C++
Gyakorlat jegyzet
1. óra

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlata alapján. (2018. május 1.)

1. Előszó

Ez a jegyzet az ELTE Informatikai Kar hallgatóinak készült a *Programozási Nyelvek C++* című tárgyhoz. A jegyzet segítségével a gyakorlaton és előadáson elhangzott anyagok elsajátíthatóak, egy saját, néhány helyen hiányos jegyzetet jól kiegészít. Úgy gondoljuk, hogy aki a jegyzetben leírtakat mélyen megérti, az olvasás mellett a példakódokkal kísérletezik, az jó eséllyel a vizsgáról nem fog sikertelenül távozni. Fontos tehát kihangsúlyoznunk, hogy a cél az aktív tudás elsajátítása, amelyet egyetlen jegyzet sem képes megadni. Az aktív tudás megszerzéséhez fontos a gyakorlás, feladatok megoldása.

A jegyzet **nem fedi le teljes mértékben az előadás és gyakorlat anyagát.** Ez azt jelenti, hogy a tárgy jó eredménnyel történő teljesítéséhez az előadáson és gyakorlaton megszerzett tudásra is szükség lesz. Például a programozási konvenciók, a programozási nyelvek történelme, és még megannyi anyagrész nem szerepel a jegyzetben, azonban ezeknek az ismerete fontos és elvárható egy programtervező informatikustól.

A jegyzettel párhuzamosan készül egy gyakorlati feladatokat tartalmazó dokumentum is, mely az eredményes tanuláshoz szintén javallott!

Hangsúlyozzuk, hogy a jegyzet elsősorban a vizsgára történő felkészülést hivatott segíteni. A vizsga közben a jegyzet használata szükséges gyakorlat nélkül nem elég.

A jegyzet első számú forrása Horváth Gábor 2015/2016/2 és 2016/2017/1 félévében tartott gyakorlatai, Pataki Norbert 2015/2016/1 előadásai, valamint a gyakorlaton megjelent helyettesítő tanárok órái: Porkoláb Zoltán, Brunner Tibor.

Külön köszönet jár Horváth Gábornak, aki a jegyzet javításában segített és aktívan segít, és mindenki másnak, aki az esetleges hibák észrevétele után szóltak.

A jegyzet teljes mértékben nyílt forráskódú, amennyiben esetleges hibába, pontatlanságba botlana, vagy szeretne segíteni az jegyzet fejlesztésében, az alábbi linken megteheti:
`https://github.com/Szelethus/ELTE-IK-CPP`.

(Felhívnánk rá a figyelmet, hogy a szerkesztés jelen pillanatában a jegyzet nem teljes terjedelmében lektorált.)

1.1. Szükséges háttértudás

Ez a jegyzet feltételezi, hogy az Olvasó elvégezte a *Programozási Alapismeretek* című tárgyat, és a *Programozás* tárgyat ezzel párhuzamosan végzi, vagy már teljesítette. Támaszkodunk arra, hogy az Olvasó tisztában van a primitív típusokkal (`int`, `float`, `double`, stb.), a szabványos bemeneten keresztül történő kiíratással és beolvasással, a konstansssággal, definiálni tud függvényeket, képes egy helyes C++ kódot lefordítani és lefuttatni (akár fejlesztői környezettel), valamint alapvető algoritmusokat ismer (pl. maximum keresés, rendezés).

2. 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. Ez is mutatja, hogy hiába fejlődtek a hardverek, továbbra is vannak olyan felhasználási területek, ahol a hatékonyság nem mellékes szempont. Az általános hatékonyság mellett a C++ alkalmas valós idejű szoftverek írására is. Mivel nem használ szemét gyűjtést, nincs nem várt szünet a program végrehajtásában a menedzselt nyelvekkel szemben. Miért előny ez? Például az önvezető autók esetében kellemetlen élményben lehetne az utasoknak része, ha az autó azért nem képes időben fékezni, mert éppen a szemétgyüjtésre vár a program. Ezen felül, a jegyzetben megmutatjuk, hogy idiomatikus C++ kód írása esetén nincs szükség szemétgyüjtésre. Sőt, az is ki fog derülni, hogy a C++ megoldása általánosabb a szemétgyüjtésnél, mivel nem csak a memória esetében működik.

A C++-szal kapcsolatban az egyik gyakori tévhit, hogy egy alacsony szintű (hardver közeli) nyelv. Bár a nyelv lehetőséget biztosít arra, hogy alacsony szinten programozzunk, számos absztrakciós lehetőséget tartalmaz. Ezeknek a használatával magas szintű kód írására is kíválóan alkalmas. A legtöbb nyelvhez képest a C++ abban emelkedik ki, hogy az itt megvalósított absztrakcióknak ritkán van futási idejű költsége.

A C++ filozófiájának fontos eleme, hogy ha nem használunk egy adott nyelvi eszközt, akkor annak ne legyen negatív 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 kíválóan alkalmas más paradigmák használatára is. A funkcionális programozástól a generatív programozáson át a deklaratív stílusig sok paradigmát támogat. A nyelv nem próbál ráerőltetni egy megközelítést a programozóra, ellenben próbál minél gazdagabb eszköztárat biztosítani, hogy a megfelelő problémát a megfelelő megközelítéssel lehessen megoldani. Még akkor is, ha ez a különböző paradigmák keverését vonja maga után. Ezért ezt a nyelvet gyakran multiparadigmás programozási nyelvnek szokták besorolni.

Cél: a tárgy során kialakítani a nyelvvel kapcsolatban egy intuíciót, amely segítségével a jegyzetben nem érintett nyelvi eszközök is könnyen megérthetőek. Emellett az idiomatikus C++ stílus elsajátítása, amely alkalmazásával könnyebben lehet hatékony és helyes kódot írni a gyakori hibalehetőségek elkerülésével. Az előzménytárgyakban az egyszerűség kedvéért gyakran féligazságok vagy kevésbé precíz definíciók hangzottak el. Bár ezek rövid távon didaktikailag megállták a helyüket, ennek a tárgynak a kapcsán rendbe rakjuk ezeket a fogalmakat.

2.1. Mi az a C++?

Alapvetően a nyelv két összetevőből áll. Az aktuális szabványból és annak implementációiból (fordítók + szabványkönyvtárak). A szabvány az, ami meghatározza a nyelv nyelvtanját, valamint a szemantikát: mit jelentenek a leforduló programok (nem definiál minden részletet). A szabvány emellett definiál egy 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 nevében a számok a szabvány elfogadásának évét jelentik.

Számos fordító (implementáció) létezik a C++ kódok fordítására, amelyek különböző mértékben támogatják az aktuális C++ szabványt: MSVC, GCC, Clang. Létezik számos fejlesztői környezet is, mint például: CLion, QtCreator, CodeBlocks, VIM. De ezek nem fordítók! Példaképp alapértelmezetten a CodeBlocks GCC-vel, a Visual Studio MSVC-vel fordít.

3. Alapok

3.1. A Hello World program

Bizonyára az alábbi program nem ismeretlen:

main.cpp

```cpp
#include <iostream>

using namespace std;

int main() { cout << "Hello World!" << endl; }
```

A `main.cpp` fájl meghatároz egy **fordítási egység**et (*compilation unit*). Fordításkor folyamata során fordítási egységeket fordítunk gépi kódra. Egy fordítási egységben az a kód található, amelyhez a fordító a fordítás során hozzáfér. A C++ fájlon túl ez a fájlban felhaszált headerek tranzitív lezártját is jelenti.

A kód legelső sorában található az ismerős `#include <iostream>`. Az `iostream` egy un. **header fájl**, mely tartalmaz valamennyi beolvasással és kiíratással kapcsolatos szabványos osztályt, függvényt és változót. Maga az `iostream` header ugyanúgy C++ kódot tartalmaz, melyet mi magunk is írunk, a `#include` direktíva hatására pedig a teljes tartalma a mi fordítási egységünkbe kerül. A fordító számára a mi fordítási egységünk a fordító számára úgy néz ki, egy fájlban szerepelne az egész `iostream`-ben található (és tranzitív módan az oda include-olt) kód, melynek a végén szerepel a Hello World programunk.

A header fájlokról később bővebben beszélünk, egyenlőre elég annyit tudni róluk, hogy C++ fájlokba szoktuk őket includeolni, önmagukban nem szokás őket fordítani.

Ez alatt található a `using namespace std;` sor. Ennek hatására az `std` névtérben található típusok, függvények, változók oly módon is elérhetővé válnak, mintha a globális névtérben lettek volna deklarálva.

A standard könyvtárban található implementációk az `std` névtéren belül találhatóak. Ennek az az oka, hogy a standard könyvtár gazdag eszközkészletet biztosít, amelynek során számos gyakran használt nevet is felhasznál, mint például `find`, `max`, stb. Ha nem az `std` névtérben lennének ezek a nevek, akkor bizonyos kontextusban nem használhatnánk fel ezeket a neveket a saját programunkban. Éppen ezért gyakran kihagyjuk ezt a sort a progamunkból, eztán pedig a standard könyvtárbeli elemekre minősített nevek segítségével hivatkozunk:

```
int main() { std::cout << "Hello World!" << std::endl; }
```

3.1.1. Megjegyzés. A fenti kódban nem írtuk ki az `#include <iostream>` sort. Ilyen rövidítésekkel gyakran fogunk élni a továbbiakban is.

Fontos, hogy `using namespace ...;` soha nem kerülhet header állományba! Ezzel ugyanis a header állomány összes felhasználójánál potenciálisan névütközéseket okozunk.

Fentebb explicit módon jeleztük a fordítónak, hogy az `std` névtérben keresse a `cout` és `endl` változókat.

A **right shift operátor** (`<<`) alternatív szintaxissal is meghívható:

```
operator<<(std::cout, "Hello World");
```

Ebből is látható, hogy az operátorok is tulajdonképpen függvények, tehát a szintaxisuktól (és néhány esetben a kiértékelési sorrendtől és rövidzártól) eltekintve ugyanazon nyelvi szabályok fognak vonatkozni rájuk, mint a többi függvényre.

Függvény deklarációnak nevezzük, amikor függvény használatáról adunk információt. Ennek része a paraméterek típúsa, visszatérési érték típusa és a függvény neve.

Függvény definíciónak nevezzük azt, amikor leírjuk a függvény törzsét is, ezzel meghatározva, hogy mit csináljon. Ez egyben deklaráció is, hiszen a paraméterekről és visszatérési értékekről is tartalmazza a szükséges információkat.

Fentebb a `main` függvényt definiáltuk. Egy függvényhez több deklaráció is tartozhat, ha azok nem mondanak egymásnak ellent:

```
int main(); // csak deklaráció

int main() // deklaráció és definíció
{
  std::cout << "Hello World!" << std::endl;
}
```

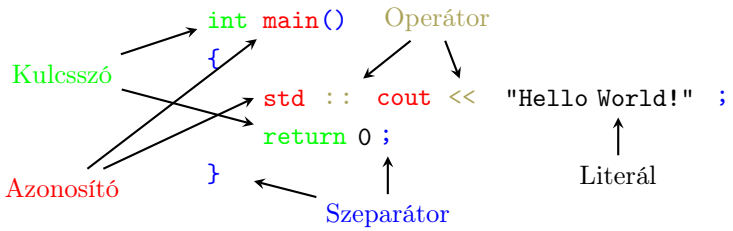
3.2. A C++ nyelvi elemei

Minden C++ kód tokenekből áll. A token a legkisebb nyelvi egység, ami még értelmes a fordító számára.

Tokeneknek az alábbiakat tekintjük:

- Kulcsszavak (*keywords*), pl. `int`, `return`. Ezek a szavak a C++ nyelvhez tartoznak, így mi újakat létrehozni nem tudunk.
- Azonosítók (*identifiers*), pl. `alma`. Lényegében ezek azok a nevek, melyket mi hozunk létre: függvénynév, osztálynév, változónév, stb. Ez csak betűkből és számokból állhat, nem kezdődhet számmal és nem lehet kulcsszó. Fontos megjegyezni, hogy a C++ nyelv különbséget tesz a kis- és nagybetűk között (*case sensitive*).
- Literálok:
  - Egész számliterálok (pl. `0xa23`, `123`)
  - Karakterliterálok (pl. `'a'`, `'\n'`)
  - Lebegőpontos számliterálok (pl. `0.12`, `3e10`)
  - Konstans szövegliterálok (pl. `"Hello"`)

  A literálokkal később egy teljes fejezet fog foglalkozni.
- Operátorok, pl. `<<`, `::`, `+`, `-`
- Szeparátorok (*punctuators*), pl. `;`, `{`, `}`



1. ábra. A Hello World programban található tokenek.

3.3. Fordítás konzolból

Windowson nyomjuk le a windows gombot és az R-t egyszerre. Ennek hatására felugrik egy ablak, melyből lehet futtatni programokat. Írjuk be hogy `cmd`, és nyomjunk entert.

Hozzunk létre egy tetszőleges mappában (pl. `C:\test\`) egy Hello World programot `main.cpp` néven. Üssük be a parancssorba következőket:

```
>cd C:\test
>g++ main.cpp
```

Amennyiben azt a hibaüzenetet kaptuk, hogy

```
"g++" is not recognized as an internal or external command.
```

az azt jelenti, hogy ha telepítve is van a GCC, a Windows nem találja. Amennyiben van CodeBlocks telepítve a gépen mely sikeresen le tud fordítani egy helyes C++ kódot, akkor közel biztosan rendelkezünk a GCCvel, mely nagy valószínűséggel itt található: `c:\Program Files (x86)\CodeBlocks\MinGW\bin\`.

Amennyiben nem rendelkezünk CodeBlocks-al, töltsük le a fordítót, és telepítsük (ennek megoldását az Olvasóra bízzuk). Másoljuk ki a `bin` mappa abszolút elérési útvonalát.

Ezek után hozzá kell adnunk ezt a mappát a PATH változóhoz. Ezáltal a parancssor újraindítása utána Windows már a megfelelő mappában meg fogja találni a GCC fordítót.

3.3.1. Megjegyzés. A PATH változóhoz való eljutás minden Windows verziónál más lehet, különösképpen, ha pl. magyar nyelvű operációs rendszerünk van. Ez okból kifolyólag ennek megoldását ismét az Olvasóra bízzuk.

Amennyiben a fordítás sikeres volt, létrejött egy `a.exe` nevű fájl, melyet tudunk futtatni. Ha módosítjuk a kódot, akkor fordítani és futtatni 1 sorban így tudunk:

```
>g++ main.cpp && a.exe
```

Linuxon általában előre telepítve van a GCC. Ennek használata szintén hasonló, azonban a létrejött fájl kiterjesztése `.out` lesz.

```
~$ g++ main.cpp && ./a.out
```

A jegyzetben később számos extra kapcsolót megismerünk a fordításhoz.

4. Különböző viselkedések kategorizálása

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

4.1. 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 fordítókkal különböző eredményeket kaphatunk. Az, hogy mikor értékelődik ki a két `i++` a kifejezésen belül, **nem specifikált.** Amikor a szabvány nem terjed ki arra, hogy pontosan milyen viselkedésű kódot generáljon a fordító, akkor a fordító bármit választhat.

Gyakran eldönthetetlen előre, hogy mi a leghatékonyabb megoldás, ez az egyik oka, hogy nem definiál mindent a szabvány. Ez többek között lehetőséget ad a fordítónak arra, hogy **optimalizáljon.**

A C++ban van úgy nevezett szekvenciapontok. A szabvány annyit mond 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, a fordítóra van bízva.

A C++ban nem meghatározott, hogy két szekvenciapont között mi a kifejezések és részkifejezések kiértékelésének a sorrendje. Az adatfüggőségek azonban definiálnak egy sorrendet.

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

Bár a fenti kódban csak az `f` meghívása után található szekvenciapont, a függvény eredményének a kiírása előtt ki kell számolni az eredményt, különben nem tudnánk, hogy mit írjunk ki. Tehát a fenti kódban garantált, hogy a `f` az eredmény kiírása előtt fog lefutni.

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. Az is elképzelhető, hogy a program pont úgy viselkedik, amire számítunk, azonban a későbbiekben ez változhat. Az ilyen viselkedés tehát egy időzített bombaként viselkedhet, amiről nem tudjuk előre, hogy mikor fog problémát okozni.
4.1.1. Megjegyzés. Az a program, amely nem definiált viselkedéseket tartalmaz, hibás.

4.2. 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
}
```

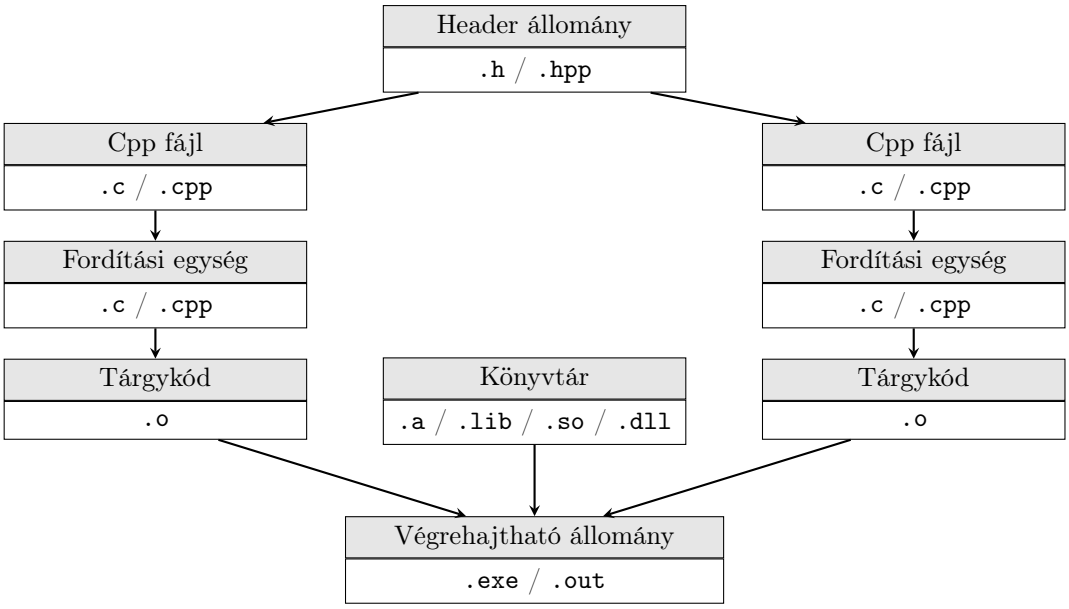
Bár azt továbbra se tudjuk, hogy `++i` vagy `++j` értékelődik ki hamarabb, (*nem specifikált*), azt biztosan tudjuk, hogy 11-et fog kiírni (a program végeredménye *jól definiált*).

4.3. 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.

5. A fordító működése

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



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

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. (ld.: 2. ábra)

A következő pár szekcióban megismerjük a fenti 3 lépést alaposabban.

5.1. 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. Emiatt 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 kezdődik. Ezeket a sorokat a fordító a 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. A cpp jelen esetben azt rövidíti, hogy c preprocessor, semmi köze nincs a C++-hoz.

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 definiáltnak fogja tekinteni. A fenti kódban rákérdezünk, hogy ez a `KORTE` makró definiálva van-e (az `#ifdef` paranccsal), és mivel ezt fent megtettük, `#else`-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 a preprocesszort kódrészletek kivágására is lehet használni. Felmerülhet a kérdés, ha az eredeti forrásszövegből a preprocesszor kivág illetve beilleszt részeket, akkor a fordító honnan tudja, hogy a hiba jelentésekor melyik sorra jelezze a hibát? Hiszen az preprocesszálás előtti és utáni sorszámok egymáshoz képest eltérnek. Ennek a problémának a megoldására az preprocesszor beszúr a fordító számára plusz sorokat, amik hordozzák azt az információt, hogy a feldolgozás előtt az adott sor melyik fájl hányadik sorában volt megtalálható.

5.1.1. 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 jelentősen meg tudják növelni a kód méretét, ami a fordítást lassítja. Ezért körültekintően kell vele bánni. A későbbiekben látni fogjuk, hogy bizonyos include-ok forward deklarációk segítségével kiválthatóak.

pp.h

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

Rekurzív include-nál, mint a fenti példában, az preprocesszor egy bizonyos mélységi limit után leállítja az előfeldolgozást.

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 beillessze `FECSKE` szöveget a preprocessor. Először megnézzük, hogy `_PP_H_` szimbólum definiálva van-e. Ha nincs, definiáljuk. Mikor legközelebb erre kerül a sor (a második `#include "pp.h"` 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 érdemes korlátozni a használatát a fent említett okok miatt.

5.2. Linkelés

Tekintsük az alábbi fordítási egységeket:

fecske.cpp

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

main.cpp

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

Fordítsuk le őket az alábbi parancsok kiadásával:

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

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 deklarál**unk, `void fecske();`-t beillesztjük a main függvény fölé, mely jelzi a fordítónak, hogy a `fecske` az egy függvény, `void` a visszatérési értékének a típuse (azaz nem ad vissza értéket) és nincs paramétere.

main.cpp

```
void fecske();

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

Ekkor `g++ main.cpp` paranccsal történő fordítás során 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, 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 linkelni. 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.

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.)

Egy adott függvényt (vagy objektumot, osztályt) akárhányszor deklarálhatunk, azonban ha a deklarációk ellentmondanak egymásnak, akkor fordítási hibát kapunk. Definiálni viszont a legtöbb esetben pontosan egyszer szabad. 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.

5.2.1. Megjegyzés. A header fájlokba általában nem szabad definíciókat rakni (kivéve, pl. template-ek, inline függvények, melyekről később lesz szó).

5.3. 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
```

5.4. 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
```

5.4.1. 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.

5.4.2. 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; }
```

Egy globális változó deklarációja hasonlít a függvényekéhez, információval látja el a fordítót arról hogy az adott szimbólum egy globális változó, és milyen a típusa. 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

Amennyiben egy lokális `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 a legtöbb 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";
```

(Nem fontos itt feltétlen értenünk a lenti inicializációt, más problémát demonstrál a kód.)

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.