C++
Gyakorlat jegyzet
1. óra

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlatán. (2017. január 6.)

1. Bevezető

A C++ többek között a hatákonyságáról is híres. Andrei Alexandrescu azt nyilatkozta, hogy amikor a Facebooknál a backend kódján 1%ot sikerült optimalizálni, több mint 10 évnyi fizetését spórolta meg a cégnek havonta csak az áramköltségen. Nem használ garbage collectort: nincs nem várt szünet a program végrehajtásában a menedzselt nyelvekkel szemben.
A C++-szal kapcsolatban az egyik gyakori tévhoz, hogy egy alacsony szintű nyelvről van szó. Bár a nyelv lehetőséget biztosít arra, hogy alacsony szinten hozzáférjünk a hardvereinkhez, számos gazdag absztrakciós lehetőséget tartalmaz. Ezeknek a használatával magas szintű kód írására is kíválóan alkalmas. A legtöbb nyelvhez képest abban emelkedik ki, hogy a C++ nyelvben ezeknek az absztrakcióknak ritkán van futási idejű költsége. Legtöbbször a fordítóprogram teljesen el tudja tüntetni ezeket az absztrakciókat a programból a fordítás során.
A C++ filozófiájának fontos eleme, hogy ha nem használunk egy adott nyelvi eszközt, akkor annak ne legyen hatása a program teljesítményére.
Fontos, hogy a C++ alapvetően nem egy objektum orientált nyelv. Bár számos nyelvi eszköz támogatja az objektum orientált stílusú programozást, de a nyelv kíválóan alkalmas más paradigmák használatára is. A funkcionális programozástól a generatív programozáson át a deklaratív stílusig sok programozási stílusra alkalmas. A nyelv nem próbál ráerőltetni egy megközelítést a programozóra, ellenben próbál minél gazdagabb eszköztárat biztosítani, hogy a megfelelő problémát a lehető legmegfelelőbb módon lehessen megoldani. Még akkor is, ha ez a különböző paradigmák keverését vonja maga után. Ezért ezt a nyelvet gyakran multiparadigmás programozási nyelvnek szokták hívni.

Cél: a tárgy során kialakítani a nyelvvel kapcsolatban egy intuíciót, ami segítségével elkerülhetőek alapvető hibák is. Az előzménytárgyakban az egyszerűség kedvéért gyakran féligazságok hangzottak el, ezeket is helyre kell rakni.

1.1. Mi az a C++?

Alapvetően a nyelv két összetevőből áll. Az aktuális szabványból és annak implementációiból (fordítók + szabványkönyvtárak). A szabvány, ami meghatározza a nyelv nyelvtanját, valamint a szemantikát: mit jelentenek a leforduló programok (nem definiál minden részletet). Emellett a szabvány definiálja a szabványkönyvtárat is, amit minden szabványos C++ fordító mellé szállítani kell. Az első C++ szabvány a C++98 volt. További szabványai: C++03, C++11, C++14, C++17.

A szabvány alapján számos fordító (implementáció) létezik a C++ kódok fordítására: MSVC (Visual Studio), GCC, Clang. Létezik számos fejlesztői környezet is, mint például: CLion, QtCreator, CodeBlocks, VIM. De ezek nem fordítók, legtöbbször a fent említett fordítók közül használnak egyet.

2. Különböző viselkedések kategorizálása

Egy reménytelen megközelítés lenne a szabványban minden szintaktikusan (nyelvtanilag) helyes kódhoz pontos szemantikát (működést) társítani. Ennek mind elméleti és gyakorlati oka van. Ezért a C++ szabvány néhány esetben nem vagy csak részben definiálja egy adott program működését. A következőkben erre fogunk példákat látni.

2.1. Nem definiált viselkedések

```
int main()
{
    int i = 0;
    std::cout << i++ << i++ << std::endl;
}
```

Lehetséges kimenet: `01` (GCC 6.1 fordítóval 64 bites x86 Linux platformon)
Lehetséges kimenet: `10` (Clang 3.9 fordítóval 64 bites x86 Linux platformon)

Fordítás és futtatás után különböző eredményeket kaphatunk, mert itt az, hogy mikor értékelődik ki a két `++i` a kifejezésen belül, az **nem specifikált.** Ha a szabvány nem terjed ki arra, hogy milyen viselkedésű kódot generáljon a fordító, akkor a fordító bármit választhat.

Gyakran eldönthetetlen előre, hogy mikor mi lesz a leghatékonyabb megoldás, ez az egyik ok, hogy nem definiál mindent a szabvány.

Ez lehetőséget ad a fordítónak arra, hogy **optimalizáljon.**

A C++ban van un. szekvenciapontok, és a szabvány csak azt mondja ki, hogy a szekvenciapont előtti kód hamarabb kerüljön végrehajtásra mint az utána levő. Mivel itt az i értékadása után és csak az `std::endl` után van szekvenciapont, így az, hogy milyen sorrendben történjen a kettő közötti kifejezés részkifejezéseinek a kiértékelése, az a fordítóra van bízva.

A C++ban nem meghatározott, hogy két szekvenciapont között mi az utasítások végrehajtásának a sorrendje.

Az, hogy két részkifejezés szekvenciaponttal történő elválasztás nélkül ugyanazt a memóriaterületet módosítja, **nem definiált** viselkedést eredményez. Nem definiált viselkedés esetén a fordító vagy a futó program bármit csinálhat. A szabvány semmiféle megkötést nem tesz.
2.1.1. Megjegyzés. Az a program, amely nem definiált viselkedéseket tartalmaz, hibás.

2.2. Nem specifikált viselkedések

Amennyiben a szabvány definiál néhány lehetséges opciót, de a fordítóra bízza, hogy az melyiket választja, akkor **nem specifikált** viselkedésről beszélünk.
A nem specifikált viselkedés csak akkor probléma, ha a program végeredményét (megfigyelhető működését) befolyásolhatja a fordító választása. Például a fenti kódot módosíthatjuk a következő képpen:

```
int main()
{
    int i = 0;
    int j = 0;
    std::cout << ++i << ++j << std::endl; // 11
}
```

Bár azt továbbra se tudjuk, hogy `++i` vagy `++j` értékelődik ki hamarabb, (*nem specifikált*), azt biztosan tudjuk, hogy 11-et fog kiírni (a program végeredménye *jól definiált*).

2.3. Implementáció által definiált viselkedés

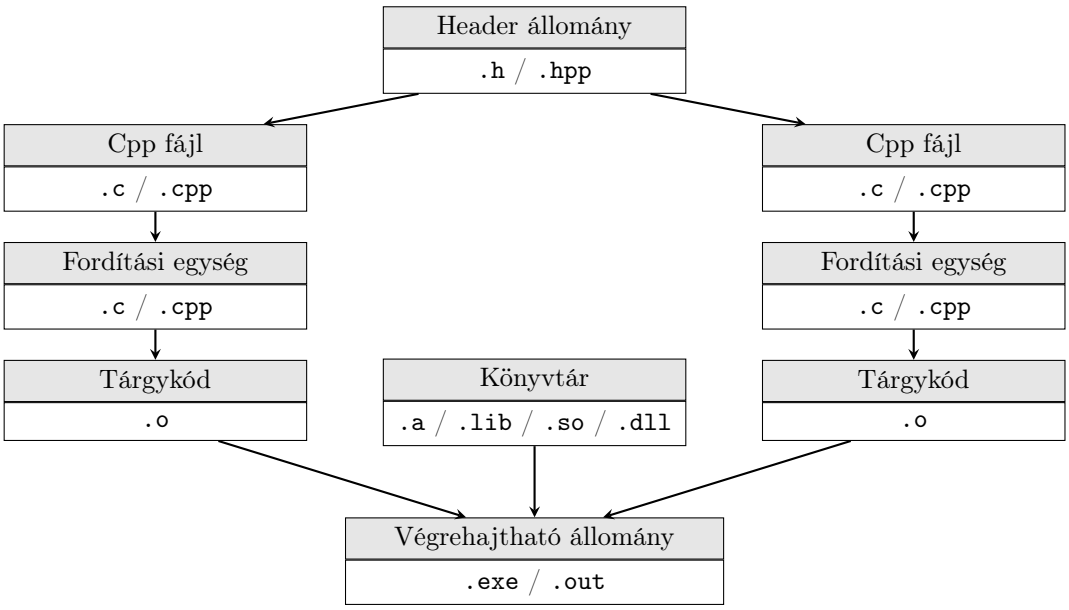
A szabvány nem köti meg, hogy egy `int` egy adott platformon mennyi byte-ból álljon. Ez állandó, egy adott platformon egy adott fordító mindig ugyanakkorát hoz létre, de platform/fordítóváltás esetén ez változhat. Ennek az az oka, hogy különböző platformokon különböző választás eredményez hatékony programokat. Ennek köszönhetően hatékony kódot tud generálni a fordító, viszont a fejlesztő dolga, hogy megbizonyosodjon róla, hogy az adott platformon a primitív típusok méretei megfelelnek a program által elvárt követelményeknek.

3. A fordító működése

A fordítás 3 fő lépésből áll:
1. Preprocesszálás
2. Fordítás (A tárgykód létrehozása)
3. Linkelés (Szerkesztés)

A fordítás a preprocesszor parancsok végrehajtásával kezdődik (például a **header fájl**ok beillesztése a **cpp fájl**okba), az így kapott fájlot hívjuk **fordítási egység**nek (*translation unit*). A fordítási egységek külön-külön fordulnak **tárgykód**dá (*object file*). Ahhoz hogy a tárgykódokból **futtatható állomány**t (*executable file*) lehessen készíteni, össze kell linkelni őket. A saját forráskódunkból létrejövő tárgykódok mellett a linker a felhasznált könyvtárak tárgykódjait is bele fogja szerkeszteni a végleges futtatható állományba.

A következő pár szekcióban megismerjük a fenti 3 lépést alaposabban.



Szürkében az adott fordítási lépés neve, alatta az így létrehozott fájl kiterjesztése (leggyakrabban).

3.1. Preprocesszálás

A preprocesszor (vagy előfeldolgozó) használata a legtöbb esetben kerülendő. Ez alól kivétel a header állományok include-olása. A preprocesszor **primitív** szabályok alapján dolgozik és **nyelvfüggetlen.** Mivel semmit nem tud a C++-ról, ezért sokszor a fejlesztő számára meglepő viselkedést okozhat a használata. Emellett nem egyszerű diagnosztizálni a preprocesszor használatából származó hibákat. További probléma, hogy az automatikus refaktoráló eszközök használatát is megnehezíti a preprocesszor túlhasználata.
A következőkben néhány preprocesszor direktívával fogunk megismerkedni. Minden direktíva `#` jellel kezdődikcl. Ezeket a sorokat a fordító a program fordítása szempontjából figyelmen kívül hagyja.

alma.h

```
#define ALMA 5

ALMA ALMA ALMA
```

A `#define ALMA 5` parancs azt jelenti, hogy minden `ALMA` szót ki kell cserélni a fájlban `5`-re.
Az előfeldolgozott szöveget a `cpp alma.h` parancs kiadása segítségével tekinthetjük meg.
Az így kapott fájlból kiolvasható előfeldolgozás eredménye: `5 5 5`.

alma.h

```
#define KORTE

#ifdef KORTE
    MEGVAN
#else
    KORTE
#endif
```

A fent leírtakon kívül a `#define` hatására a preprocesszor az első argumentumot makrónak fogja tekinteni. A fenti kódban rákérdezünk, hogy ez a `KORTE` makró definiálva van-e (az `#ifdef` paranccsal), és mivel ezt fent megtettük, `#else`-ig (vagy annak hiányában `#endif`-ig) minden beillesztésre kerül, kimenetben csak annyi fog szerepelni, hogy `MEGVAN`.

alma.h

```
#define KORTE
#undef KORTE
```

```
#ifdef KORTE
    MEGVAN
#else
    KORTE
#endif
```

Az `#undef` paranccsal a paraméterként megadott makrót a preprocesszor nem tekinti továbbá makrónak, így a kimenetben `KORTE` lesz.
Látható, hogy az előfeldolgozót kódrészletek kivágására is lehet használni. Felmerülhet a kérdés, ha az eredeti forrásszövegből az előfeldolgozó kivág illetve beilleszt részeket, akkor a fordító honnan tudja, hogy a hiba jelentésekor melyik sorra jelezze a hibát? Hiszen az előfeldolgozás előtti és utáni sorszámok egymához képest eltérnek. Ennek a problémának a megoldására az előfeldolgozó beszúr a fordító számára plusz sorokat, amik hordozzák azt az információt, hogy a feldolgozás előtt az adott sor melyik fájl hányadik sorában volt megtalálható.

3.1.1. Megjegyzés. A fordítás közbeni ideiglenes fájlokat a `g++ -save-temps hello.cpp` paranccsal lehet lementeni.

A már bizonyára ismerős `#include` egy paraméterént megadott fáj tartalmát illeszti be egy az egyben az adott fájlba, és így nagyon jelentősen meg tudják növelni a kód méretét, ami a fordítást lassítja. Ezért óvatosan kell vele bánni.

pp.h

```
#include "pp.h"
```

Rekurzív include-nál, mint a fenti példában, az előfeldolgozó egy bizonyos mélységi limit után leállítja a preprocesszálást.
Sok és hosszú include láncok esetén azonban nehéz megakadályozni, hogy kör kerüljön az include gráfba, így akaratlanul is a rekurzív include-ok aldozatai lehetünk.

pp.h

```
#ifndef _PP_H_
#define _PP_H_

    FECSKE

#endif
```

alma.h

```
#include "pp.h"
#include "pp.h"
#include "pp.h"
#include "pp.h"
#include "pp.h"
```

Egy trükk segítségével megakadályozhatjuk azt, hogy többször be legyen illesztve `FECSKE`. Először megnézzük, hogy `_PP_H_` szimbólum definiálva van-e. Ha nincs, definiáljuk. Mikor legközelebb ezt meg akarnánk tenni (a második `#include "pp."` sornál), nem illesztjük be a `FECSKE`-t, mert `#ifndef _PP_H_` kivágja azt a szövegrészt. Ez az úgy nevezett **header guard** vagy **include guard.**

A preprocesszor az itt bemutatottaknál sokkal többet tud, de általában nem érdemes túlhasználni a fent említett okok miatt.

3.2. Linkelés

fecske.cpp

```
void fecske() {}
```

main.cpp

```
int main()
{
    fecske();
}
```

Ez nem fog lefordulni, mert vagy csak a main.cpp-ből létrejövő fordítási egységet, vagy a fecske.cpp-ből létrejövő fordítási egységet látja a fordító, egyszerre a kettőt nem. Megoldás az ha **forward deklarálunk**, `void fecske();`-t beillesztjük a main függvény fölé, mely jelzi a fordítónak, hogy a `fecske` az egy függvény, `void` a visszatérési értéke és nincs paramétere.

Ekkor `g++ main.cpp` paranccsal történő fordítás a linkelési fázisánál kapunk hibát, mert nem találja a `fecske` függvény definícióját. Ezt ahogy korábban láttuk, úgy tudjuk megoldani, ha `main.cpp`-ből és `fecske.cpp`-ből is tárgykódot készítünk, majd összelinkeljük őket. `main.cpp`-ben lesz egy hivatkozás egy olyan `fecske` függvényre, melynek `void` a visszatérési értéke és paramétere nincs, és `fecske.cpp` fogja tartalmazni e függvény definícióját.

```
g++ -c main.cpp
g++ -c fecske.cpp
```

A fenti paranccsal lehet tárgykódot előállítani.

```
g++ main.o fecske.o
```

Ezzel a paranccsal pedig az eredményül kapott tárgykódokat lehet összelinkelni. Rövidebb, ha egyből a cpp fájlokat adjuk meg a fordítónak, így ezt a folyamatot egy sorral letudhatjuk..

```
g++ main.cpp fecske.cpp
```

Ha a `fecske.cpp`-ben sok függvény van, akkor nem célszerű egyesével forward deklarálni őket minden egyes fájlban, ahol használni szeretnénk ezeket a függvényeket. Ennél egyszerűbb egy header fájl megírása, amiben deklaráljuk a `fecske.cpp` függvényeit.

fecske.h

```
#ifndef _FECSKE_H_
#define _FECSKE_H_
    void fecske();
#endif
```

Ilyenkor elég a `fecske.h`-t includeolni.

Függvény definíciónak nevezzük azt, amikor megmondjuk a függvénynek hogy mit csináljon. Ez egyben deklaráció is, hiszen a paraméterekről és visszatérési értékekről is tartalmazza a szükséges információkat.

Függvény deklarációnak nevezzük azt, amikor függvény használatáról adunk információt. A paraméterek típúsáról, visszatérési értékről és a függvény nevéről.

Szokás a fecske.h-t a fecske.cpp-be is includeolni, mert ha véletlenül ellent mondana egymásnak a definíció a cpp fájlban és a deklaráció a header fájlban akkor a fordító hibát fog jelezni. (Például ha eltérő visszatérési érték típust adtunk meg a definíciónak a C++ fájlban és a deklarációnak a header fájlban.)
Valami akárhányszor deklarálhatunk, azonban ha a deklarációk ellentmondanak egymásnak, akkor fordítási hibát kapunk. Definiálni viszont mindent pontosan egyszer kell. Több definíció vagy a definíció hiánya problémát okozhat. Ezt az elvet szokás **One Definition Rule**-nak, vagy röviden **(ODR)**-nek hívni.

fecske.h

```
#ifndef _FECSKE_H_
#define _FECSKE_H_
    void fecske();
    int macska() {}
#endif
```

Ha több fordítási egységből álló programot fordítunk, melyek tartalmazzák a `fecske.h` headert, akkor a preprocesszor több macska függvény definíciót csinál, és linkeléskor a linker azt látja, hogy egy függvény többször van definiálva, és ez linkelési hibát eredményez.

3.2.1. Megjegyzés. A header fájlokba nem szabad definíciókat rakni (bár kivétel létezik, pl. template-ek, inline függvények, melyekről később lesz szó).

4. Figyelmeztetések

A fordító gyanús vagy hibás kódrészlet esetén tud figyelmeztetéseket generálni. A legtöbb fordító alapértelmezetten elég kevés hibalehetőségre figyelmeztet. További figyelmeztetések bekapcsolásával hamarabb, már fordítási időben megtalálhatunk bizonyos hibákat vagy nem definiált viselkedéseket. Ezért ajánlott a `-Wall`, `-Wextra` kapcsolókat használni.

```
g++ -Wall -Wextra hello.cpp
```

5. Optimalizálás

A fordításnál bekapcsolhatunk optimalizációkat, a GCC-nél pl. így:

```
g++ hello.cpp -O2
```

Az `-O2` paraméter a kettes szintű optimalizációk kapcsolja be. Alapértelmezetten nincs optimalizáció (`-O0`), és egészen `-O3`-ig lehet fokozni azt.

hello.cpp

```
int factorial(int n)
{
    if (n <= 0) return 1;
    else return n*factorial(n-1);
}

int main()
{
    std::cout << factorial(5) << std::endl;
}
```

A `g++ -save-temps hello.cpp` paranccsal fordítva a temporális fájlokat is meg tudjuk nézni – hello.s lesz az assembly fájl neve, mely a fordító a kódunk alapján generált. Kiolvasható benne ez a két sor:

```
movl    $5, (%esp)
call    __Z9factoriali
```

5.0.1. Megjegyzés. Az, hogy a fordító milyen assembly kódot alkot az input fájlból, implementációfüggő, ebben az esetben ezt az eredményt kaptuk.

Látható, hogy a `factorial` függvény 5 paraméterrel meg lett hívva (az hogy pontosan itt mi történik, az lényegtelen).

Amennyiben azonban `g++ -save-temps hello.cpp -O2` paranccsal fordítunk, az optimalizált assembly kódból kiolvasható, hogy a kód (kellően friss gcc-vel) a faktoriális kiszámolása helyett a végeredményt (120at) tartalmazza.

```
movl    $120, (%esp)
```

Így, mivel az eredmény már fordítási időben kiszámolásra került, futási időben nem kell ezzel plusz időt tölteni. A fordító sok ehhez hasonló **optimalizációt** végez. Ennek hatására a szabványos és csak definiált viselkedést tartalmazó kód jelentése nem változhat, viszont sokkal hatékonyabbá válhat.

5.0.2. Megjegyzés. `-O3` Olyan optimalizálásokat is tartalmazhat, amik agresszívabban kihasználják, ha egy kód nem definiált viselkedéseket tartalmaz, míg az`-O2` kevésbé aggresszív, sokszor a nem szabványos kódot se rontja el. Mivel nem definiált viselkedésekre rosszul tud reagálni az `-O3`, így néha kockázatos használni.

6. Globális változók

6.1. Féligazságok előzménytárgyakból

Előzménytárgyakból azt tanultuk, hogy a program futása a main függvény végrehajtásával kezdődik. Biztosan igaz ez?

```
std::ostream& os = std::cout << "Hello";
int main()
{
    std::cout << "valami";
}
```

Kimenet: `Hellovalami`.

Tehát ez nem volt igaz. A program végrehajtásánál az első lépés az un. **globális változók** inicializálása. Ennek az oka az, hogy a globális változók olyan objektumok, melyekre a program bármely pontján hivatkozni lehet, így ha `os`-t akarnám használni a `main` függvény első sorában, akkor ezt meg lehessen tenni. Inicializálatlan változó használata pedig nem definiált viselkedés, ezért fontos már a `main` végrehajtása előtt inicializálni a globálisokat.

```
int f()
{
    return 5;
}

int x = f();

int main()
{
    std::cout << "valami";
}
```

Itt szintén az `f()` kiértékelése a `main` függvény meghívása előtt történik, hogy a globális változót létre lehessen hozni.

6.2. Globális változók definíciója és deklarációja

Globális változókat úgy tudunk létrehozni, hogy közvetlen egy névteren belül (erről később) definiáljuk őket.

main.cpp

```
int x;

int main() {}
```

`x` egy globális változó. Azonban mit tudunk tenni, ha nem csak a `main.cpp`-ben, hanem egy másik fordítási egységben is szeretnénk rá hivatkozni?

other.cpp

```
int x;

void f()
{
    x = 0;
}
```

Sajnos ha `main.cpp`-t és `other.cpp`-t együtt fordítjuk, fordítási hibát kapunk, ugyanis megsértettük az ODR-t, hiszen `x` kétszer van definiálva. Ezt úgy tudjuk megoldani, ha `x`-et forward deklaráljuk az `extern` kulcsszóval!

other.cpp

```
extern int x;

void f()
{
    x = 0;
}
```

Csupán annyi a fontos, hogy `x`-et valamikor definiálni is kell (mely jelenleg a `main.cpp`-bentalálható).

6.2.1. Megjegyzés. A globális változók deklarációit érdemes külön header fájlba kigyűjteni.

6.3. Globális változók inicializációja

A globális változók egyedi módon kapnak kezdőértéket (inicializálódnak). Amennyiben egy nem globális `int`-et hozunk létre és nem adunk neki kezdőértéket, annak értéke nem definiált lesz (memóriaszemét).

```
int i;

int main()
{
    std::cout << i << std::endl; // 0
}
```

Azonban mégis mindig 0-t fog ez a program kiírni. Ennek oka az, hogy a globális változók mindig 0-ra inicializálódnak (legalábbis az `int`-ek). A globális változókat csak egyszer hozzuk létre a program futásakot, így érdemes jól definiált kezdőértéket adni neki.
Azonban a stacken (mellyel hamarosan megismerkedünk) rengetegszer létre kell hozni változókat, nem csak egyszer, így ott nem éri meg minden alkalommal egy jól definiált kezdőértékkel inicializálni. Sokkal nagyobb lenne a hatása a futási időre.
Annak, hogy miért épp 0-ra inicializálódnak a globális változók, az az oka, hogy ezt a modern processzorok gyorsan tudják kivitelezni minden platformon.

6.4. Problémák a globális változókkal

A linkelés vajon befolyásolhatja a program megfigyelhető viselkedését?

main.cpp

```
std::ostream& o = std::cout << "Hello";
int main() {}
```

fecske.cpp

```
std::ostream& o2 = std::cout << " World";
```

Itt nem specifikált a két globális változók inicializációs sorrendje, és ha más sorredben linkeljük a fordítási egységekből keletkező tárgykódot, mást ír ki.

`g++ main.cpp fecske.cpp` $\neq$ `g++ fecske.cpp main.cpp`

6.4.1. Megjegyzés. Ez utolsó példa nem számít jó kódnak, mert nem specifikált viselkedést használ ki. A program kimenete nem definiált. Ez is egy jó elrettentő példa, miért nem érdemes globális változókat használni.

Ezen kívül számos egyéb problémát is felvetnek a globális változók: túlzott használatuk a sok paraméterrel rendelkező függvények elkerülése végett fordulhat elő, azonban gyakran így sokkal átláthatatlanabb kódot kapunk. Mivel bárhol hozzá lehet férni egy globális változóhoz, nagyon nehéz tudni, mikor hol módosul.

6.4.2. Megjegyzés. Párhuzamos programozásnál a globális változók túl az átláthatatlanságon még sokkal több fejtörést okoznak: mi van akkor, ha két párhuzamosan futó függvény ugyanazt a változót akarja módosítani? Ennek megelőzése globális változóknál ezt rendkívül körülményes lehet. A naív megoldás (kölcsönös kizárás) pedig rosszul skálázódó programot eredményez.