett hiba megjelenése példánkban tulajdonképpen nem is lehetséges. Az alaposztályoknak jó tervezés esetén általában is elvontnak kell lenniük. Természetesen a kivételobjektumok továbbadásával kapcsolatban megfogalmazott tanácsok érvényüket vesztik, ha valóban más objektumtípust kell továbbadnunk a helyreállítás során:

```
catch( ContainerFault &fault ) {
    // Részleges helyreállítás...
    if( out_of_memory )
        throw bad_alloc(); // Új kivétel kiváltása
    fault.modifyState();
    throw; // Újbóli kiváltás
}
```

Egy másik, a kivételkezeléssel kapcsolatban általánosnak nevezhető probléma a kivételkezelő függvények elrendezése. Mivel ezek a megadás sorrendjében értékelődnek ki (akár az `if-elseif` szerkezetek logikai feltételei), a típusok vizsgálata

során általában az egyeditől az általános felé kell haladni. Az olyan kivételtípusoknál, amelyek az efféle sorrendiség megállapítását nem teszik lehetővé, valamilyen logikai sorrend felállításával próbálkozhatunk:

```
catch( ContainerFault &fault ) {
    // Részleges helyreállítás...
    fault.modifyState(); // Nem az én hibám
    throw;
}
catch( StackUnderflow &fault ) {
    // . . .
}
catch( exception & ) {
    // . . .
}
```

A fent látható kivételkezelő sorozat soha nem fogja a `StackUnderflow` objektumot kezelni, mivel azt a sorban megelőzi az általánosabb `ContainerFault` típus.

Példáinkból látható, hogy a kivételkezelés megvalósítása erősen csábít az összetettség felé. Nem kell azonban feltétlenül engednünk ennek a csábításnak. Mint annyi más helyen, a kivételek dobása és elfogása terén is törekedjünk az egyszerűségre.

66. hiba: Helyi címek téves kezelése

Ne adjunk vissza helyi változót címző mutatót vagy hivatkozást. A legtöbb fordító az ilyen hibákat észre is veszi, és figyelmeztet rájuk. Vegyük komolyan ezeket a figyelmezetéseket.

Eltűnő veremterületek

Ha egy változó automatikus, akkor a veremben jön létre, a függvény visszatérésekor pedig az általa elfoglalt terület eltűnik:

```
char *newLabel1() {
    static int labNo = 0;
    char buffer[16]; // Lásd a 2. hibát
    sprintf( buffer, "label%d", labNo++ );
    return buffer;
}
```

Ennek a függvénynek megvan az a bosszantó tulajdonsága, hogy néha működik. Visszatérése után a `newLabel1` függvény számára a verem tetején lefoglalt terület felszabadul (a `buffer` nevű karaktertömb helyével együtt), így a soron következő függvény szabadon felhasználhatja a felszabadult területet. Ha azonban a mutató által címzett értéket még a következő függvényhívás előtt lemásoljuk, jó esély van rá, hogy – bár a mutató már érvénytelen – a kód helyesen működik:

```
char *uniqueLab = newLabel1();
char mybuf[16], *pmybuf = mybuf;
while( *pmybuf++ = *uniqueLab++ );
```

Ez persze nem olyan gond, amit egy sokat megélt karbantartó ne fedezne fel. Ő aztán feltehetőleg úgy kerüli majd meg a hibát, hogy a szükséges területet dinamikusan foglalja le a szabad tárból:

```
char *pmybuf = new char[16];
```

A karbantartó azt is elhatározhatja, hogy az adott terület átmásolását nem maga oldja meg:

```
strcpy( pmybuf, uniqueLab );
```

Aztán még tovább gondolkodik, és úgy határoz, hogy az egyszerű karaktertömb helyett mégis inkább valamilyen elvont típust használ:

```
std::string mybuf( uniqueLab );
```

A fenti módosítások bármelyike okozhatja a `uniqueLab` által címzett helyi tárterület módosulását.

Ütközés a statikus változókkal

Ha a visszaadott változó statikus, az adott függvény egy későbbi hívása módosíthatja a korábbi hívások eredményét:

```
char *newLabel2() {
   static int labNo = 0;
   static char buffer[16];
   sprintf( buffer, "label%d", labNo++ );
   return buffer;
}
```

Bár a kérdéses tárterület a függvény visszatérése után is hozzáférhető, az eredményt a függvény bármilyen későbbi használata módosíthatja:

```
// 1. eset
cout << "first: " << newLabel2() << ' ';
cout << "second: " << newLabel2() << endl;

// 2. eset
cout << "first: " << newLabel2() << ' '
     << "second: " << newLabel2() << endl;
```

Az első esetben két különböző címkét fogunk kiíratni. A másodikban viszont valószínűleg (bár nem feltétlenül) két azonosat. Az első példát valószínűleg egy olyan kódkarbantartó írta, aki csendben felfedezte a `newLabel2` furcsa megvalósítását, kettébontotta a címkekiíratás műveletét, és ezzel elhelyezte a taposóaknát a kódban. Az utána következő karbantartó már nem volt ennyire figyelmes, nem ismerte a `newLabel2` megvalósítása körüli szeszélyes viszonyokat, újra egyesítette a két műveletet, és ezzel hibát okozott a kód működésében. És ami a legrosszabb, az így keletkezett egyesített kód képes a korábbi változat működését utánozni, viszont nem lehet tudni, mikor vált át hibás működésbe (lásd még a 14. hibát).

Nyelvi nehézségek

Van ezen a környéken még egy ránk leselkedő veszély. Tartsuk mindig szem előtt, hogy egy függvény felhasználói általában nem férnek hozzá annak forráskódjához, így a visszatérési érték kezelésének módját a függvény deklarációjából kénytelenek kisütni. Egy alkalmas megjegyzés természetesen sokat segíthet (lásd az 1. hibát), de azt se felejtsük el, hogy egy függvényt úgy kell megírni, hogy használatának módjából természetesen következzen annak helyessége.

Kerüljük például olyan hivatkozás visszaadását, ami a függvény által lefoglalt memóriaterülethez enged hozzáférni. Ez ugyanis a függvény felhasználóit arra fogja csábítani, hogy ne foglalkozzanak a lefoglalt terület felszabadításával, ez pedig a használható tárterület vészes csökkenését okozhatja:

```
int &f()
   { return *new int( 5 ); }
// . . .
int i = f(); // Fogyatkozó memória
```

A helyes kódnak a hivatkozást egy címmé kell alakítania, vagy másolatot kell készítenie az eredményről, majd felszabadítani a lefoglalt területet. No de mit nekünk egy ilyen tanács:

```
int *ip = &f(); // Egy rettenetes eljárás
int &tmp = f(); // Egy másik
int i = tmp;
delete &tmp;
```

Ez kifejezetten rossz ötlet, ha túlterhelt műveletfüggvényben használjuk:

```
Complex &operator +( const Complex &a, const Complex &b )
   { return *new Complex( a.re+b.re, a.im+b.im ); }
// . . .
Complex a, b, c;
a = b + c + a + b; // A memória több sebből vérzik
```

Vagy adjunk vissza egy, az adott területet címző mutatót a terület tartalma helyett, vagy ne foglaljunk területet és adjuk vissza magát az értéket:

```
int *f() { return new int(5); }
Complex operator +( Complex a, Complex b )
   { return Complex( a.re+b.re, a.im+b.im ); }
```

Az általános szokások szerint, ha egy függvénytől mutatót kapunk vissza, akkor mindig megvan az a rossz érzésünk, hogy az általa címzett területet nekünk kell majd felszabadítanunk. Ez aztán a legtöbbünket arra késztet, hogy utánanézzünk a dolognak, például úgy, hogy legalább a függvényhez mellékelt megjegyzést elolvassuk. Egy hivatkozást visszaadó függvénnyel kapcsolatban az emberek többségének valahogy nincsenek ilyen aggályai.

Gondok a helyi hatókörrel

A helyi változók élettartamával kapcsolatos gondokkal nem csak függvények határán, hanem akár egyetlen függvényen belül is találkozhatunk:

```
void localScope( int x ) {
   char *cp = 0;
   if( x ) {
       char buf1[] = "asdf";
       cp = buf1; // Rossz ötlet.
```

```
        char buf2[] = "qwerty";
        char *cp1 = buf2;
        // . . .
    }
    if( x-1 ) {
        char *cp2 = 0; // buf1 fölé kerül?
        // . . .
    }
    if( cp )
        printf( cp ); // Talán hiba...
}
```

A fordítóprogramoknak viszonylag nagy szabadságuk van abban a tekintetben, hogy hogyan oldják meg a helyi változók elrendezését a tárban. A rendszertől és a fordítás során használt kapcsolóktól függően megeshet, hogy a `buf1` és a `cp2` ugyanarra a tárterületre kerül. A fordítónak minden joga megvan ehhez az elrendezéshez, hiszen e két változó hatóköre és élettartama között nincs átfedés. Ha ez az egymásra íródás megvalósul, a `buf1` tartalma hibás lesz, és ez a `printf` függvény viselkedésére is rányomja a bélyegét (valószínűleg nem fog semmit kiírni). A hordozhatóság érdekében tehát igyekezzünk elkerülni, hogy kódunk működése a verem elrendezésétől függjön.

A statikus „javítási módszer"

Ha igazán nehezen felderíthető hibával kerülünk szembe, gyakori jelenség, hogy a hiba „elmúlik", ha a static tárolási osztályra térünk át:

```
// . . .
char buf[MAX];
long count = 0;
// . . .
int i = 0;
while( i++ <= MAX )
    if( buf[i] == '\0' ) {
        buf[i] = '*';
        ++count;
    }
assert( count <= i );
// . . .
```

Ez a kód egy ügyetlenül megírt ciklust tartalmaz, amely időnként a `buf` tömb végének elérése után beleír a `count` változóba, és ezzel a megfogalmazott logikai feltételt hamissá teszi. A hibakeresés hevében gyakran tapasztalható kapkodásnak kö-

szönhetően aztán a programozó – többek között – megpróbálja a count-ot helyi statikus változóként bevezetni, mire a program – döbbenetes módon – működni kezd:

```
char buf[MAX];
static long count;
// . . .
count = 0;
int i = 0;
while( i++ <= MAX )
    if( buf[i] == '\0' ) {
        buf[i] = '*';
        ++count;
    }
assert( count <= i );
```

Sok olyan programozó van, aki ezek után már nem is akarja tudni, miért volt ekkora szerencséje, és egyszerűen továbblép. Sajnos azonban a probléma valójában nem szűnt meg, csak máshová került. Aztán csendben vár, hogy később még nagyobb fejfájást okozhasson.

Azzal, hogy a count változót statikusként vezettük be, a számára lefoglalt memóriaterületet a veremből egy, az ilyen változók tárolására szolgáló másik memóriaterületre helyeztük át. Mivel megváltozott a helye, a hibásan megírt ciklus már nem tudja felülírni. A gond csak az, hogy ettől maga a ciklus még hibás marad, és immár egy másik helyi változót, vagy esetleg egy később megadandó változót fog felülírni. Ezzel pedig ugyanaz a helyzet áll elő, mint amit fentebb a helyi változókkal kapcsolatban bemutattunk. A helyes megoldás – mint általában – most is az, hogy meg kell szüntetni a hibát, ahelyett, hogy elrejtenénk:

```
char buf[MAX];
long count = 0;
// . . .
for( int i = 1; i < MAX; ++i )
    if( buf[i] == '\0' ) {
        buf[i] = '*';
        ++count;
    }
// . . .
```

67. hiba: „Az erőforrások lefoglalása egyben kezdeti beállítás" elv

Botrányos, hogy a fiatal C++ programozók mennyire nem látják át, és mennyire nem méltányolják a konstruktorok és destruktorok használatának gyönyörű szimmetriáját. Ezek az emberek javarészt olyan nyelveken nevelkedtek, amely távol tartotta őket a mutatók és a memóriakezelés „borzalmaitól", hamis biztonságérzetet adva ezzel. Biztonságban voltak és szabályok korlátozták őket. A tudatlanok boldog életét élhették. Úgy programoztak, ahogy azt a nyelv tervezője előírta számukra. Ez volt az egyetlen, az igaz út. Az ő útjuk.

Szerencsére a C++ ennél sokkal jobban tiszteli a programozó szabad akaratát, és ennek megfelelően sokkal nagyobb szabadságot biztosít számára a nyelvi szolgáltatások használata terén. Természetesen ezzel nem azt akarom mondani, hogy itt nincsenek alapvető szabályok vagy irányelvek (ezzel kapcsolatban lásd a 10. hibát). Ami azt illeti, az alapelvek közül az egyik legfontosabb „az erőforrás lefoglalása egyben kezdeti beállítás" (resource acquisition is initialization) elv. Ez így elsőre amolyan aranyköpésnek tűnik, de egyben egyszerű és továbbfejleszthető módszer az erőforrások és a memória összekapcsolására, és a kettő hatékony és jól tervezhető kezelésére.

A konstruktorok és a hozzájuk tartozó destruktorok működése egymás pontos tükörképe. Amikor egy adott osztályba tartozó objektum létrejön, az egyes elemek kezdeti beállításának sorrendje mindig ugyanaz. Előbb a virtuális alaposztályok alobjektumai jönnek (a szabvány szerint: „bevezetésük sorrendjében, úgy, hogy a legmélyebben fekvőtől balról jobbra haladunk az alaposztályok hurokmentes irányított gráfján"), ezután következnek a közvetlen alaposztályok az osztály meghatározásában levő alaplistában való említésük sorrendjében. Ezt követi a nem statikus adattagok beállítása bevezetésük sorrendjében, végül pedig a konstruktor törzsében szereplő kód fut le. A destruktor mindezt éppen fordítva csinálja. Előbb a függvény törzse fut le, ezt követi a tagok törlése bevezetésük fordított sorrendjében, majd a közvetlen alaposztályok következnek megjelenésük fordított sorrendjében, végül pedig a virtuális alaposztályok semmisülnek meg. Sokat segíthet, ha az egész eljárást úgy képzeljük el, mintha egy műveleti vermet kezelnénk. Az objektum létrehozása elhelyez bizonyos műveleteket a veremben, a törlés pedig „kiveszi" azokat onnan. A C++ nyelv e belső szimmetriája annyira fontos, hogy az említett műveletek még akkor is ebben a sorrendben mennek végbe, ha a tagbeállító listán más sorrendet adunk meg. (Erről már volt szó az 52. hiba kapcsán.)

A kezdeti beállítás egyfajta mellékterméke, avagy következményeként a konstruktor erőforrásokat foglal le az objektum számára. A lefoglalás sorrendje számos esetben lényeges. (Például nem írhatunk egy adatbázisba addig, amíg nem zároltuk, vagy nem írhatunk egy fájlba, amíg nincs hozzá fájlkezelőnk.) A destruktor feladata ezen erőforrások felszabadítása lefoglalásuk fordított sorrendjében. Az a tény, hogy egy objektumnak több konstruktora lehet úgy, hogy közben destruktora csak egy van, eleve feltételezi, hogy a műveleti sorrendnek mindig azonosnak kell lennie.

(Ami azt illeti, ez nem volt mindig így. A C++ nyelv fejlődésének korai szakaszában a beállítások és törlések sorrendje nem volt kötött, ami aztán komoly gondot okozott az összetettebb programok írásánál. Mint a C++ megannyi más szabálya, a kötött beállítási sorrend is részben az átgondolt tervezés, részben a gyakorlati tapasztalatok eredménye.)

A beállítás és felszámolás sorrendi szimmetriája nemcsak az osztályok megadására, hanem az összetett osztályrendszerek használatára is érvényesül. Vegyük például a következő egyszerű nyomkövető osztályt:

➡ 67. hiba/trace.h

```
class Trace {
 public:
   Trace( const char *msg )
        : m_( msg ) { cout << "Entering " << m_ << endl; }
   ~Trace()
        { cout << "Exiting " << m_ << endl; }
 private:
   const char *m_;
};
```

Ez az osztály talán túlságosan is egyszerű, hiszen vakon feltételezi, hogy az őt létrehozó objektum érvényes, és élettartama kiterjed legalább a `Trace` objektum élettartamára. Példának azért jó lesz. A `Trace` objektum megjelenít egy szöveget, amikor létrehozzák, és egy másikat, amikor megsemmisül. Ennek megfelelően alkalmas egy program végrehajtásának követésére:

➡ 67. hiba/trace.cpp

```
Trace a( "global" );
void loopy( int cond1, int cond2 ) {
   Trace b( "function body" );
it: Trace c( "later in body" );
   if( cond1 == cond2 )
        return;
   if( cond1-1 ) {
        Trace d( "if" );
```

```
        static Trace stat( "local static" );
        while( --cond1 ) {
            Trace e( "loop" );
            if( cond1 == cond2 )
                goto it;
        }
        Trace f( "after loop" );
    }
    Trace g( "after if" );
}
```

Ha a `loppy` függvényt a `4` és `2` paraméterekkel hívjuk meg, az eredmény a következő lesz:

```
Entering global
Entering function body
Entering later in body
Entering if
Entering local static
Entering loop
Exiting loop
Entering loop
Exiting loop
Exiting if
Exiting later in body
Entering later in body
Exiting later in body
Exiting function body
Exiting local static
Exiting global
```

Az üzenetek világosan mutatják, miként kapcsolódik a `Trace` objektum élettartama a végrehajtás menetéhez. Különösen érdemes megfigyelni a `goto` és a `return` hatását a pillanatnyilag aktív `Trace` típusú objektumok élettartamára. Természetesen egyik megvalósítás sem követendő példaként áll itt, bár mindkettő olyan, ami karbantartáskor fel-felbukkan a kódokban.

```
void doDB() {
    lockDB();
    // Műveleteket végzünk az adatbázissal
    unlockDB();
}
```

A fenti programban ügyeltünk arra, hogy az adatbázist a módosítás előtt zároljuk, sőt arra is, hogy miután végeztünk, töröljük a zárolást. Sajnos ez a figyelmesség a karbantartás alatt gyakorta sérül, különösen, ha a két művelet közti programblokk hosszú:

```
void doDB() {
    lockDB();
    // . . .
    if( i_feel_like_it )
        return;
    // . . .
    unlockDB();
}
```

Most bizony már van egy hibánk, ami mindannyiszor bekövetkezik, amikor a `doDB` függvény „úgy gondolja". Ilyenkor ugyanis az adatbázis zárolt állapota megmarad, és a program egy teljesen más pontján fog gondot okozni. Tulajdonképpen már az eredeti kód sem volt tökéletes, mivel az adatbázis zárolása és felszabadítása között kivétel is keletkezhetett volna. Ennek pedig ugyanaz a hatása: az adatbázis zárolt marad.

Megpróbálhatnánk persze kiküszöbölni ezt az eshetőséget a kivételek elfogásával és azzal, hogy a későbbi karbantartókat megfelelően kioktatjuk a programmal kapcsolatban:

```
void doDB() {
    lockDB();
    try {
        // Műveleteket végzünk az adatbázissal
    }
    catch(  . . .  ) {
        unlockDB();
        throw;
    }
    unlockDB();
}
```

Ez a megközelítés szószátyár, alacsony technikai színvonalra utal, lassú, nehezen karbantartható, és attól tartok, kollégáink megbecsülését sem fogjuk kivívni vele. A megfelelően megírt, a kivételek kezelésére felkészített kód általában kevés `try` blokkot tartalmaz. Ehelyett a címben említett „az erőforrás lefoglalása egyben kezdeti beállítás" elvet alkalmazza:

```
class DBLock {
 public:
```

```
    DBLock() { lockDB(); }
    ~DBLock() { unlockDB(); }
};

void doDB() {
    DBLock lock;
    // Műveleteket végzünk az adatbázissal
}
```

A `DBLock` objektum létrehozásával annak konstruktora lefoglalja az adatbázis zárolásával kapcsolatos erőforrást. Ha aztán ez az objektum bármilyen okból kifolyólag megszűnik létezni, destruktora az alapértelmezett viselkedésnek megfelelően automatikusan felszabadítja az adatbázist. Ez a módszer annyira elterjedt a C++ programozás gyakorlatában, hogy gyakorta már észre sem vesszük. Mégis valahányszor a szabványos `string`, `vector`, `list` típusokkal, vagy bármilyen más, egyéb típusok származtatására használatos típussal dolgozunk, hallgatólagosan ezt az elvet használjuk.

Egyébként ügyeljünk két olyan általános hibára, amelyek a `DBLock`-hoz hasonló erőforrás-kezelő objektumokkal kapcsolatban bukkannak fel:

```
void doDB() {
    DBLock lock1; // Helyes.
    DBLock lock2(); // Hoppá!
    DBLock(); // Hoppá!
    // Műveleteket végzünk az adatbázissal
}
```

A `lock1` objektumot helyesen vezettük be. Ennek a `DBLock` típusú objektumnak a hatóköre a bevezetést lezáró pontosvesszőtől annak a blokknak a végéig (jelen esetben a függvény végéig) tart, amely a deklarációt tartalmazza. A `lock2` azonban már olyan függvény, amely nem vár bemenő paramétereket és egy `DBLock` típusú objektumot ad vissza (lásd a 19. hibát). Formailag ez nem hiba, de valószínűleg nem ezt akartuk, hiszen itt zárolás nem történik. A következő sor egy kifejezés-utasítás, ami egy névtelem `DBLock` típusú objektumot hoz létre. Ez ugyan tényleg zárolja az adatbázist, de mivel az objektum élettartamának még a záró pontosvessző előtt vége szakad, a zárolás azonnal meg is szűnik. Formailag tehát ez a sor is helyes, de megint nagyon valószínű, hogy nem ez az, amit akartunk.

A szabványos `auto_ptr` sablon egy rendkívül hasznos, általános célú erőforrás-kezelő szolgáltatás, amely leginkább a szabad tárban létrehozott objektumokkal kapcsolatban alkalmazható. Ezzel kapcsolatban olvassuk el a 10. és 68. hibákról szóló részeket.

68. hiba: Az auto_ptr nem megfelelő használata

A szabványos `auto_ptr` sablon egyszerű és hasznos segédeszköz, ám a másolást szokatlanul értelmezi (lásd a 10. hibát). Az `auto_ptr` használata többnyire magától értetődő:

```
template <typename T>
void print( Container<T> &c ) {
    auto_ptr< Iter<T> > i( c.genIter() );
    for( i->reset(); !i->done(); i->next() ) {
        cout << i->get() << endl;
        examine( c );
    }
    // rejtett takarítás...
}
```

Ebben a példában az `auto_ptr` egy általános felhasználási módját mutatjuk be. Itt az a cél, hogy a szabad tárba helyezett objektumok területe automatikusan felszabaduljon, ha a velük kapcsolatos mutató bármilyen okból kifolyólag érvényét veszti. (A `Container` hierarchia részletesebb elemzését a 90. hibánál találjuk.) Példánk alapfeltevése szerint a `genIter` által visszaadott `Iter<T>` területét a szabad tárból dinamikusan foglaltuk le. Ennek megfelelően az `auto_ptr(Iter<T>)` automatikusan meghívja a `delete` műveletet, és felszabadítja a kérdéses területet, ha az `auto_ptr` érvényessége megszűnik.

Ugyanakkor van két általános hiba, amit az `auto_ptr` használatával kapcsolatban el szoktak követni a programozók. Az egyik az a téves feltételezés, hogy az `auto_ptr` mutathat tömbre:

```
void calc( double src[], int len ) {
    double *tmp = new double[len];
    // . . .
    delete [] tmp;
}
```

Az itt látható `calc` függvény meglehetősen érzékeny a hibákra, hiszen a `tmp` számára lefoglalt terület nem szabadul fel, ha a függvény végrehajtása során kivétel keletkezik, vagy ha egy későbbi karbantartás során belekerül egy túl korai kiléptető parancs. Itt tehát nyilvánvalóan egy erőforrás-kezelőre van szükség, és a szabványos erőforrás-kezelő az `auto_ptr`:

```
void calc( double src[], int len ) {
   auto_ptr<double> tmp( new double[len] );
   // . . .
}
```

A gond az, hogy a szabványos `auto_ptr` csak egyetlen objektumra mutathat, objektumok tömbjére nem. Ennek az a következménye, hogy amikor a `tmp` tömb érvényessége megszűnik, nem a tömbökre, hanem a skalárokra vonatkozó törlő utasítás fut majd le. Pedig esetünkben `double` típusok egy egész tömbjét kellene törölni, amit természetesen a tömbökre vonatkozó `new` művelettel hoztunk létre (lásd a 60. hibát). A fordítónak sajnos nincs esélye rá, hogy ezt a hibát felfedezze, ugyanis nem képes megkülönböztetni egy tömb, illetve egyetlen objektum mutatóját. Még ennél is nagyobb baj, hogy ez a kód számos rendszeren valószínűleg működni fog, és a hibára csak akkor derül fény, amikor programunkat egy másik operációs rendszeren akarjuk lefordítani, vagy új változatát állítjuk elő egy már kipróbált rendszeren.

Esetünkben a helyes megoldás a szabványos `vector` típus használata a közönséges tömb helyett. Ez a típus tulajdonképpen egy tömbökre vonatkozó erőforrás-kezelő, vagyis egyfajta „`auto_array`", de számos hasznos kiegészítő szolgáltatást nyújt. Ugyanakkor valószínűleg jó ötlet, ha megszabadulunk attól a primitív és veszélyes szokásunktól, hogy tömbnek álcázott mutatót használjunk formális paraméterként:

```
void calc( vector<double> &src ) {
   vector<double> tmp( src.size() );
   // . . .
}
```

Egy másik gyakori hiba, hogy az `auto_ptr` típust a szabványos sablonkönyvtár elemtípusaként használjuk. A könyvtár tároló objektumai nem sokat követelnek meg az alkalmazható elemtípusoktól, a másolás hagyományos értelmezését azonban igen.

A nyelvi szabvány amúgy kifejezetten tiltja, hogy a szabványos sablonkönyvtár alapján `auto_ptr` típusú tároló objektumot hozzunk létre. Az efféle próbálkozás normális esetben fordítási hibát eredményez (bár a hibaüzenetet valószínűleg nem lesz könnyű megérteni). Ugyanakkor arra sem árt felkészülni, hogy számos, jelenleg forgalomban levő C++ fordító még nem mindenben felel meg a szabványnak.

Ha véletlenül éppen egy ilyen elavult fordítóval van dolgunk, előfordulhat, hogy az általa ismert `auto_ptr` megvalósítása még megfelelő másolást alkalmaz ahhoz, hogy a fenti – immár nem szabványos – célra alkalmas legyen. Ez az állapot egé-

szen addig fog tartani, amíg le nem cseréljük az általunk használt szabványos könyvtárat egy újabb változatra. Akkor a hiba hirtelen előjön, hiszen programunkat már nem lehet lefordítani. Meglehetősen idegesítő, de viszonylag egyszerűen javítható, feltéve persze, hogy rájövünk, mi a baj.

Ennél egy fokkal rosszabb, ha az `auto_ptr` megvalósítása nem teljesen szabványos, így megengedi a sablonokkal kapcsolatos használatot, de a másolás értelmezése valójában nem felel meg azok elvárásainak. Amint azt a 10. hiba kapcsán leírtuk, egy `auto_ptr` lemásolása átadja a megcímzett objektum feletti vezérlést, a másolás forrásaként használt mutatót pedig nullára állítja:

```
auto_ptr<Employee> e1( new Hourly );
auto_ptr<Employee> e2( e1 );  // e1 tartalma null
e1 = e2; // e2 tartalma null
```

Ez a tulajdonság számos helyzetben nagyon hasznos lehet, de nem felel meg a szabványos sablonkönyvtár elvárásainak:

```
vector< auto_ptr<Employee> > payroll;
// . . .
list< auto_ptr<Employee> > temp;
copy( payroll.begin(), payroll.end(), back_inserter(temp) );
```

Ez a kód számos rendszeren lefordítható, sőt még futtatható is, de valószínűleg nem azt csinálja, amit elvárnánk tőle. Az `Employee` mutatókat tartalmazó `vector` át fog másolódni a `list` objektumba, de ezek után a `vector` már csupa nulla mutatót fog tartalmazni!

Összefoglalva tehát kerüljük az `auto_ptr` szabványos sablonkönyvtári tárolóként való használatát még akkor is, ha az általunk használt rendszer ezt lehetővé tenné.

7

Többalakúság

Az elvont adatábrázolás, az öröklés és a többalakúság (poliformizmus) együttesen képezik az objektumközpontú programozás alapját. A C++ nyelvben a többalakúság megvalósítása rugalmas, de összetett.

Ebben a fejezetben azokat a hibákat mutatjuk be, amelyek a többalakúságnak e két fent említett tulajdonságával kapcsolatosak. Megmutatjuk természetesen azokat a módszereket is, amelyekkel úrrá lehetünk az összetettségen. A tárgyalás során bemutatjuk az öröklés és a virtuális függvények megvalósítását, valamint hogy ez a megvalósítás milyen hatással van magára a C++ nyelvre.

69. hiba: Típuskódok

Az „első C++ programom" tünetcsoport felbukkanásának legbiztosabb jele az, ha egy osztály tagjai között találunk egy típuskódot. (Én is használtam ezt a „trükköt" az első programomban, és én is átestem mindazokon a látszólag szűnni nem akaró problémákon, amelyeket okozott.) Az objektumközpontú programozásban egy objektum típusát annak viselkedése és nem az állapota határozza meg. Egy jól megtervezett C++ programban a legritkább esetben van szükség típuskódra, arra pedig soha, hogy ezt a típuskódot osztály adattagjaként tároljuk.

```
class Base {
 public:
   enum Tcode { DER1, DER2, DER3 };
   Base( Tcode c ) : code_( c ) {}
   virtual ~Base();
   int tcode() const
       { return code_; }
   virtual void f() = 0;
 private:
   Tcode code_;
};
class Der1 : public Base {
 public:
   Der1() : Base( DER1 ) {}
   void f();
};
```

A fent látható kód a probléma jellegzetes megjelenését mutatja. A gond itt tulajdonképpen az, hogy a programozó még nem mozog elég otthonosan az új, objektumközpontú világban, és nem mer teljesen arra a dinamikus kötési rendszerre támasz-

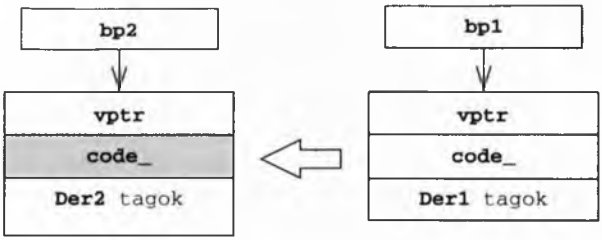
kodni, ami a jól megtervezett hierarchiákban gond nélkül működik. A típuskód arra az esetre került bele a kódba (gondolja programozónk), ha egyszer netán el kellene döntenünk mondjuk egy `switch` (ez is a jó öreg C nyelv hagyatéka) segítségével, hogy a `Base` osztálynak pontosan melyik típusával van dolgunk. Hibás gondolat! A típuskódok szerepeltetése egy objektumközpontú környezetben olyan, mintha búvárkodás során úgy próbálnánk lemerülni, hogy egyik lábunkat végig a parton tartjuk.

A C++-ban soha nem hagyatkozunk típuskódokra a döntések során, legalábbis a kód objektumközpontú részeiben. Soha. A probléma lényege azonnal nyilvánvaló, ha megvizsgáljuk a `Base` osztály `Tcode` felsoroló típusát. Ha változik a forráskód és a `Base` osztályból más osztályokat kell származtatnunk, az alaposztály tudni fog ezekről, és hozzájuk lesz csatolva. Arra azonban semmi biztosíték nincs, hogy az összes már létező, a `Tcode` elemei alapján működő kódot megfelelően át fogjuk (fogják) alakítani. A hagyományos C programok karbantartásával kapcsolatos egyik általános gond az, hogy a típuskódokra támaszkodó kódrészleteknek csak a 98 százalékát alakítják át megfelelően a programozók. Ez az a probléma, ami a virtuális függvényekkel kapcsolatban egyszerűen fel sem merülhet. Ezek után a programozónak talán nem kellene mindenféle erőfeszítéseket tennie annak érdekében, hogy egy, a nyelv által kiküszöbölt hibalehetőséget újra bevezethessen.

Az adattagokként tárol típuskódokkal kapcsolatban ennél rejtélyesebb problémák is felmerülhetnek. Előfordulhat például, hogy a típuskódot a `Base` egyik típusából véletlenül átmásoljuk egy másik típusba. A nagy és összetett, típuskódokra épülő programokban ez könnyen előfordulhat:

```
Base *bp1 = new Der1;
Base *bp2 = new Der2;
*bp2 = *bp1; // Katasztrófa!
```

Vegyük észre, hogy a `Der2` objektum típusa nem változott. A típust – mint említettük – a viselkedés határozza meg, a `Der2` objektum viselkedését pedig javarészt az, hogy létrehozásakor a konstruktor hogyan állítja be. A virtuális függvénytábla mutatója például, amit a fordító helyez el az objektumban, és amely azt határozza meg, hogy az objektum e függvények mely megvalósításával van kapcsolatban, nem változik meg a fenti kódban. Ugyanakkor a `Base` osztályban általunk megadott adattagok igen. (Ezzel kapcsolatban olvassuk el az 50. és 78. hibát.) A `bp2` által hivatkozott objektumnak csak a 7.1. ábrán szürkével jelölt területei változnak meg a hozzárendelés során.



7.1. ábra

Az alaposztály alobjektumának az egyik származtatott osztályból a másikhoz való hozzárendelése, és annak következményei. Csak az alaposztályban bevezetett adattagok másolódnak át az alobjektumból, a fordító által beillesztett osztálykezelő eljárások nem.

Ha egy objektum típusát a konstruktor a létrehozás során beállította, az többé nem változik. Ugyanakkor példánkban a `bp2` által címzett `Der2` objektum azt fogja magáról állítani, hogy `Der1` típusú. Ezt az állítást a `switch` utasításon alapuló minden döntési szerkezet el fogja hinni, a dinamikus kötésre épülők viszont nem. Az eredmény: tudathasadásos állapot.

Ha valamilyen – ritkán előforduló – tervezési szempont kifejezetten megköveteli a típuskódok használatát, akkor is célszerű két alapelvet szem előtt tartani. Először is soha ne adjuk meg a típuskódot adattagként. Használjunk inkább virtuális függvényt erre a célra, mert így sokkal közvetlenebbül rendelhetjük a típuskódot a megfelelő objektumtípushoz (viselkedésformához), és elkerülhetjük a fenti skizofrén helyzeteket is:

```
class Base {
 public:
    enum Tcode { DER1, DER2, DER3 };
    Base();
    virtual ~Base();
    virtual int tcode() const = 0;
    virtual void f() = 0;
    // . . .
};
class Der1 : public Base {
 public:
    Der1() : Base() {}
    void f();
    int tcode() const
        { return DER1; }
};
```

Másodszor, jobb, ha az alaposztálynak nem kell törődnie a belőle származtatott osztályokkal. Ez csökkenti a viszonyrendszer összetettségét, és lehetővé teszi, hogy a karbantartás során bizonyos származtatott osztályokat egyszerűen megszüntessünk, vagy éppen újakat hozzunk létre. Ez általában feltételezi, hogy a típuskódokat magán a programon kívül tartjuk karban. Ez a külső hely lehet egy hivatalos szabvány, vagy egy olyan algoritmus vagy eljárás, ami alapján a típuskódok előállíthatók. Természetesen minden származtatott osztálynak ügyelnie kell a saját típusára, de ezt az információt a program többi része elől el kell rejteni.

Az egyik olyan általános helyzet, amikor egy programozónak el kell gondolkodnia a típuskódok használatán, az, amikor egy objektumközpontú kódnak egy másik, nem objektumközpontú rendszerrel kell társalognia. Ilyen helyzet például, ha a másik rendszertől „üzenetek" érkeznek, és ezek típusa egy, az üzenet elején szereplő egész számból derül ki. Az üzenet hosszát és szerkezetét ez a típuskód határozza meg. Mit tehet ilyenkor egy tervező?

Általában az a legjobb, ha egyfajta képzeletbeli tűzfalat húzunk az objektumközpontú kód és környezete közé. Esetünkben a kívülről kapott üzenetek feldolgozásával foglalkozó kódrészlet az üzenetek elején található kódok alapján előállíthatja a megfelelő objektumokat, amelyek már nem tartalmazzák a típuskódot. A program javarésze így egyszerűen figyelmen kívül hagyhatja a típuskódokat, és a dinamikus kötésre építheti működését. Jegyezzük meg ugyanakkor, hogy szükség esetén az eredeti üzenetek könnyűszerrel visszaállíthatók, hiszen a kódok alapján előállított objektumok ezekkel való kapcsolata anélkül is fennmaradhat, hogy maguk az objektumok a típuskódot adattagként tartalmaznák.

E megközelítés egyetlen hátulütője, hogy az említett, feltehetőleg a `switch` utasításra támaszkodó döntési szerkezetet mindannyiszor újra kell fordítani, valahányszor megváltozik a lehetséges üzenetek halmaza. Ugyanakkor éppen a „tűzfalszerű" megközelítésnek köszönhetően a változás a kód egészét nem, csak a tűzfalat magát érinti:

```
Msg *firewall( RawMsgSource &src ) {
    switch( src.msgcode ) {
    case MSG1:
        return new Msg1( src );
    case MSG2:
        return new Msg2( src );
    // stb.
}
```

Egyes esetekben persze még a korlátozott újrafordítás sem megengedhető. Előfordulhat például, hogy az új üzenettípusokat úgy kell felvennünk a listába, hogy az alkalmazás közben fut. Ebben az esetben kihasználhatjuk a döntési szerkezetek helyettesíthetőségét. A `switch` szerkezet helyett bevezethetünk egy futásidőben kiértékelt adatszerkezetet, amit az előre lefordított kód rugalmasan képes kezelni. Esetünkben például objektumok egy egyszerű sorozatával helyettesíthetjük a döntési szerkezetet, amelyek mindegyike egy-egy üzenettípusnak felel meg:

➡ 69. hiba/firewall.h

```
class MsgType {
 public:
   virtual ~MsgType() {}
   virtual int code() const = 0;
   virtual Msg *generate( RawMsgSource & ) const = 0;
};
class Firewall { // Monostate
 public:
   void addMsgType( const MsgType * );
   Msg *genMsg( RawMsgSource & );
 private:
   typedef std::vector<MsgType *> C;
   typedef C::iterator I;
   static C types_;
};
```

Az értelmező működési elve ebben az esetben egyszerű: végigmegy egy sorozaton, közben egy bizonyos üzenetkódot keres, majd ha megtalálta, előállítja hozzá a megfelelő objektumot:

➡ 69. hiba/firewall.cpp

```
Msg *Firewall::genMsg( RawMsgSource &src ) {
   int code = src.msgcode;
   for( I i( types_.begin() ); i != types_.end(); ++i )
       if( code == i->code() )
           return i->generate( src );
   return 0;
}
```

Ezt az adatszerkezetet könnyen átalakíthatjuk úgy, hogy képes legyen új üzenetek automatikus felismerésére is:

```
void Firewall::addMsgType( const MsgType *mt )
   { types_.push_back(mt); }
```

Az egyedi üzenettípusok megvalósítása ismét magától értetődő:

```
class Msg1Type : public MsgType {
 public:
   Msg1Type()
        { Firewall::addMsgType( this ); }
   int code() const
        { return MSG1; }
   Msg *generate( RawMsgSource &src ) const
        { return new Msg1( src ); }
};
```

A listát különböző módon tölthetjük fel `MsgType` objektumokkal. A legegyszerűbb, ha bevezetünk egy, az üzenettípusnak megfelelő statikus változót. A konstruktor mellékhatásának köszönhetően az `MsgType` bekerül a `Firewall` statikus változókat tartalmazó listájába:

```
static Msg1Type msg1type;
```

Jegyezzük meg, hogy ezen statikus objektumok beállítási sorrendje lényegtelen. Ha lényeges lenne, akkor az 55. hiba kapcsán említett feltételekkel kellene számolnunk. A listára futás közben is felvehetünk új `MsgType` típusú objektumokat, a dinamikus betöltés módszerét alkalmazva.

Ami a statikus objektumokat illeti, vegyük észre, hogy a `Firewall` osztály fenti megvalósítása statikus adattagokat tartalmaz, de ezeket nem statikus tagfüggvények kezelik. Ez a tervezési módszer megfelel a Monostate tervezési mintának. Ez a módszer a Singleton módszer alternatívája, legalábbis ami a globális változók elkerülésének módját illeti. A Singleton módszer szerint az egyetlen objektumhoz az `instance` statikus tagfüggvényen keresztül kell hozzáférni. Ha ezt a tervezési módszert követtük volna a `Firewall` megvalósítása során, nekünk is pontosan ezt kellett volna tennünk:

```
Firewall::instance().addMessageType( mt );
```

A Monostate minta ezzel szemben tetszőleges számú objektum kezelését engedi meg, de azok valamennyien ugyanazokra a statikus adattagokra hivatkoznak. A hozzáférési protokollra vonatkozólag a módszer szintén nem tesz megkötéseket:

```
Firewall fw;
fw.genMsg( rawsource );
FireWall().genMsg( rawsource ); // Másik objektum, ugyanaz az
➥ állapot
```

70. hiba: Alaposztályok nem virtuális destruktorral

Ezt a témát gyakorlatilag az összes, C++ programozással foglalkozó tankönyv érintette az elmúlt tizenöt évben. Hogy egy osztályt alaposztálynak szánt-e a tervezője, arról semmi nem árulkodik jobban, mint maga az osztály. Ha ugyanis a destruktora virtuális, akkor valószínűleg alaposztályról van szó, ha pedig nem, akkor legalábbis igen nagy rá az esély, hogy az adott elvont típust tervezője nem alaposztálynak szánta.

Meghatározatlan viselkedés

A fenti állítást a nyelvi szabvány bevezetése még jobban aláhúzta. Ugyanis ha egy származtatott osztálynak megfelelő objektumot az alaposztály felületén keresztül semmisítünk meg, akkor az eredmény nem határozható meg, ha az alaposztály destruktora nem virtuális:

```
class Base {
   Resource *br;
   // . . .
   ~Base() // Megjegyzés: nem virtuális
       { delete br; }
};
class Derived : public Base {
   OtherResource *dr;
   // . . .
   ~Derived()
       { delete dr; }
};
Base *bp = new Base;
// . . .
delete bp; // Szép...
bp = new Derived;
// . . .
delete bp; // Csendben beleléptünk!
```

Nagy az esélye, hogy egyszerűen az alaposztály destruktora fog lefutni a származtatott osztályba tartozó objektum törlésekor. Ez viszonylag egyszerű hiba, de a fordítónak jogában áll valami egészen hajmeresztőt művelni ebben a helyzetben. (Vészleállást hajthat végre, levelet küldhet a főnöknek, vagy életfogytiglan előfizetheti számunkra az Objektumközpontú COBOL című hetilapot.)

Virtuális statikus tagfüggvények

Hogy valami biztatót is mondjak, ha az alaposztály destruktora virtuális, akkor elérhetjük vele azt a hatást, mintha egy virtuális statikus tagfüggvényt hívtunk volna meg. A virtuális és a statikus jelzők természetesen kölcsönösen kizárják egymást, így az olyan memóriakezelő műveletfüggvények, mint a `new`, a `delete`, a `new []` és a `delete []` statikus tagfüggvények. Ugyanakkor, még ha a destruktor virtuális is, akkor is meg kell hívnia az erre szakosított `operator delete` függvényt a törlés során, különösen akkor, ha annak létezik a megfelelő `operator new` párja is az adott osztályban (lásd a 63. hibát):

```
class B {
 public:
   virtual ~B();
   void *operator new( size_t );
   void operator delete( void *, size_t );
};
class D : public B {
 public:
   ~D();
   void *operator new( size_t );
   void operator delete( void *, size_t );
};
// . . .
B *bp = getABofSomeSort();
// . . .
delete bp; // A származtatott delete hívása!
```

Az alaposztályban jelen levő virtuális destruktornak köszönhetően a szabvány biztosít bennünket arról, hogy az `operator delete` tagfüggvényt fogjuk meghívni az adott ponton, mégpedig „az osztály dinamikus típusának hatókörén belül". Ez röviden azt jelenti, hogy a származtatott osztály destruktora a megfelelő `operator delete` tagfüggvényt fogja nagy valószínűséggel meghívni. Mivel a származatott osztály destruktora (természetesen) a származtatott osztály hatókörébe esik, a hívás a származtatott osztály `operator delete` függvényére fog vonatkozni.

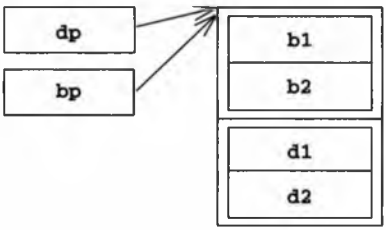
Röviden tehát, annak ellenére, hogy az `operator delete` statikus tagfüggvény, a virtuális destruktor megléte az alaposztályban gondoskodik róla, hogy a származtatott osztályra szakosított `operator delete` tagfüggvény fusson le még akkor is, ha egy, a származtatott osztályba tartozó objektumot az alaposztály mutatóján keresztül törlünk. A fenti példában a `bp` mutatóval kapcsolatos törlés a `D` destruktorát fogja meghívni, amit a `D` saját `operator delete` függvénye követ. Ugyanakkor az `operator delete` második paramétere `sizeof(D)` és nem `sizeof(B)` lesz. Csodás. Virtuális statikusok.

Bevezetés

Régebben a C++ kódok azzal a hallgatólagos feltételezéssel készültek, hogy egyszeres öröklés esetén az alaposztály alobjektumának és a teljes objektumnak azonos a címe. (Erről már volt szó a 29. hiba kapcsán.)

```
class B {
   int b1, b2;
};
class D : public B {
   int d1, d2;
};
D *dp = new D;
B *bp = dp;
```

Bár a szabvány ezzel kapcsolatban semmit sem biztosít, ebben az esetben csaknem bizonyos, hogy a `D` objektum a `B` alobjektummal kezdődik, amint azt a 7.2. ábra is mutatja.



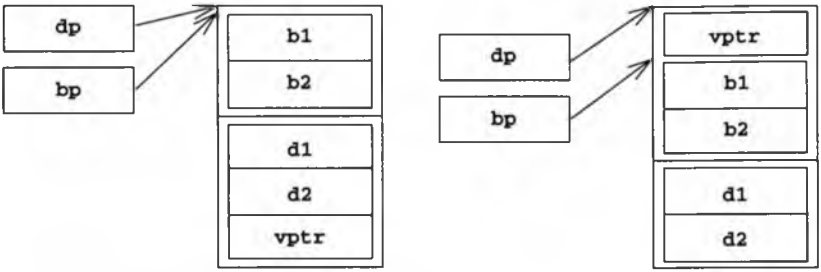
7.2. ábra

Virtuális függvényt nem tartalmazó objektum valószínű felépítése egyszeres öröklés esetén. Ennél a megvalósításnál a `B` alobjektumnak és a `D` teljes objektumnak azonos a kezdőcíme.

Ugyanakkor, ha a származtatott osztály bevezet egy virtuális függvényt, akkor a fordító elhelyez az objektumokban egy, a virtuálisfüggvény-táblát címző mutatót (`vptr`-t) is (lásd a 78. hibát). Ebben az esetben már két általánosan használt felépítés is létezik, amint azt a 7.3. ábra mutatja.

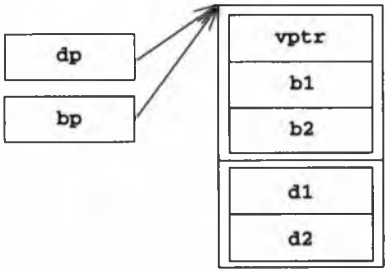
Az a kissé gyenge lábakon álló feltételezés, miszerint az alaposztály alobjektumának és a teljes objektumnak azonos a kezdőcíme, az első esetben még igaz, a második esetben azonban már nem. A legjobb megoldás természetesen az, ha átírjuk azt a kódot, amely erre a feltételezésre támaszkodott. Ez általában azt jelenti, hogy az osztálymutatók tárolásával kapcsolatban el kell felejtenünk a `void *` típust (lásd a 29. hibát).

Ha nem így teszünk, azzal azt kockáztatjuk, hogy ha az alaposztályban megjelenik egy virtuális függvény, akkor nem az előzetes feltevésünknek megfelelő memóriaképpel rendelkező objektumok keletkeznek, amint az a 7.4. ábrán is látható.



7.3. ábra

Egy objektum két lehetséges felépítése egyszeres öröklésnél, ha a származtatott osztály tartalmaz virtuális függvényt, az alaposztály azonban nem. A bal oldalon látható esetben a virtuálisfüggvény-tábla mutatója a teljes objektum végére, míg a jobb oldalon bemutatott esetben ennek az elejére kerül. Utóbbi esetben így az alaposztály alobjektumának már nullától különböző eltolási címe van a teljes objektumon belül.



7.4. ábra

Objektum valószínű felépítése egyszeres öröklés mellett, ha az alaposztály virtuális függvényt tartalmaz.

A legtöbb esetben az alaposztályok legalább egy virtuális destruktort tartalmaznak, ami azonnal megköveteli a memóriaképpel kapcsolatos előzetes feltevéseink feladását.

Kivételek

Még az olyan egyszerű szabályok alól, mint a most tárgyalt, is vannak kivételek. Néha például kifejezetten kényelmes, ha egy csinos csomagban foglalhatunk össze típusneveket, statikus tagfüggvényeket és statikus adattagokat:

```
namespace std {
    template <class Arg, class Res>
    struct unary_function {
        typedef Arg argument_type;
        typedef Res result_type;
    };
}
```

Ebben az esetben a virtuális destruktor megléte nem kötelező, hiszen az e sablon alapján létrehozott osztályok nem foglalnak le a végén visszaigénylendő erőforrásokat. Ez az osztály ráadásul elég körültekintően megtervezett ahhoz, hogy akkor se legyen hatása a végrehajtásra vagy a tárkezelésre, ha alaposztályként használjuk:

```
struct Expired : public unary_function<Deal *, bool> {
    bool operator ()( const Deal *d ) const
        { return d->expired(); }
};
```

Végül is a `unary_function` a szabványos könyvtár része. A gyakorlott C++ programozók pontosan tudják, hogy nem szabad ezt teljes értékű alaposztályként kezelni, így nem is próbálkoznak azzal, hogy belőle származtassanak saját osztályokat. A mi esetünk azonban különleges.

Itt van rögtön egy másik példa egy jól ismert, de nem szabványos könyvtárból. A tervezéssel kapcsolatos korlátok itt azonosak az előbb a szabványos könyvtárelemmel kapcsolatban említettekkel, de – mivel nem szabványos könyvtárról van szó – a tervező nem támaszkodhatott a felhasználónak a könyvtárral kapcsolatos előismereteire:

```
namespace Loki {
    struct OpNewCreator {
        template <class T>
        static T *Create() { return new T; }
      protected:
        ~OpNewCreator() {}
    };
}
```

A szerző itt azt a megoldást választotta, hogy megadott egy védett, helyben kifejtett, nem virtuális destruktort. Ez megőrzi a tárfoglalás és a futásidő hatékonyságát, megnehezíti a destruktor téves használatát, és félreérthetetlenül emlékeztet arra, hogy az osztályt tervezője alaposztálynak szánta.

Mindazonáltal az itt bemutatott példák valóban kivételek. Ennek megfelelően nem módosítják azt a korábban kimondott általános elvet, hogy egy alaposztálynak virtuális destruktort kell tartalmaznia.

71. hiba: Nem virtuális függvények elrejtése

A nem virtuális függvények az osztályhierarchia vagy részhierarchia megváltoztathatatlan elemei, amelyek gyökerei az alaposztályig nyúlnak vissza. A származtatott osztályok tervezői az ilyen függvényeket nem bírálhatják felül, és nem is rejthetik el (lásd a 77. hibát). E szabály értelme amúgy kézenfekvő: ha nem érvényesülne, az meggátolná a többalakúság használatát.

Egy többalakú (polimorf) objektumnak egyetlen megvalósítása (osztálya) van, de több típusa. Az elvont adattípusokkal kapcsolatban már megtanultuk, hogy a típust a végezhető műveletek halmaza szabja meg, ezeket a műveleteket pedig a felhasználó egy felületen keresztül érheti el. Egy `Circle` objektum például egyben `Shape` is, és akármelyik felületén keresztül használjuk is, ugyanúgy kell működnie:

```
class Shape {
 public:
   virtual ~Shape();
   virtual void draw() const = 0;
   void move( Point );
   // . . .
};
class Circle : public Shape {
 public:
   Circle();
   ~Circle();
   void draw() const;
   void move( Point );
   // . . .
};
```

A `Circle` osztály tervezője úgy határozott, hogy elrejti a `move` függvényt. (Valószínűleg azért tett így, mert a `Point` általában a bal felső saroknak felel meg, a `move`

függvény `Circle` esetében érvényes változata azonban ezt a középpontnak tekinti.) Így a `Circle` objektum eltérően viselkedhet, attól függően, melyik felületén keresztül használjuk:

```
void doShape( Shape *s, void (Shape::*op)(Point), Point p )
   { (s->*op)( p ); }
Circle *c = new Circle;
Point pt( x, y );
c->move( pt );
doShape( c, &Shape::move, pt ); // Hoppá!
```

Ha tehát elrejtjük az alaposztály egy nem virtuális függvényét, azzal megnöveljük a viszonyrendszer összetettségét, de cserébe tulajdonképpen nem kapunk semmit:

```
class B {
 public:
   void f();
   void f( int );
};
class D : public B {
 public:
   void f(); // Rossz ötlet!
};

B *bp = new D;
bp->f();  // Hoppá! A B::f() függvényt hívtuk meg a D objektumra;
D *dp = new D;
dp->f( 123 ); // Hoppá! A B::f(int) rejtett
```

Az ilyen típusfüggő viselkedésformák megvalósítására kerültek be a C++ nyelvbe a virtuális és tisztán virtuális függvények. Virtuális függvények használatával, ha egy származtatott osztály felülbírálja az alaposztály valamelyik függvényét, akkor a származtatott osztály objektumaival kapcsolatban ennek a függvénynek csak ez az utóbbi megvalósítása lesz hozzáférhető futásidőben. Ez azt jelenti, hogy az objektum csak egyféle viselkedést képes mutatni, függetlenül attól, melyik felületen keresztül férünk hozzá.

Mellesleg azért nem árt megjegyezni, hogy a virtuális függvények is meghívhatók nem virtuális értelmükben a hatókör művelet (scope operator) segítségével. Ez azonban a felület használatának sajátossága és nem a tervezési módé. Ezzel a módszerrel az alaposztály egy immár felülbírált virtuális függvénye továbbra is hozzáférhető a származtatott osztály számára:

```
class Msg {
 public:
```

```
    virtual void send();
    // . . .
};
class XMsg : public Msg {
 public:
    void send();
    // . . .
};
// . . .
XMsg *xmsg = new XMsg;
xmsg->send(); // Az XMsg::send hívása
xmsg->Msg::send(); // Az elrejtett/felülbírált Msg::send hívása
```

Ez persze megint csak egy olyan trükk, amire néhanapján szükségünk lehet ugyan, de általános tervezési módszernek nem tekinthető. Ugyanakkor egy felülbírált virtuális függvény eredeti változatának meghívhatósága akár szerves részét is képezheti egy tervnek. Ezt a módszert elterjedten használják arra, hogy az alaposztályban megadott szabványos alapfüggvénnyel időnként felülbírálják a származtatott osztályok megfelelő függvényeit.

Az itt látható `Decorator` minta szabványos megvalósítása jó példa erre a megközelítésre. A `Decorator` inkább szaporítja, semmint helyettesíti a hierarchia már meglevő függvényeit:

➡ 71. hiba/msgdecorator.h

```
class MsgDecorator : public Msg {
 public:
    void send() = 0;
    // . . .
 private:
    Msg *decorated_;
};
inline void MsgDecorator::send() {
    decorated_->send(); // Hívás továbbítása
}
```

Az `MsgDecorator` elvont osztály, hiszen egy tisztán virtuális `send` függvényt tartalmaz. Az `MsgDecorator`-ből származtatott konkrét osztályoknak természetesen felül kell bírálniuk a tisztán virtuális `MsgDecorator::send` függvényt. Ugyanakkor, bár virtuális függvényként nem hívható meg (hacsak nem valamilyen hibás, vagy erősen szokatlan helyzetben; lásd a 75. hibát), az `MsgDecorator::send` továbbra is meghívható marad a hatókör művelet használatával. Ez pedig lehetőséget ad arra, hogy az `MsgDecorator::send` függvényben olyan szabványos viselkedést állítsunk elő, amit az összes származtatott osztálynak is meg kell valósítania ezzel a művelettel kapcsolatban. Ezt a szabványos viselkedésformát pedig az utódok legegyszerűbben egy nem virtuális függvényhíváson keresztül érhetik el:

➡ 71. hiba/msgdecorator.cpp

```
void BeepDecorator::send() {
   MsgDecorator::send(); // Az alaposztály működik
   cout << '\a' << flush; // Egy további viselkedésforma
}
```

Egy másik lehetséges megoldás az lenne, ha az `MsgDecorator` osztály egy védett nem virtuális függvény formájában adná meg az említett szabványos, alapvető viselkedést. Ugyanakkor a tisztán virtuális függvény szerepeltetése sokkal világosabban kifejezi a tervezőnek azt a szándékát, hogy ezt a függvényt a származtatott osztályoknak kell használnia egyfajta közös alapként.

72. hiba: A Sablon tervezési módszer túl rugalmas alkalmazása

A Sablon tervezési módszer (Template Method) első lépése az, hogy a megvalósítandó algoritmust változó és állandó részekre osztjuk. Az állandó részeket a későbbiekben az alaposztály nem virtuális tagjaiként valósítjuk meg. Ugyanakkor ugyanezek a nem virtuális függvények lehetővé teszik, hogy a származtatott osztályok a tényleges helyzethez igazítsák a bennük megvalósított algoritmust. Ezt a programozók általában úgy oldják meg, hogy az említett nem virtuális függvények a származtatott osztályok által felülbírálható védett virtuális függvényeket hívnak meg. (Talán itt érdemes megjegyezni, hogy a Sablon tervezési módszernek az égvilágon semmi köze sincs a C++ sablonjaihoz.)

Mindez az alaposztály tervezőjének lehetővé teszi, hogy az algoritmus egészének szerkezetét megadja, de lehetővé tegye a későbbi felhasználók számára az egyes részek testreszabását is:

```
class Base {
 public:
   // . . .
   void algorithm();
 protected:
   virtual bool hook1() const;
   virtual void hook2() = 0;
};
void Base::algorithm() {
   // . . .
   if( hook1() ) {
        // . . .
```

```
        hook2();
    }
    // . . .
}
```

A Sablon tervezési módszer a virtuális és nem virtuális függvények használatának szabályozhatóságát biztosítja számunkra. Igen tanulságos amúgy, ha a munka elején összeírjuk az összes tervezési szempontot, és kiválogatjuk azokat, amelyeket a származatott osztályokra vonatkozó szabályokon keresztül rá akarunk kényszeríteni a majdani felhasználókra:

```
class Base {
 public:
   virtual ~Base(); // Én egy alaposztály vagyok
   virtual bool verify() const = 0; // Ellenőrizned kell
➥ önmagadat
   virtual void doit(); // Csinálhatod, ahogy akarod, akár úgy
➥ is, ahogy én
   long id() const; // Próbálj meg együtt élni ezzel a
➥ függvénnyel, vagy menj máshová...
   void jump(); // Ha azt mondom, „ugorj", csak annyit
➥ kérdezhetsz:
 protected:
   virtual double howHigh() const; //  ... Milyen magas, és ...
   virtual int howManyTimes() const = 0; //  ... hányszor?
};
```

Számos kezdő programozó úgy gondolja, hogy egy tervnek a lehető legrugalmasabbnak kell lennie. Ennek megfelelően gyakran követik el azt a hibát, hogy a Sablon módszert alkalmazva a teljes algoritmust virtuálisként adják meg, abban bízva, hogy a származtatott osztályok tervezői ebből a rugalmasságból majd profitálnak. Ez azonban tévedés. A származtatott osztályok tervezőinek munkáját leginkább az segíti, ha az általuk létrehozott és az alaposztály között egyértelmű a kapcsolat. Ha a teljes rendszerrel kapcsolatban tényleg vannak olyan elvárások, hogy bizonyos általános igényeket elégítsen ki, akkor ezt megfelelő származtatott osztályok formájában kell megoldani.

73. hiba: Virtuális függvények túlterhelése

Vajon mi lehet a hiba a következő kódrészletet tartalmazó alaposztállyal?

```
class Thing {
 public:
```

```
    // . . .
      virtual void update( int );
      virtual void update( double );
};
```

Képzeljük el, hogy egy ebből az alapból származtatott osztály tervezője úgy dönt, hogy az `update` függvénynek csak az egész paramétert kezelő változatát valósítja meg újra, más viselkedésformával:

```
class MyThing : public Thing {
 public:
     // . . .
     void update( int );
};
```

Ez itt a túlterhelés és a felülbírálás meglehetősen szerencsétlen összekeverése, márpedig a kettőnek semmi köze egymáshoz. Az eredmény ugyanaz, mintha elrejtettük volna az alaposztály egy nem virtuális függvényét: a `MyThing` objektum viselkedése attól függ, melyik felületen keresztül férünk hozzá:

```
MyThing *mt = new MyThing;
Thing *t = mt;
t->update( 12.3 ); // Rendben, alaposztály
mt->update( 12.3 ); // Hoppá, ez már származtatott
```

Az `mt->update(12.3)` alakú hívás például megtalálja az `update` nevet a származtatott osztályban, majd a `double` típusú paramétert sikeresen átalakítja `int` típusúvá, és ezzel látszólag minden rendben is volna. Csakhogy a programozó valószínűleg egyáltalán nem ezt akarta megvalósítani. És még ha tényleg ezt is akarta – bár ez meglehetősen furcsa világképre vall –, az ilyen kód akkor is megkeseríti a későbbi karbantartók életét, akik feltehetőleg kissé egyszerűbben gondolkodnak.

Számos szerző ahelyett, hogy vitatná a virtuális függvények túlterhelésének használhatóságát, néhány nem egészen használható C++ tankönyvben azt javasolja, hogy a származtatott osztályok tervezői minden esetben bírálják felül az összes túlterhelt virtuális függvényt. Ezzel az eljárással csak az a baj, hogy egy bizonyos programozási módot kényszerít rá mindenkire. Számos származtatott osztályt, például azokat, amelyeket egy adott keretrendszer kiegészítéseként adnak meg a tervezőik, általában az alaposztályt tartalmazó kódtól távol fejlesztenek. A kódnak ezen a pontján aztán esetleg teljesen más tervezési szempontok érvényesülnek, ami eleve meggátolhatja egy bizonyos módszer használatát.

A virtuális függvények túlterhelésének mellőzése általában semmiféle súlyos megkötést nem jelent az alaposztály felületének kialakításával kapcsolatban. Ha az alaposztály használhatósága mégis megköveteli a túlterhelést, akkor is jobb, ha más módon elnevezett virtuális függvényekre támaszkodó nem virtuális függvényeket terhelünk túl:

```
class Thing {
 public:
    // . . .
    void update( int );
    void update( double );
 protected:
    virtual void updateInt( int );
    virtual void updateDouble( double );
};
inline void Thing::update( int a )
    { updateInt( a ); }
inline void Thing::update( double d )
    { updateDouble( d ); }
```

A származtatott osztálynak így lehetősége van rá, hogy bármelyik virtuális függvényt felülbírálja, anélkül, hogy ezzel a többalakúság alapelveivel összetűzésbe keveredne. A származtatott osztálynak természetesen nem szabad `update` nevű tagfüggvényt tartalmaznia, hiszen az alaposztály nem virtuális függvényeinek elrejtésével kapcsolatos intelmek továbbra is érvényben vannak.

Ez alól a szabály alól léteznek ugyan kivételek, de ezek meglehetősen ritkák. Az egyik ilyen helyzetet a Látogató tervezési mintával kapcsolatban fogjuk látni (lásd a 77. hibát).

74. hiba: Alapértelmezett paraméterbeállításokat használó virtuális függvények

Ez jellegét tekintve tulajdonképpen ugyanolyan probléma, mint a virtuális függvények túlterhelése. A túlterheléshez hasonlóan az alapértelmezett paraméterbeállítás is egyfajta nyelvi finomság, amellyel egy osztály felületét anélkül módosíthatjuk, hogy annak alapvető viselkedését befolyásolnánk:

```
class Thing {
   // . . .
   virtual void doitNtimes( int numTimes = 12 );
};
class MyThing : public Thing {
   // . . .
   void doitNtimes( int numTimes = 10 );
};
```

Probléma akkor keletkezik, ha összekeverjük az objektumok statikus és dinamikus viselkedését. Az ezzel kapcsolatos hibákat igen nehéz felderíteni:

```
Thing *t = new MyThing;
t->doitNtimes();
```

Példánkban szemmel láthatóan azt tételeztük fel, hogy annak a bizonyos számlálónak a `MyThing` típusú objektumok esetében tíznek, az összes többi `Thing` esetében azonban tizenkettőnek kell lennie. A baj csak az, hogy a paraméterek alapértelmezett beállítása mindig statikusan értendő, vagyis esetünkben az alaposztály tizenkettője érvényesül a származtatott osztály dinamikus kötésű függvényére is.

A problémát persze áthidalhatnánk úgy, hogy elvárjuk az összes származtatott osztály tervezőjétől, hogy pontosan másolják le az alaposztályban érvényes alapértelmezett beállításokat a függvények felülbírálásakor. Ez azonban több okból is hibás megközelítés.

Először is a származtatott osztályok tervezői a maguk útját járják, és nem mindegyik hagyja magát rábeszélni valamire. (Megeshet például, hogy az illető meglátván az alaposztály alapértelmezett beállításait, egyszerűen elveszti az alaposztály tervezőjének szakértelmébe vetett hitét, és úgy határoz, hogy megoldja ezt a problémát.)

Másodszor, ez a megoldás túlságosan szoros kapcsolatot, sőt egyfajta kóros függést teremtene a származtatott osztályok és az alaposztály között, hiszen ha az utóbbi alapértelmezett beállításai a jövőben megváltoznak, a módosítást az összes származtatott osztályban is el kell végezni, ami pedig az esetek többségében nem lehetséges.

Harmadszor, az alapértelmezett beállítások jelentése más és más lehet, attól függően, hogy hol alkalmazzuk őket. A származtatott osztályok függvényeiben ugyanaz a beállítás esetleg teljesen mást jelenthet:

```
// A thing.h fájlban...
const int minim = 12;
namespace SCI {
class Thing {
   // . . .
   virtual void doitNtimes( int numTimes = minim ); //
➥ A ::minim-et használja
};
}

// A mything.h fájlban...
namespace SCI {
const int minim = 10;
class MyThing : public Thing {
   // . . .
   void doitNtimes( int numTimes = minim ); // Az SCI::minim-et
➥ használja
};
}
```

Ebben a helyzetben nehéz lenne a származtatott osztály tervezőjét hibáztatni azért, mert nem a megfelelő `minim` értéket választotta, különösen ha az `SCI::minim` deklarációja a `MyThing` osztály után került be a kódba.

A legegyszerűbb és legbiztosabb, ha teljesen elkerüljük az alapértelmezett beállítások használatát a virtuális függvényekben. Akár a virtuális függvények túlterhelését, ezt a dolgot is megkerülhetjük bizonyos trükkök bevetésével:

```
class Thing {
   // . . .
   void doitNtimes( int numTimes = minim )
        { doitNtimesImpl( numTimes ); }
 protected:
   virtual void doitNtimesImpl( int numTimes );
};
```

A `Thing` hierarchia felhasználói megkapják a statikusan megadott alapértéket az alaposztályból, a származtatott osztályok azonban anélkül módosíthatják a függvény viselkedését, hogy ezzel az alapértelmezett értékkel törődniük kellene.

75. hiba: Virtuális függvények hívása a konstruktorban és a destruktorban

A konstruktorok feladata azoknak az erőforrásoknak a lefoglalása, amelyek az adott osztályba tartozó objektumok működéséhez szükségesek. A destruktorok ugyanezeket az erőforrásokat szabadítják fel az objektum megsemmisítésekor. Mármost miért ne tükrözhetné egyértelműen ezt a logikai felépítéssel kapcsolatos alapelvet az alaposztály megvalósítása, vagyis miért ne foglalhatnánk le és szabadíthatnánk fel mi magunk azokat a bizonyos erőforrásokat?

```
class B {
 public:
   B() { seize(); }
   virtual ~B() { release(); }
 protected:
   virtual void seize() {}
   virtual void release() {}
};
```

A származtatott osztályoknak ezután megvan az a lehetősége, hogy saját erőforráskezelési igényeiknek megfelelően felülbírálják az alaposztály `seize` és `release` függvényeit:

```
class D : public B {
 public:
   D() {}
   ~D() {}
   void seize() {
        B::seize(); // Az alap erőforrások megszerzése
        // A származtatott erőforrások megszerzése...
   }
   void release() {
        // A származtatott erőforrások felszabadítása...
        B::release(); // Az alap erőforrások felszabadítása
   }
};
// . . .
D x; // Nem szabadítunk fel és nem foglalunk le erőforrásokat!
```

Az x objektum létrehozásakor a származtatott osztály konstruktora meghívja az alaposztály konstruktorát, ami maga meghívja a `seize` virtuális függvényt. Az x objektum megsemmisítésének utolsó lépéseként hasonló sorrend érvényesül: a szár-

maztatott osztály destruktora meghívja az alaposztály destruktorát, ami viszont meghívja a `release` virtuális függvényt. Csakhogy mindeközben semmiféle erőforrást nem foglalunk le, és nem is szabadítunk fel.

A probléma alapja az, hogy amikor a származtatott osztály konstruktora meghívja az alaposztály konstruktorát, az `x` objektum típusa még nem `D`. Ennek megfelelően az alaposztály konstruktora az `x` objektum `B` alobjektumát mint `B` típusú objektumot állítja be, ami pedig azt jelenti, hogy amikor a konstruktor meghívja a `seize` virtuális függvényt, valójában a `B::seize` fut le. Ugyanez a zavar érvényesül a destruktorok működése közben is. Amikor a származtatott osztály destruktora meghívja az alaposztály destruktorát, az `x` objektum többé már nem `D` típusú, az `x` `B` alobjektuma pedig `B` típusú objektumként viselkedik. Ebből pedig az következik, hogy a virtuális `release` függvény ebben a környezetben a `B::release` függvény meghívását jelenti.

Ebben a helyzetben a legegyszerűbb megoldás az lenne, ha a nyelv beépített eljárásaira hagyatkoznánk az összetett objektumok konstruktorainak és destruktorainak megalkotása terén. Az erőforrások lefoglalását és felszabadítását megvalósító kódrészleteknek természetesen továbbra is szerepelniük kellene a kódban, de a szokásos konstruktorokban és destruktorokban:

```
class B {
 public:
   B() {
        // Az alap erőforrások megszerzése...
   }
   virtual ~B() {
        // Az alap erőforrások felszabadítása...
   }
};
class D : public B {
 public:
   D() {
        // A származtatott erőforrások megszerzése...
   }
   ~D() {
        // A származtatott erőforrások felszabadítása...
   }
};
// . . .
D x; // Működik!
```

Van még egy módszer, amivel időnként célt érhetünk. Ez pedig a tisztán virtuális függvények virtuális, nem pedig statikus hívási sorozata:

```
class Abstract {
 public:
   Abstract();
   Abstract( const Abstract & );
   virtual bool validate() const = 0;
   // . . .
};
bool Abstract::validate() const
   { return true; }
Abstract::Abstract() {
   if( validate() ) // Kísérlet tisztán virtuális függvény
➥ meghívására
        // . . .
};
```

Sajnos a szabvány az ilyen hívási trükkökkel kapcsolatban azt mondja, hogy azok kimenetele nem meghatározott. Ez konkrétan azt jelenti, hogy – bár az eredmény erősen rendszerfüggő – az esetek többségében a program egyszerűen leáll, a kód egy függvényt nullmutatón keresztül próbál meghívni, vagy (és ez az igazán veszélyes) valóban meghívja az `Abstract::validate` függvényt. Bár tulajdonképpen pontosan ez az, amit el akartunk érni, az ilyen kód meglehetősen törékeny, és biztosan nem hordozható.

Vegyük észre, hogy itt a virtuális függvények hívásának csak olyan eseteiről volt szó, amikor ez az objektumok létrehozásának vagy megsemmisítésének része. Annak természetesen semmi akadálya, hogy egy konstruktor vagy destruktor egy másik, teljesen felépített objektum virtuális függvényét hívja meg:

```
Abstract::Abstract( const Abstract &that ) {
   if( that.validate() ) // Rendben
        // . . .
}
```

76. hiba: Virtuális értékadás

Az értékadás (assignment) lehet ugyan virtuális, de erre igazából ritkán van szükség. Lehet például egy olyan, tároló osztályokból álló hierarchiánk, amely az alaposztály felületén keresztül lehetővé teszi a virtuális értékadást:

```
template <typename T>
class Container {
 public:
```

```
    virtual Container &operator =( const T & ) = 0;
    // . . .
};
template <typename T>
class List : public Container<T> {
    List &operator =( const T & );
    // . . .
};
template <typename T>
class Array : public Container<T> {
    Array &operator =( const T & );
    // . . .
};
// . . .
Container<int> &c( getCurrentContainer() );
c = 12; // Világos a dolog jelentése?
```

Vegyük észre, hogy itt nem másoló értékadásról van szó, hiszen a paraméter típusa nem azonos a tároló osztályéval. (Azzal kapcsolatban, hogy miért tér el a származtatott osztály értékadó műveletének visszatérési típusa az alaposztály hasonló függvényének visszatérési típusától, olvassuk a 77. hiba leírását.) Ennek a műveletnek az a feladata, hogy a `Container` minden elemének ugyanazt az értéket adja. Sajnos az értékadásnak ezt a típusát számos programozó félreérti. Egyesek azt hiszik, hogy a művelet megváltoztatja a tároló objektum méretét, mások pedig úgy hiszik, hogy az értékadás csak az első elemet érinti (lásd a 84. hibát). Egy biztonságosabban megtervezett felületnek meg kell tiltania a túlterhelést, mert egy nem műveleti függvény használata sokkal kevésbé félreérthető:

```
template <typename T>
class Container {
 public:
    virtual void setAll( const T &newElementValue ) = 0;
    // . . .
};
// . . .
Container<int> &c( getCurrentContainer() );
c.setAll( 12 ); // A művelet jelentése világos
```

A másoló értékadás szintén lehet virtuális, de ez megint csak ritkán bizonyul jó megoldásnak, mivel a származtatott osztályban megadott másoló értékadás nem bírálja felül az alaposztály hasonló műveletét:

```
template <typename T>
class Container {
```

```
 public:
   virtual Container &operator =( const Container & ) = 0;
   // . . .
};
template <typename T>
class List : public Container<T> {
   List &operator =( const List & ); // Nem bírál felül!
   List &operator =( const Container<T> & ); // Felülbírál...
   // . . .
};
// . . .
Container<int> &c1 = getMeAList();
Container<int> &c2 = getMeAnArray();
c1 = c2; // Tömb hozzárendelése listához?!?
```

A virtuális másoló értékadás lehetővé teszi, hogy egy származtatott osztály elemét átmásoljuk egy másik, eltérő típusú származtatott osztály elemébe! Tekintettel arra, hogy meglehetősen kevés olyan helyzet képzelhető el, amikor ennek a műveletnek tényleg van értelme, célszerűbb a virtuális másoló értékadást elkerülni.

A fenti `Container` hierarchiában történetesen megpróbálkozhatunk a virtuális másoló értékadás bevezetésével, itt ugyanis értelmes művelet az egyik típusú tároló objektum (egy tömb) tartalmának egy más típusba (lista) való átmásolása. Ez azonban azt feltételezi, hogy minden tároló objektum tud az összes többi típusról, ami általában rossz tervezésre vall. A másik lehetőség persze az, hogy valamilyen kifinomult keretrendszert használunk. Egyszerűbb és ezért jobb megoldás, ha a `Container` osztály tagfüggvényeként, vagy akár nem tagfüggvényként megadunk egy `copyContent` függvényt, ami virtuális függvények vagy bejárók segítségével veszi a másolás forrásának egyes tagjait, majd áthelyezi azokat a célobjektumba:

```
Container<int> &c1 = getMeAList();
Container<int> &c2 = getMeAnArray();
c1.copyContent( c2 ); // A tömb tartalmát átmásoljuk egy listába
```

Egy ilyen megoldást a szabványos könyvtárban is találhatunk, ahol lehetőségünk van arra, hogy egy tároló objektumot egy másik, eltérő típusú tároló objektum tartalma alapján állítsunk be:

```
vector<int> v;
// . . .
list<int> el( v.begin(), v.end() );
```

Az is gyakori, hogy a virtuális másoló konstruktor alkalmazása tervezési szempontból megfelelőbb, mint a virtuális másoló értékadás. A C++-ban természetesen nincs virtuális konstruktor, van viszont egy úgynevezett Prototípus tervezési módszer, amely végül is megvalósítja ezt a szolgáltatást. Ahelyett, hogy egy ismeretlen típusú objektummal értékadást hajtanánk végre, „klónozzuk" azt. Ebben a tervezési módszerben az alaposztálynak van egy tisztán virtuális `clone` nevű függvénye, amelyet a származtatott osztályok felülbírálnak, feladata pedig az adott objektum pontos másának előállítása. Ezt a másolási műveletet persze általában a származtatott osztály másoló konstruktorával oldják meg, így a `clone` függvényt egyfajta virtuális konstruktornak tekinthetjük:

➡ 90. hiba/container.h

```
template <typename T>
class Container {
 public:
   virtual Container *clone() const = 0;
   // . . .
};
template <typename T>
class List : public Container<T> {
   List( const List & );
   List *clone() const
        { return new List( *this ); }
   // . . .
};
template <typename T>
class Array : public Container<T> {
   Array( const Array & );
   Array *clone() const
        { return new Array( *this ); }
   // . . .
};
// . . .
Container<int> *cp = getCurrentContainer();
Container<int> *cp2 = cp->clone();
```

Röviden tehát a Prototípus tervezési módszer lehetőséget ad arra, hogy programunkban valami ilyesféle parancsot adjunk ki: „Nem tudom, hogy pontosan miféle objektumra mutatok, de azt akarom, hogy legyen belőle még egy!"

77. hiba: A túlterhelés, a felülbírálás és a rejtés összekeverése

Számomra mindig megrázó, amikor egy szakmai beszélgetés sokadik percében jövök rá arra, hogy partnerem fogalmi szinten nem képes különbséget tenni a felülbírálás (overriding) és túlterhelés (overloading) között. Ha ehhez netán még a blokkszerkezetekkel kapcsolatos rejtés félreértése is társul, egészen furcsa társalgás alakulhat ki. Ennek persze nem kell feltétlenül így lennie, vagyis ideje tisztázni az említett fogalmak közti eltérést.

A C++-ban a túlterhelés azt jelenti, hogy ugyanazt a nevet használjuk különböző, de azonos hatókörben bevezetett függvényekre. Ez utóbbi kitétel nagyon fontos:

```
bool process( Credit & );
bool process( Acceptance & );
bool process( OrderForm & );
```

Ez itt világos módon három túlterhelt függvény. Ugyanaz a nevük, és azonos a hatókörük is. A fordító az alapján tesz köztük különbséget, hogy milyen típusú paraméterrel hívjuk meg a `process` nevű függvényt. Ez így minden kétséget kizáróan értelmes dolog. Ha egy `Acceptance` típusú objektumot akarunk feldolgozni, nyilván azt várjuk a fordítótól, hogy a második függvényt használja. A C++ nyelvben a függvények neve a függvény azonosítójából (ebben az esetben ez a `process`) és a bevezetésénél megadott formális paramétereinek típusaiból áll össze. Építsük be a három függvényt egy osztályba:

```
class Processor {
 public:
   virtual ~Processor();
   bool process( Credit & );
   bool process( Acceptance & );
   bool process( OrderForm & );
   // . . .
};
```

Ezek továbbra is túlterhelt függvények, és a fordító továbbra is képes őket megkülönböztetni a paraméterek típusa alapján. Az a tény, hogy a fenti `Processor` osztály destruktora virtuális, világosan jelzi tervezőjének azt a szándékát, hogy művét alaposztálynak szánta. Ennek szellemében fejlesszük tovább a `Processor` képességeit egy származtatott osztály keretében:

```
class MyProcessor : public Processor {
 public:
   bool process( Rejection & );
   // . . .
};
```

Na igen, csak ne így. A származtatott osztály `process` nevű függvénye sajnos nem túlterheli a már meglevő hármat, hanem elrejti azokat:

```
Acceptance a;
MyProcessor p;
p.process( a ); // Hiba!
```

Amikor a fordító a származtatott osztály hatókörén belül `process` nevű függvényt keres, csak egy lehetséges jelöltet talál. Ez a függvény azonban egy `Rejection` típusú paramétert vár, ami azt jelenti, hogy típuskeveredés keletkezik. Ez pedig a fordítás végét jelenti, hacsak nincs valamilyen módszer arra, hogy a `Rejection`-ből `Acceptance` típusú objektumot állítsunk elő. Pont. A fordító nem tudja megoldani a problémát, mert nem fog további lehetőségek után kutatni a szélesebb hatókörökben. A származtatott osztály `process` függvénye a származtatott osztályon belül érvényes, tehát nem terheli túl az alaposztály ugyanilyen nevű függvényeit.

Persze van lehetőségünk arra, hogy az alaposztály megfelelő függvényeit bevonjuk a származtatott osztály hatókörébe a `using` kulcsszó segítségével:

```
class MyProcessor : public Processor {
 public:
   using Processor::process;
   bool process( Rejection & );
   // . . .
};
```

Immár mind a négy függvény jelen van a származtatott osztályban, és a túlterheléssel kapcsolatos szabályok is érvényesek rájuk. A gond csak az, hogy az ilyen és ehhez hasonló tervezési eljárások túlságosan összetett szerkezeteket eredményeznek. Az összetettség pedig csak akkor jobb az egyszerűségnél, ha használható többletszolgáltatást nyújt.

Esetünkben egy `Rejection` típusú objektumot csak a `MyProcessor` származtatott osztály segítségével dolgozhatunk fel. Ha ugyanezt a `Processor` alaposztály egy objektumával próbáljuk megtenni, fordítási hibát kapunk. Ugyanakkor azonban, ha a `Rejection` típus bármilyen módon `Acceptance`, `OrderForm` vagy `Credit` típussá alakítható, a függvényhívás mindkét módon sikeres lesz, csak eltérő eredményre vezet.

Felülbírálni csak olyan függvényt lehet, ami az alaposztályban virtuális függvényként szerepel. Pont. A felülbírálásnak tehát semmi köze a túlterheléshez. Az alaposztály egy nem virtuális függvényét felülbírálni nem, csak elrejteni lehet:

```
class Doer {
 public:
   virtual ~Doer();
   bool doit( Credit & );
   virtual bool doit( Acceptance & );
   virtual bool doit( OrderForm & );
   virtual bool doit( Rejection & );
   // . . .
};
class MyDoer : public Doer {
 private:
   bool doit( Credit & ); // #1, elrejt
   bool doit( Acceptance & ); // #2, felülbírál
   virtual bool doit( Rejection & ) const; // #3, nem bírál
➥ felül
   double doit( OrderForm & ); // #4, hiba
   // . . .
};
```

(A fenti példában a `Doer` osztályok csak szemléltetési célt szolgálnak, tehát nem követendő példaként kívántam az Olvasó elé állítani ezt a tervezési stílust. Általában ritkán megengedhető egy virtuális függvény túlterhelése. Erről és az ezzel kapcsolatos problémákról a 73. hibánál olvashatunk részletesebben.)

A `#1` jelet viselő `doit` függvény nem bírálja felül az alaposztály ilyen nevű függvényét, mivel az nem virtuális. Ugyanakkor ez a deklaráció elrejti a `doit` függvénynek az alaposztályban található mind a négy megvalósítását.

A `#2`-vel jelzett függvény valóban felülbírálja az alaposztály függvényét. Jegyezzük meg, hogy a hozzáférési szint ezt a képességet semmilyen módon nem befolyásolja. Nem számít, ha az alaposztály függvénye nyilvános, a származtatott osztályé pedig védett, vagy fordítva. Általában ugyan az alap- és a származtatott osztály egymásnak megfelelő függvényei azonos hozzáférési szinthez tartoznak, néha azonban célszerű lehet ettől a szokásos megoldástól eltérni:

```
class Visitor {
 public:
   virtual void visit( Acceptance & );
   virtual void visit( Credit & );
   virtual void visit( OrderForm & );
```

```
    virtual int numHits();
};
class ValidVisitor : public Visitor {
    void visit( Acceptance & ); // felülbírál
    void visit( Credit & ); // felülbírál
    int numHits( int ); // #5, nem virtuális
};
```

Ebben az esetben a tervező úgy döntött, hogy megengedi az alaposztály viselkedésének testreszabását, de azt kikötötte, hogy a hierarchia felhasználóinak továbbra is az alaposztály felületét kell használniuk. Ezt úgy valósította meg, hogy az alaposztály függvényeit nyilvánosként, a származtatott osztály ezeknek megfelelő felülbíráló függvényeit azonban privátként vezette be.

A `virtual` kulcsszó használata a származatott osztályban nem kötelező, ha az alaposztály egy függvényét bíráljuk felül. A származtatott osztály függvényének működését e kulcsszó jelenléte vagy hiánya nem befolyásolja:

```
class MyValidVisitor : public ValidVisitor {
    void visit( Credit & ); // felülbírál
    void visit( OrderForm & ); // felülbírál
    int numHits(); // #6, virtuális, felülbírálja
➥ a Visitor::numHits tagot
};
```

Általános téveszme, hogy ha a származtatott osztály függvényének deklarációjában nem szerepel a `virtual` kulcsszó, akkor az ebből származtatott osztályok már nem bírálhatják felül ezt a függvényt. Ez persze nem igaz, így a példánkban szereplő `MyValidVisitor::visit( Credit & )` függvény felülbírálja a `ValidVisitor` és a `Visitor` osztályok megfelelő függvényeit.

Annak sincs semmi akadálya, hogy a származtatott osztályok tőlük távol eső alaposztályok függvényeit bírálják felül. A `MyValidVisitor::visit( Orderform & )` például felülbírálja a `Visitor` osztály ilyen nevű függvényét.

Az is lehetséges, hogy egy származtatott osztály az alaposztálynak olyan függvényét bírálja felül, amely az adott környezetben nem is látható. Példánkban a `#5`-tel jelzett `ValidVisitor::numHits` függvény nem bírálja felül az alaposztály `Visitor::numHits` függvényét, de elrejti azt a további származtatott osztályok elől. Ennek ellenére a `MyValidVisitor::numHits` függvény felülbírálja a `Visitor::numHits` függvényt.

A `#3`-mal jelzett `MyDoer` függvény ravasz megoldást mutat. Ez a függvény virtuális ugyan, de csak azért, mert így adtuk meg. Valójában nem bírálja felül az alaposztály megfelelő virtuális függvényét, mert állandó tagfüggvény, az alaposztályban pedig nincs neki megfelelő, szintén állandó virtuális tag. Az állandóság tehát része a függvény összetett szignatúrájának. (Ezzel kapcsolatban olvassuk el a 82. hiba leírását.)

A `#4`-gyel jelzett `MyDoer` függvény megadása hibás. Felülbírálja ugyan az alaposztály megfelelő virtuális függvényét, de visszatérési típusa nem megfelelő. Az alaposztály függvényének visszatérési típusa `bool`, a származtatotté viszont `double`. Ez pedig fordítási hibát okoz.

Általánosságban, ha egy származtatott osztály felülbírálja az alaposztály egy függvényét, akkor ugyanolyan visszatérési értékkel kell rendelkeznie. Ez a kikötés a futásidejű kötéssel kapcsolatos statikus típusbiztonságról gondoskodik. Egy származtatott osztály virtuális függvényét általában az alaposztály megfelelő függvényének felületén keresztül hívjuk meg (éppen ez a virtuális függvények használatának értelme). A fordító kénytelen olyan kódot előállítani, ami a visszatérési érték kezelésével kapcsolatban feltételezi, hogy az esetlegesen meghívott származtatott osztályban megvalósított virtuális függvénynek ugyanaz a visszatérési típusa.

A példánkban bemutatott negyedik (`#4`) esetben azonban a származtatott osztály függvénye `sizeof(double)` számú bájtot akar elhelyezni egy `sizeof(bool)` méretű lefoglalt területen. Még ha a két típus méretben valahogyan megfeleltethető lenne is egymásnak (vagyis egy `bool` legalább annyi bájtból állna, mint egy `double`), akkor is valószínűtlen, hogy a későbbi kód képes lenne `bool` típusként értelmezni egy valójában `double` értéket.

Ez alól a szabály alól egyetlen kivétel van, amit „kovariáns visszatérési típusok" néven ismerünk. (A kovariancia nem tévesztendő össze a kontravarianciával! Lásd a 46. hibát.) Egy alaposztály függvényének és a belőle származtatott osztály azt felülbíráló függvényének típusa akkor kovariáns, ha mindkét visszatérési érték olyan osztályt címző mutató vagy hivatkozás, amely osztályok egymásnak megfelelnek („is-a" kapcsolatban állnak). Ez így meglehetősen cirkalmas meghatározás, ezért szemléltessük talán két példával:

```
class B {
   virtual B *clone() const = 0;
   virtual Visitor *genVisitor() const;
   // . . .
};
class D : public B {
```

```
    D *clone() const;
    ValidVisitor *genVisitor() const;
};
```

A `clone` függvény egy olyan mutatót ad vissza, amely az őt hívó objektum pontos másolatát címzi. (Ezt a függvényt, mint korábban említettük, a Prototípus tervezési módszerben használják. Lásd a 76. hibát.) A hívás általában az alaposztály felületén keresztül történik, a lemásolt objektum pontos típusa pedig ismeretlen:

```
B *aB = getAnObjectDerivedFromB();
B *anotherLikeThat = aB->clone();
```

Egyes esetekben azonban egészen pontos elképzeléseink vannak a keletkező objektum típusáról, és nem akarjuk elveszíteni ezt az információt, vagy – ami még fontosabb – nem akarunk az alaposztályban mutatót megadni a saját származtatott osztályára (downcasting):

```
D *aD = getAnObjectThatIsAtLeastD();
D *anotherLikeThatD = aD->clone();
```

Ha nem lenne a kovariáns típusokkal kapcsolatos lehetőség, a visszatérési értéket kénytelenek lennénk `B *` típusról `D *` típusra alakítani:

```
D *anotherLikeThatD = static_cast<D *>(aD->clone());
```

Figyeljük meg, hogy ebben az esetben nyugodtan használhatjuk a hatékonyabb `static_cast` műveletet a `dynamic_cast` helyett, mivel pontosan tudjuk, hogy a `clone` függvény által visszaadott típus `D`. Más körülmények között a `dynamic_cast` használata vagy a típusátalakítás elhagyása lenne a megfelelőbb megoldás.

A `genVisitor` függvény (a 90. hibánál bemutatandó Gyár módszer része) azt mutatja, hogy a kovariáns osztálytípusoknak nem kell ahhoz a hierarchiához tartozniuk, amelyben a kérdéses függvények előfordulnak.

A C++ felülbírálattal kapcsolatos rendszere összességében rugalmas és hasznos eszköz. Ugyanakkor, ha élni akarunk vele, azzal óhatatlanul bizonyos fokú bonyolultságot viszünk programjaink szerkezetébe. A fejezet hátralevő részében éppen azt mutatjuk be, hogyan lehetünk úrrá ezen az összetettségen, és hogyan használhatjuk ki a rendszer nyújtotta előnyöket.

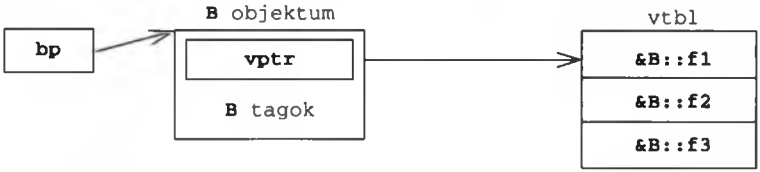
78. hiba: A virtuális függvények és a felülbírálás rendszere

Számos újdonsült C++ programozónak meglehetősen felületes ismeretei vannak a nyelv felülbírálási rendszeréről, így néhány példa sokat segíthet a dolgok átlátásában. A virtuális függvények nyújtotta lehetőségek használatának többféle bevált módszere is létezik. Ezek közül mutatunk be itt egyet.

Nézzünk először egy egyszerű példát az egyszeres öröklésre:

```
class B {
 public:
   virtual int f1();
   virtual void f2( int );
   virtual int f3( int );
};
```

A fordító a fenti osztály minden virtuális függvényéhez egy-egy sorszámot rendel. Tegyük fel például, hogy `B::f1` a `0`, `B::f2` az `1` sorszámot kapja, és így tovább. Ezeket a számokat a fordítóprogram később arra használja, hogy a virtuális függvények mutatóit tartalmazó táblát indexelje velük. E tábla `0` sorszámú eleme tehát a `B::f1` függvény mutatója, `1`-es eleme a `B::f2` függvényhez enged hozzáférést, és így tovább. Amikor az adott osztályba tartozó objektumokat hozunk létre, a fordító valamennyiben elhelyez egy mutatót, amely ezt a virtuális függvényeket felsoroló táblát címzi. Egy ilyen `B` objektum lehetséges memóriaképe a 7.5. ábrán látható.



7.5. ábra

Virtuális függvények egyszerű megvalósítása egyszeres öröklés esetén.

A „köznyelvben" a virtuálisfüggvény-tábla neve egyszerűen „v tábla" (vtbl), a táblát címző mutatóé pedig „v mutató" (vptr). A vptr-t a `B` osztály konstruktora állítja be, úgy, hogy a megfelelő táblára mutasson. Ha meghívunk egy virtuális függvényt, az a kódban egy, a táblában található mutató visszakövetését jelenti:

```
B *bp = new B;
bp->f3(12);
```

A fenti kód lefordítás után körülbelül a következővel egyenértékű:

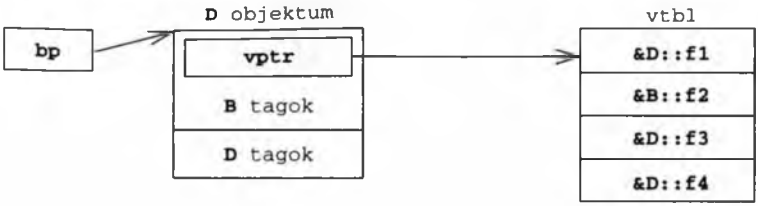
```
(*(bp->vptr)[2])(bp, 12)
```

A megfelelő függvény címét úgy kapjuk meg, hogy a virtuális függvények táblájából kikeressük a függvény sorszámának megfelelő elemet. Ezután meghívjuk a megfelelő függvényt úgy, hogy beleértett „`this`" paraméterében a hívó objektum címét helyezzük el. A virtuális függvények kezelése a C++-ban rendkívül hatékony. A közvetett függvényhívás általában minden hardveren optimális kódot eredményez, az azonos típusú objektumoknak pedig rendszerint egyetlen virtuálisfüggvény-táblája van. Egyszeres öröklésnél valamennyi objektumban egyetlen ilyen táblát címző mutató van, függetlenül attól, hány virtuális függvényt adtunk meg az adott osztályban.

Nézzük most egy olyan származtatott osztály megvalósítását, amely felülbírálja alaposztályának egyes virtuális függvényeit:

```
class B {
 public:
    virtual int f1();
    virtual void f2( int );
    virtual int f3( int );
};
class D : public B {
    int f1();
    virtual void f4();
    int f3( int );
};
```

A `D` típusú objektumok biztosan tartalmaznak egy `B` alobjektumot. Általában – de nem feltétlenül (lásd a 70. hibát) – ez az alobjektum a származtatott osztály objektumának legelején (vagyis a 0 eltolási címen) található. Minden újabb származtatott osztálynak megfelelő alobjektum ez után következik, amint az a 7.6. ábrán is látható.



7.6. ábra

Virtuális függvények megvalósításának egyszerű módja egyszeres öröklés esetén a származtatott osztály elemében. Az alaposztálynak megfelelő alobjektum továbbra is tartalmazza a virtuálisfüggvény-tábla mutatóját, de az már a származtatott osztály igényeinek megfelelő változatra mutat.

Kövessük végig ugyanannak a virtuális függvénynek a hívását, mint az előbb, de most a `B` helyett használjuk a `D` objektumot:

```
B *bp = new D;
bp->f3(12);
```

A fordító ugyanazt a műveletsort fogja előállítani, mint az előbb, de ebben az esetben futásidőben a `B::f3` helyett a `D::f3` hívódik meg:

```
(*(bp->vptr)[2])(bp, 12)
```

A virtuális függvényeket kezelő rendszer működése még jobban látszik egy valóban többalakú kód esetén, ahol a kezelendő objektumok típusa nem pontosan ismert:

```
B *bp = getSomeSortOfB();
bp->f3(12);
```

A fordító által előállított, a virtuális függvények hívására szolgáló kód újrafordítás nélkül képes meghívni bármilyen, a `B`-ből levezetett osztály `f3` függvényét, még akkor is, ha az osztály még nem is létezik.

Az alaposztály egy függvényének felülbírálása a gyakorlatban tulajdonképpen azt jelenti, hogy a virtuális függvények táblájában a megfelelő mutatót az alaposztály függvényéről a származtatott osztály saját függvényére állítjuk át az objektum létre-

hozásakor. Példánknál maradva, a D osztály felülbírálta az alaposztály f1 és f3 függvényét, örökölte az f2 nevűt, és létrehozott egy új, f4 nevű virtuális függvényt. Mindezt a D osztály virtuális függvényeit címző táblája pontosan tükrözi.

Többszörös öröklés esetén a virtuális függvények kezelésének módja valamivel bonyolultabb, de lényegét tekintve ugyanazokat az alapelveket követi. A dolog összetettsége abból ered, hogy többszörös öröklésnél az objektumok több alaposztálybeli alobjektumot is tartalmaznak, így több különböző érvényes címük lehet. Vegyük például a következő hierarchiát:

```
class B1 { /* . . . */ };
class B2 { /* . . . */ };
class D : public B1, public B2 { /* . . . */ };
```

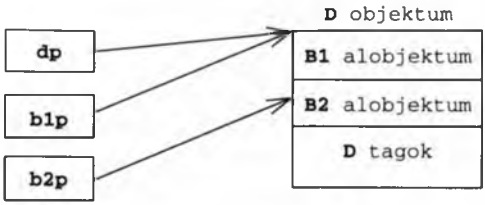
Egy származtatott osztályba tartozó objektumot bármely nyilvános alaposztályának felületén keresztül kezelhetünk. Éppen ez a típusok továbbfejleszthetőségének alapja. Ennek megfelelően egy D típusú objektumot D, B1 vagy B2 osztályt címző mutatókon vagy hivatkozásokon keresztül egyaránt kezelhetünk:

```
D *dp = new D;
B1 *b1p = dp;
B2 *b2p = dp;
```

A származtatott osztály objektumai az alobjektumok közül természetesen csak az egyiket tárolhatják a 0 eltolási címen, így az alaposztályok alobjektumai általában abban a sorrendben épülnek be a származtatott osztályok objektumaiba, amilyen sorrendben az osztály bevezeti azokat. D típusú objektumok esetében tehát először a B1, majd a B2 alaposztály alobjektuma következik, amint azt a 7.7. ábra mutatja. (Lásd még a 38. hibát.)

Egészítsük ki ezt az egyszerű többszörös öröklési hierarchiát néhány újabb virtuális függvénnyel:

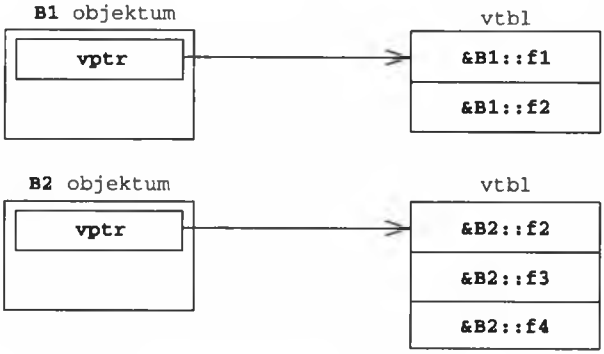
```
class B1 {
 public:
   virtual void f1();
   virtual void f2();
};
class B2 {
 public:
   virtual void f2();
   virtual void f3( int );
   virtual void f4();
};
```



7.7. ábra

Egy objektum valószínű memóriaképe többszörös öröklés esetén.

A `B1` és `B2` osztályok egyaránt tartalmaznak virtuális függvényeket, így az ilyen típusú objektumok mind tartalmazni fognak egy-egy mutatót az adott osztályra jellemző virtuális függvények táblájára. (Lásd 7.8. ábrát.)

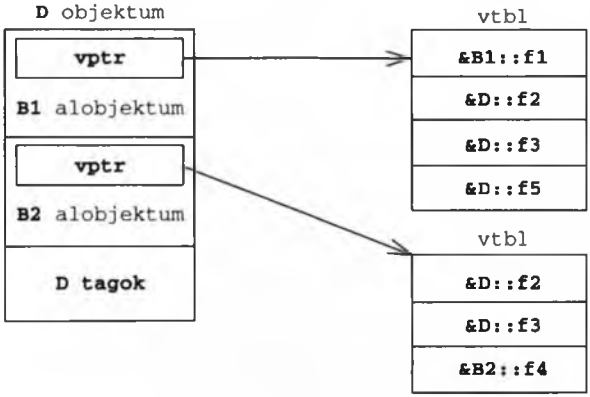


7.8. ábra

Két lehetséges alaposztály.

Egy `D` típusú objektum egyaránt tekinthető `B1` és `B2` típusúnak is, így egyszerre két mutatót is tartalmaz két különböző virtuálisfüggvény-táblára. (Lásd a 7.9. ábrát)

```
class D : public B1, public B2 {
 public:
   void f2();
   void f3( int );
   virtual void f5();
};
```



7.9. ábra

A virtuális függvények kezelésének egy lehetséges megvalósítása többszörös öröklés esetén. A teljes objektum felülbírálja mindkét alaposztályának egyes függvényeit.

Figyeljük meg, hogy a `D::f2` mindkét alaposztály megfelelő függvényét felülbírálja. Ha egy származtatott osztály egy függvényének megegyezik a neve és minden tulajdonsága több alaposztály megfelelő függvényeinek nevével és tulajdonságaival, akkor valamennyi alaposztálybeli függvényt felülbírálja. Ez a szabály nemcsak a közvetlen, hanem a közvetett alaposztályok függvényeire is vonatkozik. Vegyük észre azt is, hogy bár a `D` osztályban bevezetünk egy új virtuális függvényt (`D::f5`), a fordító nem helyez el egy újabb virtuálisfüggvény-táblát címző mutatót az objektum `D`-re vonatkozó részében. Ez azért van így, mert a származtatott osztályokban bevezetett virtuális függvények mutatói általában valamelyik alaposztály megfelelő táblájába kerülnek be.

Azért van itt egy kis gond. Nézzük a következő kódrészletet:

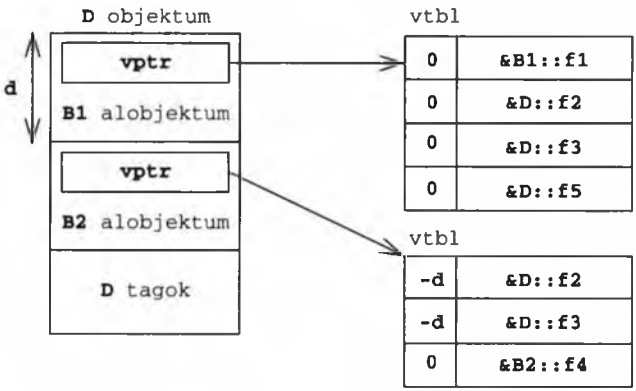
```
B2 *b2p = new D;
b2p->f3(12);
```

Itt azt a szokásosnak nevezhető eljárást alkalmazzuk, hogy a származtatott osztály egy objektumát az alaposztály felületén keresztül kezeljük. Ha azonban most is ugyanazt a hívási sorrendet próbáljuk használni, mint korábban az egyszeres öröklésnél, a `this` mutató tartalma hibás lesz:

```
(*(b2p->vptr)[1])(b2p,12)
```

A gondot az okozza, hogy a dinamikusan a `D::f3` függvényhez kötődő hívás a rejtett `this` paraméterben a `D` objektum mutatóját várja. A `b2p` azonban sajnos a `B2` alobjektum kezdetére mutat, ez pedig néhány bájtnyira az őt tartalmazó `D` objektum kezdete után található. (Lásd a 7.7. ábrát.) A `b2p` tartalmát tehát hívás közben módosítani kell úgy, hogy valóban a `D` objektum kezdetére mutasson.

Szerencsére a fordító pontosan tudja az ehhez szükséges eltolás hosszát, hiszen amikor a származtatott osztály virtuális függvényeit címző táblát készíti, tisztában kell lennie az összes alaposztály alobjektumainak pontos helyével. A javítás tényleges elvégzésének több bevált módja is létezik, kezdve a tényleges függvényhívás elé helyezett apró kódrészletektől egészen a több belépési ponttal rendelkező tagfüggvényekig. A legegyszerűbb megoldás talán az, ha a szükséges eltolás nagyságát a fordító bejegyzi a virtuális függvények táblájába, majd a hívási sorrendnél ezt figyelembe veszi (lásd a 7.10. ábrát).



7.10. ábra

A virtuális függvények egyik lehetséges megvalósítása a sok közül, többszörös öröklés esetén. Ez a módszer a this mutató eltolási értékeit a virtuális függvények táblájában tárolja.

A virtuálisfüggvény-tábla elemei immár nem egyszerű mutatók, hanem olyan kis szerkezetek, amelyek a megfelelő tagfüggvény-mutatók (`fptr`) mellett a `this` értékéhez hozzáadandó eltolási értékeket (`delta`) is tartalmazzák. Maga a hívási sorrend a következő:

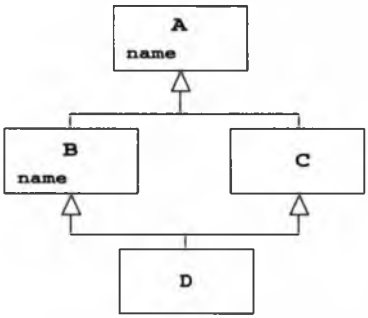
```
(*(b2p->vptr)[1].fptr)(b2p+(b2p->vptr)[1].delta,12)
```

A fenti kód hatékonysága igen jól növelhető, tehát tényleges megvalósítása egyáltalán nem olyan számításigényes, mint amilyennek ebből a sorból látszik.

79. hiba: Az elsőbbséggel kapcsolatos gondok

Talán az Olvasó is elcsodálkozott már néha azon, hogyan sikerült odáig süllyednie, hogy olyan nyelven ír programokat, amelyben „barátokról", „privát szféráról", „közvetett barátokról" és „elsőbbségi szabályokról" van szó. Most éppen ez utóbbiakat, a hierarchiák tervezésével kapcsolatos elsőbbségi kérdéseket tárgyaljuk meg: miért olyan különösek ezek, és miért lehetnek időnként nélkülözhetetlenek. Persze az Olvasó gondolhatja, hogy az ő életében ilyen kérdések soha nem merülhetnek fel, de a gyakorlott C++ programozók előbb vagy utóbb szemtől szembe találják magukat egy elsőbbségi problémával, vagy ha nem, hát kollégájuk mutat nekik egyet. Így aztán jobb, ha előre felkészülünk erre is. Aki felkészült a legrosszabbra, azt nem érheti meglepetés.

Az elsőbbség (dominancia) problémája kizárólag a virtuális öröklésre korlátozódik, és a legegyszerűbben egy rajzzal szemléltethető. A 7.11. ábrán látható rendszerben a `B::name` azonosító elsőbbséget élvez az `A::name` azonosítóval szemben, ha `A` a `B` alaposztálya. Ez az összes keresési útvonalra érvényes. Ha a fordító például a `D` osztály hatókörén belül a `name` azonosítót keresi, akkor megtalálja a `B::name` és egy másik úton az `A::name` azonosítót is. Az elsőbbségi szabályoknak köszönhetően azonban a helyzet egyértelmű: a hivatkozás a `B::name` tagra vonatkozik, annak magasabb rangja miatt.

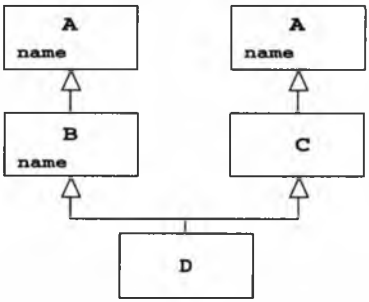


7.11. ábra

A B::name azonosító magasabb rangú, mint az A::name.

Virtuális öröklés nélkül viszont ugyanez a helyzet nem kezelhető egyértelműen. A 7.12. ábrán bemutatott rendszerben a `name` azonosító keresése nem oldható meg

egyértelműen a `D` osztály hatókörén belül. A `C` osztályban a `B::name` rangja nem magasabb az `A::name` rangjánál, így a fordító nem tudja eldönteni, mire gondoltunk.



7.12. ábra

Ebben a rendszerben nem működnek az elsőbbségi szabályok. A `B::name` azonosító elrejti az `A::name` azonosítót az egyik keresési úton, de a másikon nem.

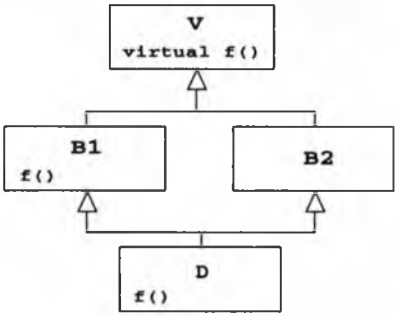
Az elsőbbségek rendszere elsőre meglehetősen furcsa nyelvi szabály látszatát kelti, az igazság azonban az, hogy nélküle számos esetben lehetetlen lenne előállítani a virtuális függvények tábláját a virtuális öröklést használó rendszerekben. Röviden tehát az elsőbbségi szabályok létezése a dinamikus kötés és a virtuális öröklés szükségszerű következménye.

Nézzünk egy egyszerű virtuális öröklési rendszert (7.13. ábra). A `D` típusú objektumok három alobjektumot tartalmaznak, a három alaposztálynak megfelelően. Ezek valamennyien tartalmaznak egy-egy mutatót a megosztott `V` alobjektumra (lásd a 7.14. ábrát). (Számos különböző megvalósítás lehetséges. Az itt bemutatott egy kissé túlhaladott, de könnyen követhető, és logikailag azonos a többivel.)

Amint az el is várható, a `D::f` tagfüggvény deklarációja (lásd a 7.13. ábrán) egyaránt felülbírálja a `B1::f` és `V::f` megvalósításokat:

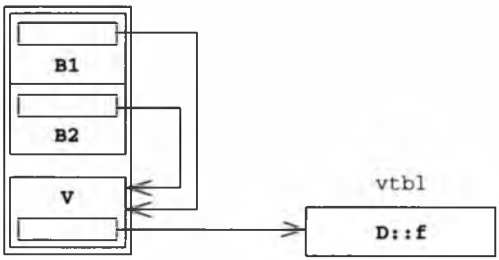
```
B2 *b2p = new D;
b2p->f();  // A D::f függvényt hívja meg
```

Lássunk egy másik esetet (7.15. ábra). Ez a rendszer nem valósítható meg, mivel mind a `B1::f`, mind a `B2::f` felülbírálhatja a `D`-ben található `V::f` függvényt. Mivel az ilyen helyzetekben a fordító nem tud dönteni, fordítási hibát kapunk.



7.13. ábra

A D::f tagfüggvény egyaránt felülbírálja a B1::f és V::f függvényeket. A B1 és V alobjektumokban található virtuálisfüggvény-táblák a D::f hívására vonatkozó információt tartalmaznak.



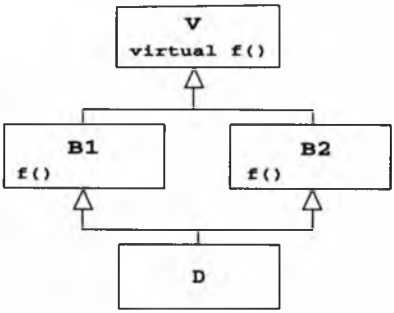
7.14. ábra

A D teljes objektum egy lehetséges felépítése a V alobjektum virtuálisfüggvény-táblájával.

Végezetül nézzünk egy olyan esetet, amikor az elsőbbségi szabályok életbe lépnek:

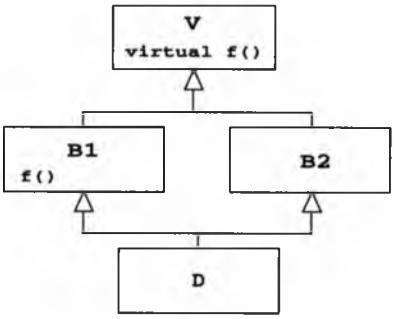
```
B2 *b2p = new D;
b2p->f(); // A B1::f() függvényt hívja meg
```

Amint a 7.16. ábrán látható, a `B1::f` azonosító minden keresési úton elsőbbséget élvez a `V::f` azonosítóval szemben, így a `D` objektum `V` alobjektumának virtuálisfüggvény-táblájába a `B1::f` címe kerül be. Az elsőbbségi szabály nélkül ezt az esetet sem lehetne kezelni, mivel a `V` táblázatába a `V::f` és a `B1::f` címe egyaránt bekerülhetne. A szabály azonban feloldja az ellentmondást és a fordító a `B1::f` megvalósításra voksol.



7.15. ábra

Egy megoldhatatlan helyzet. Mind a B1::f, mind a B2::f felülbírálhatja a D-ben található V::f függvényt a V alobjektum virtuálisfüggvény-táblájában.



7.16. ábra

Az elsőbbségi szabályok alkalmazásával a rendszerben levő ellentmondás feloldható. A B1::f elsőbbséget élvez a V::f azonosítóval szemben, így a V alobjektum virtuális függvényeket címző táblájába az előbbi címe kerül be.

8

Az osztályok megtervezése

Az elvont adattípusok hatékony megtervezése legalább annyira művészet, mint amennyire tudomány. A jó felületek tervezéséhez nagyjából egyenlő arányban van szükségünk technikai tudásra, pszichológiai ismeretekre és szakmai gyakorlatra. A könnyen érthető és jól karbantartható kód létrehozása során semmi sem olyan fontos, mint az átlátható felületek megvalósítása.

Ebben a fejezetben az osztályok felületének megtervezésével kapcsolatos leggyakoribb hibákról esik majd szó. Közben megemlítek néhány olyan, a megvalósítással kapcsolatos részletet, amelyek az osztályok felületére is kihatással lehetnek.

80. hiba: Beállító és lekérdező felületek

Az elvont adattípusoknál valamennyi adattagot privátként kell bevezetnünk. Ugyanakkor, ha egy osztály nem egyéb, mint privát adatok halmaza a hozzájuk tartozó nyilvános beállító és lekérdező függvényekkel, akkor az osztály nem igazán nevezhető elvontnak.

Emlékezzünk rá, hogy az adatelvonatkoztatás célja az, hogy a kód írói és olvasói számára lehetővé tegyük, hogy az egyes típusokról egymás közt folyó „tárgyalást" magának a problémának a nyelvén folytathassák le. Ennek megfelelően az elvont adattípus nem egyéb, mint a felhasználható által végezhető műveletek egy halmaza. Ezek a műveletek határozzák meg azt a „látásmódot", ahogy a kérdéses típust szemléljük, róla gondolkodunk. Vegyünk például egy vermet:

```
template <class T>
class UnusableStack {
 public:
   UnusableStack();
   ~UnusableStack();
   T *getStack();
   void setStack( T * );
   int getTop();
   void setTop( int );
 private:
   T *s_;
   int top_;
};
```

Az egyetlen pozitívum, amit erről a sablonról el lehet mondani, az, hogy legalább van rendes neve. Nincs itt szó semmiféle elvontságról, csupán valami vékony álruháról, ami az adatok egy halmazát hivatott álcázni. A megadott nyilvános felület nem teszi lehetővé a felhasználók számára, hogy a veremről elvont módon gondolkodjanak, sőt a verem megvalósításának változásaival szemben sem nyújtja az állandóság érzetét. Egy megfelelően megtervezett osztálynak tiszta elvonatkoztatási lehetőségeket kell nyújtania, felületének pedig el kell szigetelnie a felhasználót a tényleges megvalósítástól:

```
template <class T>
class Stack {
 public:
   Stack();
   ~Stack();
   void push( const T & );
   T &top();
   void pop();
   bool empty() const;
 private:
   T *s_;
   int top_;
};
```

Mindezeket szem előtt tartva valójában egyetlen programozó sem akarna olyan felületet előállítani, mint amit az `UnusableStack` kapcsán bemutattunk. Minden, a szakmájához valamit is értő programozó pontosan tudja, hogy egy verem megvalósításához milyen műveletek társulnak, így egy hatékony felület előállítása számukra tulajdonképpen automatikus. Ugyanakkor nem lehet ugyanezt elmondani valamennyi elvont adattípussal kapcsolatban, különösen akkor nem, ha olyan területen végezzük a programozást, ahol nem számítunk szakértőnek. Az ilyen esetekben nagyon fontos, hogy szorosan együttműködjünk a téma szekértőivel, hiszen csak tőlük tudhatjuk meg, milyen elvont adattípusokat érdemes létrehozni, és azoknak milyen műveleteket kell támogatniuk. Annak, hogy egy elvont típusrendszert nem szakértő tervezett, az egyik legbiztosabb jele az, ha rengeteg olyan osztály van a megvalósításban, amelyek beállító és lekérdező műveleteket valósítanak meg.

Megeshet persze, hogy egy osztály működéséhez valóban szükség van ilyen egyszerű hozzáférést, vagyis beállítást és lekérdezést megvalósító függvényekre. Vajon hogyan valósíthatjuk meg ezeket? Nos, többféle stílus képzelhető el:

```
class C {
 public:
   int getValue1() const  // Beállítás és lekérdezés, 1. stílus
       { return value_; }
```

```
    void setValue1( int value )
        { value_ = value; }
    int &value2()           // Beállítás és lekérdezés, 2. stílus
        { return value_; }
    int setValue3( int value )        // Beállítás és lekérdezés,
➥ 3. stílus
        { return value_ = value; }
    int value4( int value ) {         // Beállítás és lekérdezés,
➥ 4. stílus
        int old = value_;
        value_ = value;
        return old;
    }
  private:
    int value_;
};
```

A másodikként bemutatott stílus a legtömörebb és legrugalmasabb, de egyben a legveszélyesebb is. Ha az osztály privát területére mutató leírót (handle) adunk vissza, azzal a `value2` függvényt tulajdonképpen szükségtelenné tesszük, hiszen semmivel sem nyújt többet, mint egy nyilvános adattag. Az osztály felhasználói továbbfejleszthetik szolgáltatásait, és közvetlenül hozzáférhetnek a privát részeihez. Ez a tervezési forma még akkor is gondot okoz, ha csak olvasási hozzáférést engedünk a kérdéses adatokhoz. Vegyünk például egy, a szabványos könyvtárból származó tároló segítségével megvalósított osztályt:

```
class Users {
 public:
   const std::map<std::string,User> &getUserContainer() const
       { return users_; }
   // . . .
 private:
   std::map<std::string,User> users_;
};
```

A lekérdező („get") függvény kiszolgáltatja azt a meglehetősen személyes információt, amit a felhasználói tároló objektum a szabványos `map` függvénnyel valósít meg. Minden kód, ami ezt a nyilvános függvényt meghívja, függőségi viszonyba kerülhet (és valószínűleg fog is kerülni) a `Users` osztály adott megvalósításával. Ha a későbbiekben valaki úgy dönt, hogy a `vector` típus használata mégis hatékonyabb megoldást eredményez, akkor a `Users` osztályt használó összes kódot át kell írni. Ilyen típusú hozzáférési függvényt tehát egyszerűen nem szabad alkalmazni a megvalósítás során.

A harmadik stílus kissé szokatlan, mivel ténylegesen nem ad hozzáférést a kérdéses taghoz, de képes annak új értéket adni, és visszatérési értéke is ez az új érték lesz. (A programozó itt feltételezi, hogy a régi értéket tudjuk valahonnan. Végül is tudjuk, hogy új értéket kell helyette beállítani. Vagy nem?) Ez a megvalósítás lehetővé teszi, hogy a felhasználó olyan kifejezéseket alkalmazzon, mint a `+= setValue3(12)`, ahelyett, hogy a szokásos kétlépéses minta szerint gondolkodna: `setValue(12); a += getValue1();`. Az egyetlen gond ezzel a módszerrel az, hogy a legtöbb programozó úgy fogja gondolni, hogy a visszaadott érték a régi beállítás. Ez pedig később igen nehezen felderíthető hibák forrása lehet.

A negyedik megvalósításnak megvan az a szépsége, hogy ugyanazzal a függvénnyel kérdezhetjük le, és állíthatjuk be a változót. Ugyanakkor, ha csak lekérdezni akarunk, szükség van egy kis találékonyságra:

```
int current = c.value4( 0 ); // Lekérdezés és beállítás
c.value4( current ); // Helyreállítás
```

Az aktuális érték egyszerű lekérdezéséhez meg kell adnunk egy „segédértéket", majd az ezzel „elrontott" tartalmat egy következő hívással vissza kell állítanunk. Bár a módszer elsőre kissé pongyolának tűnhet, valójában nagy múltra tekint vissza a C++ programozás történetében, olyannyira, hogy a szabványos könyvtár `set_new_handler`, `set_unexpected` és `set_terminate` függvényei is használják. Ezekkel a függvényekkel általában visszahívó függvényeket (callback function) kezelünk verem üzemmódban, anélkül, hogy ténylegesen létrehoznánk egy veremtípust:

```
typedef void (*new_handler)(); // A visszahívás típusa
// . . .
new_handler old_handler = set_new_handler( handler ); //
➥ Elhelyezés a verem tetején
// Csinálunk valamit...
set_new_handler( old_handler ); // Levesszük a verem tetejéről
```

Az aktuális leíróhoz való hozzáférés ezzel a módszerrel kissé bonyolult. A következő – a C++-ban általánosnak tekinthető – módszer erre ad megoldást:

```
new_handler handler = set_new_handler( 0 ); // Az aktuális leíró
➥ megszerzése
set_new_handler( handler ); // Helyreállítás
```

Az elmondottak ellenére – eltekintve a szabványos visszahívó függvények használatától – óvakodjunk ennek a módszernek az alkalmazásától. Túlságosan összetett,

ha egyszerűen csak olvasási hozzáférésre van szükségünk, gondokat okozhat a többszálú alkalmazásokkal és a kivételek kezelésével kapcsolatban, ráadásul sokan össze fogják téveszteni a korábban leírt harmadik módszerrel.

A mindenkinek mindenkorra javasolható beállítási és lekérdezési módszer az elsőként bemutatott. Ez a létező legegyszerűbb megvalósítás, hatékony, és ami még ennél is fontosabb, félreérthetetlen mind az író, mind az olvasó számára:

```
int a = c.getValue1(); // Lekérdezés, természetesen
c.setValue1( 12 ); // Beállítás, természetesen
```

Összefoglalva tehát, ha egy osztály megvalósításában lekérdező és beállító függvényekre van szükségünk, használjuk az első módszert.

81. hiba: Állandó és hivatkozás típusú adattagok

Az egyik általános jótanács, amit a C++ programozóknak adni lehet, úgy hangzik, hogy „mindent, ami állandónak tekinthető, állandóként adjunk meg". Egy másik, szintén igen hasznos tanács szerint „ha valaminek az értéke nem mindig állandó, az véletlenül se legyen állandó". Összevonva e két bölcsességet, a helyes tanács talán úgy fogalmazható meg, hogy „tegyél mindent állandóvá, amit csak lehet, de tudd, hogy hol a határ".

Ebben a részben arról próbálom meggyőzni az Olvasót, hogy állandó vagy hivatkozás típusú adattagokat ritkán van értelme felvenni egy osztályba. Az ilyen tagok nehezebben kezelhetővé teszik az osztályokat, rendhagyó módon értelmezik a másolást, a karbantartókat pedig arra késztetik, hogy veszélyes változtatásokat vezessenek be.

Nézzünk egy egészen egyszerű osztályt, amelynek állandó és hivatkozás típusú tagjai is vannak:

```
class C {
 public:
   C();
   // . . .
 private:
   int a_;
```

```
    const int b_;
    int &ra_;
};
```

Az állandó és hivatkozás típusú tagokat a konstruktornak kell beállítania:

```
C::C()
    : a_( 12 ), b_( 12 ), ra_( a_ )
    {}
```

Eddig semmi gond. Vezessünk most be néhány `C` típusú objektumot, és állítsuk be őket:

```
C x; // Alapértelmezett konstruktor
C y( x ); // Másoló konstruktor
```

Hoppá! Honnan is származik az a bizonyos másoló konstruktor? A fordító írta meg nekünk, ami azt jelenti, hogy tagról tagra haladva hajtja végre a másolást, és `y` minden tagját az `x` megfelelő tagja alapján állítja be (lásd a 49. hibát). Ebből viszont az következik, hogy az `y` objektum `ra_` hivatkozása az `x` objektum `a_` tagjára fog mutatni. Ha már az általános jótanácsoknál tartunk, fogalmazzunk meg egyet erre is: „Ha az általad tervezett osztály más adatra vonatkozó leírót tartalmaz (általában mutatót vagy hivatkozást), akkor gondolkodj el rajta, nem lenne-e jobb, ha a másoló műveleteket magad írnád meg."

```
C::C( const C &that )
    : a_( that.a_ ), b_( that.b_ ), ra_( a_ )
    {}
```

Használjuk tovább az imént megadott `C` osztályt:

```
x = y; // Hiba!
```

Itt az a gond, hogy a fordító képtelen előállítani az értékadáshoz szükséges függvényt. Alapértelmezésként ez egy olyan függvény lenne, amely `x` minden tagjához `y` megfelelő tagjának értékét rendeli hozzá. A gond ezzel csak az, hogy a `C` típusú objektumok esetében a `b_` és a `ra_` nem szerepelhet értékadó műveletben. És ez jól is van így, hiszen még ha a fordító elő is állítana ilyen kódot, az akkor is ugyanolyan helytelenül működne, mint az alapértelmezett másoló konstruktor.

Ha jól meggondoljuk, valóban nem is egyszerű feladat megírni egy értékadó műveletet. Nézzük az első próbálkozást:

```
C &C::operator =( const C &that ) {
    a_ = that.a_; // Rendben.
    b_ = that.b_; // Hiba!
    return *this;
}
```

Egy állandónak nem adhatunk értéket. Az igazi veszély itt az, hogy egy „kreatív" karbantartó azért mégis meg fogja próbálni végrehajtani az értékadást. Az első ötlete valószínűleg a típusátalakítás lesz:

```
int *pb = const_cast<int *>(&b_);
*pb = that.b_;
```

Ez a kód futásidőben valószínűleg semmi gondot nem fog okozni, hiszen kevéssé valószínű, hogy a `b_` tag csak olvasható memóriaterületre kerül, hacsak nem egy állandó `C` típusú objektum tagja. Ugyanakkor ezt a megoldást nehezen lehetne természetesnek nevezni, hivatkozás típusú tagra pedig nem is működne. (Jegyezzük meg, hogy esetünkben az értékadó műveleten belül szükségtelen volt a `C` objektum hivatkozás típusú tagjának ismételt beállítása, mivel az már eleve a saját `a_` tagjára hivatkozott.)

Néhány kivételesen ötletes karbantartó más úton is próbálkozhat. A tényleges értékadás helyett megkísérelhetik megsemmisíteni az `x` objektumot, és újra létrehozni azt az `y` tartalma alapján:

```
C &C::operator =( const C &that ) {
    if( this != &that ) {
        this->~C(); // A destruktor hívása
        new (this) C(that); // Másoló konstruktor
    }
    return *this;
}
```

Az elmúlt években nagyon sok tintát fecséreltek mind e módszer bemutatására, mind arra, hogy az alkalmatlanságát bizonygassák. Bár ebben a konkrét esetben működne – legalábbis egy ideig –, túlságosan összetett, nem méretezhető, és nagyon valószínű, hogy a jövőben gondokat fog okozni. Nézzük például, mi történik, ha `C`-ből végül alaposztály lesz. Nagyon valószínű, hogy a származtatott osztályok

a C értékadó műveletét fogják használni. A destruktor meghívása – ha virtuális –, a teljes objektumot meg fogja semmisíteni, nem csak a C típusú részét. Ha a destruktor nem virtuális, működése megjósolhatatlan lesz. Vagyis kerüljük ezt a módszert.

A legegyszerűbb és legkönnyebb módszer az említett problémák elkerülésére, ha egyszerűen nem használunk állandó és hivatkozás típusú tagokat. Mivel osztályunk valamennyi adattagja privát (ugye az?!), tartalmuk már kellőképpen védett a véletlen módosításokkal szemben. Ha az állandó és hivatkozás típusokat csak azért használjuk, mert a fordítót meg akarjuk akadályozni az alapértelmezett értékadó művelet létrehozásában, annak egyszerűbb és szabványosabb módja is létezik:

```
class C {
   // . . .
 private:
   int a_;
   int b_;
   int *pa_;
   C( const C & ); // Másolva előállítás letiltása
   C &operator =( const C & ); // Értékadás letiltása
};
```

Összefoglalva tehát ritka az a helyzet, amikor állandó vagy hivatkozás típusú tagokra van szükségünk. Kerüljük ezek használatát.

82. hiba: Az állandó tagfüggvények félreértelmezése

Utasításforma

Az első szembeötlő dolog az állandó tagfüggvényekkel kapcsolatban a szokatlan utasításforma (szintaxis). A `const` minősítő a deklaráció végén szerepel, mintha csak egy elvetemült trükkről lenne szó. Pedig szó sincs ilyesmiről. Akár a C++-nak a C nyelvtől örökölt többi része, ez a megoldás is egyszerre következetes és megtévesztő:

```
class BoundedString {
 public:
   explicit BoundedString( int len );
   // . . .
   size_t length() const;
   void set( char c );
   void wipe() const;
```

```
  private:
    char * const buf_;
    int len_;
    size_t maxLen_;
};
```

Nézzük először a privát `buf_` adattag bevezetését. Ez egy karakter típust címző állandó mutató. (A példa szemléltető jellegű; lásd a 81. hibát.) A mutató, és nem az általa címzett tartalom az állandó, így a `const` minősítőnek a `*` után kell következnie. Ha a `const` szót a csillag elé írtuk volna, akkor állandó karaktert címző nem állandó mutatónk lenne.

Ugyanez a helyzet a `length` tagfüggvénnyel. Ha a `const` szót a függvény neve elé írtuk volna, azzal olyan tagfüggvényt vezettünk volna be, amelynek nincsenek paraméterei, visszatérési típusa pedig egy állandó `size_t` típus. A `const` szónak a deklaráció végén való szerepeltetése azt jelzi, hogy maga a függvény az állandó, és nem a visszatérési értéke.

Egyszerű jelentés és működés

Mit is jelent pontosan az, ha egy függvény állandó? A szokásos válasz erre az, hogy az állandó függvény nem módosítja azt az objektumot, amelynek tagja. Ez egy egészen egyszerű kijelentés, és a fordító is egész egyszerűen valósítja meg a gyakorlatban.

Minden nem statikus tagfüggvénynek van egy beleértett paramétere, amely nem egyéb, mint az őt tartalmazó objektum címe. A függvényen belül ezt a címet a `this` kulcsszó helyettesíti:

```
BoundedString bs( 12 );
cout << bs.length(); // A "this" most &bs
BoundedString *bsp = &bs;
cout << bsp->length(); // A "this" most bsp
```

Egy `X` típusú osztály nem állandó tagfüggvényénél a `this` mutató típusa `X * const`, vagyis egy állandó mutató egy nem állandó objektumra. A `this` mutató tartalma nem módosítható (mindig ugyanarra az `X` típusú objektumra fog mutatni), viszont `X` tagjait módosíthatjuk rajta keresztül. Egy nem állandó tagfüggvény belsejében minden nem statikus taghoz való hozzáférés alapja egy nem állandót címző mutató:

```
void BoundedString::set( char c ) {
    for( int i = 0; i < maxLen_; ++i )
```

```
        buf_[i] = c;
    buf_[maxLen_] = '\0';
}
```

Állandó tagfüggvény esetén a `this` mutató típusa `const C * const`, vagyis állandó objektumot címző állandó mutató. Ez azt jelenti, hogy sem a mutató, sem az általa címzett objektum tagjait nem lehet módosítani:

```
size_t BoundedString::length() const
    { return strlen( buf_ ); }
```

Lényegét tekintve tehát a függvényekkel kapcsolatban használható `const` kulcsszó arra ad lehetőséget, hogy a függvény rejtett `this` paraméterének típusát meghatározzuk. Nézzük például a `BoundedString` osztály egyenlőséget vizsgáló nem tag műveletének bevezetését:

```
bool operator ==( const BoundedString &lhs,
                  const BoundedString &rhs );
```

Ez a függvény nem módosítja a paramétereit, csak megvizsgálja azokat. Ennek megfelelően mindkét paramétert állandóként vezettük be. Ugyanezt megtehetnénk egy ehhez hasonló tagfüggvénnyel is:

```
class BoundedString {
    // . . .
    bool operator <( const BoundedString &rhs );
    bool operator >=( const BoundedString &rhs ) const;
};
```

Emlékezzünk rá, hogy a kétténye zős túlterhelt műveleti tagfüggvények bal paraméterüket a rejtett `this` mutató tartalmaként kapják meg. A jobb oldali tényezőt az általunk kifejezetten megadott formális paraméter beállítására használjuk. (Ez esetünkben az `rhs` nevű változó.) Példánkban a „nagyobb vagy egyenlő" művelet bevezetése megfelelő, ez ugyanis garantálja, hogy a függvény semmilyen módon nem változtatja meg a paraméterek értékét. A „kisebb, mint" művelet megadása ugyanakkor helytelen, mert csak a jobb oldal sértetlenségét biztosítja. A hibára valószínűleg akkor derül fény, amikor a „nagyobb vagy egyenlő" műveletet a lehető legegyszerűbb módon akarjuk megvalósítani:

```
bool BoundedString::operator >=( const BoundedString &rhs ) const
    { return !(*this < rhs); }
```

Erre a kódra fordítási hibát fogunk kapni, egész pontosan az `operator <` meghívása lesz hibás. Amikor ugyanis a `*this` értéket adjuk át ennek a műveletnek első paraméterként, egy nem állandó tagfüggvény `this` mutatóját akarjuk feltölteni egy állandó objektum címével.

Az állandó tagfüggvények jelentése

Az előzőekben leírtuk az állandó tagfüggvények kezelésének módját, azt azonban nem árultuk el, mi is pontosan ezek használatának értelme, vagy hogy mit illik egyáltalán állandó tagfüggvényként bevezetni és mit nem. Erre az utóbbi problémára nincs is szigorú szabály, mivel azt, hogy pontosan mit illik és mit nem, javarészt a C++ programozók társadalma állapítja meg, egyfajta hallgatólagos megállapodás alapján. Vegyük például a `BoundedString` osztály `wipe` nevű függvényét:

```
void BoundedString::wipe() const
   { buf_[0] = '\0'; }
```

Ez a kód működik, de attól, hogy valami működőképes, még nem feltétlenül sorolható a megengedett, a szakmabeliek által elvárt, vagy elfogadott dolgok közé. A `wipe` függvény valóban nem változtatja meg azt az objektumot, amelynek része, vagyis nem nyúl a `BoundedString` adattagjaihoz. Ugyanakkor megváltoztat olyan külső adatokat, amelyek aztán visszahatnak a kérdéses objektum működésére. A `this` mutató állandósága, amit a `const` kulcsszó biztosít, csak a `BoundedString` objektum elemeit védi a változtatás ellen, a külső adatokra ez a védelem nem terjed ki. Ugyanakkor ezek a külső adatok szerves részét képezik a `BoundedString` objektum logikai állapotának.

A `BoundedString` osztály felhasználóinak többségét kellemetlen meglepetésként fogja érni, amikor észreveszik, hogy egy állandó tagfüggvény meghívása után az objektum viselkedése megváltozott. Ezt a hatást jelzendő a `wipe` függvényt nem szabadna állandóként megadni, és ez az, amiért korábban a `set` függvény deklarációja nem tartalmazta a `const` kulcsszót, holott a helyzet és a fordító ezt megengedte volna.

Nézzük ellenben a `length` tagfüggvény megvalósítását. Ez az a függvény, amelynek teljesen világos módon állandónak kell lennie, hiszen a karakterlánc hosszának lekérdezése nem változtatja meg a vele kapcsolatos objektum logikai állapotát. A legegyszerűbb megvalósításnak nyilván a szabványos könyvtár `strlen` nevű függvényére kell támaszkodnia, akárcsak mi tettük a fenti példában. Valószínűleg ez a lehetséges megoldások legjobbika, hiszen egyszerű, gyors, és minden helyzet-

ben a megfelelő eredményt szolgáltatja. Ugyanakkor képzeljük el, hogy a karakterláncok felhasználásának átvizsgálásakor azt tapasztaljuk, hogy sok karakterlánc hosszára senki sem kíváncsi, mások hosszát ezzel szemben igen gyakran kérdezik le, és valamennyi karakterlánc meglehetősen hosszú. Ebben az esetben egy másik megoldás talán célravezetőbb lehet:

```
size_t BoundedString::length() const {
    if( len_ < 0 )
        len_ = strlen( buf_ );
    return len_;
}
```

A fent látható kód szerzője úgy döntött, hogy a karakterlánc hosszát magában a `BoundedString` objektumban fogja tárolni. Ez egy kis többletet jelen a beállítás során azokban az esetekben, ha a hosszra végül senki sem lesz kíváncsi, viszont nagyot nyerünk általa, ha a `length` függvényt többször is meghívja a felhasználó. Sajnos a fenti kódra fordítási hibát fogunk kapni azon a ponton, ahol a `len_` nevű változónak akarunk értéket adni. Elfelejtettük ugyanis, hogy éppen egy állandó tagfüggvényt írunk, amelynek nem áll módjában módosítani saját objektumát.

A problémát megoldhatnánk persze egyszerűen úgy, hogy a kérdéses függvény bevezetéséből elhagyjuk a `const` kulcsszót, csakhogy ezzel egyrészt feláldozzuk a célszerűség oltárán azt a logikus igényt, hogy a `length` függvény ne változtathassa meg az objektum logikai állapotát, másrészt ha magát az objektumot vezetjük be állandóként, lehetetlenné válik a karakterlánc hosszának meghatározása. (Ezzel kapcsolatban olvassuk el a 6. és 31. hibát.) Ezt a kompromisszumot persze megkötjük, de soha ne felejtsük el, hogy a megvalósítás nehézségeinek elvileg nem szabad a tervezést és az elvont adattípusok felületét befolyásolniuk.

Az ilyen helyzetekben általános és eléggé el nem ítélhető az a megoldás, hogy az egyébként állandó tagfüggvényben megfelelő típusátalakítás segítségével megszüntetjük az állandóság okozta kellemetlenségeket:

```
size_t BoundedString::length() const {
    if( len_ < 0 )
        const_cast<int &>(len_) = strlen( buf_ );
    return len_;
}
// . . .
BoundedString a(12);
int alen = a.length(); // Működni fog...
const BoundedString b(12);
int blen = b.length(); // Nem meghatározott!
```

Ha egy állandó objektumot annak konstruktorán és destruktorán kívül máshol is módosítani próbálunk, az eredmény megjósolhatatlan lesz. A fenti kódban a `b` objektum `length` függvényének meghívása lehet, hogy működni fog, de az is lehet, hogy rejtélyes dolgokat fog művelni valamikor a távoli jövőben, amikor a megrendelőnek már régen leszállítottuk a programot. Ezen pedig a legkisebb mértékig sem segít az, hogy az újhullámosok kedvenc típusátalakító műveletét, a `const_cast`-ot használtuk.

A megfelelő megoldás esetünkben az, ha a `len_` adattag deklarációját a `mutable` minősítővel egészítjük ki. Ebbe a tárolási osztályba csak nem statikus, nem állandó és nem hivatkozás típusú adattagok tartozhatnak. Maga a jelzés azt tudatja a fordítóval, hogy ezt az adattagot biztonságosan módosíthatják állandó és nem állandó tagfüggvények egyaránt.

```
class BoundedString {
   // . . .
 private:
   char * const buf_;
   mutable int len_;
   size_t maxLen_;
};
```

Összefoglalva tehát, a C++ közösség számára a `const` kulcsszó felbukkanása egy tagfüggvény bevezetésében „logikai állandóságot" jelent, vagyis azt, hogy az adott függvény hatására az objektum logikai állapota nem változik meg még akkor sem, ha fizikai értelemben változás áll be benne.

83. hiba: Az összerendelés és az ismertség megkülönböztetése

Magában a C++ nyelvben nem létezik mód a tulajdonjog (összerendelés) és a használhatóság (ismertség) megkülönböztetésére. Ez aztán számos különböző hibának lehet a forrása, a „memóriaszivárgástól" az álnevekig:

```
class Employee {
 public:
   virtual ~Employee();
   void setRole( Role *newRole );
   const Role *getRole() const;
```

```
    // . . .
  private:
    Role *role_;
    // . . .
};
```

A fent bemutatott felület alapján nem világos, hogy az `Employee` objektum tulajdonosa a `Role` nevű objektumnak, vagy csak hivatkozik egy ilyen külső objektumra, amelyen több különböző `Employee` típusú objektum osztozik. Gond ebből persze csak akkor származik, ha a felhasználó mást hisz a tulajdonlással kapcsolatban, mint amit a tervező gondolt:

```
Employee *e1 = getMeAnEmployee();
Employee *e2 = getMeAnEmployee();
Role *r = getMeSomethingToDo();
e1->setRole( r );
e2->setRole( r ); // hiba #1!
delete r; // hiba #2!
```

Ha a tervező úgy gondolta, hogy a `Role` objektum és annak tartalma a megfelelő `Employee` objektum tulajdona, akkor a `#1` jelű sor hibás, hiszen két `Employee` objektum osztozik egyetlen `Role` objektumon. Ennek annyi hátulütője szinte biztosan lesz, hogy a `Role` objektumot kétszer fogja törölni a két különböző destruktor, hiszen az `e1` és `e2` objektumok nem szándékosan osztoznak az egyetlen `Role` objektumon, így felkészítve sem lehetnek ennek a helyzetnek kezelésére.

A `#2` jelű sor már sokkal problémásabb. Itt az `Employee` osztály felhasználója úgy gondolja, hogy a `setRole` függvény másolatot készít a `Role` típusú paraméteréről, így az eredeti `Role` objektum már megsemmisíthető. Ha a tervező nem így gondolkodott, a művelet végrehajtása után mind az `e1`, mind az `e2` objektum semmit címző mutatókat fog tartalmazni.

Egy gyakorlottabb fejlesztő persze belenézhet a `setRole` függvény megvalósításába, és a következők egyikével találkozhat:

```
void Employee::setRole( Role *newRole ) // #1 változat
   { role_ = newRole; }

void Employee::setRole( Role *newRole ) { // #2 változat
   delete role_;
   role_ = newRole;
}
```

```
void Employee::setRole( Role *newRole ) { // #3 változat
   delete role_;
   role_ = newRole->clone();
}
```

A `setRole` első megvalósítása azt tükrözi, hogy az `Employee` tervezőjének elképzelése szerint a `Role` típusú objektumnak az `Employee` nem a tulajdonosa. Ez abból látszik, hogy a függvény meg sem kísérli törölni a már esetleg létező `Role` objektumot, mielőtt a mutatót egy új objektum címére állítaná át. (A kódot olvasva természetesen csak feltételezhetjük, hogy mindez az `Employee` osztály tervezőjének szándéka volt, és nem programozási hiba.)

A `#2`-vel jelzett megvalósításból az látszik, hogy az `Employee` tulajdonosa a saját `Role` objektumának és a függvény paramétere által címzett külső objektum fölött átveszi az uralmat. Ugyanez látszik a harmadik megoldáson is, de ott a függvény másolatot készít a paraméterben megadott külső objektumról. Ebben az esetben talán szerencsésebb lett volna a paramétert `const Role *` típusúként megadni a `Role *` helyett, hiszen a másoló művelet biztosan nem módosítja a paraméterként megadott objektum tartalmát. Hasonlóan szokatlan az első megvalósítás is, ahol egy megosztott objektumot adunk át egy mutatón keresztül, de azt nem állandóként adjuk meg.

Ugyanakkor egy elvont adattípus felhasználói az esetek többségében nem férnek hozzá annak konkrét megvalósításához. Ez amúgy jobb is így, hiszen ellenkező esetben az adott típusra támaszkodó programok túlságosan kötődnének egy konkrét megvalósításhoz, és a lehető legteljesebb adatrejtés elve is sérülne. A fenti `#1` jelű megvalósításban például abból, hogy a `setRole` függvény nem törli a már létező `Role` objektumot, nem egyértelmű, hogy az `Employee` osztály tervezője is meg akart osztani egy `Role` objektumot több `Employee` között, vagy ez csupán egy félreértésen alapuló programozási hiba. Ha azonban a felhasználói kód jelentős része ezen a feltételezésen alapul, akkor ez már nem hiba, hanem kiegészítő szolgáltatás.

Mivel – mint említettük – a tulajdonjog közvetlen jelzésére maga a C++ nyelv nem ad lehetőséget, saját elnevezési szabályokhoz, formális paramétertípusokhoz és (igen, ebben az esetben mégis) megjegyzésekhez kell folyamodnunk:

```
class Employee {
 public:
   virtual ~Employee();
   void adoptRole( Role *newRole ); // A tulajdonjog megszerzése
   void shareRole( const Role *sharedRole ); // Nem tulajdonos
void copyRole( const Role *roleToCopy ); // Klónozás
   const Role *getRole() const;
   // . . .
};
```

Az `adoptRole`, `shareRole` és `copyRole` nevek kellően szokatlanok ahhoz, hogy felkeltsék az `Employee` osztály felhasználójának figyelmét, és arra késztessék, hogy elolvassa a dokumentációt. Ha a megjegyzések rövidek és világosak, a karbantartók is használni és főleg frissíteni fogják azokat. (Lásd az 1. hiba leírását.)

A tulajdonlással kapcsolatos zűrzavar egyik gyakori felbukkanási helye a mutatókat tartalmazó tároló objektumok környéke. Vegyünk például egy mutatókat tartalmazó listát:

➡ 83. hiba/ptrlist.h

```
template <class T> class PtrList;
template <> class PtrList<void> {
   // . . .
};
template <class T>
class PtrList : private PtrList<void> {
 public:
   PtrList();
   ~PtrList();
   void append( T *newElem );
   // . . .
};
```

A gond itt megint annak a valószínű veszélye, hogy a tervező és a felhasználó nem ugyanazt fogja gondolni a tulajdonjogokkal kapcsolatban:

```
PtrList<Employee> staff;
staff.append( new Techie );
```

A fenti kód alapján a `PtrList` felhasználója valószínűleg úgy képzeli, hogy a tároló objektum tulajdonaként kezeli az `append` paramétereként megadott objektumokat, vagyis a `PtrList` destruktora törölni fogja az összes olyan objektumot, amelynek a mutatója szerepel a listában. Ha a tároló elem nem végzi el ezt a feltételezett törlést, az eredmény a szabad memória vészes fogyatkozása lesz. Az alább látható kód írója más feltételezéssel élt:

```
PtrList<Employee> management;
Manager theBoss;
management.append( &theBoss );
```

Ebben az esetben a `PtrList` felhasználója úgy képzeli, hogy a tároló objektum megsemmisítésekor nem törli az általa címzett objektumokat. Ha mégis így tenne, akkor a `PtrList` destruktora már felszabadított területet fog még egyszer törölni.

Az ilyen félreértések elkerülésének legbiztosabb módja, ha a szabványos könyvtár szolgáltatásaira és típusaira támaszkodunk. Ezeket éppen szabványosságuk miatt valamennyi gyakorlott programozó ismeri, és viselkedésükkel kapcsolatban nem kell feltételezésekre hagyatkoznia. A szabványos könyvtár mutatókkal kapcsolatos tároló objektumai egyébként magukat a mutatókat törlik megsemmisítésükkor, a mutatók által címzett objektumokat azonban nem:

```
std::list<Employee *> management;
Manager theBoss;
management.push_back( &theBoss ); // Helyes
```

Ha azt akarjuk, hogy a tároló az objektumokat is törölje, több választásunk is lehet. A legegyszerűbb természetesen az, ha a törlést magunk hajtjuk végre:

```
template <class Container>
void releaseElems( Container &c ) {
   typedef typename Container::iterator I;
   for( I i = c.begin(); i != c.end(); ++i )
       delete *i;
}
// . . .
std::list<Employee *> staff;
staff.push_back( new Techie );
// . . .
releaseElems( staff ); // Takarítás
```

Sajnos az ilyen, általunk megírt törlésről később könnyű megfeledkezni, a kódrészlet rossz helyre kerülhet, a karbantartás során törölhetik, a kivételekre pedig különösen érzékeny. Gyakran jobb választásnak bizonyul, ha a közönséges mutatók helyett valamilyen „okos mutató" objektumot használunk. (Jegyezzük meg, hogy a szabványos `auto_ptr` sablon nem használható tároló elemként, mivel a másolás értelmezése itt eltér a szabványos könyvtár által megkövetelttől. Ezzel kapcsolatban olvassuk el a 68. hibát.) Egy ilyen okos mutatóra láthatunk a következőkben egy egyszerű példát:

➡ 83. hiba/cptr.h

```
template <class T>
class Cptr {
 public:
   Cptr( T *p ) : p_( p ), c_( new long( 1 ) ) {}
   ~Cptr() { if( !--*c_ ) { delete c_; delete p_; } }
   Cptr( const Cptr &init )
       : p_( init.p_ ), c_( init.c_ ) { ++*c_; }
   Cptr &operator =( const Cptr &rhs ) {
```

```
        if( this != &rhs ) {
            if( !--*c_ ) { delete c_; delete p_; }
            p_ = rhs.p_;
            ++*(c_ = rhs.c_);
        }
        return *this;
    }
    T &operator *() const
        { return *p_; }
    T *operator ->() const
        { return p_; }
  private:
    T *p_;
    long *c_;
};
```

A tárolót itt úgy valósítottuk meg, hogy elemei okos mutatók legyenek az eddig alkalmazott közönséges mutatók helyett (lásd a 24. hibát). Amikor a tároló objektum törli az elemeit, az okos mutató törli az általa címzett objektumot:

```
std::vector< Cptr<Employee> > staff;
staff.push_back( new Techie );
staff.push_back( new Temp );
staff.push_back( new Consultant );
// Nincs szükség sajátkezű törlésre
```

Az ilyen okos mutatók lehetséges felhasználási területe az ennél összetettebb helyzetekre is kiterjed:

```
std::list< Cptr<Employee> > expendable;
expendable.push_back( staff[2] );
expendable.push_back( new Temp );
expendable.push_back( staff[1] );
```

Amikor az `expendable` tároló érvényessége megszűnik, helyesen törli a második, `Temp` nevű elemét, és csökkenti első és harmadik elemének hivatkozásszámát. Ez utóbbiaknak a `staff` tárolóval együtt tulajdonosa, így ezeket majd a `staff` destruktora fogja törölni.

84. hiba: Műveletek nem megfelelő túlterhelése

Ha a szükség úgy kívánja, meg lehet lenni a műveletek túlterhelése nélkül is:

```
class Complex {
 public:
   Complex( double real = 0.0, double imag = 0.0 );
   friend Complex add( const Complex &, const Complex & );
   friend Complex div( const Complex &, const Complex & );
   friend Complex mul( const Complex &, const Complex & );
   // . . .
};
// . . .
Z = add( add( R, mul( mul( j, omega ), L ) ),
   div( 1, mul( j, omega ), C ) ) );
```

A műveletek túlterhelésének lehetősége csak amolyan nyelvi finomság, bár a kódot kétségkívül ízlésesebbé teszi, és abban is segít, hogy a tervező átadhassa a gondolatait a program olvasójának vagy karbantartójának:

```
class Complex {
 public:
   Complex( double real = 0.0, double imag = 0.0 );
   friend Complex operator +( const Complex &, const Complex & );
   friend Complex operator *( const Complex &, const Complex & );
   friend Complex operator /( const Complex &, const Complex & );
   // . . .
};
// . . .
Z = R + j*omega*L + 1/(j*omega*C);
```

A váltóáramú ellenállás kiszámításának közbevetett műveleti jelet használó változata helyes, az előző, függvényhívással megoldott azonban helytelen. Ugyanakkor ez a hiba nem különösebben feltűnő, éppen a megfelelő műveletek használatának hiánya miatt.

A műveletek túlterhelését akkor is célszerű használni, ha egy már létező keretrendszert fejlesztünk tovább, egészítünk ki, vagy csak használunk. Ilyen lehet például az `iostream` könyvtár vagy a szabványos sablonkönyvtár:

```
ostream &operator <<( ostream &os, const Complex &c )
   { return os << '(' << c.r_ << ", " << c.i_ << ')'; }
```

Ezek a sikeres felhasználások aztán néhány újdonsült programozót arra bátorítanak, hogy túlságosan is támaszkodjanak a műveletek túlterhelésére:

```
template <typename T>
class Stack {
 public:
   Stack();
   ~Stack();
   void operator +( const T & ); // Verembe helyezés (push)
   T &operator *(); // top
   void operator -(); // Kivétel a veremből (pop)
   operator bool() const; // Nem üres?
   // . . .
};
// . . .
Stack<int> s;
s + 12;
s + 13;
if( s ) {
   int a = *s;
   -s;
   // . . .
```

Nevezhetjük-e ezt a megoldást okosnak? Nem, ez gyerekesen ostoba. A műveletek túlterhelésének lehetősége elsősorban azt a célt szolgálja, hogy segítségével a kódot világosabbá tehessük. Nem azért van, hogy a programozó parádézhasson vele. Egy túlterhelt műveletnek mindenekelőtt úgy kell viselkednie, ahogy azt bármely gyakorlott olvasó elvárná tőle. Ennek megfelelően egy verem megfelelő megvalósításának közismert függvényneveket kell használnia és nem műveletekét:

```
template <typename T>
class Stack {
 public:
   Stack();
   ~Stack();
   void push( const T & );
   T &top();
   void pop();
   bool isEmpty() const;
   // . . .
};
// . . .
Stack<int> s;
s.push( 12 );
```

```
s.push( 13 );
if( !s.isEmpty() ) {
   int a = s.top();
   s.pop();
   // . . .
```

Jegyezzük meg, hogy egy túlterhelt művelet jelentésének mindenki számára világosnak kell lennie ahhoz, hogy használható is legyen. Ha mi és kollégáink 75 százaléka érti, miről van szó, az sajnos nem megfelelő. 25 százaléknyi félreértés és téves használat több gondot okoz, mint amennyit a művelet megold.

Ezen a ponton töredelmesen be kell vallanom, hogy én is követtem már el ilyen hibát egy egyszerű tömbsablon tervezése során:

➡ 5. hiba/array.h

```
template <class T, int n>
class Array {
 public:
   Array();
   explicit Array( const T &val );
   Array &operator =( const T &val ); // Mindenki számára
➥ egyértelmű?
   // . . .
 private:
   T a_[n];
};
// . . .
Array<float,100> ary( 0 );
ary = 123; // Egyértelmű?
```

Meg voltam róla győződve, hogy a fenti értékadás mindenki számára egyértelmű. Úgy gondoltam, mindenki számára világos, hogy a `123` értéket akarom hozzárendelni a tömb valamennyi eleméhez. Ugye világos? Nos, kollégáim többségének sajnos nem volt az. Némelyik gyakorlott programozó például azt hitte, hogy át akarom méretezni a tömböt úgy, hogy `123` eleme legyen. Egyesek úgy gondolták, hogy csak az első elemnek akarom a `123`-at értékül adni. Én persze pontosan tudom, hogy amit csináltam, az helyes, és hogy a többiek tévedtek, ez a gyakorlati tapasztalat mégis arra késztetett, hogy visszavonjam a fenti megoldást, és egy egyszerű, mindenki számára egyértelmű függvénnyel valósítsam meg ugyanezt a szolgáltatást:

```
ary.setAll( 123 ); // Unalmas, de világos.
```

A tanulságot talán úgy lehetne összefoglalni, hogy hacsak nem teljesen világos egy művelet túlterhelésének jelentése, ne használjuk ezt a megoldást.

85. hiba: Elsőbbség és túlterhelés

Egy műveletnek más műveletekkel szembeni rangja (elsőbbsége, precedenciája) a művelet viselkedésének szerves része, hiszen a felhasználónak ezzel kapcsolatban bizonyos tapasztalatai, elvárásai vannak. Ha az általunk megvalósított túlterhelt művelet nem felel meg ezeknek az elvárásoknak, a felhasználó tévesen fogja azt alkalmazni. Vegyük például a komplex számok egy nem szabványos megvalósítását:

```
class Complex {
 public:
   Complex( double = 0, double = 0 );
   friend Complex operator +( const Complex &, const Complex & );
   friend Complex operator *( const Complex &, const Complex & );
   friend Complex operator ^( const Complex &, const Complex & );
   // . . .
};
```

Láthatólag azt szeretnénk, ha létezne egy komplex számokra működő hatványozó műveletünk. A gond csak az, hogy magának a C++-nak nincs hatványozó művelete. Mivel új műveletet nem vezethetünk be, úgy döntöttünk, hogy egy már létező, de a komplex számokra nem alkalmazható műveletet fogunk felhasználni a célra. Ez a „logikai kizáró vagy" művelete.

Ezzel már önmagában is okoztunk egy érdekes problémát, hiszen minden gyakorlott C vagy C++ programozó úgy fogja kiolvasni az `a^b` műveletet, hogy „hajtsunk végre kizáró vagy műveletet a és `b` tartalma között". Van azonban itt egy súlyosabb baj:

```
a = -1 + e ^ (i*pi);
```

A matematikában és minden olyan nyelvben, amely támogatja a hatványozás műveletét, a hatványozásnak meglehetősen magas rangja van. Ennek megfelelően az alább látható kód olvasója minden bizonnyal úgy fogja gondolni, hogy a hatványozás elsőbbséget élvez az összeadással szemben:

```
a = -1 + (e ^ (i*pi));
```

A fordítóprogramnak persze fogalma sincs a hatványozással kapcsolatos elvárásokról. Felismeri a kizáró vagy műveletet, és ennek megfelelően értelmezi a kifejezést:

```
a = (-1 + e) ^ (i*pi);
```

Ebben a helyzetben az egyértelműség szent nevében természetesen sokkal hasznosabb, ha nem kísérletezünk a művelettúlterheléssel, hanem közönséges függvényhívást használunk:

```
a = -1 + pow( e, (i*pi) );
```

A művelet rangja része a művelet felületének. Ennek megfelelően túlterhelés esetén is gondoskodnunk kell arról, hogy az új művelet a felhasználók elvárásainak megfelelően működjön.

86. hiba: Barát és tag műveletek

Egy túlterhelt műveletnek lehetővé kell tennie mindazokat az átalakításokat, amelyek a paramétereivel kapcsolatban általában is megengedettek:

```
class Complex {
 public:
   Complex( double re = 0.0, double im = 0.0 );
   // . . .
};
```

A `Complex` osztály konstruktora például lehetővé teszi a szokványos számtípusok `Complex` típussá alakítását. A nem tag `add` függvény ugyanezt az átalakítást „beleértve" teszi lehetővé a paramétereivel kapcsolatban:

```
Complex add( const Complex &, const Complex & );
Complex c1, c2;
double d;

add(c1,c2);
add(c1,d); // add( c1, Complex(d,0.0) )
add(d,c1); // add( Complex(d,0.0), c1 )
```

A nem tag `operator +` ugyancsak lehetővé teszi tényezői típusának beleértett átalakítását:

```
Complex operator +( const Complex &, const Complex & );
c1 + c2;
operator +(c1,c2); // Ugyanaz, mint feljebb
c1 + d;
```

```
operator +(c1,d); // Ugyanaz, mint feljebb
d + c1;
operator +(d,c1); // Ugyanaz, mint feljebb
```

Ha azonban a `Complex` osztály elemeivel kapcsolatos összeadást egy tagfüggvény segítségével valósítjuk meg, aszimmetriát viszünk a rendszerbe a típusátalakítással kapcsolatban:

```
class Complex {
 public:
   // tag műveletek
   Complex operator +( const Complex & ) const; // Kéttényezős
   Complex operator -( const Complex & ) const; // Kéttényezős
   Complex operator -() const; // Egytényezős
   // . . .
};
// . . .
c1 + c2; // Remek.
c1.operator +(c2); // Remek.
c1 + d; // Remek.
c1.operator +(d); // Remek.
d + c1; // Hiba!
d.operator +(c1); // Hiba!
```

A tagfüggvények első paraméterére a fordító nem tud beleértett, felhasználó által meghatározott típusátalakítást alkalmazni. Ha egy ilyen átalakítás része a megvalósításnak, az azt jelzi, hogy a kéttényezős művelet tagfüggvényként való megvalósítása nem megfelelő megoldás. A tagviszonyban nem álló barátok (friend) lehetővé teszik első paraméterük típusának átalakítását, a tagok azonban csak az osztálytípusok bővítését.

87. hiba: A növeléssel és csökkentéssel kapcsolatos gondok

Még a legjobb C programozók is keverik az előtagként használt (prefix) és utótagként használt (postfix) növelő és csökkentő műveleteket azokon a helyeken, ahol ezt megtehetik:

```
int j;
for( j = 0; j < max; j++ ) /* A C nyelvben jó. */
```

Ez a C++-ban már divatjamúlt. Mindazokon a helyeken, ahol az előtagkénti és az utótagkénti írásmód is megengedett, az előtag előnyt élvez. A magyarázat a műveletek túlterhelésének lehetőségében rejlik.

A csökkentő és növelő műveletek túlterhelésével gyakran élünk a bejárók (iterátorok) és az okos mutatók megvalósítása során. Ezek a túlterhelt példányok lehetnek tag és nem tag műveletek egyaránt, de a leggyakoribb megoldás az, hogy osztályok tagjaiként adjuk meg őket:

```
class Iter {
 public:
    Iter &operator ++(); // prefix
    Iter operator ++(int); // postfix
    Iter &operator --(); // prefix
    Iter operator --(int); // postfix
    // . . .
};
```

A prefix műveletektől elvárható, hogy módosítható balértéket adjanak vissza, hiszen ez felel meg a nyelvben eleve meglevő társaik működésének. Ez a gyakorlatban azt jelenti, hogy a műveletnek egy saját objektumára vonatkozó hivatkozást kell visszaadnia:

```
Iter &Iter::operator ++() {
    // A *this növelése
    return *this;
}
// . . .
int j = 0;
++++j; // Rendben, de j+=2 jobb lett volna
Iter i;
++++++++i; // Rendben, de furcsa
```

Az utótag formát az előtagtól úgy különböztetjük meg, hogy egy használaton kívüli egész paramétert adunk meg. A fordító ebben egyszerűen egy nullát fog elhelyezni a hívás során, a két használat megkülönböztetése végett:

```
Iter i;
++i; // ugyanaz, mint az i.operator ++();
i++; // ugyanaz, mint az i.operator ++(0);
i.operator ++(1024); // Megengedett, de furcsa
```

A postfix műveletek megvalósítása általában nem támaszkodik erre az egész paraméterre. A nyelv saját utótagos növelő és csökkentő műveleteinek tökéletes utánzása végett egy túlterhelt utótag műveletnek vissza kell adnia tényezőjének (operandusának) egy másolatát, mielőtt a műveletet (növelést vagy csökkentést) elvégezné rajta. Az utótagos műveletet általában a megfelelő előtagos művelet segítségével valósítják meg:

```
Iter Iter::operator ++(int) {
    Iter temp( *this );
    ++*this;
    return temp;
}
```

Általános elvárás, hogy egy postfix művelet érték szerint adja vissza az eredményt. Ez pedig azt jelenti, hogy ha a tényező osztályelem, az utótagos művelet működése akkor is lassabb lesz, mint a megfelelő előtagos változaté, ha alkalmazzuk az olyan kódátalakítási fogásokat, mint amilyen a névvel rendelkező visszatérési érték használata (lásd az 58. hibát). Vegyük például a szabványos sablonkönyvtár egy lehetséges felhasználását:

```
vector<T> v;
// . . .
vector<T>::iterator end( v.end() );
for( vector<T>::iterator vi( v.begin() ); vi != end;
    vi++ ) { // Esetlen!
    // . . .
```

A `vector` típus bejáróját megvalósíthatjuk egy egyszerű mutató formájában. Ebben az esetben az utótagos növelés okozta számítási többlet elenyésző. Ugyanakkor, ha a bejáró osztály típusú, az eltérés a két használat között már jelentős. A C++ programozás gyakorlatában tehát általában célszerűbb az előtagos növelést és csökkentést használni. Ugyanakkor vannak olyan esetek is, amikor a programozó annyira komolyan veszi ezt az alapszabályt, hogy mindenáron igyekszik elkerülni az utótagos műveletek felbukkanását:

```
template <typename In, typename Out>
Out myCopy( In b, In e, Out r ) {
    while( b != e ) {
        // inkább, mint *r++ = *b++
        *r = *b;
        ++r;
        ++b;
```

```
    }
    return r;
}
```

Jegyezzük meg, hogy a nyelv saját utótagos növelő és csökkentő műveletei eredményként jobbértéket adnak vissza, aminek nincs címe és nem használható tovább olyan műveletekben, amelyek balértéket igényelnek (lásd a 6. hiba leírását):

```
int a = 12;
++a = 10; // Rendben
++++a; // Rendben
a++ = 10; // Hiba!
a++++; // Hiba!
```

Sajnos az utótagos ++ művelet általunk előbb megadott megvalósítása egy, a fordító által előállított névtelen ideiglenes objektumot ad vissza. A szabvány szerint ez nem balérték, de arra lehetőségünk van, hogy egy ilyen objektum egy tagfüggvényét meghívjuk. Ez pedig azt jelenti, hogy az objektum növelhető és egyben értékadás része is lehet. Arra azonban ügyeljünk, hogy a művelet végén az ideiglenes objektum megszűnik létezni!

```
Iter i;
Iter j;
++i = j; // Rendben
i++ = j; // Megengedett, de hibának kellene lennie!
```

Az `i` értékét nem befolyásolja az, hogy `j` tartalmát adjuk neki értékül, mivel az értékadás egy névtelen ideiglenes objektumra vonatkozik, ami az `i` növelés előtti értékét tartalmazza. Sokkal biztonságosabb megoldás, ha egy felhasználó által megadott, utótagos növelő vagy csökkentő művelet állandót ad vissza:

```
class Iter {
 public:
    Iter &operator ++(); // előtag
    const Iter operator ++(int); // utótag
    Iter &operator --(); // előtag
    const Iter operator --(int); // utótag
    // . . .
};
// . . .
i++ = j; // Hiba!
i++++; // Hiba!
```

Ez valószínűleg sikerrel gátolja majd meg a véletlen félreértéseket, viszont továbbra sem véd a szándékosság ellen. A művelet visszatérési értéke – bár van címe – nem módosítható:

```
const Iter *ip = &i++;
```

Ez a programozó ügyesen megszerezte – no nem az `i` objektum, hanem a kezelése során a fordító által előállított ideiglenes objektum címét. Persze ez utóbbi a művelet befejezése után azonnal eltűnik, tehát ténylegesen semmit nem lehet vele kezdeni. Összefoglalva tehát, amit programozónk művelt, azt nevezi a jog „előre megfontolt szándéknak". (A témával kapcsolatban olvassuk el a 11. hibát is.)

Az imént azt mondtam, hogy a felhasználó által megadott növelő és csökkentő műveletek általában tagfüggvények. A felsoroló típusokkal kapcsolatban ez természetesen nem lehet így, hiszen ezeknek nem lehetnek tagfüggvényeik:

```
enum Sin { pride, covetousness, lust, anger,
           gluttony, envy, sloth, future_use, num_sins };

inline Sin &operator ++( Sin &s )
   { return s = static_cast<Sin>(s+1); }

inline const Sin operator ++( Sin &s, int ) {
   Sin ret( s );
   s = ++s;
   return ret;
}
```

Figyeljük meg, hogy ezekből a függvényekből hiányzik az érvényességi tartomány ellenőrzése. Ha egy programozó hosszas megfontolás után úgy döntött, hogy osztály használata helyett felsoroló típussal old meg valamit, azt valószínűleg a hatékonyság érdekében tette. Ha a felsoroló típussal kapcsolatban tartományellenőrzésre van szükség, az önmagában megkérdőjelezi az előbbi döntés helyességét.
Az ilyen kód nemcsak hogy nem hatékony, hanem rengeteg felesleges ellenőrzést is tartalmaz:

```
for( Sin s = pride; s != num_sins; ++s ) // . . .
```

88. hiba: A sablonokra támaszkodó másoló műveletek félreértése

A sablon tagfüggvények leggyakoribb felhasználási területe a konstruktorok megvalósítása. Számos szabványos tárolótípusnak van például olyan sablon konstruktora, amely lehetővé teszi az adott objektumtípus kezdeti beállítását egy sorozat segítségével:

```
template <typename T>
class Cont {
 public:
   template <typename In>
       Cont( In b, In e );
   // . . .
};
```

A sablon konstruktor használatával az objektumot egy bármilyen forrásból származó bemenő sorozattal beállíthatjuk, ami a kérdéses tároló osztályt sokkal hasznosabbá teszi. A szabványos `auto_ptr` sablon emellett sablonként megadott tagfüggvényeket is használ:

```
template <class X>
class auto_ptr {
 public:
   auto_ptr( auto_ptr & ); // Másoló konstruktor
   template <class Y>
       auto_ptr( auto_ptr<Y> & );
   auto_ptr &operator =( auto_ptr & ); // Másoló értékadás
   template <class Y>
      auto_ptr &operator =( auto_ptr<Y> & );
   // . . .
};
```

Jegyezzük meg ugyanakkor, hogy az `auto_ptr` a sablonként megadott konstruktor és értékadó művelet mellett saját másoló műveleteit is pontosan meghatározza. Ez elengedhetetlen a helyes működéshez, mivel a sablonként megadott tagfüggvények nem használhatók másoló műveletek megvalósításához. Mint mindig, ha hiányzik a kifejezetten megadott másoló konstruktor és másoló értékadás, a fordító előállítja azokat. A tapasztalatok szerint ez a sablonokkal kapcsolatos kivételes viselkedés számos programozási hiba forrása:

➡ 88. hiba/money.h

```
enum Currency { CAD, DM, USD, Yen };

template <Currency currency>
class Money {
 public:
   Money( double amt );
   template <Currency otherCurrency>
        Money( const Money<otherCurrency> & );
   template <Currency otherCurrency>
        Money &operator =( const Money<otherCurrency> & );
   ~Money();
   double get_amount() const
         { return amt_; }
   // . . .
 private:
   Curve *myCurve_;
   double amt_;
};
// . . .
Money<Yen> acct1( 1000000.00 );
Money<DM> acct2( 123.45 );
Money<Yen> acct3( acct2 ); // Sablon konstruktor
Money<Yen> acct4( acct1 ); // Fordító által előállított másoló
➥ konstruktor
acct3 = acct2; // Értékadás sablon alapján
acct4 = acct1; // Fordító által előállított értékadás
```

Az itt látható hiba csupán új változata egy, a C++ osztályok tervezésével kapcsolatos nagyon régi problémának. Valahányszor egy osztály mutatót, vagy bármilyen más erőforrásleírót tartalmaz egy olyan objektumra, amely nem kezeli önmagát, nagyon fontos a körültekintés a megfelelő másoló műveletek megvalósításakor. Csak így kerülhető el ugyanis a memória fogyatkozása vagy a külső objektumok szándékolatlan megosztása. Vegyük például az előbbi sablon egy részletét:

➡ 88. hiba/money.h

```
template <Currency currency>
template <Currency otherCurrency>
Money<currency> &
Money<currency>::operator =( const Money<otherCurrency> &rhs ) {
   amt_ = myCurve_->
        convert( currency, otherCurrency, rhs.get_amount() );
}
```

Világos, hogy a `Money` megvalósítása során különös jelentőséggel bír az, hogy a `myCurve_` által hivatkozott `Curve` objektum ne legyen megosztva több ilyen típusú objektum között. Sajnos azonban éppen ez az, amit a fordító által előállított másoló műveletek tenni fognak, ráadásul teljes csendben:

```
template <Currency currency>
Money<currency> &
Money<currency>::operator =( const Money<currency> &that ) {
   myCurve_ = that.myCurve_; // Memóriaszivárgás, álnév
➥ használata, és a curve megváltozása!
   amt_ = myCurve_->
        convert( currency, otherCurrency, rhs.get_amount() );
}
```

A megoldás természetesen az, hogy a `Money` osztálynak meg kell adnia saját másoló műveleteit.

Sablonfüggvények segítségével nem valósíthatunk meg másoló műveleteket. Ennek megfelelően valamennyi osztály esetében fontoljuk meg a saját másoló műveletek megvalósítását (lásd a 49. hibát).

9

A hierarchia megtervezése

A hierarchiák (viszonyrendszerek) megtervezése nehéz feladat. Az osztályhierarchiáknak kellően rugalmasnak kell lenniük a továbbfejleszthetőség, viszont kellően konkrétnak a világos tervezés érdekében. A lehető legegyszerűbb szerkezettel kell rendelkezniük, de tükrözniük kell a megoldandó feladat valamennyi elvont tulajdonságát is. Eltérően más programelemektől, az osztályhierarchiákat a karbantartók még jóval azután is módosítani és fejleszteni fogják, hogy az eredeti tervező megvalósította és lefordította azokat. A hierarchia tervezőjének ezért egyértelművé kell tennie a tervezés módjával azt, hogy milyen mértékű testreszabást enged a felhasználóknak.

A hierarchia megtervezése egyben mindig hatékonyságnövelés, melynek legfőbb elemei a tervet befolyásoló külső tényezők és a megvalósítás hatékonysága. A helyzet meglehetősen hasonló a lineáris programozáshoz, ahol egy adott feladatnak több optimális megoldása is létezhet. A hierarchiák hatékony tervezéséhez inkább tapasztalatra és egyfajta tisztánlátásra van szükség, mintsem bizonyos betanult szabályok alkalmazására. Ennek megfelelően az ebben a fejezetben megfogalmazott tanácsok körvonalai is lágyabbak lesznek. Mindaz, amit elmondok, helyenként inkább valamiféle makacskodó személyes meggyőződésnek fog tetszeni, mint általános igazságnak.

Ugyanakkor a hierarchiák tervezésében is előfordulnak bizonyos általánosnak nevezhető hibák. Egyesek oka az, hogy a más nyelvekből magunkkal hozott programozási tapasztalatokra próbálunk építeni a C++ használata során is. Mások magyarázata egyszerűen a rutintalanság. Megint mások olyan új vagy annak ható módszerek, amelyek valahogy beleivódtak a köztudatba. Mindegyik általános típusra látunk majd példákat.

89. hiba: Osztály objektumok tömbjei

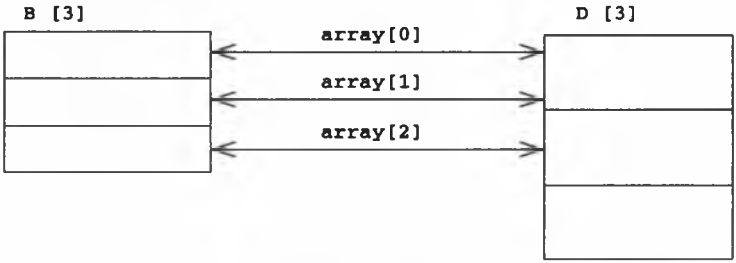
Legyünk bizalmatlanok az osztályelemeket tartalmazó tömbökkel szemben, különösen akkor, ha az elemek alaposztályok. Képzeljünk el például egy „végrehajtó függvényt", amely egy bizonyos műveletet a tömb összes elemével elvégez:

➡ 89. hiba/apply.cpp

```
void apply( B array[], int length, void (*f)( B & ) ) {
    for( int i = 0; i < length; ++i )
        f( array[i] );
}
// . . .
D *dp = new D[3];
apply( dp, 3, somefunc ); // Katasztrófa!
```

A gond itt az, hogy az `apply` függvény első formális paraméterének típusa „`B`-t címző mutató" és nem „`B` típusú elemek tömbje". Ez a fordító szempontjából azt jelenti, hogy egy `B *` mutatót egy `D *` mutatóval töltünk fel. Ennek semmi akadálya, ha a `B` nyilvános alaposztálya a `D`-nek, hiszen így a két típus rokonsági kapcsolatban áll. Ugyanakkor `D` típusú objektumok tömbje már nem feleltethető meg ilyen egyszerűen `B` objektumok tömbjének. Ha egy `D` objektumokat tartalmazó tömbön akarunk mutatóműveleteket végezni olyan eltolásokkal, amelyek a `B` típusra jellemzők, igen érdekes hibákat fogunk tapasztalni.

A helyzetet a 9.1. ábrán vázoltuk. Az `apply` függvény egy `B` típusú objektumokat tartalmazó tömb mutatóját várja (bal oldali ábra), a kapott mutató azonban valójában `D` objektumok tömbjére mutat (jobb oldali ábra). Emlékezzünk rá, hogy az indexek használata csupán egyfajta rövidített írásmódja a mutatóműveleteknek, vagyis az `array[i]` a fordító számára pontosan ugyanazt jelenti, mint a `*(array+i)` forma (lásd a 7. hiba leírását). Sajnos a fordító a mutatóműveleteket azzal a feltételezéssel fogja végrehajtani, hogy a mutató az alaposztálynak megfelelő objektumokat címez. Ha a származtatott osztály objektumai nagyobbak, vagy egyszerűen csak más a belső szerkezetük, az összes címzés hibás lesz.



9.1. ábra

Azok a mutatóműveletek, amelyek megfelelnek az alaposztály elemeit tartalmazó tömbök címzésére, általában nem használhatók a származtatott osztályok elemeit tartalmazó tömböknél.

Minden arra irányuló kísérlet, hogy a fenti tömb működését kijavítsuk, kudarcra van ítélve. Ha a `B` alaposztályt elvontként adjuk meg (ami általában jó ötlet), az automatikusan meggátolja bármilyen, `B` típusú elemeket tartalmazó tömb létrehozását, az `apply` függvény deklarációja azonban továbbra is helyes marad, hiszen az `B` típusú objektumok mutatóival és nem magukkal az objektumokkal foglalkozik. (Működését tekintve a függvény természetesen továbbra is rossz.) Ha a formális paramétert

tömbre való hivatkozásként adjuk meg (úgy, mint a `B (array&)[3]` kifejezésben), akkor a megoldás helyes ugyan, de nem praktikus, mivel rögzítenünk kell a tömb méretét (ebben az esetben ez 3), mutatót pedig (például egy már lefoglalt tömb címét) nem adhatunk át paraméterként.

Mindezek alapján az alaposztályok tömbjeit egyszerűen el kell kerülni, az egyéb osztályok elemeit tartalmazó tömbök használatát pedig igen alaposan meg kell fontolni.

Ha egy adott típushoz megírt függvény helyett valamilyen általános algoritmust használunk, az általában fejlődés:

```
for_each( dp, dp+3, somefunc );
```

A szabványos `for_each` algoritmus használata lehetővé teszi, hogy a fordító levezesse a függvénysablon paramétereinek típusát. A származtatott osztály típusának a megfelelő nyilvános alaposztály típusává alakítása nem gond, mivel ilyen átalakítás nem megy végbe. A fordító a `for_each` sablon alapján elkészíti azt a függvényt, amely pontosan igazodik a `D` származtatott típushoz. Sajnos ez a megoldás már nem teljesen azonos az eredeti célkitűzéssel, hiszen itt a futásidejű többalakúság helyett áttértünk a fordítási időben történő kezelésre.

Sokkal jobb megoldás lehet, ha objektumok tömbje helyett osztály objektumok mutatóinak tömbjét használjuk. Ez lehetővé teszi a tömb többalakú felhasználását, anélkül, hogy a mutatókkal kapcsolatos aritmetikai problémák felmerülnének:

```
void apply_prime( B *array[], int length, void (*f)( B * ) ) {
    for( int i = 0; i < length; ++i )
        f( array[i] );
}
```

Gyakran az a legjobb megoldás, ha teljesen elhagyjuk a tömbök használatát, és helyettük valamelyik szabványos tárolót, általában a `vector` típust használjuk. Az osztály objektumokat tartalmazó tömbökkel kapcsolatban egy ilyen „erős típusú" tároló használata gyakorlatilag kizárja a mutatóműveletekkel kapcsolatos gondokat, az alaposztályba tartozó objektumok mutatóit tartalmazó tömbök esetében pedig lehetővé teszi a többalakú használatot:

```
vector<B> vb; // D típusok nem megengedettek!
vector<B *> vbp; // többalakú
```

90. hiba: A tárolók nem megfelelő helyettesítése

Ha tárolókat kell használniuk, a C++ programozók első ötlete mindig a szabványos sablonkönyvtár (STL). Ugyanakkor ez a könyvtár sem elégít ki minden igényt, részben éppen azért, mert a hatékony megoldások gyakran csak az általánosság rovására valósíthatók meg. A szabványos sablonkönyvtár tárolótípusainak egyik leghasznosabb tulajdonsága az, hogy – mivel sablonként vannak megvalósítva – a szerkezetükkel és működésükkel kapcsolatos döntések fordítási időben születnek meg. Az eredmény olyan kicsi és hatékony kód, amely teljesítmény tekintetében pontosan alkalmazkodik az adott felhasználási módhoz.

Ugyanakkor számos információ fordítási időben egyszerűen nem áll rendelkezésre. Vegyünk például egy olyan egyszerűsített keretrendszernek megfelelő szerkezetet, amely a „nyílt zártság" elv alapján készült, vagyis anélkül fejleszthető tovább, hogy ehhez az egész kódot újra kellene fordítani. Ez az egyszerűsített keretrendszer két párhuzamos hierarchiát tartalmaz. Az egyiket a tárolók alkotják, a másikat a nekik megfelelő bejárók:

➡ 90. hiba/container.h

```
template <typename T>
class Container {
 public:
   virtual ~Container();
   virtual Iter<T> *genIter() const = 0; // Gyári módszer
   virtual void insert( const T & ) = 0;
   // . . .
};
template <typename T>
class Iter {
 public:
   virtual ~Iter();
   virtual void reset() = 0;
   virtual void next() = 0;
   virtual bool done() const = 0;
   virtual T &get() const = 0;
};
```

Ezekre az elvont alaposztályokra építve előállíthatunk valamilyen kódot, majd ezt később továbbfejleszthetjük új származtatott osztályok és bejáró osztályok megadásával:

➡ 90. hiba/main.cpp

```
template <typename T>
void print( Container<T> &c ) {
   auto_ptr< Iter<T> > i( c.genIter() );
```

```
    for( i->reset(); !i->done(); i->next() )
        cout << i->get() << endl;
}
```

Az ilyen párhuzamos hierarchiák használata tervezési szempontból általában problémás, mivel az egyikben történt módosítást azonnal követnie kell a másik változásainak is. Ennél nyilván sokkal kényelmesebb lenne, ha a módosításokat elegendő volna egyetlen ponton elvégezni. A Gyár tervezési módszer (Factory Method) használata a `Container` megvalósítása során enyhítheti ezt a `Container/Iter` hierarchiapáros körüli gondot. A Gyár tervezési módszer lehetővé teszi egy elvont alaposztály felhasználója számára, hogy az adott típusnak megfelelő származtatott objektumokat hozzon létre anélkül, hogy a típussal kapcsolatos részleteket ismerné. A `Container` elvont alaposztály esetében ez azt jelenti, hogy a `genIter` „gyári eljárásnak" azt mondja: „Hozz nekem létre egy, a saját típusodnak megfelelő `Iter` objektumot, de kímélj meg a részletektől". A Gyár módszer gyakran használhatóbbnak bizonyul, mint a sokak által tévesen javasolt, típusokon alapuló feltételes kód (lásd a 96. hibát). Másként fogalmazva, soha ne írjunk olyan kódot, ami gyakorlatilag a következő kacifántos döntési szerkezetnek felel meg: „`Container`, ha te valójában `Array` vagy, akkor adj nekem egy `ArrayIter` típust. Ha netán `Set` lennél, akkor `SetIter`-t adj nekem. Ha pedig...".

Egészen egyszerű egymással felcserélhető származtatott `Container` típusokat előállítani. Egy `Set<T>` behelyettesíthető egy `Container<T>` helyére, a `Set<T> *` típusról `Container<T> *` típusra való átalakítás pedig továbbra is működik. A tisztán virtuális `genIter` „gyári eljárás" jelenléte a `Container` alaposztályban azonnal emlékezteti a konkrét tárolótípus tervezőjét arra, hogy a szükséges változtatásokat az `Iter` hierarchiában is el kell végeznie:

```
template <typename T>
SetIter<T> *Set<T>::genIter() const
    { return new SetIter<T>( *this ); } // Jobb a SetIter-t
➥ megírni!
```

Szerencsétlen módon sokan gondolják úgy, hogy a tárolók elemeinek felcserélhetősége egyenértékű maguknak a tárolóknak a felcserélhetőségével. Azt már tudjuk, hogy a C++ alapértelmezett tárolóira, a tömbökre, ez a feltételezés nem érvényes. Egy származtatott osztály elemeit tároló tömb nem helyettesíthető egy, az alaposztályba tartozó objektumokat tárolóval (lásd a 89. hibát). Ugyanez igaz a felhasználó által megadott, felcserélhető elemeket tartalmazó tárolókra. Vegyük például a következő tárolók hierarchiáját:

➡ 90. hiba/bondlist.h

```
class Object
    { public: virtual ~Object(); };
```

```
class Instrument : public Object
   { public: virtual double pv() const = 0; };
class Bond : public Instrument
   { public: double pv() const; };
class ObjectList {
 public:
   void insert( Object * );
   Object *get();
   // . . .
};
class BondList : public ObjectList { // Rossz ötlet!!!
 public:
   void insert( Bond *b )
        { ObjectList::insert( b ); }
   Bond *get()
        { return static_cast<Bond *>(ObjectList::get()); }
   // . . .
};
```

➡ 90. hiba/bondlist.cpp

```
double bondPortfolioPV( BondList &bonds ) {
   double sumpv = 0.0;
   for( each bond in list ) {
          Bond *b = current bond;
          sumpv += b->pv();
   }
   return sumpv;
}
```

Azzal természetesen semmi baj nincs, hogy létrehoztuk `Bond` mutatók egy listáját, és ezzel párhuzamosan egy másik listát `Object` mutatókból. (Bár ami azt illeti, ügyesebb tervezés lett volna, ha `void *` típusból hozunk létre egy listát, és ebben az egyben helyezzük el az `Object` osztállyal kapcsolatos összes információt; lásd a 97. hibát.) A probléma a nyilvános öröklés használata a privát öröklés vagy a tagsági viszony helyett, ezzel ugyanis rokonsági kapcsolatot kényszerítünk ki egymással fel nem cserélhető típusok között. Azzal, hogy az eredetileg felcserélhető mutatókat egy tárolóban fogtuk össze, gyakorlatilag fel nem cserélhetővé tettük őket. Ugyanakkor, eltérően attól az esettől, amikor mutatót (vagy mutatók egy egész tömbjét) címző mutatónk van, a fordító nem tud bennünket a hibára figyelmeztetni (lásd a 33. hibát):

➡ 90. hiba/bondlist.cpp

```
class UnderpaidMinion : public Object {
 public:
   virtual double pay()
        { /* Elhelyezünk 1 millió dollárt minion számláján */ }
};
void sneaky( ObjectList &list )
```

```
    { list.insert( new UnderpaidMinion ); }

void victimize() {
    BondList &blist = getBondList();
    sneaky( blist );
    bondPortfolioPV( blist ); // Végrehajtva!
}
```

Ebben a kódban felcseréltünk két rokon objektumot. Ott, ahol a keretrendszer `Bond` típust várt volna, mi `UnderpaidMinon` típust használtunk. A legtöbb környezetben ennek az az eredménye, hogy a `Bond::pv` helyett az `UnderpaidMinon::pay` fut le, és ezzel egy gyakorlatilag felderíthetetlen futásidejű hiba áll elő. Ugyanúgy, ahogy felcserélhető származtatott objektumok tömbje nem cserélhető fel az alaposztály elemeit tartalmazó tömbbel, felhasználó által megadott, felcserélhető származtatott objektumokat tartalmazó tároló sem cserélhető fel az alaposztályba tartozó objektumokat tartalmazó tárolóval.

Összefoglalva tehát, a tárolók felcserélhetőségét a tárolók, és nem az általuk tárolt elemek szerkezete határozza meg.

91. hiba: A védett hozzáférés félreértése

Egy osztály tagjaihoz való hozzáférés érvényessége gyakran „nézőpont kérdése". Egy alaposztály nyilvános tagjához például nem férhetünk hozzá egy privát módon származtatott osztályon keresztül:

```
class Inst {
 public:
    int units() const
        { return units_; }
    // . . .
 private:
    int units_;
    // . . .
};

class Sbond : private Inst {
    // . . .
};
// . . .
void doUnits() {
```

```
    Sbond *bp = getNextBond();
    Inst *ip = (Inst *)bp;        // Szükség van a régi stílusú
➥ típusátalakításra...
    bp->units(); // hiba!
    ip->units(); // megengedett
}
```

A fenti helyzet a gyakorlatban ritkán fordul elő. Ha az alaposztály felületét elérhetővé akarjuk tenni a származtatott osztályon keresztül, nyilvánvalóan nyilvános öröklést használunk. A privát öröklést csaknem kizárólag egy adott megvalósítás öröklésére használjuk. Ha a programozás során azt látjuk, hogy a származtatott osztály mutatóját az alaposztály mutatójává kell alakítanunk, az csaknem biztosan tervezési hiányosságra utal.

Mellesleg figyeljük meg, hogy régi stílusú típusátalakítást használtunk a származtatott osztály típusáról a privát alaposztály típusára való áttéréskor. Normális esetben valószínűleg a biztonságosabb `static_cast` műveletet használtuk volna, ez azonban ebben a helyzetben nem lehetséges. A `static_cast` nem képes a származtatott osztály típusáról egy hozzáférhetetlen alaposztály típusára váltani. Sajnos a régi stílusú típusátalakítás gyakran elfedi a hibákat, ha később az `Sbond` és az `Inst` viszonya megváltozik. (Ezzel kapcsolatban olvassuk el a 40. és 41. hibát.) Személyes véleményem szerint az ilyen esetekben a legmegfelelőbb megoldás az, ha teljesen megszabadulunk a típusátalakítástól, úgy, hogy a teljes hierarchiát áttervezzük.

Ennek szellemében hozzunk tehát létre egy virtuális destruktort, a hozzáférési függvényt tegyük védetté, és hozzunk létre néhány származtatott osztályt:

```
class Inst {
 public:
   virtual ~Inst();
   // . . .
 protected:
   int units() const
        { return units_; }
 private:
   int units_;
};
class Bond : public Inst {
 public:
   double notional() const
        { return units() * faceval_; }
   // . . .
 private:
   double faceval_;
};
```

```
class Equity : public Inst {
 public:
   double notional() const
       { return units() * shareval_; }
   bool compare( Bond * ) const;
   // . . .
 private:
   double shareval_;
};
```

Az alaposztály azon függvénye, amely a pénzügyi eszközök számát adja vissza, immár védett, jelezvén, hogy azt a származtatott osztályoknak kell használniuk. A kötvények és részvények névértékét kiszámító függvények például ezt az információt használják a kalkuláció során.

Ugyanakkor jelen gazdasági helyzetben nem árt összehasonlítani egy kötvényt egy részvénnyel, így az `Equity` osztályban bevezettünk egy `compare` függvényt, amely mindössze a következőt teszi:

```
bool Equity::compare( Bond *bp ) const {
   int bunits = bp->units(); // Hiba!
   return units() < bunits;
}
```

Számos programozó megdöbben azt látván, hogy a védett `units` taghoz való első hozzáférés tiltott. A hibát az okozza, hogy bár a hozzáférés kiindulópontja az `Equity` származtatott osztály egy tagja, az egy `Bond` típusú objektumra vonatkozik. A nem statikus tagokkal kapcsolatban a védett hozzáférés nemcsak azt követeli meg, hogy a hozzáférést kezdeményező egy megfelelő származtatott osztály tagja vagy barátja legyen, hanem azt is, hogy a megcélzott objektum típusa ugyanaz legyen, mint azé, amelynek az adott függvény tagja, vagy amely „baráti alapon" hozzáférést enged számára. (Természetesen az is megfelelő, ha azt az osztályt, amelynek az illető függvény a tagja, nyilvánosan származtattuk a megfelelő alaposztályból.)

Esetünkben nem bízhatunk abban, hogy az `Equity` osztály egy tagja vagy barátja képes megfelelően értelmezni a `Bond` objektumok `units` tagját. Az `Inst` alaposztály ugyan valamennyi belőle származtatott osztály rendelkezésére bocsátotta saját `units` függvényét, de az már a származtatott osztályok dolga, hogy ezt megfelelően értelmezzék. A fenti `compare` függvény esetében a részvények és kötvények számának összehasonlítása feltehetőleg semmiféle értelmes eredményt nem szolgáltat. Ha nem támaszkodunk a származtatott osztályokban jelen levő olyan privát

információkra, mint a kötvények névértéke vagy a részvények árfolyama, nem tudjuk megoldani a feladatot. A hozzáférési jogosultságoknak ez az újabb ellenőrzési szintje a származtatott osztályok határozottabb leválasztását eredményezi.

Az sem segít, ha a `Bond` objektumot `Inst`-ként igyekszünk átadni:

```
bool Equity::compare( Inst *ip ) const {
   int bunits = ip->units(); // Hiba!
   return units() < bunits;
}
```

Az örökölt védett elemekhez való hozzáférés joga a származtatott osztályok (valamint a származtatott osztályokból nyilvánosan származtatott osztályok) azon objektumaira korlátozódik, amelyek a függvényhívást kezdeményezik. Egy olyan függvényt, mint amilyen a `compare` – ha egyáltalán szükség van rá –, valószínűleg célszerűbb a hierarchia egy magasabb pontján elhelyezni, ahol jelenléte nem okoz semmiféle csatolást a származtatott osztályok között:

```
bool Inst::unitCompare( const Inst *ip ) const
   { return units() < ip->units(); }
```

Ha ez a megoldás valamiért nem megvalósítható, és nem bánjuk, ha csatolás keletkezik az `Equity` és `Bond` osztályok között (amúgy illene bánnunk), egy kölcsönös barátfüggvény is segíthet:

```
class Bond : public Inst {
 public:
   friend bool compare( const Equity *, const Bond * );
   // . . .
};
class Equity : public Inst {
 public:
   friend bool compare( const Equity *, const Bond * );
   // . . .
};
bool compare( const Equity *eq, const Bond *bond )
   { return eq->units() < bond->units(); }
```

92. hiba: Nyilvános öröklés a kód újrahasznosítása végett

Az osztályhierarchiák két módon segítik a kód újrahasznosítását. Először is lehetővé teszik, hogy a különböző származtatott osztályok által közösen használt kódot egy megosztott alaposztályban helyezzük el. Másodszor lehetővé teszik, hogy az alaposztály felületét megosszuk minden nyilvánosan származtatott osztállyal. Mind a kódok, mind a felületek megosztása fontos tervezési szempont, de a felületek kezelése talán fontosabb.

Ha a nyilvános öröklést elsősorban a kód újrahasznosítása végett használjuk, az gyakran áttekinthetetlen, karbantarthatatlan, és végső soron kevésbé hatékony kódot eredményez. Ennek pedig az a magyarázata, hogy a nyilvános öröklés ilyen célú felhasználása annyira korlátozhatja az alaposztály felületét, hogy ezzel már megnehezíti a helyettesíthető származtatott osztályok tervezését. Ez aztán egyben azt is megnehezíti, hogy az alaposztály kezeléséhez írt általános kódokat a származtatott osztályok átemelhessék. Általában nagyobb mértékű kód-újrahasznosítás érhető el a nagyobb terjedelmű általános kódrészletek átemelésével, mint azzal, hogy az alaposztály egy szerény részét megosztjuk.

Az alaposztály kezeléséhez írt általános kód átemeléséből származó haszon olyan elsöprő, hogy számos osztályhierarchia gyökere egy úgynevezett „felületi osztály" („interface class"). A felületi osztály olyan alaposztály, amelynek nincsenek adattagjai, nincs megadott konstruktora, a destruktora virtuális, tagfüggvényei pedig tisztán virtuálisak. A felületi osztályokat szokás „protokoll osztályoknak" is hívni, ugyanis egy közelebbről meg nem határozott viszonyrendszer használati módját határozzák meg. (A „bekeveredő" osztály – mix-in – hasonló a felületi osztályhoz, de tartalmazhat néhány adattagot, valamint néhány ténylegesen megvalósított függvényt is.)

Ha egy hierarchia gyökereként felületi osztályt használunk, az később számos különböző tervezési módszer használatát megkönnyítheti. (A felületi osztályok a virtuális alaposztályok használatával kapcsolatos gondokat is enyhítik. Erről az 53. hiba kapcsán ejtettünk szót.)

A felületi osztályokkal kapcsolatban általában említett példa egy elvont visszahívási rendszer megvalósítása a Parancs tervezési módszer (Command pattern) segítségével. Képzeljük el például, hogy van egy grafikus felhasználói felületünk, amelynek `Button` nevű osztálya az `Action` függvényt hívja meg, ha a gombra kattintunk:

➡ 92. hiba/button.h

```
class Action {
 public:
```

```
    virtual ~Action();
    virtual void operator ()() = 0;
    virtual Action *clone() const = 0;
};
class Button {
 public:
    Button( const char *label );
    ~Button();
    void press() const;
    void setAction( const Action * );
 private:
    string label_;
    Action *action_;
};
```

A Parancs módszer alkalmazása esetén minden műveletet egy-egy objektumként adunk meg, így az objektumok használatának valamennyi előnyét élvezhetjük. Amint hamarosan látni fogjuk, ez a megközelítési mód azt is lehetővé teszi, hogy egyidejűleg más tervezési módszerekre is támaszkodjunk.

Figyeljük meg a túlterhelt `operator ()` használatát az `Action` megvalósításában. Használhattunk volna egy egyszerű – mondjuk `execute`-nak nevezett – tagfüggvényt is, a művelet használata azonban egyértelművé teszi, hogy az `Action` egy függvény elvont ábrázolása. Ez ugyanolyan jelzés, mint amikor a túlterhelt `operator ->` használatát látva egyértelműen tudjuk, hogy az adott osztály objektumai „okos mutatóként" használhatók (lásd a 24. és 83. hibákat). Maga az `Action` objektum a Prototípus tervezési módszerre (Prototype pattern) épül, hiszen tartalmaz egy `clone` nevű tagfüggvényt, ami a típus pontos ismerete nélkül képes előállítani egy `Action` objektum pontos mását (lásd a 76. hibát).

Az első konkrét `Action` típusunk a Null objektum tervezési módszer (Null Object pattern) alapján épül fel, hiszen olyan `Action` típust képvisel, ami megfelel ezen alaptípus minden ismérvének, de nem csinál semmit. A `NullAction` tehát egy `Action`-változat:

➡ 92. hiba/button.h

```
class NullAction : public Action {
 public:
    void operator ()()
        {}
    NullAction *clone() const
        { return new NullAction; }
};
```

Az `Action` keretrendszerrel a kezünkben immár egészen egyszerűen valósíthatunk meg egy biztonságos és rugalmas `Button` típust. A módszer gondoskodik róla, hogy a gomb mindig csinálni fog valamit, amikor megnyomják, még akkor is, ha a „valamit csinálni" egészen konkrétan semmittevést jelent (lásd a 96. hibát):

➡ 92. hiba/button.cpp

```
Button::Button( const char *label )
    : label_( label ), action_( new NullAction ) {}
void Button::press() const
    { (*action_)(); }
```

A Prototípus tervezési módszer gondoskodik róla, hogy a `Button` objektumnak legyen másolata a megfelelő `Action` objektumról még akkor is, ha nem ismeri a lemásolt `Action` objektum tényleges típusát:

➡ 92. hiba/button.cpp

```
void Button::setAction( const Action *action )
    { delete action_; action_ = action->clone(); }
```

Ezzel el is készült a `Button/Action` keretrendszer, amit immár kiegészíthetünk tényleges műveletekkel (olyanokkal, amelyek a `NullAction`-től eltérően tényleg csinálnak valamit). Fontos szempont, hogy mindezt a keretrendszer újrafordítása nélkül tehetjük meg.

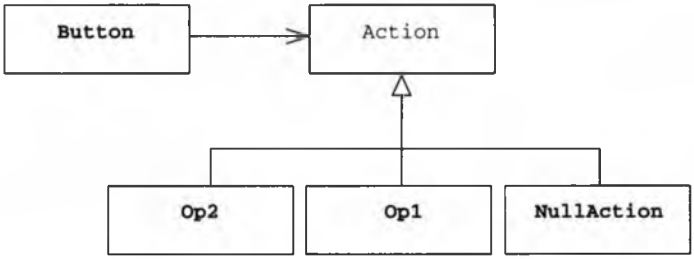
Az, hogy a hierarchia gyökerét egy felületi osztály alkotja, azt is lehetővé teszi, hogy a későbbiekben kiegészítsük a hierarchia képességeit. A Kompozit tervezési módszert (Composite pattern) alkalmazva például elérhetjük, hogy egy `Button`-hoz az `Action` objektumok egész logikai fája csatlakozzon:

➡ 92. hiba/moreactions.h

```
class Macro : public Action {
 public:
   void add( const Action *a )
        { a_.push_back( a->clone() ); }
   void operator ()() {
        for( I i(a_.begin()); i != a_.end(); ++i )
            (**i)();
   }
   Macro *clone() const {
        Macro *m = new Macro;
        for( CI i(a_.begin()); i != a_.end(); ++i )
            m->add((*i).operator ->());
        return m;
   }
 private:
   typedef list< Cptr<Action> > C;
```

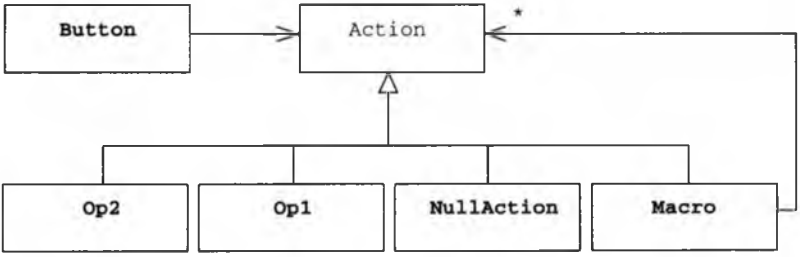
```
    typedef C::iterator I;
    typedef C::const_iterator CI;
    C a_;
};
```

Amint a 9.3. ábra mutatja, egy pehelysúlyú felületi osztály jelenléte a hierarchia gyökerénél lehetővé tette számunkra a Null objektum és a Kompozit módszer egyidejű használatát. Ugyanakkor, ha az `Action` alaposztály jelentős mennyiségű, ténylegesen megvalósított kódot tartalmazna, az arra késztetné a származtatott osztályok tervezőit, hogy örököltessék ezt a kódot, annak minden, beállítással és törléssel kapcsolatos mellékhatásával együtt. Ez pedig minden bizonnyal megakadályozná a Kompozit, a Null objektum és más gyakran alkalmazott tervezési módszerek használatát.



9.2. ábra

Példa a Parancs és a Null objektum módszerek használatára a Button visszahívó műveleteinek megvalósítása során.

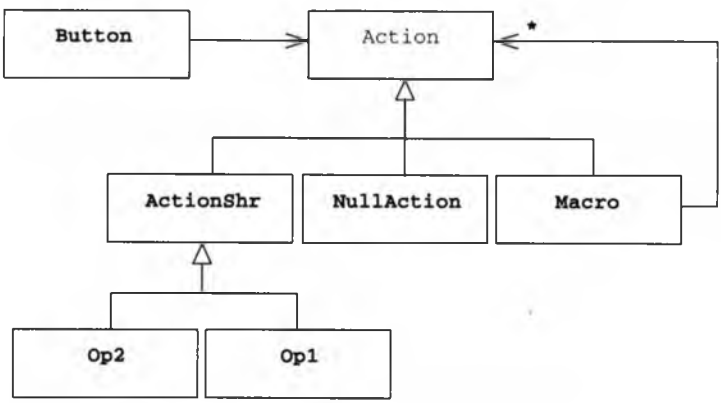


9.3. ábra

Az Action hierarchia kiegészítése a Kompozit tervezési módszer alkalmazásával.

Tervezési szempontból van némi feszültség a felületi osztály nyújtotta rugalmasság, és egy viszonylag „kiadósabb" alaposztály megosztása, illetve teljesítménytöbblete között. (Bár ez utóbbi gyakran elhanyagolható.) Előfordulhat például, hogy bizo-

nyos függvények a jelenlegi `Action` alaposztályból levezetett valamennyi származtatott osztályban változatlan formában előfordulnak. Ilyenkor ezt a közös függvényt el lehetne helyezni az alaposztályban is, ezzel azonban meggátolnánk önmagunkat abban, hogy a Kompozit módszer segítségével kiegészítsük a hierarchia képességeit, úgy, ahogy az előbb tettük. Az ilyen esetekben megengedhető, hogy mindkét megközelítés legjavát vegyük egy mesterséges alaposztály bevezetésével, amelynek egyetlen célja bizonyos függvénymegvalósítások megosztása (lásd a 9.4. ábrát).



9.4. ábra

Mesterséges alaposztály bevezetése a felület és a kódok egy részének öröklése végett.

Ha viszont túl bátran élünk ezzel a lehetőséggel, akkor túl sok olyan mesterséges osztály fog megjelenni a hierarchiában, amelynek a gyakorlatban nincs értelmes megfelelője. Ez pedig végső soron nehezen érthetővé és nehezen karbantarthatóvá teszi a rendszert.

Általánosságban tehát mégis jobb, ha a felületek öröklődésére összpontosítunk. A kódok megfelelő és hatékony újrahasznosítása ebből már egyértelműen következni fog.

93. hiba: Konkrét nyilvános alaposztályok

Tervezési szempontból a nyilvános alaposztályoknak elvontnak kell lenniük, mivel a konkrét problémával kapcsolatos elvont fogalmakat hivatottak leírni. Ugyanúgy, ahogy a fizikai térben sem szoktunk – és nem is szeretnénk – találkozni vándorló

elvont dolgokkal (képzeljük el például, hogyan nézhet ki egy általános alkalmazott, egy elvont gyümölcs, vagy egy ugyanilyen I/O eszköz), nyilván azt sem szeretnénk, ha az általunk írt kódban az elvont objektumok föl-alá „sétálnának".

A C++ programok esetében felmerülnek bizonyos gyakorlati szempontok is az osztályok megvalósításával kapcsolatban. Ezek közül elsődlegesen a másoló műveletek megvalósítása és a kivágás jelensége az, ami érdekel bennünket. (Erről részletesen volt szó a 30., 49. és 65. hibák kapcsán.) Általánosságban tehát a nyilvános alaposztályoknak elvontnak kell lenniük.

94. hiba: A leegyszerűsített hierarchiák használatának elmulasztása

Az alap- és az önálló osztályok tervezésével kapcsolatos szempontok meglehetősen különböznek, és a felhasználók is máshogy kezelik e két osztálytípust. Ezért mielőtt nekiállnánk megtervezni egy osztályt, mindenképpen érdemes eldönteni, hogy alap- vagy önálló osztályként fogjuk a jövőben használni.

„Előrelátó tervezés" az, amikor idejében felismerjük, hogy egy osztály a jövőben alaposztályként fog működni, és eleve egy kétosztályos hierarchiát hozunk létre a segítségével. Ezzel a lépéssel a hierarchia felhasználóit eleve arra kényszerítjük, hogy az alaposztálynak megfelelő elvont felületeket valósítsanak meg, ami aztán számunkra is megkönnyíti a rendszer későbbi kiegészítését. Ha az alaposztály helyett kezdetben egy konkrét osztályt használnánk, és csak később alakítanánk alaposztállyá, az valószínűleg azt jelentené, hogy mind nekünk, mind az osztály felhasználóinak át kell írniuk az elkészült keretrendszereket. Az előbb említett egészen egyszerű osztályhierarchiákat leegyszerűsített hierarchiáknak nevezzük. („Degenerált hiererchiák" – a „degenerált" szónak itt matematikai és nem morális jelentése van.)

Az olyan önálló osztályok, amelyeket később mégis alaposztályként használunk, a teljes kódra veszélyt jelentenek. Az önálló osztályok megvalósítása során hajlamosak vagyunk „érték szerint" értelmezni (value semantics), vagyis olyan szerkezetet alakítunk ki, ami lehetővé teszi az objektumok hatékony érték szerinti átadását. Ezzel természetesen arra késztetjük az osztály felhasználóit, hogy ezeket az objektumokat valóban érték szerint adják paraméterként függvényeknek, érték szerint adják vissza, és érték szerint rendeljék azokat más objektumokhoz.

Ha az ilyen osztály később alaposztállyá változik, minden másolási művelet magában hordozza a kivágás veszélyét (lásd a 30. hibát). Az önálló osztályok osztály objektumok tömbjeinek használatára is csábíthatnak, ami aztán később a címműveletek körül okozhat súlyos bajokat (lásd a 89. hibát). Számos rejtélyes hiba keletkezhet abból is, ha a felhasználó feltételezi egy bizonyos objektum méretének állandóságát, vagy azt, hogy az adott objektumtípus rendelkezik a képességek egy bizonyos halmazával. Mindez természetesen fel sem merülhet, ha az osztályt eleve elvont alaposztályként tervezzük meg.

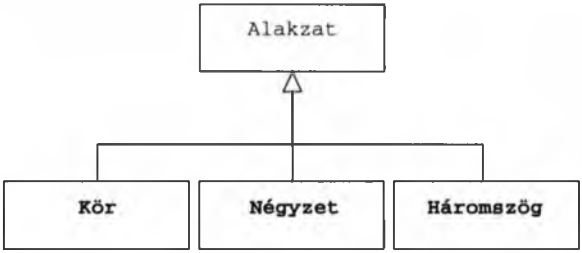
Persze mint már annyiszor, az elmondottaknak néha az ellenkezője is igaz. Számos olyan osztály képzelhető el, ami soha nem lesz alaposztály, és ezért soha nem is szabad a fent leírt módon megvalósítani. Ugyancsak szükségtelen, sőt kifejezetten káros a fenti tervezési módszert használni olyan kicsi típusokkal kapcsolatban, amelyeknél a fő cél a hatékonyság. Az elvont számtípusok, a karakterláncok, vagy a dátumtípusok jellemzően olyan osztályok, amelyek soha nem részei valamilyen nagyobb hierarchiának. Tervezőként végső soron a mi feladatunk eldönteni, mikor melyik módszer használható. Ehhez pedig, mint már említettük, szakmai tapasztalat, ítélőképesség és egyfajta tisztánlátás szükséges.

95. hiba: Az öröklés túlzott használata

A feltűnően széles vagy mély hierarchiák általában hibás tervezésre utalnak. Ilyesmi rendszerint akkor fordul elő, ha rosszul mérjük fel vagy határozzuk meg a viszonyrendszer egyes elemeinek feladatait. Vegyük például alakzatoknak a 9.5. ábrán bemutatott rendszerét.

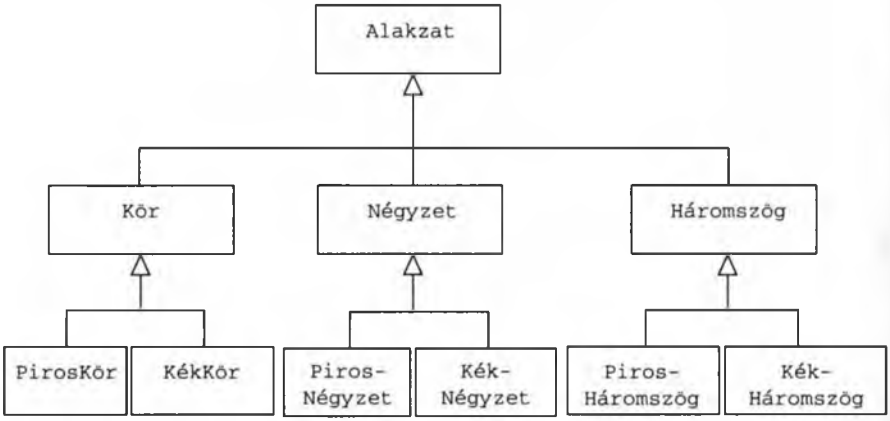
Ezek a mértani objektumok eredetileg valamennyien kék színnel jelennek meg a képernyőn, ha kirajzoljuk őket. Jön aztán egy ifjú programozó, akinek az öröklés lehetősége még izgatóan új, és azt a feladatot kapja, hogy alakítsa kétszínűvé ezt a hierarchiát. A jövőben kék és piros objektumokat is akarunk rajzolni. Semmi gond, gondolja emberünk.

Munkájának eredményét a 9.6. ábra mutatja. Ez az exponenciálisan növekvő rendszerek jellegzetes esete. Ha új színt akarunk bevezetni, akkor minden egyes alakzathoz új objektumot kell létrehoznunk. Ha ellenben egy új alakzatot akarunk felvenni a rendszerbe, akkor minden egyes színhez meg kell azt valósítanunk egy-egy új osztály formájában. Ez az öröklési rendszer nyilvánvalóan elhibázott, a megoldás pedig csaknem magától értetődő. Öröklés helyett a 9.7. ábrán bemutatott összetételi rendszert kell alkalmaznunk.



9.5. ábra

Alakzatok viszonyrendszere.

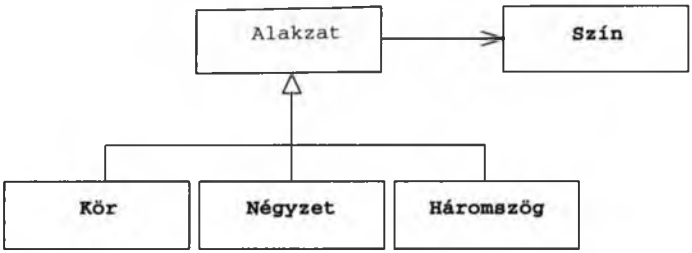


9.6. ábra

Egy helytelenül megtervezett, exponenciálisan növekvő rendszer.

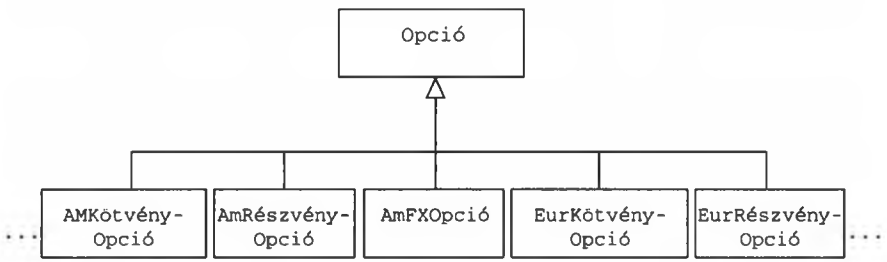
A helyesen megvalósított rendszerben a `Négyzet` egy `Alakzat` típusú objektum, amelynek van egy `Szín` nevű tulajdonsága. Az öröklés túlzott használatából eredő hiba persze nem mindig ennyire nyilvánvaló. Vegyünk például egy olyan rendszert, ami különböző befektetési formákat ábrázol (9.8. ábra).

Itt egyetlen, az opciót mint elvont fogalmat jelölő alaposztályt használunk, a konkrét osztályok pedig az opciótípus és a kérdéses befektetési forma kombinációi. Ismét egy exponenciálisan növekvő rendszerrel van dolgunk, hiszen egy újabb befektetési forma vagy egy új opciótípus megjelenése számos új osztály keletkezését vonja maga után. Ilyen esetekben a megfelelő tervezés több kisebb hierarchia létrehozása lenne (lásd a 9.9. ábrát) az egyetlen hatalmas rendszer helyett.



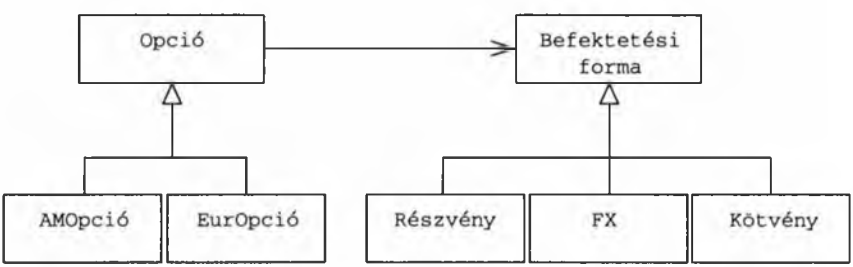
9.7. ábra

A helyes tervezés egyszerre alkalmazza az öröklést és az összetételt.



9.8. ábra

Egy hibásan megtervezett, tömbszerű rendszer.



9.9. ábra

A helyes tervezés: egyszerű hierarchiák összetétele.

Ebben a felállásban az `Opció` objektum rendelkezik egy `Befektetési forma` objektummal. Az említett probléma általában annak az eredménye, hogy nem fordítunk elég időt és energiát a modellezni kívánt valós rendszer megértésére, ugyan-

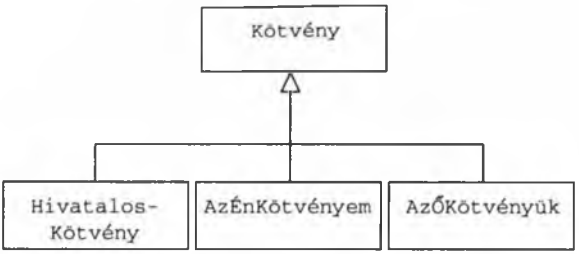
akkor az sem ritka, hogy bár kifogástalan munkát végeztünk az előzetes elemzés terén, a megtervezett osztályhierarchia mégis esetlenre sikerül. Folytatva előző gazdasági példánk elemzését, nézzük most a kötvény típus egy egyszerű megvalósítását:

```
class Kötvény {
 public:
   // . . .
   Pénz pv() const; // A jelenlegi érték kiszámítása
};
```

A kötvény jelenlegi értékét a `Kötvény` osztály `pv` tagfüggvénye számítja ki. Ugyanakkor az érték kiszámítására számos különböző algoritmus létezhet. A problémát például úgy hidalhatjuk át, hogy az összes számítási módot egyetlen függvényben gyűjtjük össze, majd a megfelelő számítási algoritmust egy kód segítségével választjuk ki:

```
class Kötvény {
 public:
   // . . .
   Pénz pv() const;
   enum Modell { Hivatalos, Enyém, Övék };
   void setModell( Modell );
 private:
   // . . .
   Modell model_;
};
Pénz Kötvény::pv() const {
   Pénz result;
   switch( model_ ) {
   case Hivatalos:
        // . . .
        return result;
   case Enyém:
        // . . .
        return result;
   case Övék:
        // . . .
        return result;
   }
}
```

Ez a megközelítés nagyon megnehezíti új árszámítási modellek felvételét a rendszerbe, mivel ez a teljes kód újrafordítását igényli. A szabványos objektumközpontú megközelítés az ilyen esetekre írja elő az öröklés és dinamikus kötés használatát, amint azt a 9.10. ábrán is láthatjuk.



9.10. ábra

Az öröklés helytelen használata: egyetlen tagfüggvény változékonyságát igyekszünk biztosítani az öröklés által.

Sajnos azonban ez a megközelítés rögzíti a `pv` függvény viselkedését a `Kötvény` objektum létrehozásakor, így az később már nem változtatható meg. A `Kötvény` megvalósításának egyéb szempontjai ugyanakkor idővel a `pv` függvénytől függetlenül változhatnak, ami megint csak a származtatott osztályok számának exponenciális növekedését fogja okozni.

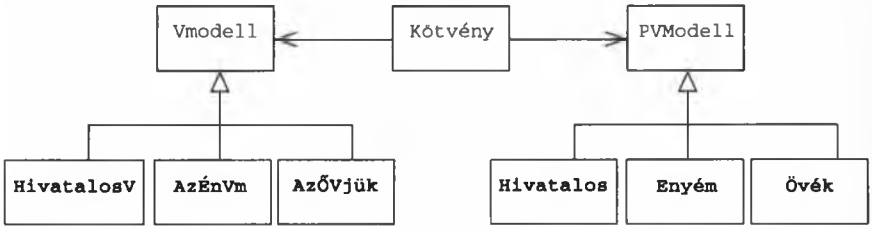
Egy `Kötvény` objektumnak lehet például egy olyan tagfüggvénye, ami az árral kapcsolatos kockázatot számítja ki. Ha ez az algoritmus független a jelenlegi érték kiszámítási módjától, akkor egy újabb árszámítási vagy kockázatbecslő algoritmus bevezetése származtatott osztályok tömegének létrehozását igényli, hiszen minden egyes ár–kockázat kombinációt meg kell valósítani a hierarchiában. Összefoglalva tehát, az öröklést általában egy egész objektum, és nem egy adott művelet változékonyságának biztosítására célszerű használni.

Akárcsak az előbbi, színes alakzatokkal kapcsolatos példánknál, itt is az összetétel (kompozíció) alkalmazása a helyes megoldás. Egész konkrétan a Stratégia tervezési módszert használhatjuk a tömbszerű (monolitikus) `Kötvény` hierarchia felbontására és a részek logikus összekapcsolására (lásd a 9.11. ábrát).

A Stratégia tervezési módszer lényege, hogy az algoritmus tényleges megvalósítását egy függvény törzséből kiemelve, önálló hierarchiában helyezzük el:

```
class PVModell { // Stratégia módszer
 public:
   virtual ~PVModell();
   virtual Pénz pv( const Kötvény * ) = 0;
};
class VModell { // Stratégia módszer
```

```
  public:
    virtual ~VModell();
    virtual double volatility( const Kötvény * ) = 0;
};
class Kötvény {
    // . . .
    Pénz pv() const
        { return pvmodel_->pv( this ); }
    double volatility() const
        { return vmodel_->volatility( this ); }
    void adoptPVModell( PVModell *m )
        { delete pvmodel_; pvmodel_ = m; }
    void adoptVModell( VModell *m )
        { delete vmodel_; vmodel_ = m; }
  private:
    // . . .
    PVModell *pvmodel_;
    VModell *vmodel_;
};
```



9.11. ábra

A Stratégia tervezési módszer használata két tagfüggvény egymástól független változékonyságának kifejezésére.

Ezzel a módszerrel egyrészt egyszerűsíthetjük a `Kötvény` hierarchiát, másrészt könnyen módosíthatjuk a `pv` vagy a `volatility` függvény viselkedését futásidőben.

96. hiba: Típus alapú vezérlőszerkezetek

Objektumközpontú kódban soha ne használjunk típuskódokon alapuló `switch` szerkezeteket:

```
void process( Employee *e ) {
   switch( e->type() ) { // Rettenetes kód!
```

```
    case SALARY: fireSalary( e ); break;
    case HOURLY: fireHourly( e ); break;
    case TEMP: fireTemp( e ); break;
    default: throw UnknownEmployeeType();
    }
}
```

A többalakú megközelítés sokkal megfelelőbb erre a célra:

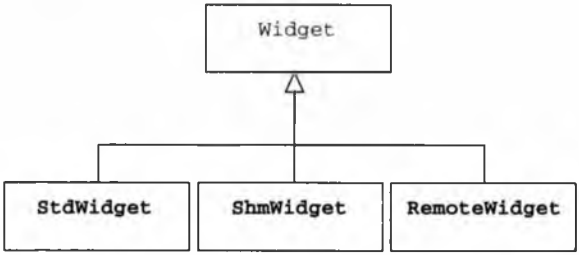
```
void process( Employee *e )
    { e->fire(); }
```

Ennek a módszernek óriási előnyei vannak az előbbivel szemben. Sokkal egyszerűbb és nem kell újrafordítani a teljes kódot, ha új típusok keletkeznek. A típusok keveredése miatti futásidejű hibák keletkezése teljesen kizárható. A keletkező kód kisebb és valószínűleg gyorsabb is. Összességében tehát a típusokon alapuló döntéseket célszerűbb dinamikus kötéssel, semmint feltételes vezérlőszerkezetekkel megoldani (lásd még a 69., 91. és 98. hibákat).

A feltételes kódok dinamikus kötéssel való helyettesítése annyira hatékony, hogy gyakran még az is értelmes, ha a feltételes kifejezéseket típus alapú kérdésekké fogalmazzuk át. Vegyünk például egy kódot (`process`), amely egy grafikus elemet (`Widget`) akar feldolgozni. A `Widget` osztály nyilvános felületének részeként rendelkezik egy `process` nevű függvénnyel, de attól függően, hogy a grafikus elem pontosan hol található, bizonyos más műveleteket is kell végezni a `process` meghívása előtt:

```
if(Widget a helyi memóriában van)
w->process();
else if(Widget a megosztott memóriában van)
csináljunk rettenetes dolgokat a feldolgozása közben
else if(Widget távoli)
még rettenetesebb dolgokat művelünk a feldolgozás közben
else
error();
```

Ez a feltételes kód nemcsak hogy hibázhat („Fel akarok dolgozni a `process` függvénnyel egy `Widget` objektumot, de fogalmam sincs, hol lehet!"), hanem ráadásul számos különböző helyen bukkanhat fel a kódban. A karbantartás és fejlesztés során, amikor a `Widget` osztály felbukkanási helyeinek száma változik, ezeket a független felbukkanási helyeket folyamatosan összhangban kell tartanunk. Sokkal jobb megoldás lenne, ha a `Widget` típus eleve tartalmazna információt az objektumok helyéről (lásd a 9.12. ábrát).



9.12. ábra

Egy módszer a feltételes szerkezetek elkerülésére. A Proxy módszer segítségével megoldhatjuk, hogy egy objektum típusa eleve tartalmazza annak hozzáférési protokollját.

Ez annyira gyakori, hogy még neve is van: Proxy tervezési módszernek hívják. A különböző helyeken található `Widget` típusú objektumokhoz való hozzáférés módja immár bele van kódolva minden `Widget` típusba, amelyek megkülönböztetéséhez elég egyetlen virtuális függvény meghívása. Ez a kód ráadásul nem ismétlődik sehol, és az is kizárt, hogy egy virtuális függvény eltévessze a `Widget` objektum helyét:

```
Widget *w = getNextWidget();
w->process();
```

A feltételes kódok elkerülésének másik módja annyira egyszerű, hogy gyakran észre sem vesszük a kínálkozó lehetőséget. Nem lehet hibás döntést hozni, ha egyáltalán nem hozunk döntést. Egyszerűbben fogalmazva, minél kevesebb feltételes kódot írunk, annál kevesebb lesz belőle hibás.

Ennek a tanácsnak az egyik megvalósulása a Null objektum módszer használata. Képzeljünk el egy függvényt, amely egy mutatót ad vissza egy olyan „eszközre" (device), ami „leírót" (handle) igényel:

```
class Device {
 public:
   virtual ~Device();
   virtual void handle() = 0;
};
// . . .
Device *getDevice();
```

A `Device` elvont alaposztály, amely számos különböző eszköztípust ábrázolhat. Az sem zárható ki, hogy a `getDevice` függvény valamiért nem tud visszaadni egy `Device` objektumot, így a biztonságos kód a következőképpen néz ki:

```
if( Device *curDevice = getDevice() )
    curDevice->handle();
```

Ez egészen egyszerű kód, de még mindig tartalmaz döntést. Ebben az esetben attól kell tartanunk, hogy egy figyelmetlen karbantartó elfelejti ellenőrizni a `getDevice` visszatérési értékét, mielőtt a `handle` függvényt használni kezdené.

A Null objektum módszer alapján egy olyan mesterséges `Device` alaposztályt hozhatunk létre, ami rendelkezik ugyan az ilyen típusú objektumok minden ismérvével (például kezelhető a `handle` függvénnyel), de amúgy nem csinál semmit. Ezt a semmit azonban pontosan a megfelelő módon csinálja:

```
class NullDevice : public Device {
 public:
   void handle() {}
};
// . . .
Device &getDevice();
```

A `getDevice` immár soha nem tévedhet, így eltávolíthatjuk a feltételes kódot, és ezzel ki is küszöböltük az említett lehetséges jövőbeni hibát:

```
getDevice().handle();
```

97. hiba: Kozmikus hierarchiák

Több mint egy évtizeddel ezelőtt a C++ programozók közössége úgy döntött, hogy az úgynevezett „kozmikus" hierarchiák használata a C++ nyelv esetében nem hatékony tervezési módszer. („Kozmikusnak" nevezzük azt a hierarchiát, amelynek minden osztálya egy közös gyökérből származik, amit általában egyszerűen `Object`-nek neveznek.) E döntésnek több oka is volt, amelyek részben a tervezéssel, részben a megvalósítással álltak kapcsolatban.

Tervezési szempontból a kozmikus hierarchiákkal az a baj, hogy túlságosan sok általános tároló osztályt tartalmazhatnak. E tárolók tényleges tartalma ráadásul gyak-

ran megjósolhatatlan, ami aztán számos igen furcsa futásidejű viselkedésforma megjelenéséhez vezethet. Bjarne Stroustrup klasszikus hasonlata szerint ez olyan, mintha egy „csatahajó" nevezetű objektumot egy „ceruzatartó" nevű tárolóban helyeznénk el. A kozmikus hierarchiákban ennek semmi akadálya, de a ceruzatartó tulajdonosa valószínűleg nagyon meg lesz lepve.

A gyakorlatlan programozók körében elterjedt az az igen veszélyes nézet, miszerint egy logikai szerkezetnek olyan rugalmasnak kell lennie, amennyire csak lehet. Ez hiba. Én inkább azt mondanám, hogy egy logikai szerkezetnek a lehető legszorosabb kapcsolatban kell állnia az általa modellezett problémával. Eközben persze célszerű fenntartani a lehető legnagyobb mértékű rugalmasságot, hiszen ez segíti a rendszer későbbi továbbfejlesztését. Amikor beáll a zűrzavar állapota, és a meglevő szerkezetet már nem tudjuk kiegészíteni az új elvárásoknak megfelelően, akkor egyszerűbb a teljes kódot újratervezni. A lehető legrugalmasabb szerkezet hajszolása olyan, mintha a leghatékonyabb kódot akarnánk előállítani egy alapjaiban ismeretlen probléma megoldására. Ha így gondolkodunk, egyetlen szerkezetet sem fogunk hosszú távon megfelelőnek találni, viszont jó esély van rá, hogy értékes teljesítményt áldozunk fel az általánosság oltárán (lásd a 72. hibát).

A logikai szerkezetek alapvető céljának ez a fajta félreértelmezése, párosulva az összetett problémák megfelelő elvont leírásának hiányával, oda vezethet, hogy az imént említett kozmikus hierarchiák közül az egyik legkártékonyabb típust kezdjük el használni:

```
class Object {
 public:
   Object( void *, const type_info & );
   virtual ~Object();
   const type_info &type();
   void *object();
   // . . .
};
```

Itt az látható, hogy a tervező semmiféle erőfeszítést nem tett a valós probléma elvont megfogalmazására. Helyette egy olyan burkoló objektumot állított elő, amivel egymással semmilyen logikai kapcsolatban nem álló típusokat is összeköthetünk. Az `Object` típus bármilyen más objektumot tartalmazhat, az `Object`-ek halmazából pedig további ilyen, egyre „kozmikusabb" tárolókat hozhatunk létre, amelyekbe bármi befér. (És gyakran beléjük is hánynak mindent.)

Mindezek megfejeléseként programozónk esetleg még arról is gondoskodhat, hogy az `Object` típust az általa tartalmazott objektum típusára cserélhessük:

```
template <class T>
T *dynamicCast( Object *o ) {
    if( o && o->type() == typeid(T) )
        return reinterpret_cast<T *>(o->object());
    return 0;
}
```

Ez a megközelítés elfogadhatónak tűnhet (bár elég suta), de képzeljük csak el, hogyan vesszük ki és használjuk fel egy olyan objektum tartalmát, amiben bármi lehet:

```
void process( list<Object *> &cup ) {
    typedef list<Object *>::iterator I;
    for( I i(cup.begin()); i != cup.end(); ++i ) {
        if( Pencil *p =
            dynamicCast<Pencil>(*i) )
            p->write();
        else if( Battleship *b =
            dynamicCast<Battleship>(*i) )
            b->anchorsAweigh();
        else
            throw InTheTowel();
    }
}
```

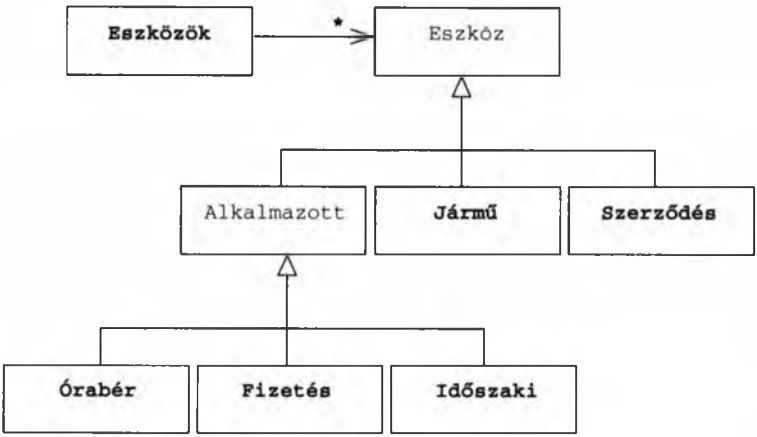
Kozmikus hierarchiánk valamennyi felhasználóját belekényszerítjük egy buta és gyerekes kitalálósdiba, aminek az a célja, hogy kiderítsük, mi van a kalapban. Persze semmi sem kényszerített bennünket arra, hogy az elején ezt az információt elveszítsük. Másként fogalmazva az, hogy egy ceruzatartóban nem lehet csatahajót tartani, nem a ceruzatartó tervezőjének a hibája. A hiba abban a kódban van, amelyik azt „képzeli", hogy ennek a megoldásnak értelme van. Valószínűtlen, hogy egy csatahajónak ceruzatartóban való tárolása bármiféle értelmes kapcsolatban lenne a program által leírni kívánt valósággal. Ez tehát nem az a tervezési módszer, amit követendő példaként emlegethetnénk. Ha munkánk bármelyik pontján egy kozmikus hierarchia megalkotása válik szükségessé, akkor valószínűleg a terv egy másik pontján van valami hiba.

Mivel a fenti, ceruzatartóval és csatahajóval megfogalmazott hasonlatunk a való világ modellje, érdemes elképzelnünk egy neki megfelelő valós helyzetet is. Képzeljük el tehát, hogy a ceruzatartó tervezőjeként egyszer csak érkezik hozzánk egy minőségi kifogás, miszerint az ügyfél csatahajója nem fért bele a ceruzatartóba. Kijavítanánk a hibát, vagy esetleg más jellegű szaksegítség igénybevételét javasolnánk?

A tervezés hanyagságának utóhatásai általában igen súlyosak, és sokba kerülhetnek. Bármilyen, az `Object` típusra épülő tároló használata a típuskeveredéssel kapcsolatos gondok tömkelegét vonhatja maga után. Senki nem tudja megmondani, mennyi kódot kell átírnunk, ha megváltozik az `Object` által tárolható típusok halmaza. És még csak az sem biztos, hogy az átírandó kód egyáltalán hozzáférhető. Végezetül, mivel tulajdonképpen semmilyen konkrét logikai szerkezetet nem valósítottunk meg, valamennyi felhasználó kénytelen lesz szembesülni azzal a problémával, hogy „külsőre" teljesen egyforma `Object` típusok alapján próbálja meg kideríteni, milyen típust tartalmaznak ténylegesen.

Az egésznek ráadásul az a szépsége is meglesz, hogy értelmes felépítés hiányában a hibák felderítését is ötletszerűen végzik a felhasználók. Előfordulhat például, hogy az egyik felhasználó elunja az olyan buta kérdések ismételgetését hogy „Te egy ceruza vagy? Nem? Talán csatahajó? Nem?..." és megkísérli magától az objektumtól megtudni, milyen tulajdonságokkal rendelkezik. (Erről hamarosan lesz szó a 99. hiba kapcsán.) A végeredmény így sem lesz sokkal jobb.

A hierarchia „kozmikus jellege" nem minden esetben annyira nyilvánvaló, mint a fenti példában. Vegyük például a 9.13. ábrán bemutatott hierarchiát.

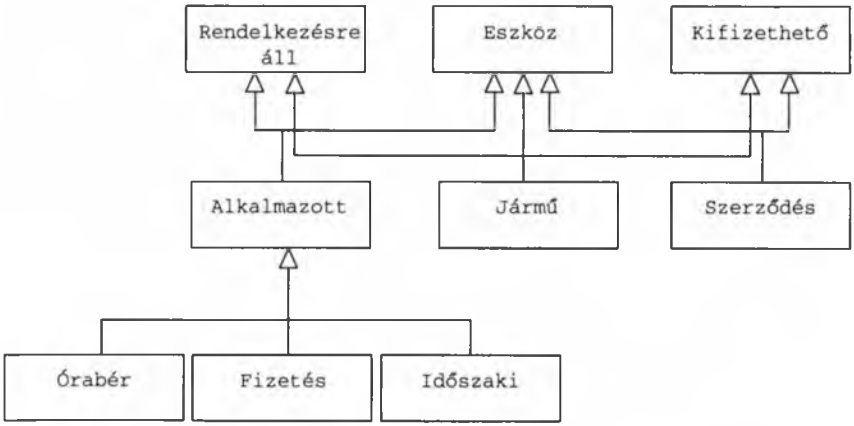


9.13. ábra

Egy meglehetősen esetleges hierarchia. Nem világos, hogy az Eszköz osztály túl általános vagy sem.

Ebben az esetben nem világos azonnal, hogy az `Eszköz` osztály túlságosan általános vagy sem, különösen egy ilyen általános „tervrajzot" szemlélve. Gyakori, hogy egy tervezési elképzelés alkalmatlanságára csak akkor derül fény, amikor már az alacsonyabb szintek tervezésénél tartunk, és lépten-nyomon beleütközünk olyan tervezési eljárásokba, amelyekről pontosan tudjuk, hogy veszélyesek vagy alkalmatlanok (lásd a 98. és 99. hibát). Ilyenkor minden valószínűség szerint egyfajta kozmikus hierarchiával van dolgunk, amit a teljes rendszer áttervezésével lehet (és kell) megszüntetni. Ha az alsóbb szinteken is mindent rendben találunk, akkor a terv működik.

Néha az is elég egy ilyen probléma megoldásához, ha más szempontból kezdjük szemlélni rendszerünket. A hibás hierarchiát esetleg anélkül is kijavíthatjuk, hogy a forráskódhoz számottevően hozzá kellene nyúlnunk. A kozmikus hierarchiák megjelenésének oka általában az, hogy túl általánosított alaposztályt tartalmaznak. Ha ezt átalakítjuk felületi osztállyá, a módosításról pedig értesítjük a hierarchia felhasználóit (lásd a 9.14. ábrát), könnyen elkerülhetjük a korábban említett romboló kódolási szokások kialakulását.



9.14. ábra

Egy hatékony szerkezetváltás. Az objektumok közti logikai kapcsolat kellőképpen meggyengül, ha az `Eszköz` osztályt protokollnak és nem alaposztálynak tekintjük.

Átalakítás után az eszközök hierarchiája immár három kisebb hierarchiára bomlott, amelyek a megfelelő felületen keresztül emelik át a másik kettőből a szükséges információt. Ez csak szerkezeti módosítás, de nagyon is lényeges. Az alkalmazottakat, a járműveket és a szerződéseket egyaránt eszközként kezelheti a megfelelő alrend-

szer, de ez működése közben nem akarja kideríteni a különböző `Eszköz` objektumok pontos típusát. Ugyanez igaz a másik két felületre is, így a futásidejű típuskeveredésből származó hibák valószínűsége elenyésző.

98. hiba: Személyes kérdések intézése egy objektumhoz

Ebben a részben az objektumközpontú rendszerek egy általános problémájával, a futásidőben megszerzett típusinformáció használatával foglalkozunk. A C++ nyelv szabványosította a futásidejű típuslekérdezés módját, így tulajdonképpen hallgatólagosan jóváhagyta ezen eljárás használatát. Bár a típuslekérdezésnek megvan a maga helye a C++ programozás gyakorlatában, nem szabad vele túl gyakran élni, és legfőképpen nem tanácsos egy egész felépítményt erre alapozni. Sajnos a C++ közösség hierarchiákkal és típusokkal kapcsolatos tapasztalatait gyakran félresöprik olyan rosszul megtervezett, túl általános, összetett, karbantarthatatlan és hibáktól hemzsegő rendszerek megvalósítása során, amelyek javarészt futásidejű típusinformációkra építenek.

Vegyük például az alant látható, és igen tiszteletreméltó módon megvalósított `Employee` alaposztályt. Néha egy-egy nagyobb alrendszer megvalósítása és tesztelése után szükségessé válhat néhány új szolgáltatás bevezetése. Az `Employee` alaposztály felületének van például egy egészen szembeszökő hiányossága:

```
class Employee {
 public:
   Employee( const Name &name, const Address &address );
   virtual ~Employee();
   void adoptRole( Role *newRole );
   const Role *getRole( int ) const;
   // . . .
};
```

Nos igen. Meg kell tudnunk határozni az alkalmazottak fizetését. (Aztán persze ki is kell fizetni őket, de az várhat a program következő változatáig.) Az igazgatóság egy napon azt kéri tőlünk, hogy tegyük lehetővé egy alkalmazott elbocsátását, de úgy, hogy csak egy mutatóval rendelkezünk az őt ábrázoló objektumra, és az `Employee` típust tartalmazó hierarchiát sem lehet újrafordítani. Az ugyebár világos, hogy az órabérben dolgozó alkalmazottakat máshogy kell kirúgni, mint a fix fizetéssel rendelkezőket:

```
void terminate( Employee * );
void terminate( SalaryEmployee * );
void terminate( HourlyEmployee * );
```

A fenti kívánalmak megvalósításának legegyszerűbb módja, ha egy kérdéssorozatot teszünk fel az alkalmazottat ábrázoló objektum pontos típusának meghatározása végett:

```
void terminate( Employee *e ) {
   if( HourlyEmployee *h = dynamic_cast<HourlyEmployee *>(e) )
       terminate( h );
   else if( SalaryEmployee *s = dynamic_cast<SalaryEmployee *>(e) )
       terminate( s );
   else
       throw UnknownEmployeeType( e );
}
```

E megközelítésnek nagy ára van: nem hatékony, és futásidejű hibába is torkollhat, ha a kód ismeretlen típussal találkozik. Mivel a C++ statikus típusokra épülő nyelv, és mivel a dinamikus kötés (a virtuális függvények) ellenőrzése is statikus, elvileg képesnek kellene lennünk az ilyen, típusokkal kapcsolatos hibák elkerülésére. Ezzel pedig el is jutottunk ahhoz a felismeréshez, hogy az itt látható `terminate` függvény inkább nevezhető programozási trükknek, mint egy továbbfejleszthető tervezési módszer alapjának.

A tervezés hiányossága valószínűleg még nyilvánvalóbbá válik, ha a fenti programot visszafordítjuk a való élet nyelvére, és megvizsgáljuk azt az élethelyzetet, amit a program modellezni kíván:

> Az alelnök beviharzik az irodájába és láthatóan igen dühös. A számára fenntartott parkolóhelyet ebben a hónapban immár harmadszor foglalja el az a trágyásszekér, amit az a kóbor programozó vezet, akit a múlt hónapban szerződtetett. „Ide hozzám Dewhursttel!" – üvölti a távbeszélőbe.
>
> Másodpercekkel később átható tekintettel méregeti a bánatos programozót, majd fejhangon azt üvölti: „Ha magát órabérben alkalmaztuk, akkor órabérben fizetett alkalmazottként van kirúgva. Ha fix fizetést kap, akkor ilyen minőségében van kirúgva. Ha pedig másként fizetjük magát, akkor hagyja el az irodámat, és mától legyen valaki másnak a problémája."

Én egyébként tanácsadó vagyok, és soha nem vesztettem még el olyan igazgatóval kötött szerződést, aki futásidejű információk alapján igyekezett megoldani a problémáit. Az előző példa helyes megoldása természetesen az, ha a megfelelő művele-

teket az `Employee` alaposztályban helyezzük el, és a szabványos, a típusok szempontjából biztonságos dinamikus kötést használjuk a futás közben a típusokkal kapcsolatban keletkező gondok megoldására.

```
class Employee {
 public:
    Employee( const Name &name, const Address &address );
    virtual ~Employee();
    void adoptRole( Role *newRole );
    const Role *getRole( int ) const;
    virtual bool isPayday() const = 0;
    virtual void pay() = 0;
    virtual void terminate() = 0;
    // . . .
};
```

A típusinformációk futásidejű lekérdezése persze néha hasznos is lehet. Amint az imént láttuk, jó szolgálatot tehet, ha egy eleve rosszul megtervezett rendszert kell karbantartanunk. Kényszerűségből használhatjuk akkor is, ha más módon egyszerűen nem megoldható feladattal kerülünk szembe. Ilyen lehet például az a követelmény, hogy módosítsunk egy kódot annak újrafordítása nélkül, de úgy, hogy közben pontosan tudjuk: az eredeti tervező erre nem készítette fel a rendszerét. A típuslekérdezésnek ritkán ugyan, de hasznát vehetjük hibakeresés során, vagy olyan különleges célú programok fejlesztésekor, mint amilyenek a nyomkövetők vagy a böngészők. Végezetül, ha a modellezni kívánt problémának eleve van valamilyen belső szabálytalansága, ez megmutatkozhat abban is, hogy a kódban nem tudjuk elkerülni a futásidejű típuslekérdezést.

Amióta a C++-ban szabványosították a futásidejű típuskezelést, számos programozó él is vele, ahelyett, hogy egyszerűbb, hatékonyabb, jobban karbantartható kódot írna. A futásidejű típuslekérdezés általában a logikai szerkezet hiányosságait fedi el, amelyek pedig rendszerint a túlbonyolított programozási megoldásokból, a feladat felületes elemzéséből, illetve abból az általános félreértésből adódnak, miszerint egy rendszernek mindenáron muszáj a lehető legrugalmasabbnak lennie.

A gyakorlatban a legritkább esetben van szükség arra, hogy egy objektumnak „személyes kérdést" tegyünk fel a típusával kapcsolatban.

99. hiba: Képességekkel kapcsolatos kérdések

A futásidejű típuslekérdezés felbukkanása egy kódban – amint azt az előző rész `terminate` függvényével kapcsolatban láthattuk – gyakran nem is annyira tervezési hiba, inkább bizonyos elvetemült programozási trükkök és a hibás munkaszervezés eredménye. Ugyanakkor előfordulnak olyan hierarchiák is, amelyek alapját néhány „kifinomult" dinamikus típusátalakítás és a többszörös öröklés összjátéka képezi.

> Az alkalmazott első munkanapján jelentkezik a személyzeti osztályon. Azt mondják neki, hogy álljon be az „eszközök" sorába. Egy hosszú sort mutatnak neki, amelyben állnak ugyan más alkalmazottak is, de van ott egy csomó irodai gép, jármű, meg bútor, sőt még szerződések is.
>
> Végül sorra kerül, ahol is számos igen furcsa kérdésre kell válaszolnia: „Fogyaszt ön benzint?" „Tud ön programozni?" „Tudunk önnel iratokat másolni?" Minden kérdésre nemmel felel, mire végül hazaküldik. Emberünk persze nagyon csodálkozik, hiszen azt senki sem kérdezte tőle, hogy tud-e padlót sikálni. Pedig takarítónak vették fel.

Elég eszement történet, nem? (Persze ha az Olvasó dolgozott már nagyvállalatnál, esetleg ismerős lehet.) A megérzés természetesen helyes: ez itt a képességlekérdezés hibás használatának alapesete.

Hagyjuk most egy időre a személyzeti osztályt a gondjaival együtt, és vizsgáljuk meg befektetési formák egy lehetséges rendszerét. Tegyük fel, hogy értékpapírokkal üzletelünk. Rendelkezésünkre áll egy árképző, valamint egy likviditási alrendszer, amelyeket át szeretnénk emelni a most megtervezendő rendszerbe. Mindkét alrendszer igényei világosan kiderülnek egy felületi osztályból, amiből a felhasználónak saját osztályait származtatnia kell:

```
class Megtakarítható { // Likviditási felület
 public:
   virtual ~Megtakarítható();
   virtual void save() = 0;
   // . . .
};
class Árazható { // Árképző felület
 public:
   virtual ~Árazható();
   virtual void price() = 0;
   // . . .
};
```

Az `Ügylet` hierarchia egyes konkrét osztályai eleget tesznek a két említett alrendszer által támasztott megkötéseknek, így átemelhetik az alrendszerek kódját. Ez a többszörös öröklés használatának szabványos, hatékony és helyes módja:

```
class Ügylet {
 public:
   virtual void validate() = 0;
   // . . .
};
class Kötvény
 : public Ügylet, public Árazható
   {/* . . . */};
class Csere
 : public Ügylet, public Árazható, public Megtakarítható
   {/* . . . */};
```

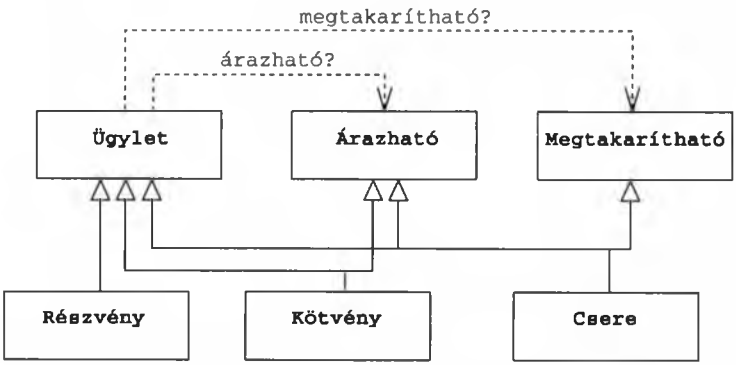
A rendszert most fel kell ruháznunk azzal a képességgel, hogy kezelni tudjon egy ügyletet akkor is, ha csak egy mutatója van a kérdéses objektumhoz. A naiv megközelítés az lenne, hogy egyenes kérdéseket teszünk fel az objektumnak annak típusával kapcsolatban. Ez a módszer természetesen semmivel sem jobb, mint az előző példánkban szereplő `terminate` függvény, amely ezzel a módszerrel igyekezett `Employee` objektumokat kezelni (lásd a 98. hibát):

```
void processDeal( Ügylet *d ) {
   d->validate();
   if( Kötvény *b = dynamic_cast<Kötvény *>(d) )
       b->price();
   else if( Csere *s = dynamic_cast<Csere *>(d) ) {
       s->price();
       s->save();
   }
   else
       throw IsmeretlenÜgylet( d );
}
```

Egy másik aggasztóan gyakori módszer az, hogy nem azt kérdezik meg az objektumtól, hogy kicsoda, hanem azt, hogy mire képes. Ezt hívják képességlekérdezésnek:

```
void processDeal( Ügylet *d ) {
   d->validate();
   if( Árazható *p = dynamic_cast<Árazható *>(d) )
       p->price();
   if( Megtakarítható *s = dynamic_cast<Megtakarítható *>(d) )
       s->save();
}
```

Valamennyi alaposztály képességek egy halmazát ábrázolja. Ha a `dynamic_cast` segítségével egy, a hierarchián átívelő típusátalakítást hajtunk végre, az olyan, mintha azt kérdeznénk az objektumtól, hogy képes-e betölteni egy bizonyos szerepet vagy szerepek egy csoportját (lásd a 9.15. ábrát). A `processDeal` második változata gyakorlatilag a következőt mondja: „`Ügylet` objektum, azonosítsd magad! Ha lehet hozzád árat rendelni, akkor árazd be magad. Ha megtakarítható vagy, takarítsd meg magad."



9.15. ábra

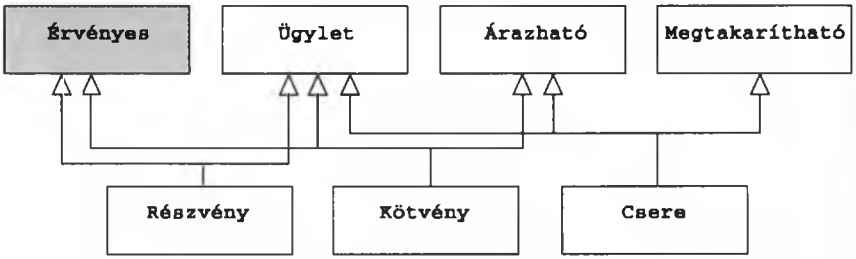
Képességek lekérdezése hierarchián átívelő típusátalakítással.

Ez a megközelítés valamivel kifinomultabb, mint a `processDeal` előző megvalósítása. A hibák felbukkanásának lehetősége is kisebb, hiszen az új függvény képes új típusokat kezelni anélkül, hogy kivétel keletkezne. Ugyanakkor ez a megoldás még mindig nem elég hatékony, és karbantartása is nehézkes. Gondoljuk át, mi történne, ha az `Ügylet` hierarchiában a 9.16. ábrán bemutatott módon egy új felületi osztály jelenne meg.

Az új képesség megjelenését a függvény nem veszi észre. A kód gyakorlatilag soha nem fogja megkérdőjelezni egy ügylet érvényességét (ami viszont a gyakorlatban nem biztos, hogy helyes). Végső soron tehát – akárcsak az előző részben bemutatott `terminate` függvény – a jelenlegi, képességek lekérdezésén alapuló megoldás is csupán ideiglenes kezelése egy hirtelen felmerült problémának, nem pedig egy szilárd felépítmény alapja.

A típusok és képességek lekérdezésének objektumközpontú környezetben való használata tehát kétségkívül problémás. A bajok közös gyökere az, hogy egy objektum típusát vagy képességeit egy külső kód határozza meg. Ez a megközelítés

pedig az objektumközpontú tervezés alapeszméjével, az elvont adatábrázolással áll szöges ellentétben. Ha lekérdezéseket használunk, akkor egy elvont adat típusát már nem a neki megfelelő objektum tárolja: az információ szétoszlik az egész forráskódban.



9.16. ábra

A képességlekérdezésen alapuló rendszer törékenységének bemutatása. Mi történik, ha elfelejtjük feltenni a megfelelő kérdést?

Ha új képességet akarunk bevezetni az `Ügylet` rendszerébe, a leghatékonyabb mód – akárcsak korábban az `Employee` hierarchiánál – egyben a legegyszerűbb is:

```
class Ügylet {
 public:
   virtual void validate() = 0;
   virtual void process() = 0;
   // . . .
};
class Kötvény : public Ügylet, public Árazható {
 public:
   void validate();
   void price();
   void process() {
        validate();
        price();
   }
};
class Csere : public Ügylet, public Árazható, public
➥ Megtakarítható {
 public:
   void validate();
   void price();
   void save();
   void process() {
```

```
        validate();
        price();
        save();
    }
};
// stb.
```

Olyan eljárások is használhatók a képességek lekérdezésének javítására, amelyek nem igénylik a viszonyrendszer módosítását, feltéve persze, hogy az eredeti szerkezet lehetővé teszi alkalmazásukat. A Látogató tervezési módszer (Visitor pattern) lehetővé teszi új képességek felvételét a rendszerbe, de nehezíti a karbantartást. Az Acyclic Visitor változat kevésbé növeli a hibák valószínűségét, de olyan (egyetlen) képességlekérdezés meglétét igényli, amely hibát okozhat. Mindazonáltal bármelyik módszer jobb, mint a képességlekérdezés rendszeres használata.

Az ilyen lekérdezések iránti igény megjelenése általában tervezési hiányosságra utal, helyettük szinte mindig egyszerűbb, hatékonyabb és biztonságosabb egy megfelelő virtuálisfüggvény-hívást használni.

> Az alkalmazott első munkanapján jelentkezik a személyzeti osztályon. Azt mondják neki, hogy álljon egy hosszú sor végére, amelyben más alkalmazottak is állnak. Végül sorra kerül, és az Illetékes Elvtárs így szól hozzá: „Menj és dolgozz!" És mivel emberünk takarítóként áll alkalmazásban, fog egy felmosórongyot, és a nap hátralevő részét a padló tisztításával tölti.

Irodalomjegyzék

Andrei Alexandrescu: *Modern C++ Design*
(Addison-Wesley, 2001.)

Association for Computing Machinery: *ACM Code of Ethics and Professional Conduct*
(www.acm.org/constitution/code.html)

Association for Computing Machinery: *Software Engineering Code of Ethics and Professional Practice*
(www.acm.org/serving/se/code.htm)

Marshall P. Cline, Greg A. Lomow, Mike Girou: *C++ FAQs*
(Second Edition, Addison-Wesley, 1999.)

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*
(Addison-Wesley, 1995.)

Nicolai Josuttis: *The C++ Standard Library*
(Addison-Wesley, 1999.)

Robert Martin: *Agile Software Development*
(2nd ed., Prentice Hall, 2003.)

Scott Meyers: *Effective C++*
(2nd ed. Addison-Wesley, 1998.)

Scott Meyers: *Effective STL*
(Addison-Wesley, 2001.)

Scott Meyers: *More Effective C++*
(Addison-Wesley, 1996.)

Stephen C. Dewhurst, Kathy T. Stark: *Programming in C++*
(2nd ed., Prentice-Hall, 1995.)

William Strunk, E. B. White: *The Elements of Style*
(3d ed., Macmillan, 1979.)

Herb Sutter: *More Exceptional C++*
(Addison-Wesley, 2002.)

E. B. White: *Writings from The New Yorker*
(HarperCollins, 1990.)

Tárgymutató

A, Á

B

C, Cs

D

E, É

F

G, Gy

H

I

J

K

L

M

N, Ny

O, Ö

P

Q

R

S, Sz

T, Ty

U, Ü

V

W

Z