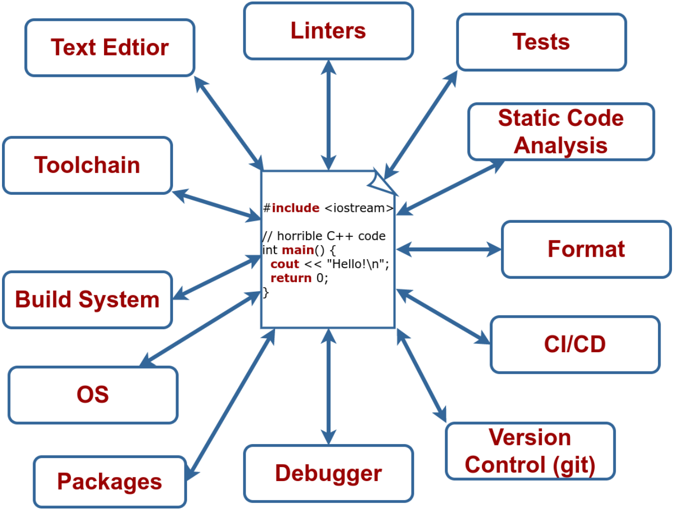
# Modern C++ for Computer Vision Lecture 1: Build Systems

**Ignacio Vizzo, Rodrigo Marcuzzi, Cyrill Stachniss**

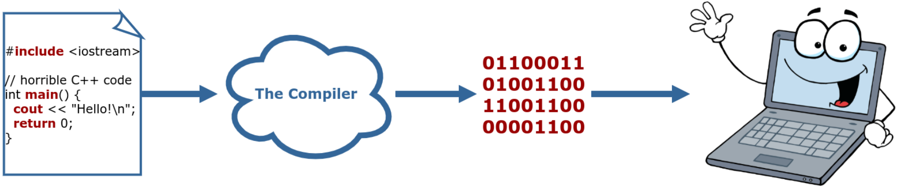
# SW dev ecosystem



# What is a compiler?

- A compiler is basically… a 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:**

- `c++ main.cpp`
- This will build a program called `a.out` that it’s ready to run.

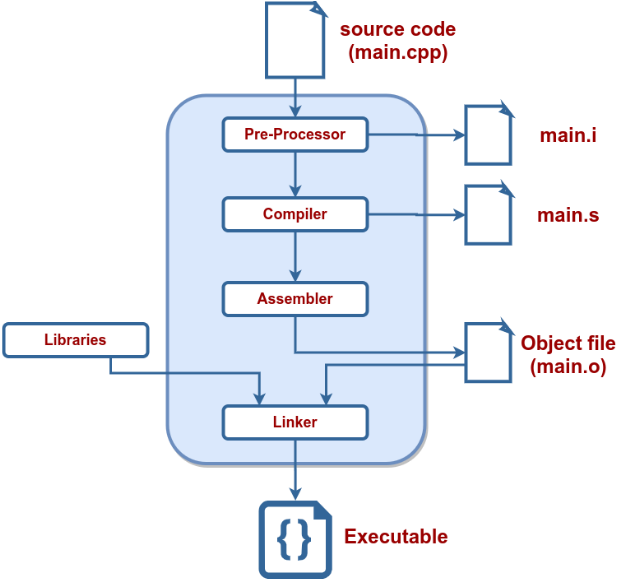
**Will be always this easy?**

# The Compiler: Behind the scenes

**The compiler performs 4 distinct actions to build your code:**

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

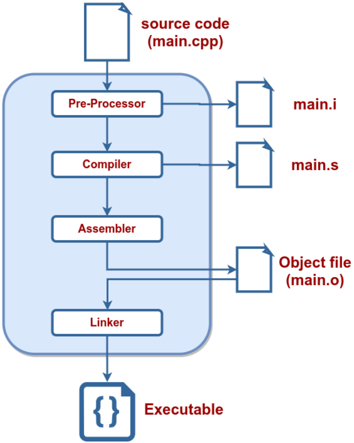
# The Compiler: Behind the scenes



# Compiling step-by-step

## 1. Preprocess:

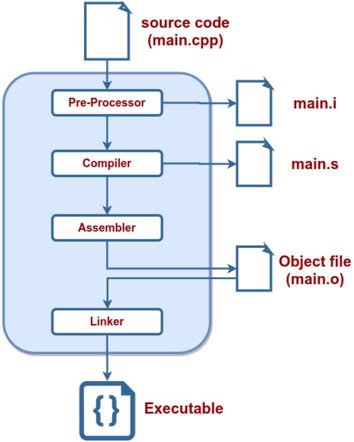
- `c++ -E main.cpp > main.i`



# Compiling step-by-step

## 2. Compilation:

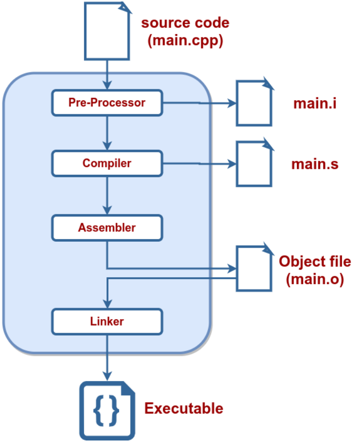
- `c++ -S main.i`



# Compiling step-by-step

## 3. Assembly:

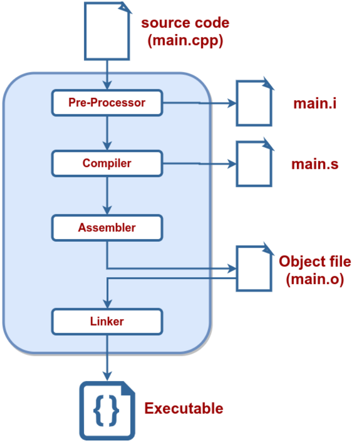
- `c++ -c main.s`



# Compiling step-by-step

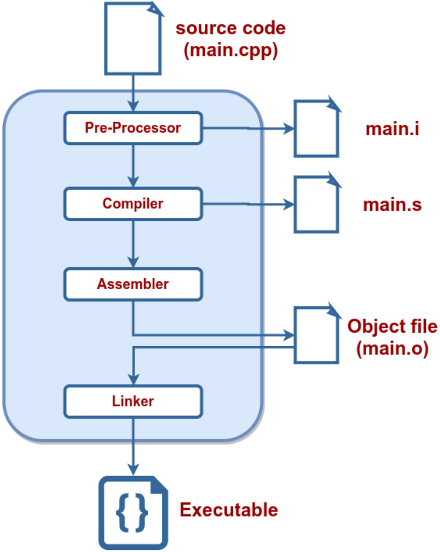
## 4. Linking:

- `c++ main.o -o main`



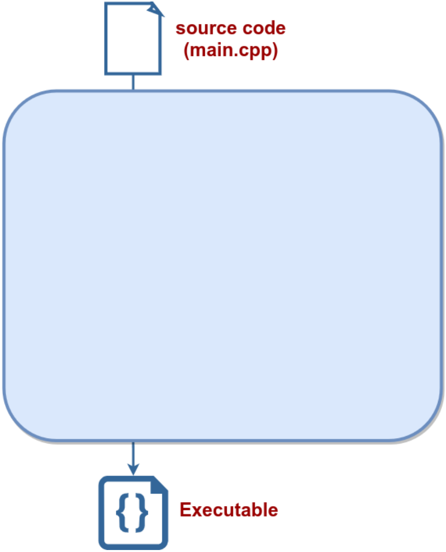
# Compiling recap

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



# Compiling recap

1. `c++ 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:
  - `-O0` — no optimizations **[default]**
  - `-O3` or `-Ofast` — full optimizations
- Keep debugging symbols: `-g`

Play with them with Compiler Explorer: https://godbolt.org/

# 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

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

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

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

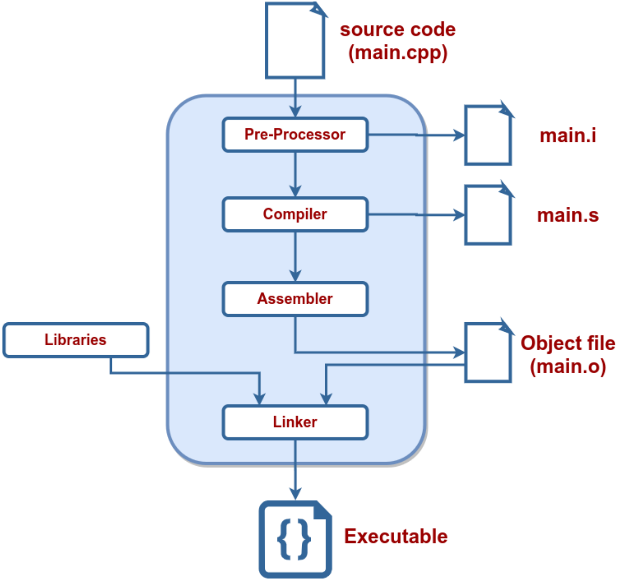
# Just build it as before?

`c++ -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_api.h`
  2. The compiled library object `libmylibrary.a`

# How to build libraries?

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

**Short:** we separate the code into modules
**Declaration:** `tools.hpp`

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

# How to build libraries?

## Definition: `tools.cpp`

```cpp
#include "tools.hpp"

#include <iostream>
void Greet() { std::cout << "Hello There!\n"; }
```

## Calling: `main.cpp`

```cpp
#include "tools.hpp"
int main() {
  Greet();
  return 0;
}
```

# Use modules and libraries!

## Compile modules:

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

## Organize modules into libraries:

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

## Compile main appliaction:

```
c++ -std=c++17 -c main.cpp
```

## Link main application to libraries:

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

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

## For a script in the form of:

```
add_library(tools tools.cpp)          # Steps 1 and 2
add_executable(main main.cpp)         # Step 3
target_link_libraries(main tools) # Step 4
```

# Use CMake to simplify the build

- One of the most popular build tools
- Does not build the code, generates 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 children `src/CMakeLists.txt`, etc.

# 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
```

# Typical 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.hpp
|    |-- src/
|    |    |-- CMakeLists.txt
|    |    |-- project_name
|    |        |-- CMakeLists.txt
|    |        |-- tools.hpp
|    |        |-- 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

```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

# CMake language

- 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/`

# `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.

# Watch for Homeworks


https://youtu.be/hwP7WQkmECE

# 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