# 2. előadás: Fordítás első lépése → preprocessing

Gyakorlaton volt már szó a fordítás fázisairól.

## Fordítás lépései

1. Előfordítás (preprocessing)
2. Nyelvi fordítás
3. Linkelés (Összeszerkesztés, linker)

# Előfordítás

Csak szöveg átalakítást végez, nem fordít. Szövegrészek kivágása vagy benntartása, feltételektől függően. ($\#ifdef$, $\#endif$)

Fájlokat tud bemásolni ($\#include$)

Olyan, mint a Search & Replace. Nem kell neki C++ kód, akár plain text / szöveges fájlokon is tud dolgozni.

Ez hátrány / veszély is.

## Tipikus preprocesszor direktívák

* $\#include <iostream>$ – Az "input-output stream" fájl beszúrása. Előre definiált helyről.
* $\#include "név.h"$ – Az aktuális könyvtárhoz képest relatív módon keresi a fájlt.
(Include vajon miért tudja a pontos sorszámot visszaadni?)
* $\#define N 100$ – Minden $N$ előfordulást a szövegben lecserél $100$-ra.
Például tömb méretének definiálásához: $int v\left[N\right]\rightarrow int v\left[100\right]$
Ciklusban az $i<N$ ciklusfeltétel is lecserélődik $i<100$-ra.
Így csak egy helyen kell átírni, ha változtatni szeretnénk a tömb méretét később. Hibák esélye kisebb.
$-wall –werror$ esetén lokális változó nem lehet tömbméret. (Mert fordítás időben nem ismert érték a változó.)
C++-ban már van konstans ($const$) változó, ami kiváltja ezt.
* $\#define ABS\left(x\right) x<0 ? \left(-1\right)\*x : x$ – Abszolút érték makró. $?:$ operátort használ. Ezzel az a gond, hogy nem igazi függvény.
$cout<<ABS\left(-4\right) \rightarrow cout<<-4<0?\left(-1\right)\*x:x$ ekkor még jó.
$cout<<ABS\left(i-3\right)$ esetén nem mindig működik:
Ha $i=-2$, akkor $5$-öt várnánk, de $-1$-et ad vissza: $cout<<\left(-1\right)\*i-3$
Azért szeretnénk mégis használni, mert a függvényhívás költsége magas. Stack-re be kell írni a paramétereket, bázispointert frissíteni kell, stb.
Túlzárójelezéssel javítható: $\#define ABS\left(x\right) \left(\left(\left(x\right)<0\right) ?\left(\left(-1\right)\*\left(x\right)\right):\left(x\right)\right)$ de így sem tökéletes.
Ma már nem használunk ilyen makrókat. Az inline függvények jobbak.
* Include guard $\#ifndef NEV\\_\\_H$ $\#define NEV\\_\\_H$ $\#endif$ Ezt ma is használjuk, többek között névütközések elkerülése miatt. De főként, hogy ne legyen kétszer beszúrva ugyan az a header fájl. Hogy ne lehessen kétszer beszúrva.
Csak akkor van rá szükség, ha több mint függvénydeklaráció van a header fájlunkban. Például osztálydeklaráció esetén érdemes.
(*Ritkán használt magyar neve: "Állomány őrszem".*)
Példa:

complex**.**h**:**

#ifndef COMPLEX\_\_H

#define COMPLEX\_\_H

 class Complex **{**

 double re**,**im**;**

 public**:**

 //...

 **}**

#endif

main**.**cpp**:**

 #include "complex.h"

 #include "equation\_solver.h"

 int main**(){**

 Complex a**,** b**,** c**;**

 //...

 **}**

equation\_solver**.**h**:**

 #include "complex.h"

 Complex solve**(**Complex a**,** Complex b**,** Complex c**);**

* $\#undef NEV$ Definíció / preprocesszor szimbólum megszüntetése.
Példa:

#if N < 120

//...

#elif N > 160

//...

#else

#endif

* $\#ifdef \\_\\_cplusplus$ Milyen fordító fordít. C verziójának lekérdezése.
C++ már engedi a függvények túlterhelését (function overloading). A c ezt még nem tudta. Minden függvénynév egyedi kell, legyen.
Ez akkor jó, ha szeretnénk, hogy programunk fordítható legyen akár c-vel akár c++-szal. Azonos nevű, különböző paraméterezésű funkciók kezelése.
Példa:

#ifdef \_\_cplusplus

extern "C" **{**

#endif

 void f**(**int**){...}**

#ifdef \_\_cplusplus

**}**

#endif

* $\#undef COMMENT$ $\#ifdef COMMENT$ … $\#endif$ Jobb kommentkezelés. Csak sor elején működik. Vigyázzunk, hogy a COMMENT változó ne legyen definiálva. Miért van erre szükség: Régi c-ben csak /\* ... \*/-szerű komment van. Ez nem ágyazható egymásba. (Nesting)

## Linking, linker (Összeszerkesztés)

$a.cpp$-ből lesz az object fájl. $a.obj$ vagy $a.o$.

$b.cpp$-ből hasonlóan.

Az object fájlokból ($a.o$, $b.o$) lesz az összeszerkesztett fájl. $x.exe$ Ez futtatható fájl.

Könyvárat is lehet írni, ez linkelés után $.dll$ lesz, ami nem futtatható le.

### Statikus linkelés

Mielőtt eljutna a felhasználóhoz, már összeáll az egész program. Linkeléskor mindent feloldunk, minden szükséges könyvtárat belerakunk a futtatható állományba.

Mindenről egyértelműen kiderül, hogy megvan-e. De nekünk kell újra linkelni, ha új verzió jön ki valamelyik könyvtárunkból.

### Dinamikus linkelés

A $.dll$ ilyen komponens. Dynamic linked library

Futási időre hagyjuk a külső hivatkozások feloldásának nagy részét.

Ettől lesz a sok "missing $.dll$" hiba. De cserébe kisebb lesz az $.exe$.

Nem kell újrafordítani, ha új dll verzió jön ki.

Lehet "on demand" lusta, lassú feloldás vagy szigorú, csak akkor indul el a szoftver, ha minden hivatkozás él.