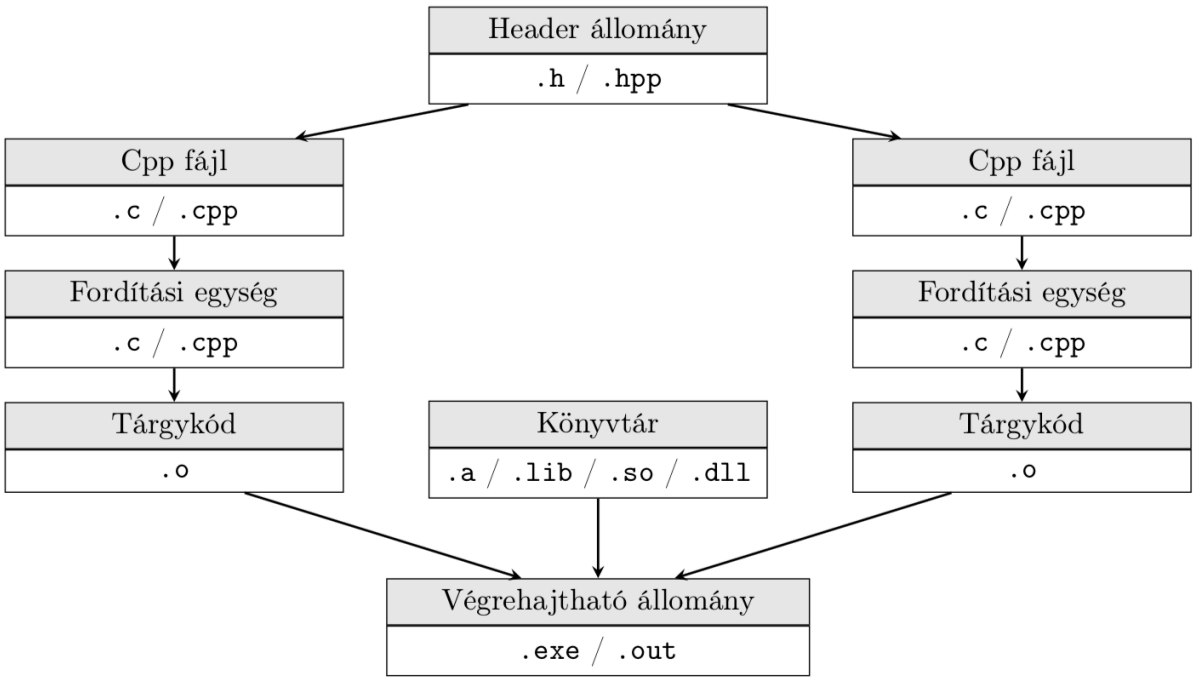
I. Gyakorlat

Fordító működése, fordítás parancssori eszközökkel (CLI), több forrásfájl, object, linkelés

A fordítás 3 fő lépésből áll:

- **Preprocesszálás** (előfeldolgozás)
- **Fordítás** (A tárgykód létrehozása)
- **Linkelés** (Szerkesztés)

A fordítás a preprocesszor parancsok végrehajtásával kezdődik, mint például a header fájlok beillesztése a `.cpp` fájlokba, az így kapott fájlokat hívjuk fordítási egységeknek `(translation unit)`. A fordítási egységek külön-külön fordulnak tárgykóddá `(object file)`. Ezekben a gépi utasítások már megvannak, de hiányoznak belőle a hivatkozások, például változók vagy függvények, melyek más fájlokban vannak megvalósítva. Ahhoz, hogy a tárgykódból futtatható állományt `(executable file)` lehessen készíteni, össze kell linkelni őket. A szerkesztő `(linker)` feladata, hogy ki töltse a tárgykódokban hiányzó referenciákat. A linkelés lehet statikus, amikor a fordító tölti fel a hiányzó referenciákat, vagy dinamikus, amikor fordítási időben, jellemzően egy másik fájlból (pl: `.dll`) tölti be a hiányzó kódot. Utóbbi akkor praktikus, ha egy modult több, különálló program használ.



Preprocesszálás

Az előfeldolgozó (`preprocesszor`) 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 meglepő viselkedést okoz a használata a fejlesztő számára. Emiatt nem egyszerű meghatározni az általa okozott hibákat. Továbbá az automatikus refaktoráló eszközök használatát is megnehezíti a túlzott használata.

A következőkben nézzünk meg pár preprocesszor direktívát. Minden direktíva `'#'` jellel kezdődik.

```
#define NUMBER 5

NUMBER NUMBER NUMBER

// A #define NUMBER 5 parancs azt jelenti, hogy minden NUMBER szót ki kell
cserélni a fájlban 5-re.

#define NUMBER

#ifdef NUMBER
      std::cout << ”Definiálva van”;
#else
      std::cout << ”Nincs definiálva”;
#endif
```

A fent leírtakon kívül a `'#define'` hatására a preprocesszor az első argumentumot definiáltnak fogja tekinteni. A fenti kódban megvizsgáljuk, hogy a NUMBER makró definiálva van-e (`'#ifdef'` parancs), és mivel ezt megtettük, `'#else'`-ig minden beillesztésre kerül, kimenetben csak annyi fog szerepelni, hogy ”Definiálva van”

```
#define NUMBER
#undef  NUMBER

#ifdef NUMBER
      std::cout << ”Definiálva van”;
#else
      std::cout << ”Nincs definiálva”;
#endif
```

Az `'#undef'` paranccsal a paraméterként megadott makrót a preprocesszor nem tekinti továbbá makrónak, így a kimenetben ”Nincs definiálva” lesz. Definiálni függvényeket is tudunk.

```
#include <iostream>

#define MAX(x, y) (x > y ? x : y)

int main()
{
	std::cout << MAX(10, 12) << std::endl;
	return 0;
}
```

Látható, hogy a preprocesszort kódrészletek kivágására is lehet használni. Felmerülhet a kérdés, hogy 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 előfeldolgozás előtti és utáni sorszámok egymáshoz képest eltérnek. Ennek a problémának a kiküszöbölése érdekében a preprocesszor beszúr sorokat a fordító számára, amik hordozzák azt az információt, hogy a feldolgozás előtt az adott sor melyik fájl hányadik sorában volt megtalálható. Rekurzív include-oknál a preprocesszor egy bizonyos mélységi limit után leállítja az előfeldolgozást. Ezt egy trükk segítségével megakadályozhatjuk azt, hogy ugyan azt a fájl többször is beillesztésre kerüljön többszörös include esetén.

```
#ifndef _H_HPP_
#define _H_HPP_
	SYMBOL
#endif

#include ”h.hpp”
#include ”h.hpp”
#include ”h.hpp”
#include ”h.hpp”
```

Leellenőrizzük, hogy a `'_H_HPP_'` szimbólum definiálva van-e? Ha nincs, akkor definiáljuk. A második `'#include "h.hpp" '` -nál, nem fogjuk újra beilleszteni a `'SYMBOL'`-t, mert az `'#ifndef _H_HPP_'` kivágja azt a szövegrészt, mivel már egyszer definiálásra került. Ez az úgy nevezett **header guard** vagy **include guard.** Védi a header-t a többszörös include-tól.

Linkelés

Tekintsük az alábbi fordítási egységeket

```
symbol.cpp
  void symbol() {}

main.cpp
  int main() { symbol(); }
```

fordítsuk le őket az alábbi sorrendben.

```
clang++ (g++) main.cpp
clang++ (g++) symbol.cpp
```

Nem fog lefordulni, hiszen vagy csak a `main.cpp`-ből létrejövő fordítási egységet, vagy a `symbol.cpp`-ből létrejövő fordítási egységet látja a fordító, egyszerre a kettőt nem. Megoldás az, ha **forward deklará**lunk, azaz `'void symbol();'`-t beillesztjük a `'main()'` függvény fölé, mely jelzi a fordítónak, hogy a `symbol()` az egy függvény, visszatérési értékének a típusa void (azaz nem ad vissza értéket) és nincs paramétere.

```
symbol.cpp
  void symbol() {}

main.cpp
  void symbol(); // forward deklaráció

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

Ekkor a `clang++ (g++) main.cpp` paranccsal történő fordítás során a linkelés fázisánál kapunk hibát, mert nem találja a `'symbol()'` függvény definícióját. Ezt úgy tudjuk megoldani, ha a `main.cpp`-ből és a `symbol.cpp`-ből először tárgykódot készítünk, majd összelinkeljük őket. A `'main.cpp'`-ben lesz egy hivatkozás a `'symbol()'` függvényre, és a `'symbol.cpp'` fogja tartalmazni a függvény definícióját.

```
clang++ (g++) -c main.cpp
clang++ (g++) -c symbol.cpp
```

ezekkel a parancsokkal tudunk tárgykódot előállítani.

```
clang++ (g++) main.o symbol.o
```

ezzel a paranccsal pedig az eredményül kapott tárgykódokat lehet linkelni.

Rövidebb, ha egyből a `.cpp` fájlokat adjuk meg a fordítónak.

```
clang++ (g++) main.cpp symbol.cpp
```

Egy adott függvényt (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.
Ha egy `.cpp`-ben több függvény is van, akkor nem célszerű ezeket egyesével forward deklarálni minden egyes fájlban, ahol használni szeretnénk. Ennél egyszerűbb egy header fájl megírása, amiben deklaráljuk a függvényünket.

```
symbol.hpp
  #ifndef _SYMBOL_H_
  #define _SYMBOL_H_
    void symbol();
  #endif
```

Ilyenkor elég a `symbol.hpp`-t `include`-olni. Szokás a `symbol.hpp`-t a `symbol.cpp`-ben is `include`-olni, mert ha véletlenül ellent mondana egymásnak a definíció a `cpp` fájlban és a deklaráció a `header` fájlban akkor a fordító hibát fog jelezni. (Pl.: eltérő visszatérési értékeket adunk meg.)

Tekintsük a következő kódrészletet.

```
symbol.hpp
  #ifndef _SYMBOL_H_
  #define _SYMBOL_H_
    void symbol() {}
  #endif
```

Ha több fordítási egységből álló programot fordítunk, melyek tartalmazzák a `symbol.hpp` `headert`, akkor a preprocesszor több `'symbol()'` függvény definíciót csinál és linkeléskor a linker azt látja, hogy egy függvény többször van definiálva, és ez linkelési hibához vezet.
Megjegyzés: a `header` fájlokba általában nem szabad definíciót írni, kivéve pl. `template`-ek, `inline` függvények.

Hasznos kapcsolók fordításnál

- `-Wall -Wextra` – warningok megjelnítése
- `-std=c++17` (17 helyett bármelyik C++ szabvány szám elfogadott[98, 03, 11, 14]) – különböző szabványokkal való fordítás bekapcsolása
- `-fsanitize=address` – létrehoz ellenőrzéseket, amik azelőtt észrevesznek bizonyos nem definiált viselkedéseket, mielőtt azok megtörténnének

Hello World C++-ban, névtér, std, iostream

Tekintsük a következő programot. Legyen a neve 'main.cpp'

```
#include <iostream>

using namespace std;

int main(int argc, char const* argv[])
{
  cout << ”Hello World” << endl;
  return 0;
}
```

A 'main.cpp' fájl meghatároz egy **fordítási egység**et (compilation unit). Fordítás 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 ebben a fájlban használt header-ök tranzitív lezártját is jelenti.

A kód legfelső sorában található az '#include <iostream>'. A kettős kereszttel '#' jelzett sorok az előfordítónak (precompiler) szóló utasítások. Az "include" utasítás behelyettesíti a hívás helyére a megadott fájl tartalmát. Ez a helyettesítés fordítási időben történik. Az 'iostream' fejállomány tartalmazza a megfelelő I/O (**I**nput/**O**utput) utasításokat a kiíratáshoz. Két különböző elérési útvonallal tudjunk megadni az include-okat:

- '#include <...>' // Előre bekonfigurált include path-ban keresi a fájlt
- '#include "..."' // Relatív az aktuális fájlokhoz képest

Mik az előnyei az #include<iostream>-nek az <cstdio>-val szemben C++-ban ?

Típus biztonságot növel, csökkenti a hiba lehetőségek számát, lehetővé teszi a bővíthetőséget és örökölhetőséget. printf() vitathatatlanul jól működik és a scanf() is működik annak ellenére, hogy nem túl nagy a hibatűrő képessége. Mindazon által mindkettő korlátozott a C++ I/O-val szemben. Ha szeretnénk összehasonlítani a C++ I/O '<<' és '>>' operátorai nagyjából megfelelnek a C-s 'printf()' és 'scanf()' függvényeknek. Azonban több előnyük is van:

1. **Típus biztosabb**: az <iostream> esetén az objektumok típusai fordítási időben ismertek a fordító számára. Ezzel ellentétben az <cstdio> '%' mezőket használ a típusok dinamikus meghatározásához.
2. **Kevesebb hibalehetőség**: az <iostream> használatakor nincsenek redundáns '%' tokenek, amelyeknek konzisztensnek kell lenniük az 'I/O'-zni kívánt objektumokkal. Ezen redundancia eltávolítsa a hibák eltávolítását is jelenti.

3. **Bővíthető**: a C++ `<iostream>` lehetővé teszi az új, felhasználó által definiált típusok `'I/O'`-zását, anélkül, hogy bármi problémát okozna a már meglévő kódban. Képzeljük el azt a káoszt, ha mindenki egyidejűleg inkonzisztens `'%'` mezőket adna hozzá a `printf()` és `scanf()` függvényekhez.
4. **Örököltethető**: a C++ `<iostream>` mechanizmus olyan valós osztályokból épül fel, mint az `std::ostream` vagy az `std::istream`. A `<cstdio>` `FILE*`-val ellentétben ezek valódi osztályok, ezért örököltethetők. Ez azt jelenti, hogy más felhasználók által definiált dolgok is viselkedhetnek `'stream'`-ként. Automatikusan használhatjuk a rengeteg soros I/O kódot, amelyet más felhasználók írtak, akik nem is tudják és nem is kell tudniuk a saját magunk által kiterjesztett `'stream'` osztályról.

Kiírás és beolvasás, C vs. C++

C-ben a kiírásra és beolvasásra elsősorban a `'printf()'` és `'scanf()'` függvényeket használjuk. Ezekkel két fő probléma lehet: nincs típus ellenőrzés, és nem tudjuk megtanítani, hogyan kell a felhasználók által definiált típusokat kiírni illetve beolvasni. A `'scanf()'` használatánál arra is oda kell figyelni, hogy cím szerint kell kapnia a változókat, könnyű a `'&'` karakter lefelejteni, vagy olyankor is kitenni, amikor nincs rá szükség pl. `string`-eknél.

```
int a;
scanf("%s", a);
```

Ilyenkor a fordító ne ellenőrzi a paraméterek típusát, így lefordul, pedig nyilvánvalóan helytelen a kód. A `'scanf()'` string-et fog beolvasni, `char*` paramétert vár, és `int*`-ot fog kapni egyetlen `int`-nyi hellyel. Hosszabb `string`-nél a kapott helyet túlírja és a következő memóriaterület sérül, vagy összeomlik a program. Az újabb fordítók erre már figyelmeztetést adnak.

Mindkét problémára megoldást nyújt a C++ megoldása. Kiírásra az `std::cout` (C-ben `stdout`), beolvasásra az `std::cin` (C-ben `stdin`), míg hiba kezelésére a C-s `stderr`-hez hasonlóan az `std::cerr` való. Használatukhoz az `iostream` header szükséges. Sor vége karakter kiírható akár az `std::endl` használatával is, ami annyiban különbözik a `'\n'`-től, hogy azonnal kiüríti a puffert. (Általában ez azt jelenti, hogy azonnal megjelenik a képernyőn.) Ezért ez több számítással jár. Az `std::endl` meghívja a `os.put(os.widen('\n'))`, majd ezt követően az `os.flush()` függvény.

Az `std::cin` – a `scanf`-től eltérően – képes referencia szerint átvenni a változókat, így nincs probléma a címképzéssel. Az `std::cin` és az `std::cout`, és minden azonos típusú objektum, automatikusan tud konvertálódni igaz vagy hamis logikai értékekre, attól függően volt-e hiba

Mind ezek az előnyök eltörpülnek amellett, hogy az `std::cin` és az `std::cout` megtanítható arra, hogyan kezelje a felhasználók által definiált típusokat, nem csak beépített típusokat tud kezelni.

Ez alatt található a `using namespace std;` sor. A standard (`std`) névtér globális használata. Ennek hatására az `std` névtérben található típusok, függvények és változók oly módon is elérhetővé válnak, mintha a globális névtérben lettek volna deklarálva. A standard könyvtárban található implementációk az `std` névtérben találhatók. Ennek az az oka, hogy a standard könyvtár gazdag eszközkészletet biztosít, amelyek során számos gyakran használt nevet is felhasznál, mint pl. `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 programunkból. A standard könyvtárbeli elemekre minősített nevek megadásával hivatkozhatunk:

```
#include <iostream>


int main(int argc, char const* argv[])
{
  std::cout << ”Hello World” << std::endl;
  return 0;
}
```

FONTOS! A `using namespace ...;` soha nem kerülhet `header` állományban. Ezzel ugyanis a `header` állomány összes felhasználójánál potenciálisan névtükrözéseket okozunk. Fentebb explicit módon jeleztük a fordítónak, hogy az `std` névtérben keresse a `cout` és `endl` változókat. Létezik úgynevezett névtelen névtér (`unnamed namespace`), amit arra használhatunk, hogy ne szemeteljük tele a globális névteret, megvédjük magunkat a név ütközésektől. Csak az adott fájlon belül lesznek elérhetőek a scope operátor `(::)`segítségével.

```
namespace
{
 // amit el akarunk keríteni
}
```

A `'main()'` függvény a program belépési pontja `(entry point)`. Minden C++ nyelven írt programnak tartalmaznia kell egyet. PONTOSAN egyet. Paraméterei közül az `'argc'` a parancssori paraméterek számát adja meg, míg az `'argv[]'` egy nullpointerrel terminált, karaktermutatókat tartalmazó tömb, amelyben a paraméterek vannak C-stílusú `string`-ként. A C++-ban a tömböket 0-tól indexeljük, az `'argv[]'` nulladik eleme a futtatható állomány neve, első eleme pedig az első paraméter.

```
./program elso masodik (vagy program.exe elso masodik)
argv[0] == „program”
argv[1] == „elso”
argv[2] == „masodik”
```

A két kapcsos zárójel '{ }' közti részt blokknak nevezzük.
A '::' az ún. hatókör operátor (scope operator). A 'return' kulcsszó visszaadja a vezérlést az őt hívó függvénynek, jelen esetben ez a program befejezését jelenti, ezért az operációs rendszernek. A 'return' mögé írt szám visszatérési értéke a 0. Ez általában azt jelzi, hogy a program rendben lefutott. A 'main()'–ben ez nem kötelező, ha elhagyjuk akkor automatikusan 0-t ad vissza, továbbá nem muszáj kiírni az 'int argc, char const *argv[]' paramétereket sem. A fordító ezeket automatikusan legenerálja.

Referenciák és pointerek

A **mutatók** olyan nyelvi elemek, melyek egy adott típusú memóriaterületre mutatnak. Segítségükkel anélkül is tudunk hivatkozni egy adott objektumra, hogy közvetlenül az objektummal dolgoznánk. Ahhoz, hogy értéket tudjunk adni egy mutatónak, egy memóriacímet kell neki értékül adni, erre való a **címképző operátor (&)**. Ha a mutató által mutatott értéket szeretnénk módosítani, akkor dereferálnunk kell a **dereferáló operátor**ral **(*).** Definiálásnál nem kötelező az inicializálásuk. Megkülönböztetünk konstans-ra mutató mutatót és konstans mutatót. Konstansra mutató mutató-n keresztül nem tudjuk megváltoztatni a mutatott értéket, de magát a mutatót át tudjuk állítani, hogy más memóriacímre mutasson. Konstans mutató esetében megtudjuk változtatni a mutatott értéket, viszont nem tudjuk átállítani a mutatót másik memóriacímre.

A **referencia** egy létező objektum alternatív neve. Definiálásánál meg kell adni azt az objektumot is, amelyet alternatív névvel látunk el. A referencia nem egy változó, mint a pointer, hanem csak egy azonosító, ezért nincs is címe és nem is változtatható meg, amíg a referencia létezik mindig ugyan oda mutat. Két leggyakoribb felhasználása:

- függvény paraméter
- függvény visszatérési érték

Mindkét esetben lehet konstans és nem konstans.

- Ha konstans a referencia, akkor azért használjuk, hogy kevesebb adatot kelljen mozgatni. Ez jellegzetesen struktúrák esetén jelent problémát. Sok adattagja lehet, akár tömbök is. Ha referenciát használunk, akkor csak egy pointer másolódik.
- Ha nem konstans a referencia, akkor paraméter esetében a hívás helyén lévő adat módosítható. Referencia visszatérési típus esetén pedig a visszaadott változó lesz módosítható.