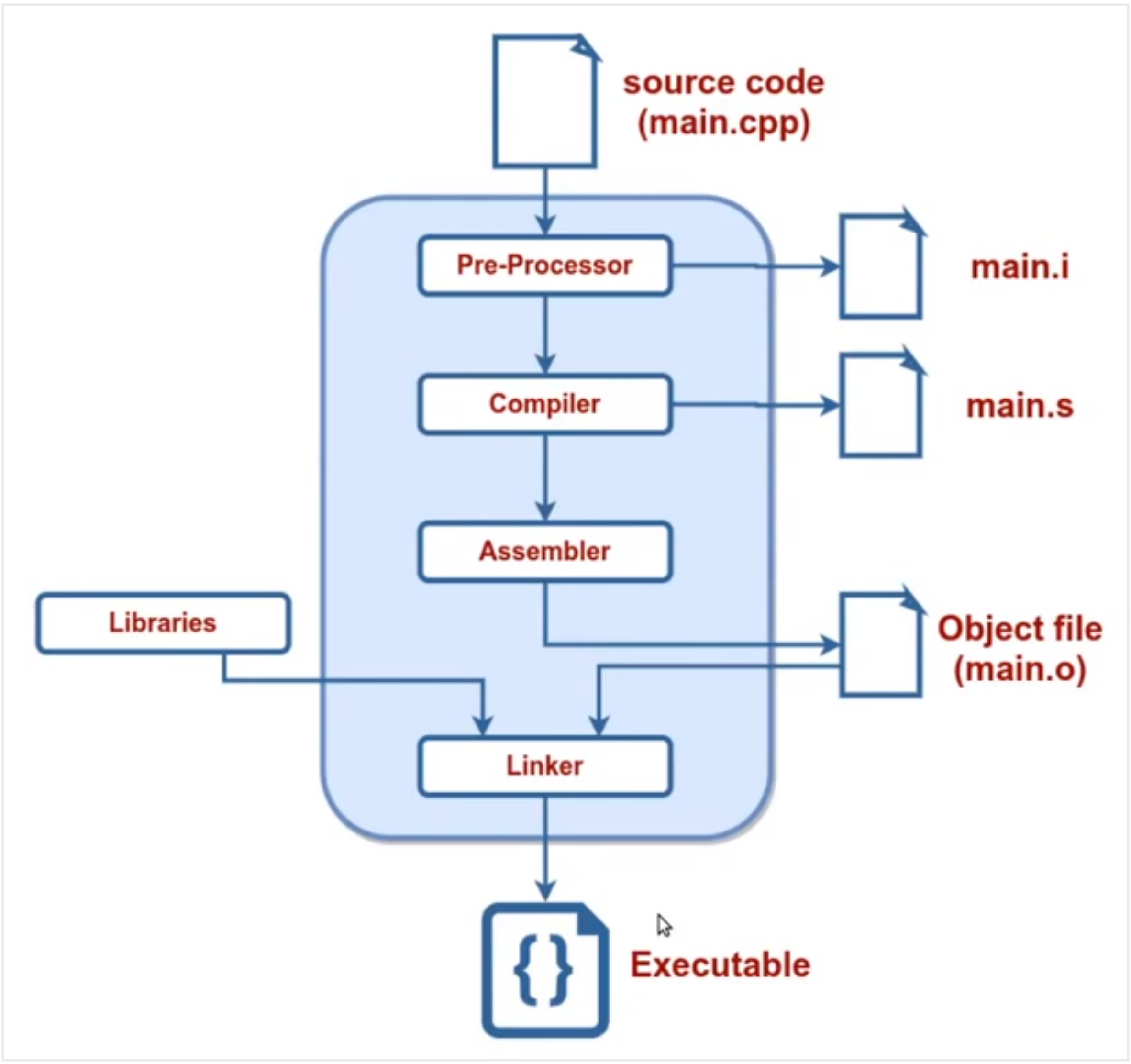
Tools & Build Systems

Tools

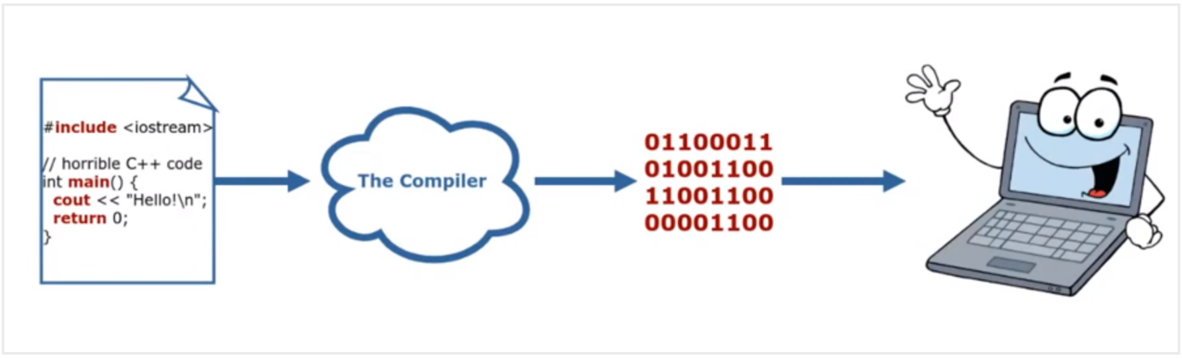


# c++ and clang++ can be used interchangeably in the commands

- Preprocess: c++ -E hello_world.cpp > main.i
- Compilation: c++ -S main.i
- Assembly: c++ -c main.s
- Linking: c++ main.o -o main

  Run ./main to execute the program

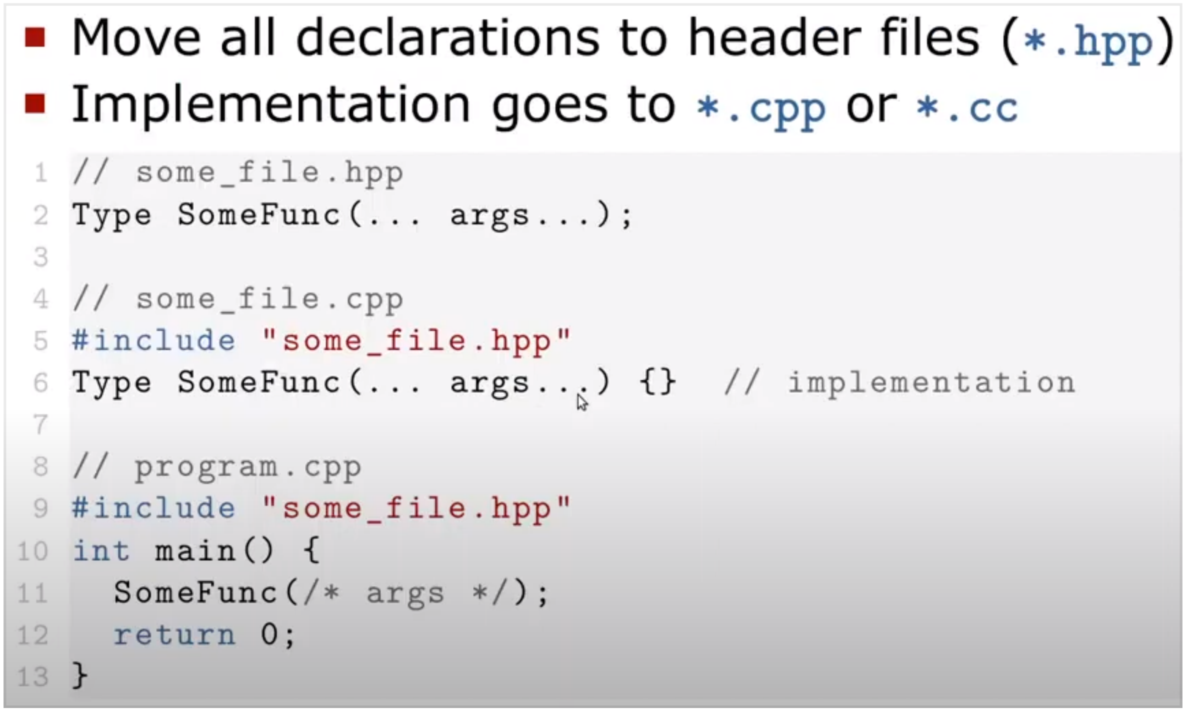
Compiler



Libraries

Library: multiple object files that are logically connected.
Types:

- Static: faster, takes lot of space, named: lib*.a
- Dynamic: slower, can be copied, named: lib*.so



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

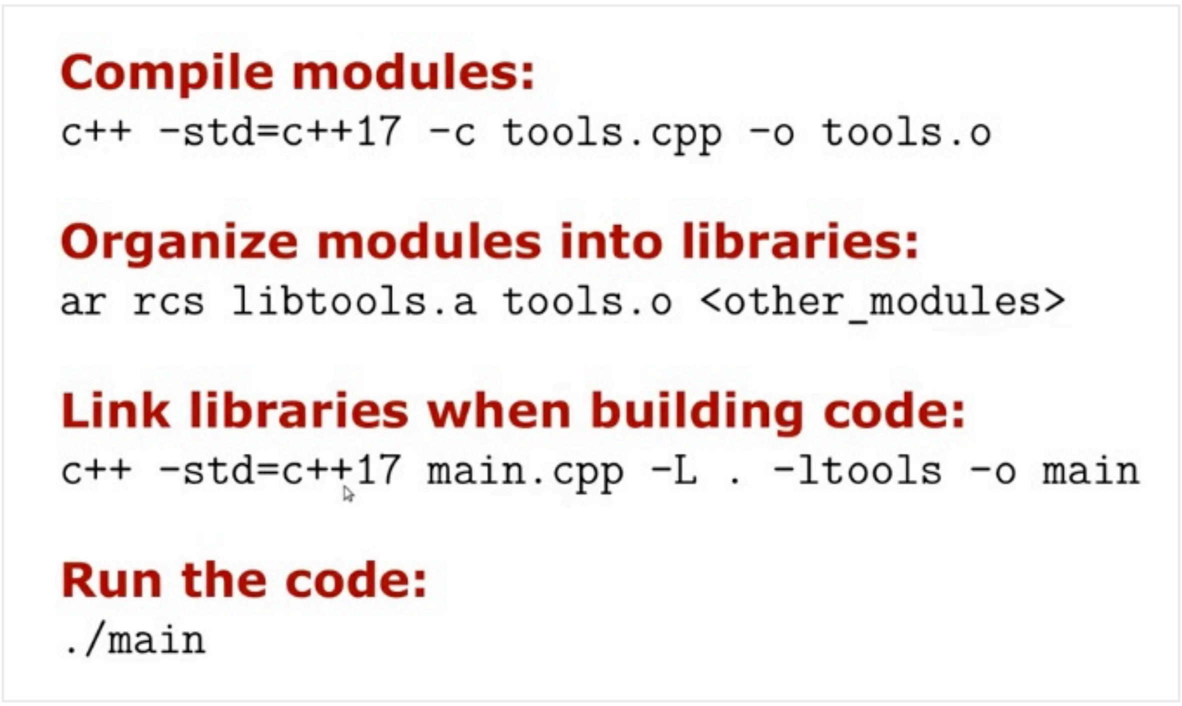
```
// some_file.hpp
Type SomeFunc(... args...);

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

// program.cpp
#include "some_file.hpp"
int main() {
  SomeFunc(/* args */);
  return 0;
}
```

Linker

The library is a binary object that contains the compiled implementation of some methods.
Linking maps a function declaration to its compiled implementation.

Example: Building a library:

main.cpp	tools.cpp	tools.hpp
#include "tools.hpp" int main(){ MakeItSunny(); MakeItRain(); return 0; }	#include "tools.hpp" #include <iostream> void MakeItSunny(){ std::cout << "It's now sunny\n"; } void MakeItRain(){ std::cerr << "Not yet implemented\n"; }	void MakeItSunny(){ std::cout << "It's now sunny\n"; }



Compile modules:

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

Organize modules into libraries:

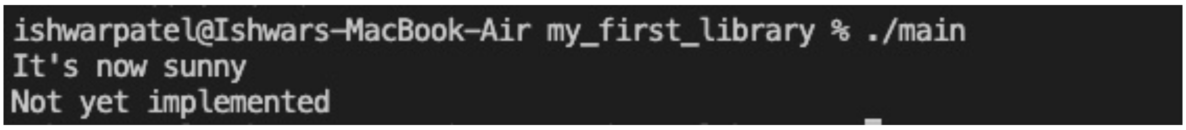
```
ar rcs libtools.a tools.o <other_modules>
```

Link libraries when building code:

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

Run the code:

```
./main
```



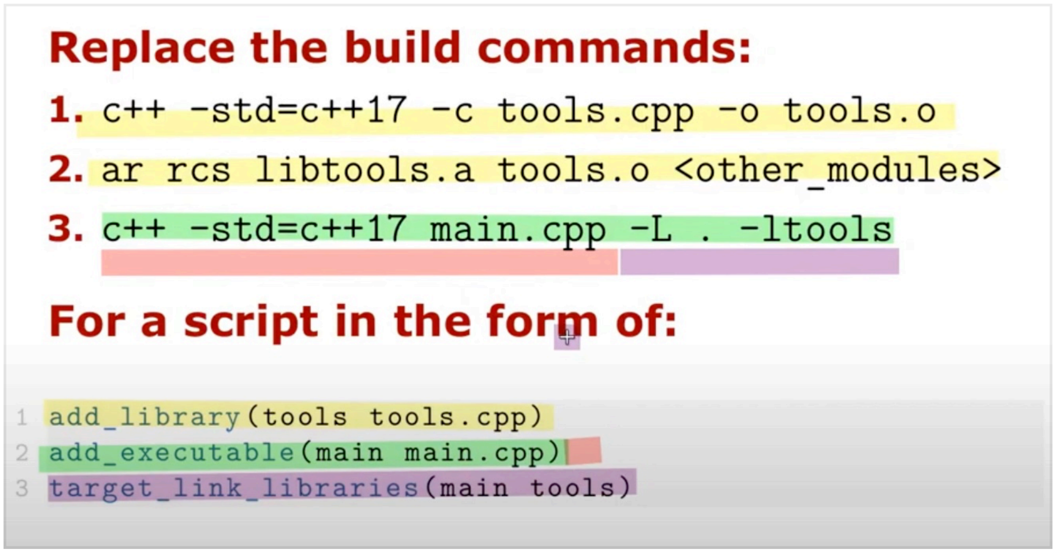
```
ishwarpatel@Ishwars-MacBook-Air my_first_library % ./main
It's now sunny
Not yet implemented
```

Build Systems

Build systems automate the build process of projects.
They began as shell scripts and turn into MakeFiles.
And now into meta-build systems like Cmake.

- Cmake is not a build system.
- It's a build system generator.
- We need to use actual build system like Make and Ninja.

Use Cmake to simplify the build.



Replace the build commands:

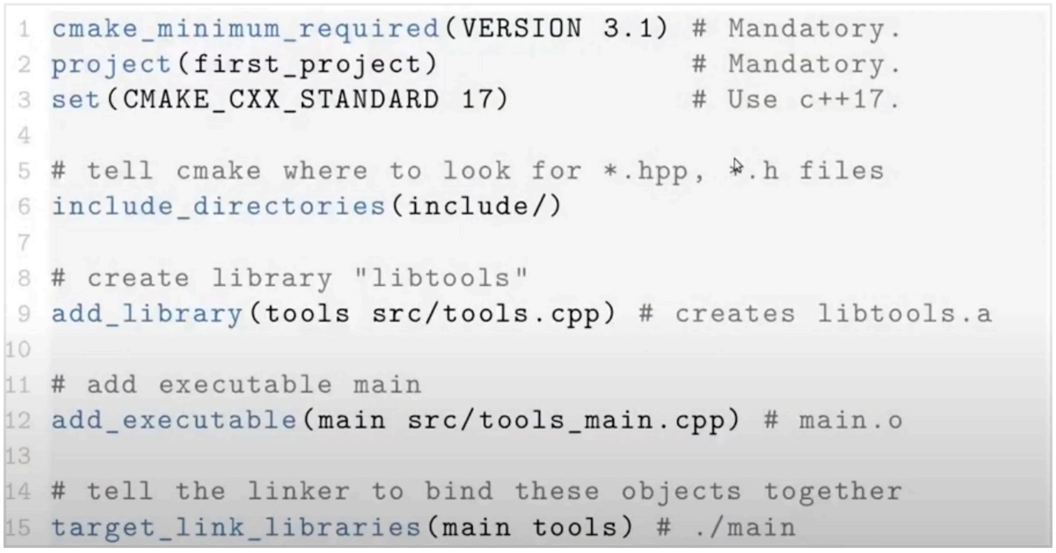
```
1. c++ -std=c++17 -c tools.cpp -o tools.o
2. ar rcs libtools.a tools.o <other_modules>
3. c++ -std=c++17 main.cpp -L . -ltools
```

For a script in the form of:

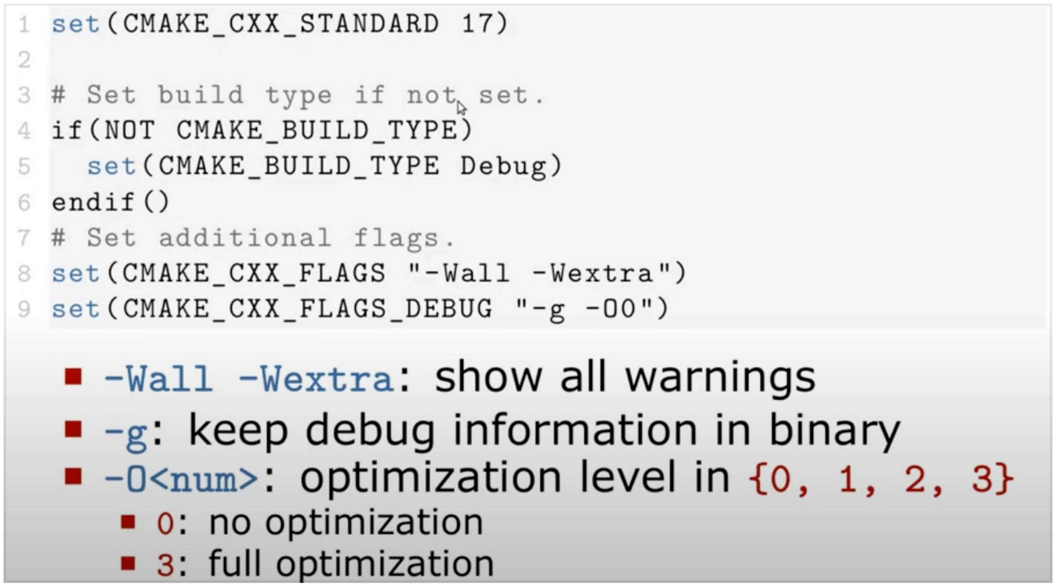
```
add_library(tools tools.cpp)
add_executable(main main.cpp)
target_link_libraries(main tools)
```

Build a Cmake project

- Build process from the user's perspective:
  - cd <project folder>
  - mkdir build
  - cd build
  - cmake ..
  - 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
```



```
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 -O0")
```

- `-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