

A programozás alapjai 2.

Hivatalos segédlet az első laborhoz.

Fejlesztői környezet

Laborokon a Visual Studio Code vagy a Visual Studio 2019 fejlesztői környezetet (IDE: Integrated Development Environment) használjuk, mivel az van a gépekre telepítve, de a régebbiek is megfelelőek.


További lehetséges fejlesztő eszközök

- **JetBrains termékek** (<https://www.jetbrains.com/student/>)
 - **ReSharper:** <https://www.jetbrains.com/resharper/> (Visual Studio-hoz)
 - **CLion:** <https://www.jetbrains.com/clion/> (cross-platform)
- **Xcode:** <https://developer.apple.com/xcode/> (macOS)
- **Code::Blocks:** <http://www.codeblocks.org/> (cross-platform)
- **Dev-C++:** <https://sourceforge.net/projects/orwelldevcpp/>
- **NetBeans:** <https://netbeans.org/features/cpp/index.html>
- **Szövegszerkesztő + (Clang vagy MinGW vagy g++):**

Visual Studio Code saját gépre


Letöltés

A Visual Studio Code letöltése a <https://code.visualstudio.com/download> oldalon történik, a megfelelő platform kiválasztásával.



↓ **Windows**
Windows 8, 10, 11


User Installer	x64	x86	Arm64
System Installer	x64	x86	Arm64
.zip	x64	x86	Arm64
CLI	x64	x86	Arm64



↓ **.deb**
Debian, Ubuntu

↓ **.rpm**
Red Hat, Fedora, SUSE

.deb	x64	Arm32	Arm64
.rpm	x64	Arm32	Arm64
.tar.gz	x64	Arm32	Arm64
Snap	Snap Store		
CLI	x64	Arm32	Arm64



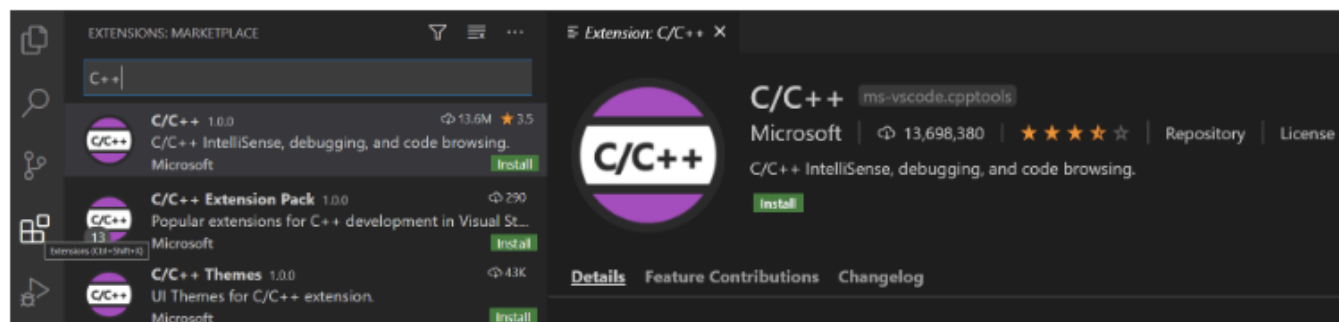
↓ **Mac**
macOS 10.11+

.zip	Intel chip	Apple silicon	Universal
CLI	Intel chip	Apple silicon	

Maga a VS Code egy nagyon jól használható szerkesztő mindenféle kódolási nyelvhez, környezethez. Ahhoz, hogy c++ kódot tudjunk vele fordítani, szükséges, hogy legyen ennek megfelelő szerkesztőfelületi támogatás (extension) és fordítást, linkelést végző programok.

A Microsoft által biztosított C++ extensiont az alábbi leírás alapján telepíthetjük:

<https://code.visualstudio.com/docs/languages/cpp>

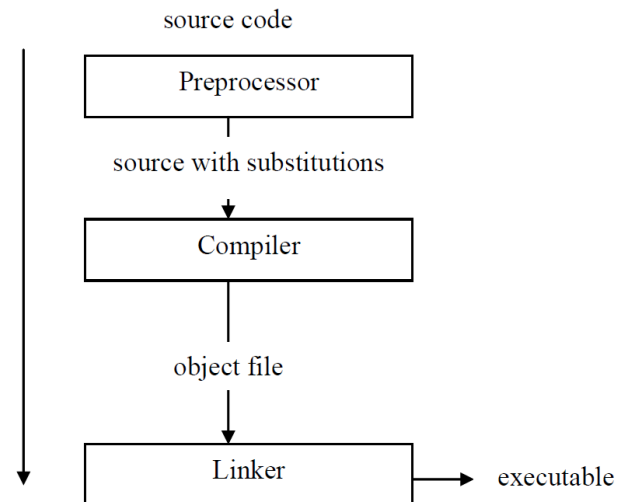


Ha nincs compiler a gépünkön, ugyancsak innen érdemes a howto doksit végigvenni, és a MinGW-t telepíteni. Érdemes a <https://www.msys2.org/> oldalon levő telepítési útmutatót követni ehhez (pl. gdb telepítése is a pacman használatával).

A fordítás menete

Preprocesszor

Végignézi a forráskódot, és minden neki szóló utasításnál (preprocesszor direktíva), melyek mindegyike #-gel kezdődik (pl. `#include`) változtatást végez a kódon. A kimeneti fájlok C/C++ nyelvűek és már nem tartalmaznak preprocesszor direktívákat. Továbbá speciális jelöléseket tesz a kódba a compiler számára, melyekből visszakövethető, hogy melyik behelyettesített sor honnan származik. Így későbbi error előfordulásakor tudni lehet, hol történt a hiba.



Szemantika

`#include <fájl>` esetén a hivatkozott fájl egész tartalmát bemásolja a kódba.

`#define azonosito helyettesites` makrók esetén a kódban az “azonosito” minden előfordulását lecseréli a megadott behelyettesítendő értékre. pl. `#define PI 3.14` makrónál, ha 100 helyen használtuk a PI-t, akkor 100 helyre behelyettesíti a 3.14 értéket.

Header fájlokban:

```

#ifndef VALAMI // ha nincs definiálva a "VALAMI", akkor...
#define VALAMI // definiálunk egy "VALAMI" direktívát
// itt vannak a változók és függvények deklarációi...
#endif // az ifndef vége
  
```

Ez a formátum lehetővé teszi, hogyha véletlenül többször include-olnánk be egy header fájlt (ami nagy szoftverek esetén előfordulhat), akkor csak egyszer másolja be a kódba. Így elkerüljük a függvények, változók stb., dupla definícióját, aminek az eredménye compilation error lenne.

`#pragma`

once

Az előbbi helyettesíti, ha a header fájl első sorában van. Előny: rövidebb.

```

#ifdef _LINUX
#include "LINUX_FUNC.h"
#elif _WIN32
#include "WINDOWS_FUNC.h"
#else
#error OS not supported
#endif
  
```

Modern fejlesztőeszközök definiálnak operációs rendszer azonosító direktívákat, amik alapján operációs rendszer független programot tudunk írni. Tehát ha megírunk egy programot, azt ugyanúgy le tudjuk fordítani jelen esetben linuxon és windowson egyaránt. Például: linuxon és windowson máshogy működik a fájlkezelés és más a könyvtárszerkezet, ezért amikor hivatkozunk egy fájlra, más formátumú útvonalat kell megadnunk. Ezt a problémát célszerű nem úgy megoldani, hogy a függvényen belül ellenőrizzük, hogy milyen

operációs rendszeren fut a program, mert ezt egyrészt meg kellene tennünk minden platform függő függvényben, ráadásul bonyolulttá és átláthatatlanná válna az egész. (Ezt nem kell megtanulni, de hasznos érdekesség).

Compiler

A preprocesszor kimenetéből object fájlt készít. A C/C++ nyelvű kódot átírja előbb assembly-be, majd az assembler bináris kódba. Ebben a fájlban még lehetnek olyan hivatkozások, melyeket a fordító nem ismer, például ha egy függvény definíciója egy másik forrásfájlban van. Ez a compilernek nem gond, csak a nevét, visszatérési értékének típusát és paraméterezését, azaz deklarációját kell ismernie, azt nem, hogy ez hol található fizikailag. A deklarációt úgy is elképzelhetjük, mint egy ígéretet a compilernek, hogy ez bizony valahol létezik, nyugodtan használja. Tehát az object fájl referenciákat, hivatkozásokat tartalmaz más fájlokban lévő elemekre - függvényekre, változókra. Minden forrásfájlhoz külön object fájl jön létre, melyeknek kiterjesztése .o (linuxon) vagy .obj (windowson). Ebben a lépésben fordulhat elő pl. syntax error.

Linker

A fordítás végső kimenetét állítja elő, egy futtatható fájlt (.exe) vagy dinamikusan linkelhető fájlt (.dll), melyhez felhasználja az összes object fájlt, ami a programunkhoz tartozik. Behelyettesíti az előbb említett hivatkozásokat a memóriacímükkel a többi object fájlt vagy hivatkozott libraryt felhasználva. Ha egy függvény vagy változó definíciót egyik object fájlban sem találja, vagy többet is talál, hiba keletkezik.

C és C++ fordítója

- gcc: GNU C Compiler
- g++: GNU C++ Compiler
- g++ a c nyelvű forrásállományokat is c++ kódként fordítja
- több forrásállományunk van → először külön fordítani őket, aztán linkelni
- GNU: **G**NU is **N**ot **U**nix
- Visual Studio fordítójának neve, verziója:
 - Command Prompt
 - cd C:\Program Files (x86)\Microsoft Visual Studio xx.x\VC\bin
 - cl.exe
- gcc, g++ alternatíva: clang
- Linux parancs: `man g++`
- Néhány kapcsoló:
 - `-g` : debug bekapcsolása
 - `-Wall` : a legtöbb warningot bekapcsolja
 - `-O` vagy `-O2` : optimalizáló bekapcsolása (figyelem! bonyolultabb programoknál rendellenes működést is okozhat (pl. szálkezelésnél))
 - `-o <név>` : kimeneti fájl neve
 - `-c` : object fájl kimenet (.o)
 - `-I<include elérési út vonal>` : include mappa megadása
 - `-L<library elérési út vonal>` : library (függvény könyvtár) mappa megadása
 - `-l<library>` : library-vel való linkelés `lib<library>.a`
 - `-x <nyelv>` - opció után megadott állományok nyelvének megadása
- több állomány: először fordítani, aztán linkelni
 - fordítás a `-c` kapcsolóval → forrásállománnyal azonos nevű, .o végződésű tárgykódú állományt hoz létre
 - valamelyik forrásállományban megtalálhatónak kell lennie a main függvénynek, de csak egyben
- tárgykódú állományból `ar` segédprogrammal statikus programkönyvtár készíthető

- egyik sem tartalmaz main függvényt (ez használó kódban található általában)
- pl. ar rvs pelda.a hello.o hi.o

extern

- Változó vagy függvény deklaráció előtt áll pl. **extern int** i;
- Azt mondja meg a fordítónak, hogy a függvény/változó "valahol létezik, definiálva van", nem kell tudnia, hogy hol, csak a típusát
- Minden függvényen kívül (file scope) definiált változó alapértelmezetten extern is. File scope a main() függvényen is kívül van.
- Az extern változónak/függvénynek pontosan egyszer kell definiálva lennie. File scope-ban kell lennie.
- Ha több helyről is tud definíciót szerezni, a linker hibát dob, mert nem tudja, hogy melyiket kell használnia, ha pedig nincs definiálva akkor azért.
- *extern: "Compiler, hidd el, hogy ez valahol márpedig létezik, Linker, te pedig keresd meg a definícióját!"*

Deklaráció vs Definíció

Deklaráció	Definíció
compiler-nek van rá szüksége	linker-nek van rá szüksége
bevezet egy azonosítót, és megadja a típusát (függvény esetén a visszatérési értékét, és paramétereinek típusát)	<ul style="list-style-type: none"> • teljesen specifikál egy entitást pl. mit csinál egy függvény • ekkor jön létre a memóriában
prototípus: függvény deklarációja int add(int , int)	implementáció: függvény definíciója int add(int a, int b) { return a+b; }
akárhányszor lehet deklarálni double f(int , int); double f(int , int); // megengedett	pontosan egyszer lehet és kell definiálni, különben linkage error-t kapunk
nem definíció	egyben deklaráció is
<ul style="list-style-type: none"> • függvény prototípus (nincs törzse) double func(double); • extern szerepelhet előtte, és nincs inicializálva / nincs megadva függvény törzs extern double a; • extern double func(double); • typedef ... // lehet típust definiálni belőle • statikus adattag osztálydeklaráción belül • osztálynév deklaráció, ha nem követi definíció class A; <p>// az utolsó két pontról később tanulunk, nem kell ezeket tudni, amíg nem ismerkedtünk meg az osztályokkal, csak a teljesség kedvéért szerepelnek itt</p>	<ul style="list-style-type: none"> • double func(double a) { return a*a; } • extern double a = 10; • int c = 5; • class A {}

Statikus könyvtárak

- Változók, függvények gyűjteménye, melyeket programjaink felhasználhatnak, ha a statikus library hozzájuk van linkelve. A programunk futtatható fájljának része lesz. A program futtatásához a futtatható fájlon kívül nincs szükség másra. A használt statikus könyvtár kicseréléséhez az egész futtatható fájlt le kell cserélni.
- *(Dinamikus könyvtárak esetén a benne található függvények betöltése futási időben történik, a könyvtár nem része a futtatható fájlnak. Egyszerűen lecserélhető, a futtatható fájl lecserélése nélkül.)*
- Kiterjesztése: .lib (Windows) vagy .a (Linux)

Létrehozása

Ha nem solution-on belüli statikus programkönyvtárat szeretnénk használni:

Jobb gomb a projekt nevére -> Properties -> Linker -> Input / Additional Dependencies -> meg kell adni az előbb generált *.lib fájl nevét (ez általában a projekt neve - a kiterjesztést ne felejtsük el).

Nullterminált string-ek

Karaktertömb, ami az érvényes karakterek után egy null (0x00) karaktert tartalmaz. Ez jelzi a string végét. A string méretét úgy tudjuk kiszámolni, hogy a karaktertömb elejétől kezdve megkeressük az első null karaktert. Ebből következik, hogy a string nem tartalmazhat (a végét leszámítva) null karaktert.

Fontos, hogy ne felejtsünk el helyet foglalni a termináló karakternek. Ha pl. egy 50 karakter hosszú tömböt szeretnénk, akkor valójában 51 karakternek kell helyet foglalni a memóriában.

További hibalehetőség, ha elfelejtjük kiírni a null karaktert a tömb végére. Ekkor, ha le szeretnénk kérdezni a string hosszát, a null karaktert kereső algoritmus (mivel nem találja a karaktert) kifut a string-ünknek lefoglalt memóriaterületből.

Konstans string literálok (idézőjelek között megadott szöveg) típusa explicit jelzés nélkül is nullterminált string. Pl. "Hello World" végén ott van 12. karakterként a '\0', de ezt nem kell kiírunk. Tehát ha egy legalább 12 méretű karaktertömbnek a "Hello World" string-et adjuk értékül, akkor nem kell külön törődnünk a null karakter meglétével.

```
pl. char hello[20] = "Hello World";
```

Ezzel ellentétben, ha karakterenként adunk értéket egy karaktertömbnek, a végén nekünk kell gondoskodni arról, hogy a végére odakerüljön a null.

```
pl. char bye[10] = {'B', 'y', 'e', '\0'};
```

sizeof() vs strlen()

Először is, a sizeof() nem szöveghossz számolásra való, ellentétben a címben említett másik két függvénnyel.

Egyik példa

```
char month[] = "october";  
char month[8] = "october"; // 7+1 null terminátor; teljesen megegyezik az előző sorban lévő definícióval
```

- sizeof(month) értéke 8, hiszen 8 byte-ot foglalunk a memóriában (konkrétan a stack-en) mindkét esetben
- strlen(month) értéke 7, mert 7 hasznos hosszúságú maga a string (null terminátort nem számolja bele)

Másik példa

```
char month[100] = "october"  
• sizeof(month) értéke 100  
• strlen(month) értéke 7
```

Referencia

A referencia egy alias, tehát egy másik név egy már létező másik változóra.

Tegyük fel, hogy létrehozunk egy x nevű, int típusú változót. Gondoljunk úgy a változó nevére, mint egy felirat, amit a változó memóriaterületéhez kapcsoltunk ('tessék, ezt a memóriaterületet x-nek fogom hívni'). Amikor létrehozunk erre az x nevű változóra egy y nevű referenciát, akkor a változóhoz tartozó memóriaterületre még egy feliratot teszünk ('tessék, ezt a memóriaterületet most már y-nak is hívom'). Innentől kezdve ugyanahhoz a memóriaterülethez hozzáférhetünk x-en és y-on keresztül is (és mindketőn keresztül lehet módosítani is, nem csak olvasni).

```
int x = 0;  
int& y = x; // y egy integer referencia x-re inicializálva
```

Paraméter átadása függvénynek referenciával

Amennyiben egy változót referenciaként adunk át egy függvénynek, abban a függvényben a referencián keresztül módosítható az eredeti változó értéke. Megvan az az előnyük a pointer-ekkel szemben, hogy paraméterátadáskor nem kell a változó címét képezni, így ezt nem is tudjuk lefelejtetni (ami pedig sok fejfájást okozhat, hogy miért nem működik a program). Továbbá nem kell bonyolult pointer-szintaktikákat alkalmazni a függvényen belül (*p)

```
void func(int& number){  
    number++;  
}  
// a függvény hívása  
int num = 0;  
func(num);
```


Fontos! Függvény soha ne adjon vissza referenciát lokális változójára! Amikor a függvény visszatér, a lokális változói törődnek a stack-ből, így egy olyan memóraterrületre fog hivatkozni a referencia, ami már nem a programunké.

Pointer vs referencia

	Pointer	Referencia
NULL	lehet	nem lehet
inicializálás	nem kell	kötelező
változtatás	lehet	nem lehet
használat	dereferálható (pl. *pa = 5)	transzparens, nem kell dereferálni, ugyanúgy használható, mint a változó (pl. ra = 5)
referálható-e	igen (**pa, &pb)	nem (&ra a változó címe)
	Muszáj használni, ha: - tömbről van szó - át kell állítani máshova - lehetséges, hogy nem mutat sehova - C nyelvű függvényt kell hívni C++-ból, ami csak ezt érti	- kívülről olyan, mint egy változó - használatában olyan, mint egy pointer - függvényen belül a változótól használatban nem különbözik (nem memóriacím, hanem maga a változó)

(Forrás: Ács Judit)

Mikor használjunk referencia/érték szerinti átadást?

- **beépített típusú** változó (pl. int, double)
 - ha nem akarjuk megváltoztatni az eredeti példányt → érték szerint, mert kicsi, a másolása gyors
 - ha meg akarjuk változtatni az eredeti példányt → referencia
- **nem beépített** típusú → referencia, mert a másolás lassú
 - ha nem akarjuk megváltoztatni az eredeti példányt → konstans referencia, hogy ténylegesen megakadályozzuk
 - ha meg akarjuk változtatni az eredeti példányt → referencia

Fordítás cl.exe (VS compiler) használatával

- Jobb klikk start menüre → System → Change settings → System Properties → Advanced → Environment Variables → User variables for ... → PATH-ot egészítsük ki ezzel:
;D:¥Install¥Microsoft Visual Studio 14.0¥VC¥bin (vagy ami neked van)
- Ezután képesek leszünk a bin mappába lépés nélkül meghívni a cl.exe fájlt bármelyik mappából a CMD használatával
- Hozzunk létre egy új mappát, benne egy hello.c-vel, írjunk egy egyszerű programot
- CMD-ben lépünk be az új mappába, majd a parancssorba írjuk be: **vcvars32** (parancssori változók létrehozása (INCLUDE, LIBRARY), hogy a cl.exe tudja őket használni)
- fordítsunk: cl[.exe] /Wall hello.c

Fordítás gcc használatával, assembly elemzés linuxon:

- Compile object fájl (.o) készítéséhez: gcc -c hello.c
- Assembly: objdump -D hello.o

Ajánlott irodalom

- http://www.diffen.com/difference/C_vs_C%2B%2B
- <http://www.cprogramming.com/tutorial/c-vs-c++.html>
- [https://msdn.microsoft.com/en-us/library/0kw7hsf3\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/0kw7hsf3(v=vs.100).aspx)
- <https://msdn.microsoft.com/en-us/library/xhkh4zs.aspx>
- <https://msdn.microsoft.com/en-us/library/3awe4781.aspx>
- <https://www.visualstudio.com/vs/compare/>
- <https://www.jetbrains.com/student/>
- <https://msdn.microsoft.com/en-us/library/ms175759.aspx>