

Modern C++ for Computer Vision and Image Processing

Lecture 1: Build and Tools

Ignacio Vizzo and Cyrill Stachniss

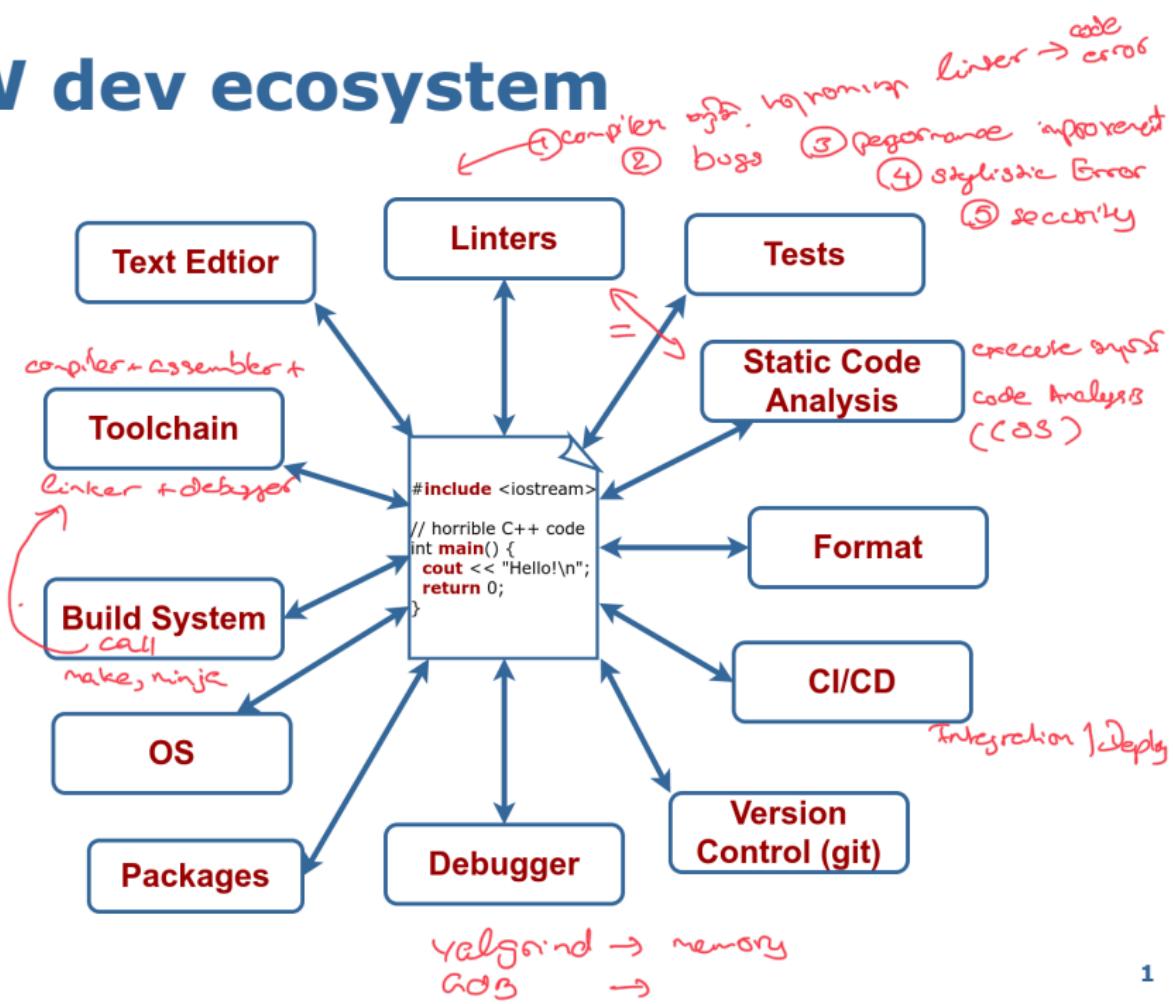
CMake

Sudo apt install libopencv-dev python3-opencv

Pkg-config --modversion opencv4

g++ -I/path -L/path -lxxx

SW dev ecosystem

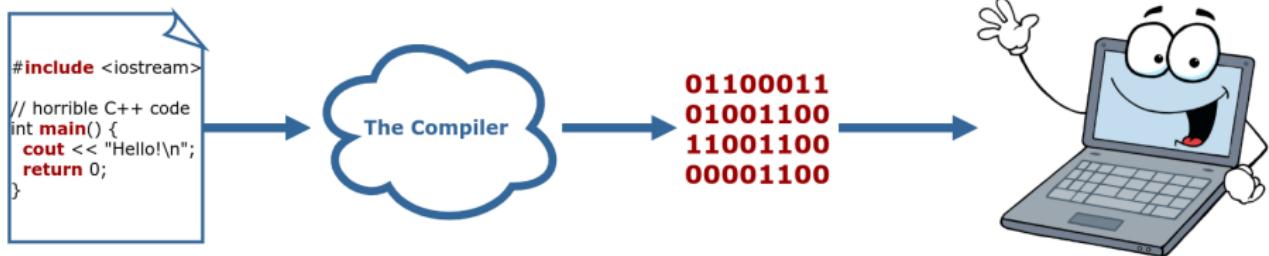


The compilation process

What is a compiler?

- A compiler is basically... a program.
- But not any program.
- Is in charge on transforming your horrible source code into binary code.
- Binary code, `0100010001`, is the language that a computer can understand.

What is a compiler?



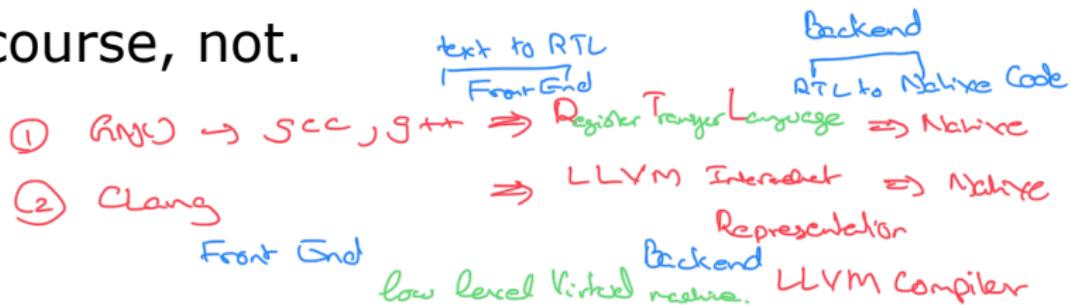
Compilation made easy

The easiest compile command possible:

- ~~clang++~~^{g++} main.cpp
- This will build a program called a.out that it's ready to run.

Will be always this easy?

- Of course, not.

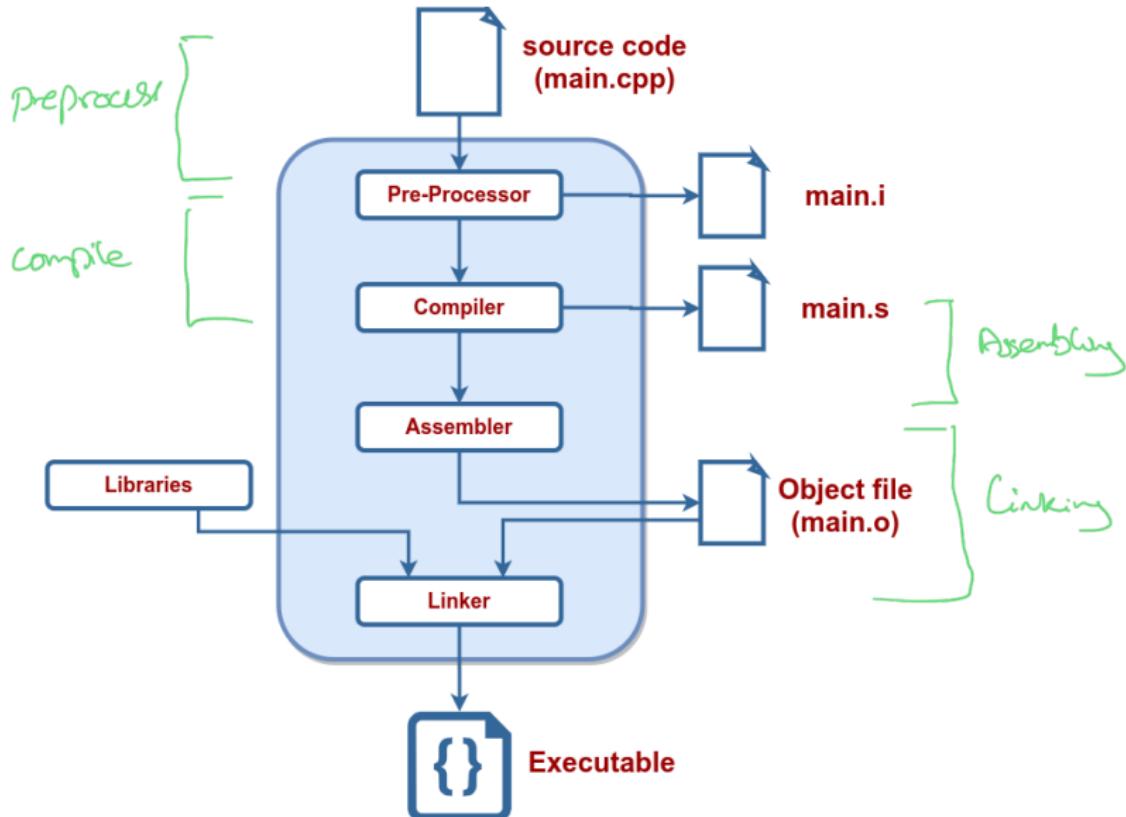


The Compiler: Behind the scenes

The compiler performs 4 distinct actions to build your code:

- 1.** Pre-process
- 2.** Compile
- 3.** Assembly
- 4.** Link

The Compiler: Behind the scenes

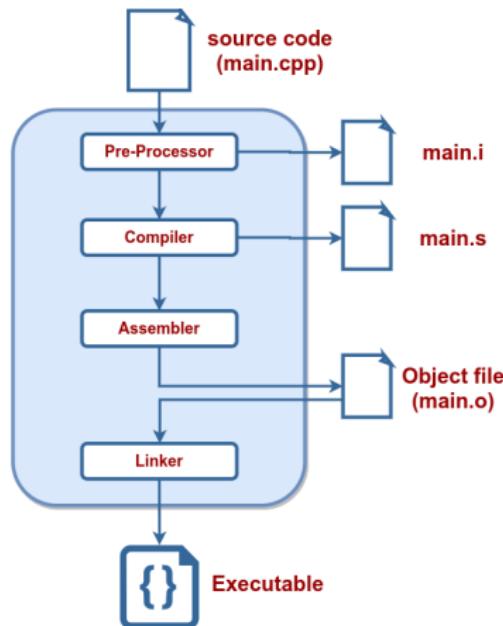


Compiling step-by-step

1. Preprocess:

■ clang++ -E main.cpp > main.i

Preprocess Only
Do not Compile
Don't Assemble
Don't Link



Compiling step-by-step

2. Compilation:

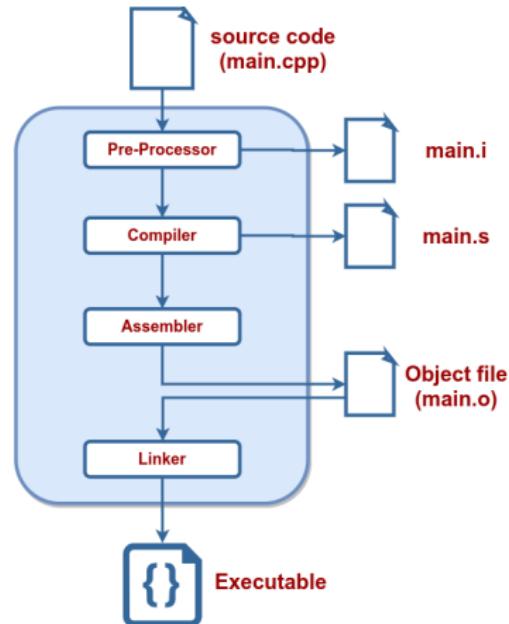
■ clang++ -S main.i



Compile Only

Do not assemble

Do not link



Compiling step-by-step

3. Assembly:

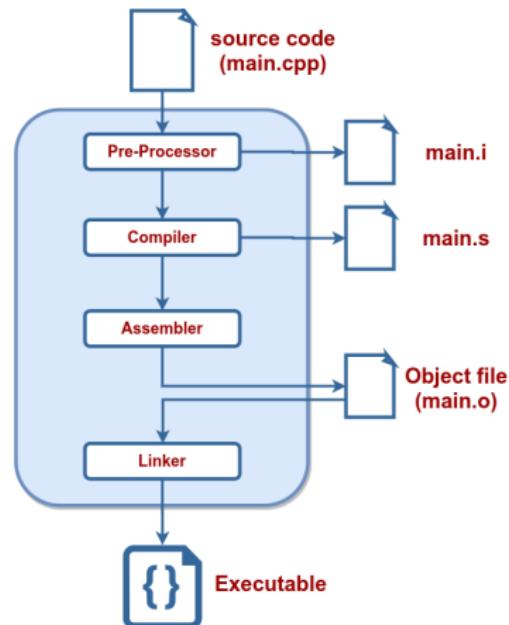
■ clang++ -c main.s



Compile and

Assemble

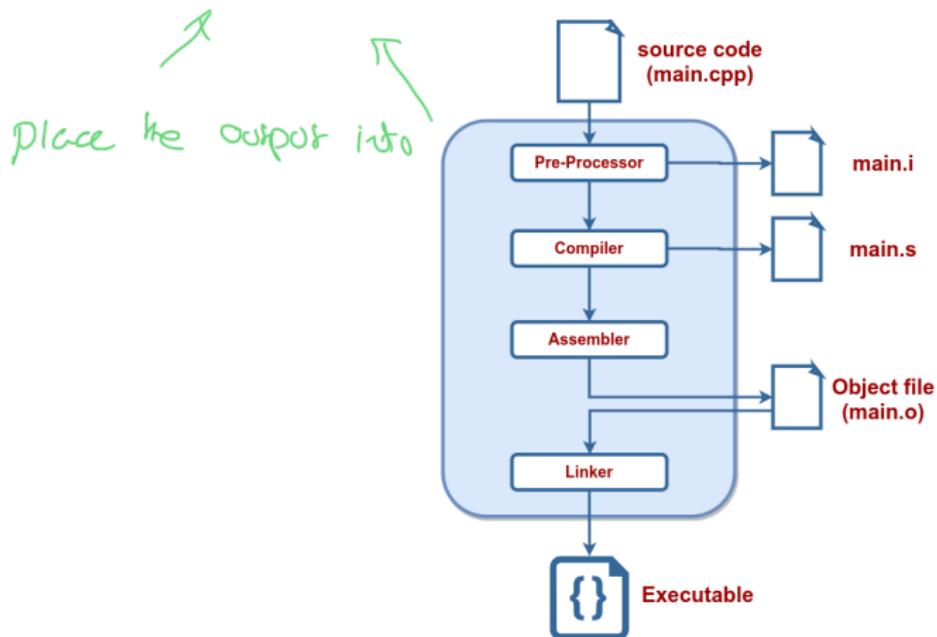
Do not link



Compiling step-by-step

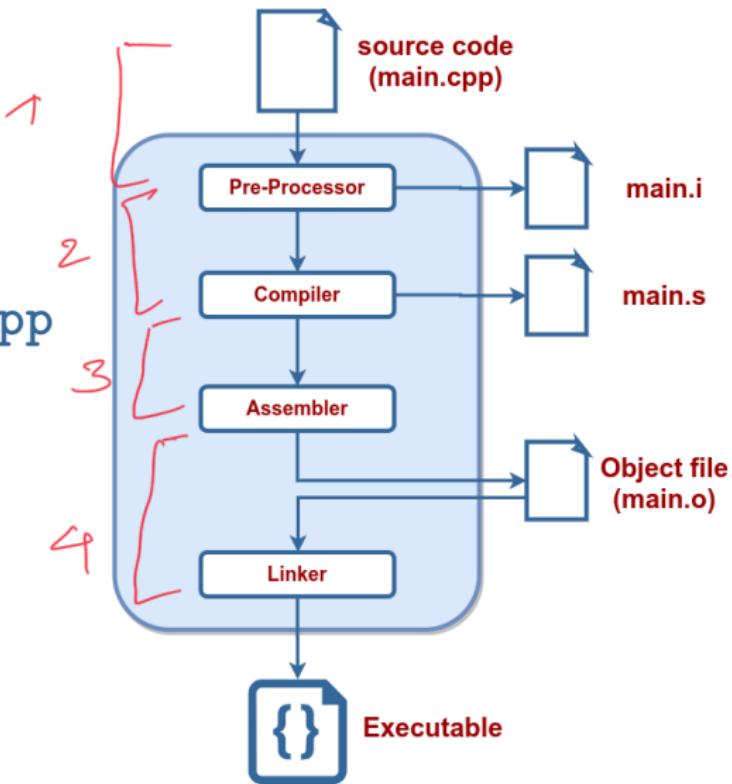
4. Linking:

- `clang++ main.o -o main`



Compiling recap

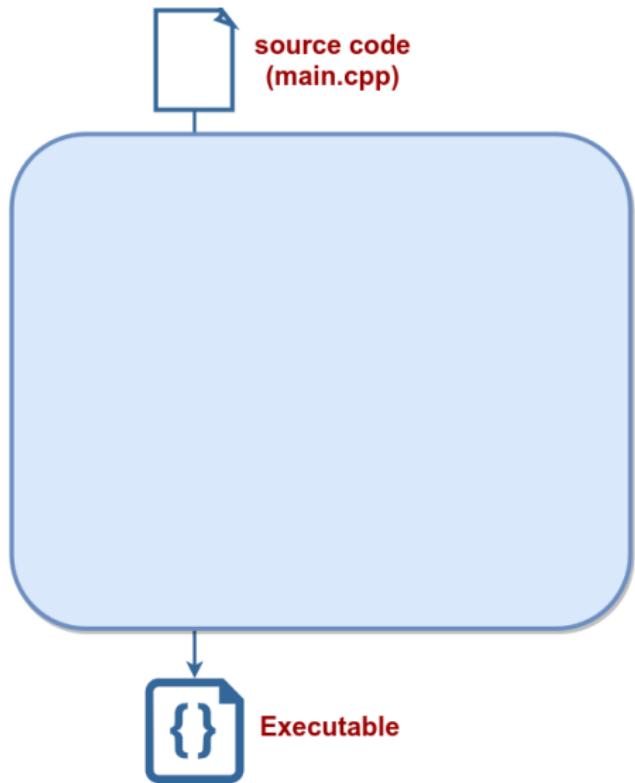
1. clang++ -E main.cpp
2. clang++ -S main.i
3. clang++ -c main.s
4. clang++ main.o



Compiling recap

1. ~~clang++~~ main.cpp

Try D2_hello



Compilation flags

- There is a lot of flags that can be passed while compiling the code
- We have seen some already:
`-std=c++17, -o, etc.`

g++ -std=c++17 main.cpp -o

Other useful options:

- Enable all warnings, treat them as errors:
`-Wall, -Wextra, -Werror`
- Optimization options:
 - `-O0` — no optimizations **[default]** *-O0*
 - `-O3` or `-Ofast` — full optimizations *fastest code*
- Keep debugging symbols: `-g` *symbolic information*

Libraries

What is a Library

- Collection of symbols.
- Collection of function implementations.



Libraries

ar -rcs

Archive

c → libname replace

c → name create

s → sort index (replace ranlib)

- **Library:** multiple object files that are logically connected
- Types of libraries:
 - **Static:** faster, take a lot of space, become part of the end binary, named: lib*.a
 - **Dynamic:** slower, can be copied, referenced by a program, named lib*.so
- Create a static library with
`ar rcs libname.a module.o module.o ...`
ar, rcs, libname.a, module.o, ...
- Static libraries are just archives just like zip/tar/...

Declaration and definition

- Function declaration can be separated from the implementation details
- Function **declaration** sets up an interface

```
1 void FuncName(int param);
```

prototype

- Function **definition** holds the implementation of the function that can even be hidden from the user

```
1 void FuncName(int param) {  
2     // Implementation details.  
3     cout << "This function is called FuncName! ";  
4     cout << "Did you expect anything useful from it?";  
5 }
```

implementation

Header / Source Separation

- Move all declarations to header files (*.hpp)
- Implementation goes to *.cpp or *.cc

```
1 // some_file.hpp
2 Type SomeFunc(... args...);
3
4 // some_file.cpp
5 #include "some_file.hpp"
6 Type SomeFunc(... args...) {} // implementation
7
8 // program.cpp
9 #include "some_file.hpp"
10 int main() {
11     SomeFunc(/* args */);
12     return 0;
13 }
```

Just build it as before?

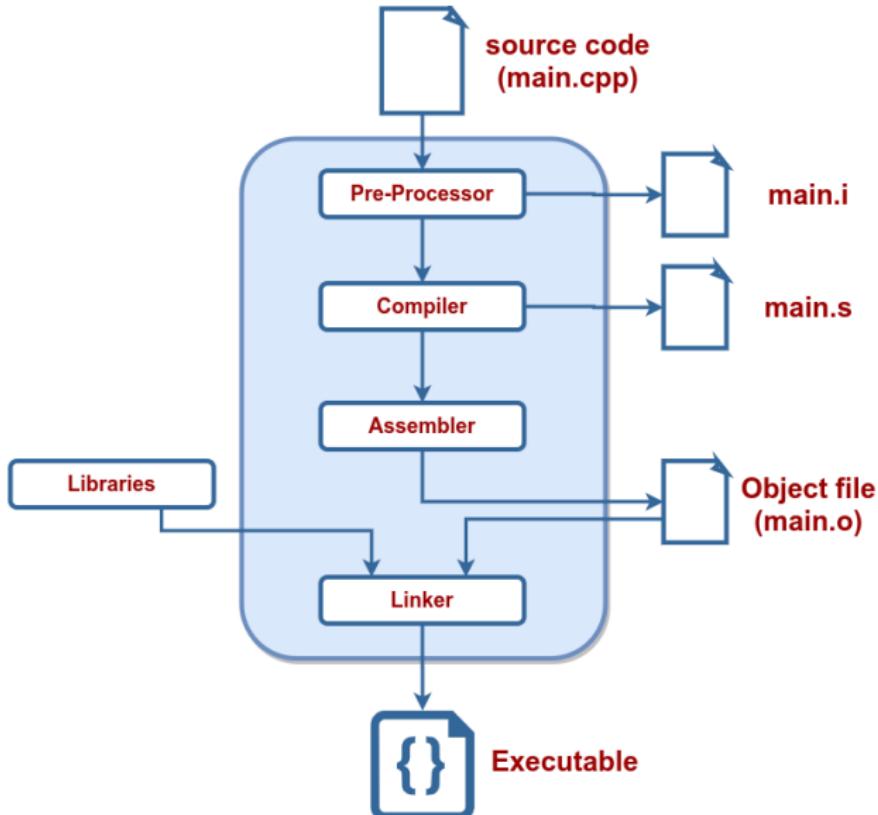
```
clang++ -std=c++17 program.cpp -o main
```

Error:

```
1 /tmp/tools_main-0eacf5.o: In function `main':  
2 tools_main.cpp: undefined reference to `SomeFunc()'  
3 clang: error: linker command failed with exit code 1  
4 (use -v to see invocation)
```

Try Q3-lib-test

What is linking?



What is linking?

- The library is a binary object that contains the compiled implementation of some methods
- Linking maps a function declaration to its compiled implementation
- To use a library we **need:**
 1. A header file `library_api.h` ✓ *header f.*
 2. The compiled library object `libmylibrary.a` ✓

*static library
follows*

How to build libraries?

```
1 folder/  
2     --- tools.hpp  
3     --- tools.cpp  
4     --- main.cpp
```

Short: we separate the code into modules

Declaration: tools.hpp

```
1 #pragma once // Ensure file is included only once  
2 void MakeItSunny();  
3 void MakeItRain();
```

Prototype

#ifndef
#define
#endif

How to build libraries?

Definition: tools.cpp

Implement

```
1 #include "tools.hpp"
2 #include <iostream>
3 void MakeItRain() {
4     // important weather manipulation code
5     std::cout << "Here! Now it rains! Happy?\n";
6 }
7 void MakeItSunny() { std::cerr << "Not available\n"; }
```

Calling: main.cpp

Use it.

```
1 #include "tools.hpp"
2 int main() {
3     MakeItRain();
4     MakeItSunny();
5     return 0;
6 }
```

Use modules and libraries!

Compile modules:

```
c++ -std=c++17 -c tools.cpp -o tools.o
```

compile and assemble, do not link
out put into

Organize modules into libraries:

```
ar rcs libtools.a tools.o <other_modules>
```

Link libraries when building code:

```
c++ -std=c++17 main.cpp -L . -ltools -o main
```

Run the code:

```
./main
```

Dynamic library

& touch main.cpp myDynamic.cpp

Made Dynamic Library

Error ⚡ g++ -shared -o libmyDynamic.so myDynamic.cpp

Ok ⚡ g++ -shared -fPIC -o " "

Now we get libmyDynamic.so

Use Dynamic lib

& g++ main.cpp -L . -lmyDynamic

Not Ok ⚡ ./a.out # But Not Ok in new Terminal

Ok ⚡ export LD_LIBRARY_PATH=. && ./a.out

C++ header & Implementation 파일

Dynamic Library C++ 구조체 정의

cd 02

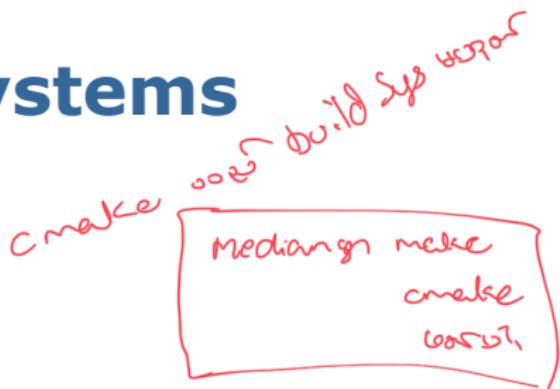
Building by hand is hard

၃၂။ Command ကျော်များ၏
၃၃။

- 4 commands to build a simple hello world example with 2 symbols.
- How does it scales on big projects?
- Impossible to mantain.
- Build systems to the rescue!

What are build systems

- Tools.
- Many of them.
- Automate the build process of projects.
- They began as shell scripts
- Then turn into MakeFiles.
- And now into MetaBuild Systems like CMake.
 - Accept it, CMake is not a build system.
 - It's a build system generator
 - You need to use an actual build system like Make or Ninja.



build system

What I wish I could write

Replace the build commands:

1. `c++ -std=c++17 -c tools.cpp -o tools.o`
2. `ar rcs libtools.a tools.o <other_modules>`
3. `c++ -std=c++17 main.cpp -L . -ltools`

For a script in the form of:

```
1 add_library(tools tools.cpp) 1,2 ←  
2 add_executable(main main.cpp) .  
3 target_link_libraries(main tools) ] 3
```

Use CMake to simplify the build

- One of the most popular build tools
- Does not build the code, generates files to feed into a build system
- Cross-platform
- Very powerful, still build receipt is readable



Build a CMake project

- **Build process** from the user's perspective
 1. `cd <project_folder>`
 2. `mkdir build`
 3. `cd build`
 4. `cmake ..`
 5. `make`
- The build process is completely defined in `CMakeLists.txt`
- And childrens `src/CMakeLists.txt`, etc.

First CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.1) # Mandatory.
2 project(first_project)           # Mandatory.
3 set(CMAKE_CXX_STANDARD 17)        # Use c++17.
4
5 # tell cmake where to look for *.hpp, *.h files
6 include_directories(include)
7
8 # create library "libtools"
9 add_library(tools src/tools.cpp) # creates libtools.a
10
11 # add executable main
12 add_executable(main src/tools_main.cpp) # main.o
13
14 # tell the linker to bind these objects together
15 target_link_libraries(main tools) # ./main
```

CMake is easy to use

- All build **files are in one place**
- The build **script is readable**
- Automatically **detects changes**
- After doing changes:
 1. `cd <project_folder>/build`
 2. `make`

~~—~~ just "make"

Typical project structure

```
1  |-- project_name/
2  |  |-- CMakeLists.txt
3  |  |-- build/    # All generated build files
4  |  |-- results/ # Executable artifacts
5  |  |  |-- bin/
6  |  |  |  |-- tools_demo
7  |  |  |-- lib/
8  |  |  |  |-- libtools.a
9  |  |-- include/ # API of the project
10 |  |  |-- project_name
11 |  |  |  |-- library_api.hpp
12 |  |-- src/
13 |  |  |-- CMakeLists.txt
14 |  |  |-- project_name
15 |  |  |  |-- CMakeLists.txt
16 |  |  |  |-- tools.hpp
17 |  |  |  |-- tools.cpp
18 |  |  |  |-- tools_demo.cpp
19 |  |-- tests/  # Tests for your code
20 |  |  |-- test_tools.cpp
21 |  |  |  |-- CMakeLists.txt
22 |  |-- README.md # How to use your code
```

Compilation options in CMake

```
1 set(CMAKE_CXX_STANDARD 17)
2
3 # Set build type if not set.
4 if(NOT CMAKE_BUILD_TYPE)
5   set(CMAKE_BUILD_TYPE Debug)
6 endif()
7 # Set additional flags.
8 set(CMAKE_CXX_FLAGS "-Wall -Wextra")
9 set(CMAKE_CXX_FLAGS_DEBUG "-g -O0")
```

اندیزه

- `-Wall -Wextra`: show all warnings
- `-g`: keep debug information in binary
- `-O<num>`: optimization level in {0, 1, 2, 3}
 - 0: no optimization
 - 3: full optimization

Useful commands in CMake

- Just a scripting language
- Has features of a scripting language, i.e. functions, control structures, variables, etc.
- All variables are string
- Set variables with `set(VAR VALUE)`
- Get value of a variable with `${VAR}`
- Show a message `message(STATUS "message")`
- Also possible `WARNING`, `FATAL_ERROR`

Build process

- CMakeLists.txt defines the whole build
- CMake reads CMakeLists.txt **sequentially**
- **Build process:**
 1. cd <project_folder>
 2. mkdir build
 3. cd build
 4. cmake ..
 5. make -j2 # pass your number of cores here

↑
cpd cores

Everything is broken, what should I do?

- Sometimes you want a clean build
- It is very easy to do with CMake
 - 1. `cd project/build`
 - 2. `make clean` [remove generated binaries]
 - 3. `rm -rf *` [make sure you are in build folder]
- Short way (If you are in `project/`):
 - `rm -rf build/`

Don't push build to git.
use gitignore

Use pre-compiled library

- Sometimes you get a compiled library
- You can use it in your build
- For example, given libtools.so it can be used in the project as follows:

```
1 find_library(TOOLS NAMES tools PATHS ${LIBRARY_OUTPUT_PATH})  
2 # Use it for linking:  
3 target_link_libraries(<some_binary> ${TOOLS})
```

CMake find_path and find_library

- We can use an external library
- Need headers and binary library files
- There is an easy way to find them
- **Headers:**

```
1 find_path(SOME_PKG_INCLUDE_DIR include/some_file.hpp  
2           <path1> <path2> ...)  
3 include_directories(${SOME_PKG_INCLUDE_DIR})
```

- **Libraries:**

```
1 find_library(SOME_LIB  
2               NAMES <some_lib>  
3               PATHS <path1> <path2> ...)  
4 target_link_libraries(target ${SOME_LIB})
```

`find_package` ✓

- `find_package` calls multiple `find_path` and `find_library` functions
- To use `find_package(<pkg>)` CMake must have a file `Find<pkg>.cmake` in `CMAKE_MODULE_PATH` folders
- `Find<pkg>.cmake` defines which libraries and headers belong to package `<pkg>`
- Pre-defined for most popular libraries, e.g. OpenCV, libpng, etc.

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.1)
2 project(first_project)
3
4 # CMake will search here for Find<pkg>.cmake files
5 SET(CMAKE_MODULE_PATH
6     ${PROJECT_SOURCE_DIR}/cmake_modules)
7
8 # Search for Findsome_pkg.cmake file and load it
9 find_package(some_pkg)
10
11 # Add the include folders from some_pkg
12 include_directories(${some_pkg_INCLUDE_DIRS})
13
14 # Add the executable "main"
15 add_executable(main small_main.cpp)
16 # Tell the linker to bind these binary objects
17 target_link_libraries(main ${some_pkg_LIBRARIES})
```

cmake_modules/Findsome_pkg.cmake

```
1 # Find the headers that we will need
2 find_path(some_pkg_INCLUDE_DIRS include/some_lib.hpp <
            FOLDER_WHERE_TO_SEARCH>)
3 message(STATUS "headers: ${some_pkg_INCLUDE_DIRS}")
4
5 # Find the corresponding libraries
6 find_library(some_pkg_LIBRARIES
                 NAMES some_lib_name
                 PATHS <FOLDER_WHERE_TO_SEARCH>)
7
8 message(STATUS "libs: ${some_pkg_LIBRARIES}")
```

Watch for Homeworks



<https://youtu.be/hwP7WQkmECE>

Watch for Homeworks



<https://youtu.be/OZEGnam2M9s>

Suggested Video

“Free software, free society” by Richard Stallman



https://youtu.be/Ag1AKII_2GM

References

- **CMake Documentation**

cmake.org/cmake/help/v3.10/

- **GCC Manual**

gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/

- **Clang Manual**

releases.llvm.org/10.0.0/tools/clang/docs/index.html