SW dev ecosystem

```
#include <iostream>

// horrible C++ code
int main() {
    cout << "Hello!\n";
    return 0;
}
```
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?

```cpp
#include <iostream>

// horrible C++ code
int main() {
    cout << "Hello!\n";
    return 0;
}
```

The Compiler

01100011
01001100
11001100
00001100
Compilation made easy

The easiest compile command possible:

- `clang++ 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

source code (main.cpp)

Pre-Processor

main.i

Compiler

main.s

Assembler

Object file (main.o)

Linker

Executable
Compiling step-by-step

1. Preprocess:
   - `clang++ -E main.cpp > main.i`
Compiling step-by-step

2. Compilation:

- `clang++ -S main.i`
Compiling step-by-step

3. Assembly:
   - `clang++ -c main.s`
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`
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.

Other useful options:
- Enable all warnings, treat them as errors:
  `-Wall, -Wextra, -Werror`
- Optimization options:
  - `-00` — no optimizations [default]
  - `-03` or `-Ofast` — full optimizations
- Keep debugging symbols: `-g`

Play with them with Compiler Explorer: https://godbolt.org/
Libraries
What is a Library

- Collection of symbols.
- Collection of function implementations.
Libraries

- **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 ...
```

- 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.
  
  ```
  void FuncName(int param);
  ```

- Function **definition** holds the implementation of the function that can even be hidden from the user.
  
  ```
  void FuncName(int param) {
    // Implementation details.
    cout << "This function is called FuncName! ";
    cout << "Did you expect anything useful from it?";
  }
  ```
Header / Source Separation

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

```cpp
// some_file.hpp
Type SomeFunc(... args...);

// some_file.cpp
#include "some_file.hpp"
Type SomeFunc(... args...) {} // implementation

// program.cpp
#include "some_file.hpp"
int main() {
  SomeFunc(/* args */);
  return 0;
}
```
Just build it as before?

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

**Error:**

```sh
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)
```
What is linking?

source code (main.cpp)

Pre-Processor

main.i

Compiler

main.s

Assembler

Object file (main.o)

Linker

Libraries

Executable
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`
  2. The compiled library object `libmylibrary.a`
How to build libraries?

**Short:** we separate the code into modules

**Declaration:** tools.hpp

```cpp
#pragma once // Ensure file is included only once
void MakeItSunny();
void MakeItRain();
```
How to build libraries?

**Definition:** tools.cpp

```cpp
#include "tools.hpp"
#include <iostream>

void MakeItRain() {
    // important weather manipulation code
    std::cout << "Here! Now it rains! Happy?\n";
}

void MakeItSunny() { std::cerr << "Not available\n"; }
```

**Calling:** main.cpp

```cpp
#include "tools.hpp"

int main() {
    MakeItRain();
    MakeItSunny();
    return 0;
}
```
Use modules and libraries!

Compile modules:
c++ -std=c++17 -c tools.cpp -o tools.o

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
Build Systems
Building by hand is hard

- 4 commands to build a simple hello world example with 2 symbols.
- How does it scales on big projects?
- Impossible to maintain.
- 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.
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)
2 add_executable(main main.cpp)
3 target_link_libraries(main tools)
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.
cmake_minimum_required (VERSION 3.1)  # Mandatory.
project(first_project)  # Mandatory.
set(CMAKE_CXX_STANDARD 17)  # Use c++17.

# tell cmake where to look for *.hpp, *.h files
include_directories(include/)

# create library "libtools"
add_library(tools src/tools.cpp)  # creates libtools.a

# add executable main
add_executable(main src/tools_main.cpp)  # main.o

# tell the linker to bind these objects together
target_link_libraries(main tools)  # ./main
CMake is easy to use

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

```plaintext
|-- project_name/
 | |-- CMakeLists.txt
 | |-- build/   # All generated build files
 | | |-- bin/
 | | | |-- tools_demo
 | |-- lib/
 | | |-- libtools.a
 | |-- include/ # API of the project
 | | |-- project_name
 | | | |-- library_api.hpp
 | |-- src/
 | | |-- CMakeLists.txt
 | | |-- project_name
 | | | |-- CMakeLists.txt
 | | | |-- tools.hpp
 | | | |-- tools.cpp
 | | | |-- tools_demo.cpp
 | | |-- tests/   # Tests for your code
 | | | |-- test_tools.cpp
 | |-- README.md # How to use your code
```
Compilation options in CMake

```cpp
set(CMAKE_CXX_STANDARD 17)

# Set build type if not set.
if(NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Debug)
endif()

# Set additional flags.
set(CMAKE_CXX_FLAGS "-Wall -Wextra")
set(CMAKE_CXX_FLAGS_DEBUG "-g -00")
```

- **-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
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/
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:

```c
1 find_library(TOOLS
2     NAMES tools
3     PATHS ${LIBRARY_OUTPUT_PATH})
4 # Use it for linking:
5 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:**

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

**Libraries:**

```cpp
find_library(SOME_LIB
             NAMES <some_lib>
             PATHS <path1> <path2> ...)
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

cmake_minimum_required (VERSION 3.1)
project(first_project)

# CMake will search here for Find<pkg>.cmake files
SET(CMAKE_MODULE_PATH
    ${PROJECT_SOURCE_DIR }/cmake_modules)

# Search for Findsome_pkg .cmake file and load it
find_package(some_pkg)

# Add the include folders from some_pkg
include_directories(${some_pkg_INCLUDE_DIRS })

# Add the executable "main"
add_executable(main small_main.cpp)

# Tell the linker to bind these binary objects
target_link_libraries(main ${some_pkg_LIBRARIES })
# Find the headers that we will need
find_path(some_pkg_INCLUDE_DIRS include/some_lib.hpp <FOLDER_WHERE_TO_SEARCH>)
message(STATUS "headers: ${some_pkg_INCLUDE_DIRS}")

# Find the corresponding libraries
find_library(some_pkg_LIBRARIES
  NAMES some_lib_name
  PATHS <FOLDER_WHERE_TO_SEARCH>)
message(STATUS "libs: ${some_pkg_LIBRARIES}")
Watch for Homeworks

100 seconds of git

STARTED!

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