Robotic Mapping & Localization

Kaveh Fathian
Assistant Professor
Computer Science Department
Colorado School of Mines

Lab 02: C++ & CMake

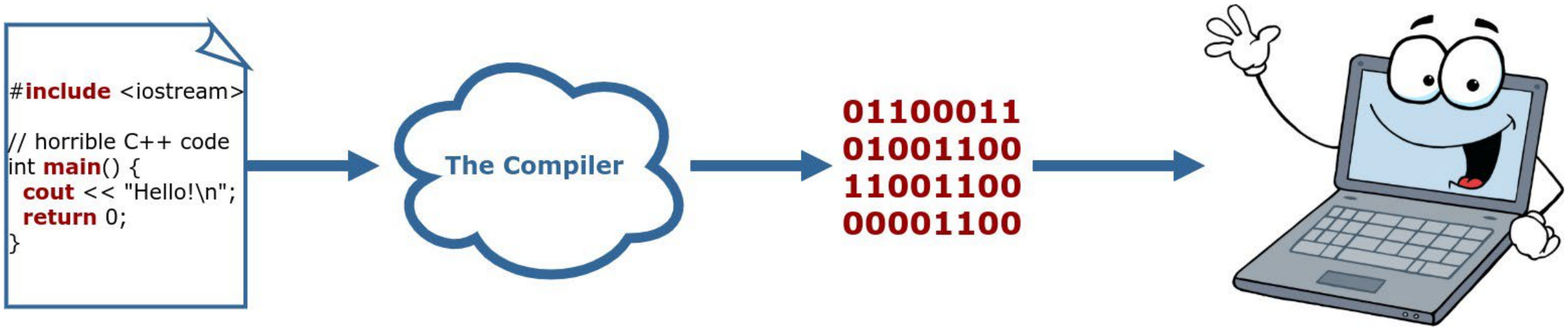
*Courtesy of Ignacio Vizzo, Igor Bogoslavskyi, Cyrill Stachniss

Lecture Outline

- **Introduction to C++**
  - Review
  - Compiler
  - CMake

The compilation process

What is a compiler?



- A compiler is basically a program!
- Is in charge of transforming source code into binary code.
- Binary code (**0100010001**) is the language that a computer can understand.

Compilation made easy

The easiest compile command possible:

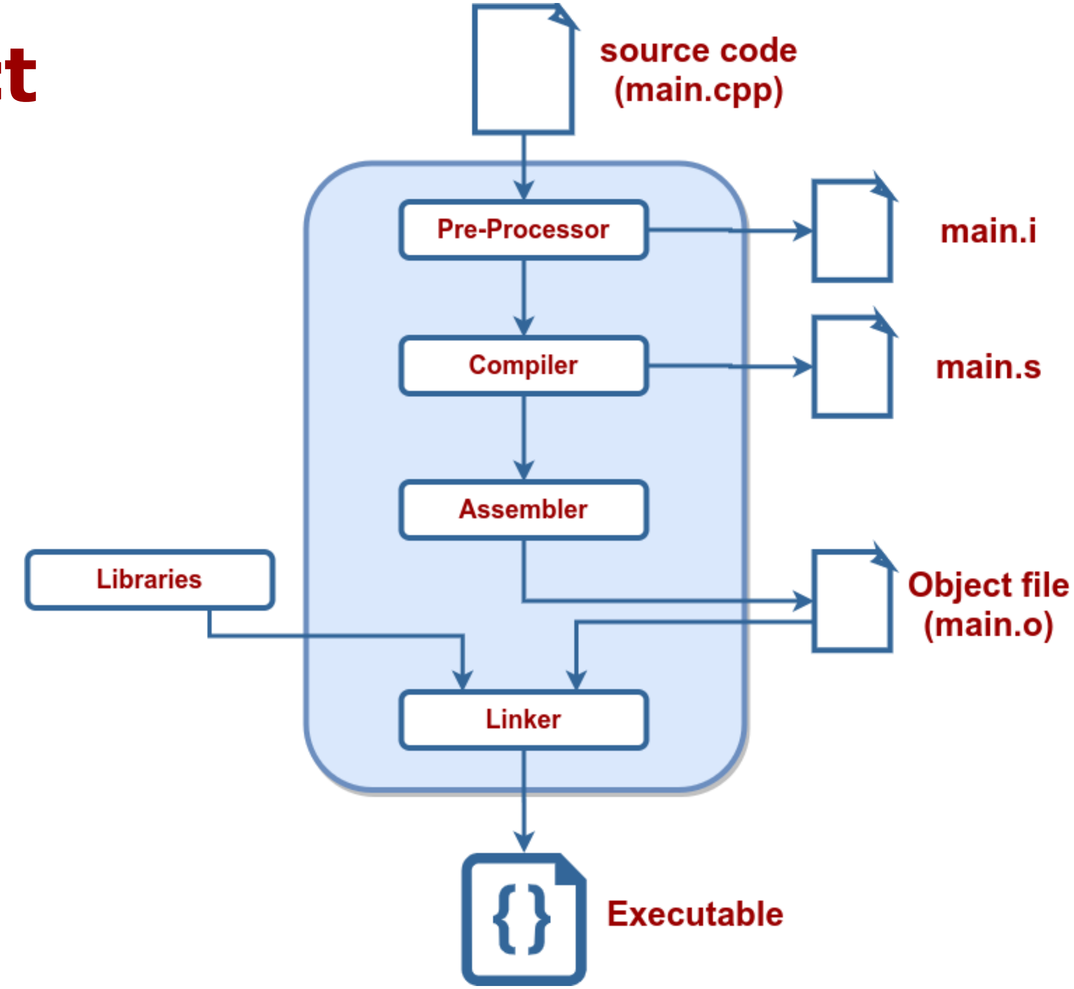
- **g++ main.cpp**
- This will build a program called **a.out** that it's ready to run.

- **g++ -o hello main.cpp**
- This will name the output **hello.out**

The Compiler: Behind the scenes

The compiler performs 4 distinct actions to build your code:

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



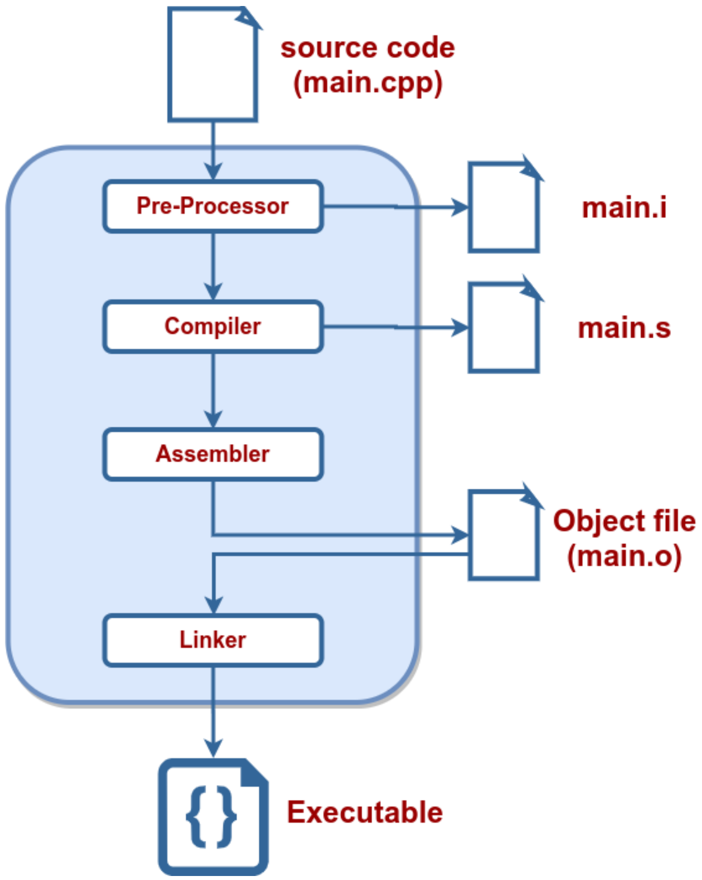
Compiling step-by-step

1. Pre-Preprocessing:

- g++ -E main.cpp > main.i

- **Role:** Performs tasks such as including header files, macro expansion, and conditional compilation.

- **Example:** #include statements, macros, & conditional compilation directives like #ifdef and #define are evaluated.

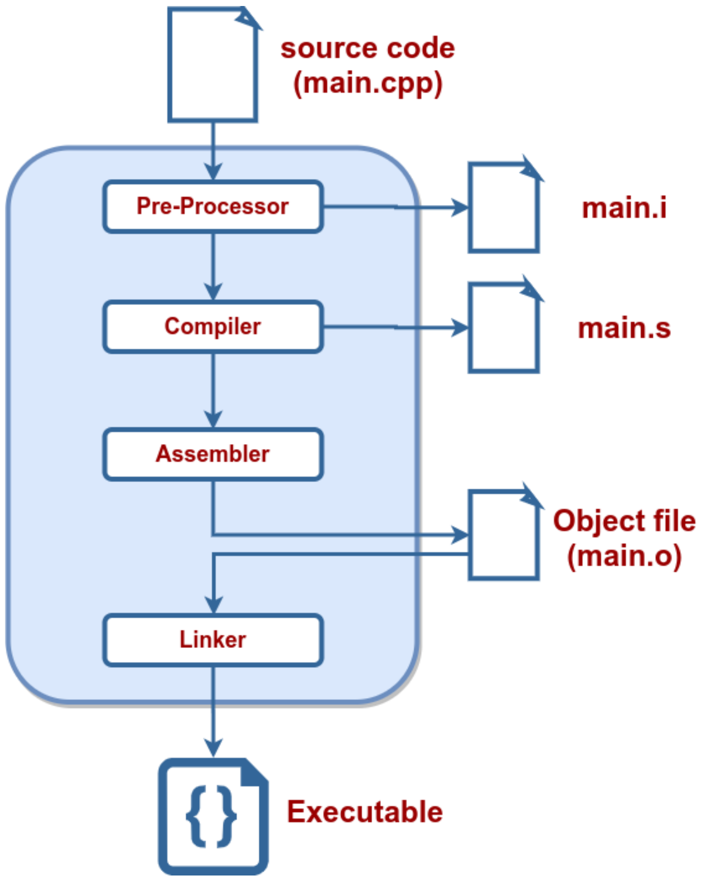


Compiling step-by-step

2. Compilation:

- g++ -S main.i

- **Role:** The compilation stage takes the pre-processed source code & translates it into assembly code. It involves syntax checking, semantic analysis, and the generation of intermediate code.

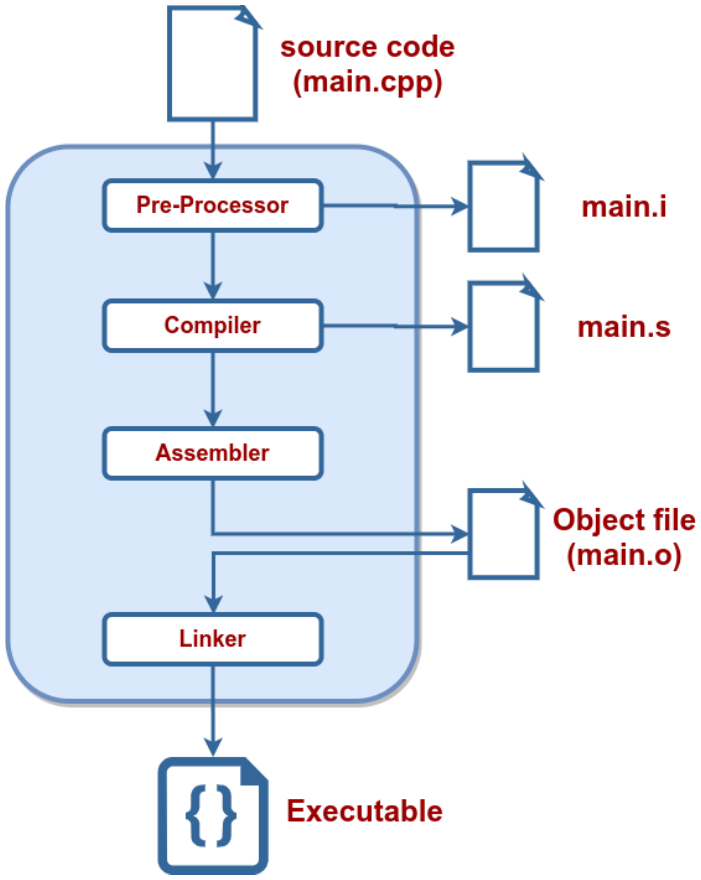


Compiling step-by-step

3. Assembly:

- g++ -c main.s

- **Role:** The assembler converts the assembly code generated in the compilation stage into machine code (binary code) specific to the target architecture. The binary code can be understood & executed by the computer's CPU.

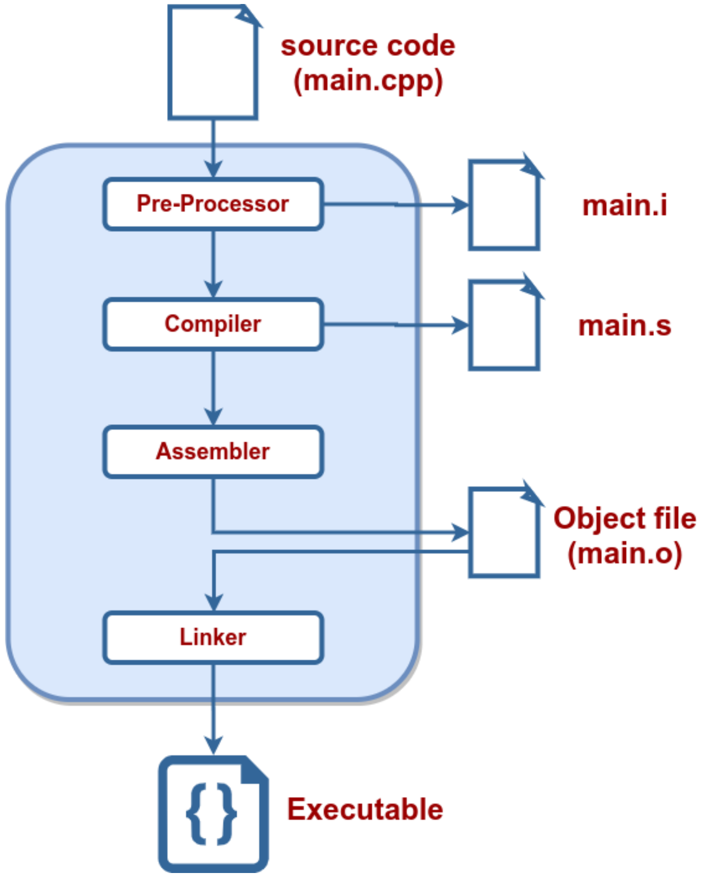


Compiling step-by-step

4. Linking:

- `g++ main.o -o main`

- **Role:** The linking stage combines multiple object files (resulting from assembly stages) & resolves references between them. It creates a single executable file.
- If program uses functions from external libraries or other source files, linking resolves these references, ensuring that the final executable has all the necessary components.

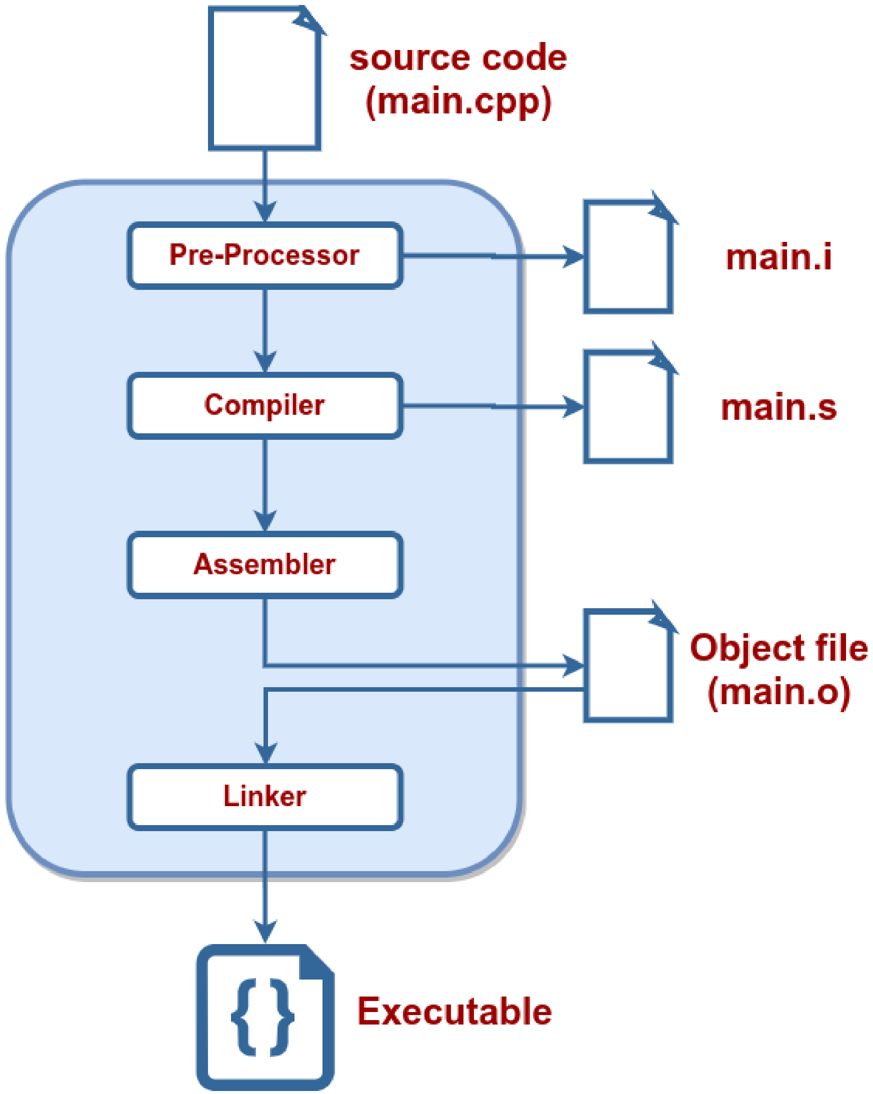


Compiling recap

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

All steps above:

- g++ main.cpp



Compilation flags

- There are lots of flags that can be passed while compiling the code
- We have seen some already:
- **-o**, **-E**, **-S**, **-c**, etc.

Other useful options:

- Enable all warnings, treat them as errors:
  - **-Wall**, **-Wextra**, **-Werror**
- Optimization options:
  - **-O0** — no optimization **[default]**
  - **-O3** or **-Ofast** — full optimizations
- Keep debugging symbols: **-g**
- C++ standard used when compiling: **-std=c++17**

Libraries

What is a Library?

- A C++ library is a collection of **pre-compiled** functions, classes, & procedures

- Libraries provide **reusable** code that can be included in programs, saving time by avoiding the need to write programs from scratch

- C++ libraries are collections of object files (**.o**) that are logically connected

Libraries

Types of libraries:

- **Static:** faster, takes a lot of space, becomes 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 & definition

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

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

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

```cpp
void FuncName(int param) {
  // Implementation details
  std::cout << "This function is called FuncName! ";
  std::cout << "Did you expect anything useful from it?";
}
```

Header/Source Separation

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

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

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

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

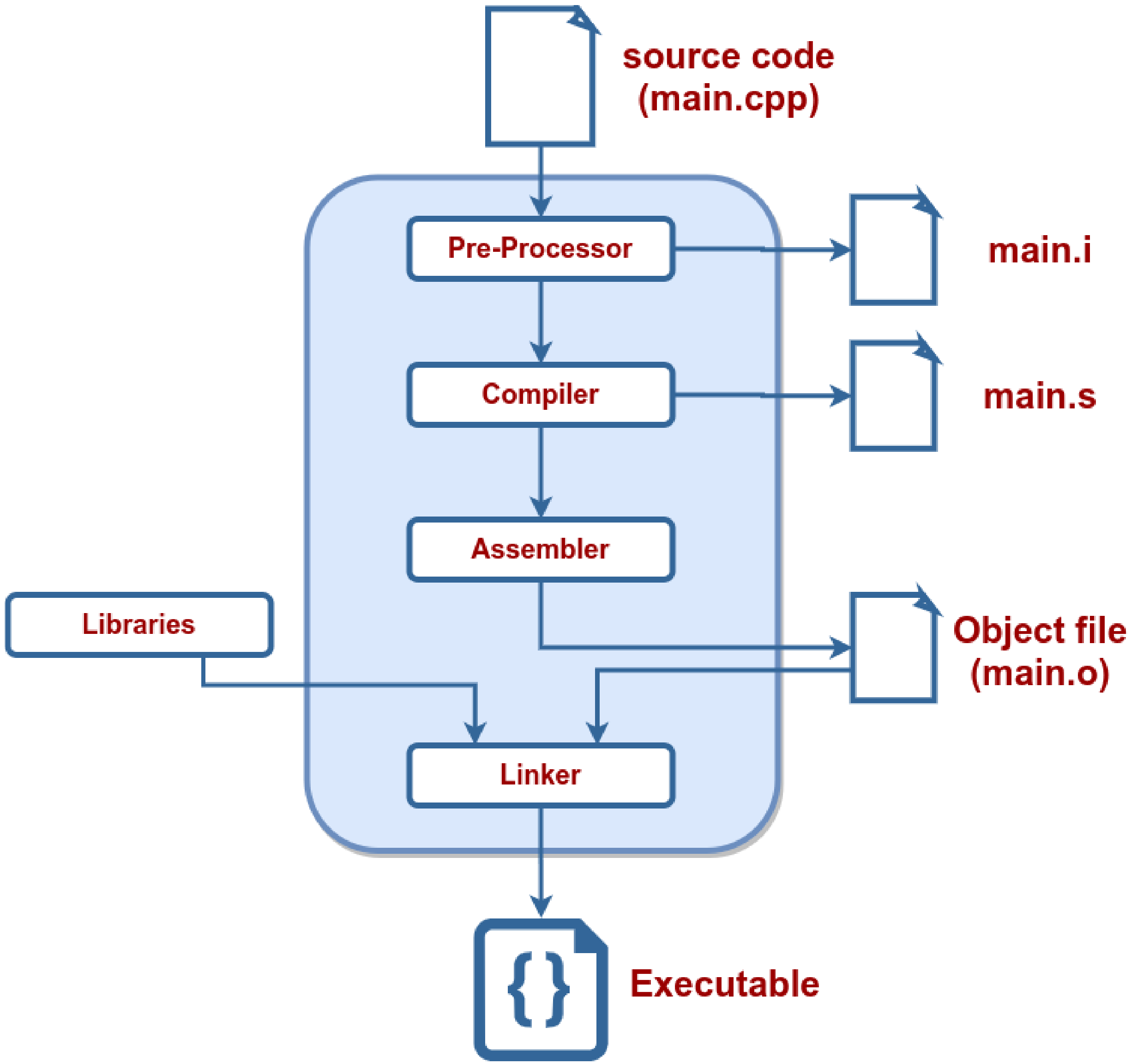
Just build it as before?

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

Error:

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

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.h**
  2. The compiled library object **libmylibrary.a**

How to build libraries?

Short: we separate the code into modules

```
folder/
    --- tools.h
    --- tools.cpp
    --- main.cpp
```

Declaration: tools.h

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

How to build libraries?

Definition: tools.cpp

```cpp
#include "tools.h"
#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.h"
int main() {
  MakeItRain();
  MakeItSunny();
  return 0;
}
```

Use modules and libraries!

Compile modules:

```
g++ -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:

```
g++ -std=c++17 main.cpp -L . -l tools -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 scale on big projects?
- Impossible to maintain!
- Build systems to the rescue!

What are build systems?

- They are tools!
- Many of them.
- Automate the build process of projects.
- They began as **shell** scripts
- Then turned into **MakeFiles**.
- And now into MetaBuild Sytems like **CMake**.
    - Technically, **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. g++ -std=c++17 -c tools.cpp -o tools.o
2. ar rcs libtools.a tools.o <other_modules>
3. g++ -std=c++17 main.cpp -L . -l tools

For a script in the form of:

```
add_library(tools tools.cpp)
add_executable(main main.cpp)
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 (Make)
- Cross-platform
- Very powerful, still build receipt is readable

First CMakeLists.txt

```
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
```

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 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**

Standard Project Structure

```
|-- project_name/
|   |-- CMakeLists.txt
|   |-- build/   # All generated build files
|   |-- results/ # Executable artifacts
|   |   |-- bin/
|   |       |-- tools_demo
|   |   |-- lib/
|   |       |-- libtools.a
|   |-- include/ # API of the project
|   |   |-- project_name
|   |       |-- library_api.h
|   |-- src/
|   |   |-- CMakeLists.txt
|   |   |-- project_name
|   |       |-- CMakeLists.txt
|   |       |-- tools.h
|   |       |-- tools.cpp
|   |       |-- tools_demo.cpp
|   |-- tests/  # Tests for your code
|   |   |-- test_tools.cpp
|   |   |-- CMakeLists.txt
|   |-- README.md # How to use your code
```

Compilation options in CMake

```
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

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:

```
find_library(TOOLS
             NAMES tools
             PATHS ${LIBRARY_OUTPUT_PATH})
# Use it for linking:
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:**

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

- **Libraries:**

```
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})
```

cmake_modules/Findsome_pkg.cmake

```
# Find the headers that we will need
find_path(some_pkg_INCLUDE_DIRS include/some_lib.h
      <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}")
```

References

- Modern C++ for Computer Vision: https://www.ipb.uni-bonn.de/modern-cpp/index.html
- Google Code Styleguide: https://google.github.io/styleguide/cppguide.html
- CMake Documentation: https://cmake.org/cmake/help/v3.28/
- GCC Manual: https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/