Compilation

Raul P. Pelaez

October 31, 2024

Contents

1 Introduction

In this session we will learn about the compilation process in C++. We will learn about the different steps that the compiler takes to transform our source code into an executable file. We will also learn about the different types of files that are generated in the process and how they are used.

2 The compilation process

Thus far you have been using the compiler directly to create an executable file like this:

Source code

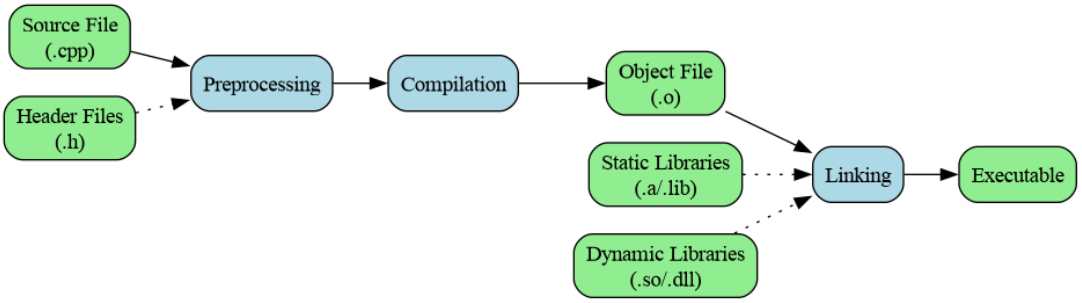
```
g++ -o myprog myprog.cpp
```

Or using clang++ instead. You also know how to pass flags to the compiler to enable warnings (-Wall) or to enable a particular standard (-std=c++20):

Source code

```
g++ -Wall -std=c++20 -o myprog myprog.cpp
```

Lets go a little deeper into the compilation process. The full process looks like this:



When calling `g++ code.cpp`, the compiler is doing all three blue steps for us in sequence.

In our simple examples thus far our program consists on a single source file that contains the main function and all the other functions it calls. In more complex programs we will have multiple source files, each containing some functions, classes or definitions that are used by other files. Still, g++ will do the correct thing for us if we give it all the files at once:

Source code

```
g++ -o myprog file1.cpp file2.cpp file3.cpp
```

Note that only one of these files should contain the main function.

Let us now do each step manually instead. Let us ignore the preprocessing step, which is highly integrated into the compiler and is not something we usually need to worry about.

Advanced

Preprocessing
Preprocessing is such an automatic step that virtually never have to worry about, but I am describing it here for your enlightenment.

The compiler does not really want to see things like `#include <iostream>`. It needs to see a giant file that contains all the necessary code in one place. The preprocessor takes care of this in addition to things like solving macros (stuff like `#define PI 3.1415`).

To see the preprocessed code, we can use the -E flag:

Source code

```
g++ -E myprog.cpp -o myprog.i
```

Suggestion

Try to compile one of your programs from the last session using this flag and inspect the generated file.
You can count the lines in this file using the UNIX command `wc -l myprog.i`.
You will notice this file has tens of thousands of lines.

2.1 Compilation

The compilation step is where the compiler takes a preprocessed code and translates it into machine code that the CPU can understand, creating a file thats called an "object" file. This file is not yet an

executable, but it contains all the necessary information to create one.

At this stage a main function is not needed, the compiler will just translate any functions, classes, definitions, etc. that are in the file.

To compile a single file into an object we can use the "-c" flag:

Source code

```
g++ -c myprog.cpp -o myprog.o
```

Info

Note that the extension (and actual format) of the object file changes depending on the platform. In Linux and OSX its ".o", in Windows its ".obj".

Advanced

This stage will work whether or not we preprocessed the source file. g++ will detect if its not preprocessed and do it for us. You can also just provide `myprog.i` as input to the compiler and it will work too.

Advanced

If you try to open an object file with a text editor you will see that it is not human readable. It is a binary file that contains the machine code for the functions in the source file.
In order to "read" the code in an object file we can use the objdump command in order to disassemble it:

Source code

```
objdump -D myprog.o
```

A bunch of assembler code will be printed to the screen. This is possible to do because there is a direct translation between machine code and assembler code.
If your object code contains a main function, you will find it inside it. For instance:

Source code

```
$ objdump -D a.out  | grep main
  4010cf:       ff 15 03 2f 00 00            call   *0x2f03(%rip)
  ↪  # 403fd8 <__libc_start_main@GLIBC_2.34>
0000000000401196 <main>:
  4011d5:       74 26                        je     4011fd <main+0x67>
```

Grep is a very powerful UNIX tool for searching for text inside files. In this case we are looking for the word "main" inside the output of objdump.

2.2 Linking

The linking step is where the compiler takes all the object files and links them together into an executable file. This is where the main function is needed, as it is the entry point of the program.

The linker will look for an object containing a main function and then it will pick from the rest of the objects any definition that is needed by it.

We can manually link the object file we created before into an executable like this:

Source code

```
g++ -o myprog my_object1.o my_object2.o my_object3.o
```

g++ takes care of adding some libraries for us already, like the c++ standard library[1], which is located in some default system folder.

2.3 Objects and Libraries

In the previous example we linked object files together to create an executable. Objects are intermediate files and are not meant to be shared with others. If we want to share code with others (or other parts of the system) we can create a shared library instead.
Your system already has hundreds of shared libraries available to you, look for files with the extension ".so" in linux, ".dll" in windows or ".dylib" in OSX.

While the link stage in g++ picks the entities it needs from object files to compose an executable file, it will only store pointers (or links) to shared libraries[2] . This means that the executable file will be much smaller than if it contained all the code itself, but also means that in order for the executable to run, the shared libraries must be available in the system.

You can check which shared libraries an executable need by using the ldd command in linux:

Source code

```
$ ldd myprog # Prints a list of shared libraries needed by myprog
linux-vdso.so.1 (0x00007f26fac8c000)
libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f26fa800000)
libm.so.6 => /lib64/libm.so.6 (0x00007f26fab7c000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f26fab4e000)
libc.so.6 => /lib64/libc.so.6 (0x00007f26fa60f000)
/lib64/ld-linux-x86-64.so.2 (0x00007f26fac8e000)
```

Advanced

There are also static libraries, that are embedded into the executable at compile time instead of just linked against. These are files with the extension ".a" in linux and ".lib" in windows. These are not commonly used.

You can create a shared library containing the functions in your library like this:

Source code

```
g++ -shared -o libmy_functions.so my_functions.cpp
```

And then link against it like this:

[1] In linux, this is called libstdc++.so, in windows its called libstdc++.dll, in OSX its called libstdc++.dylib
[2] Think about it like storing inside the executable "when you need function foo, look for it inside file libwhatever.so

Source code

```
g++ -o myprog myprog.cpp -lmy_functions
```

Note the naming convention: the library is called `libmy_functions.so`, but we only use `-lmy_functions` to link against it. The compiler will automatically add the "lib" prefix and ".so" suffix.

Advice

Why would you want to compile in multiple steps?
Let me give you some reasons:

- Compilation can be really slow, so when developing you might want to compile only the files that changed.
- Compilation is hard and requires build-time dependencies and a compilation environment, you might want to share an already compiled piece of code instead of a bunch of source code files.
- You might want to share a library with others, but not the source code.

Apart from that, you need to know about it because it is a widespread practice. For instance, when you install software in your computer, you are installing precompiled binaries (shared libraries), not source code.

2.4 Key new concepts

Info

The compilation pipeline
The compilation process is divided into three steps: preprocessing, compilation and linking. The compiler does all three steps for us when we call it with `g++ myfile1.cpp myfile2.cpp`.

We can also do each step manually by using the -E (preprocess), -c (compile) flags. And linking when simply providing g++ with a bunch of object files.

Info

Object files
Object files are intermediate files that contain the machine code for the functions in the source file. They are not meant to be shared with others.

Info

Shared libraries
Shared libraries are files that contain code that can be shared between different programs. They are not embedded into the executable, but are linked against it at runtime. Every program in your system needs and looks for shared libraries in some default system folder.
We can compile our code into a shared library using the -shared flag and link against it using the -l flag.

3 Compilation scripts

When your project grows it quickly becomes unfeasible to compile each file manually. Furthermore, the specific commands that will work for your system will probably not work for others. For this reason, it is common to write a script that will compile your project for you.

One of the most popular tools to do this is CMake. One describes the characteristics of the project in a file called CMakeLists.txt and then CMake will generate the necessary build files for your system. The typical workflow when encountering a CMake-enabled project is to run the following commands:

Source code

```
mkdir build
cd build
cmake ..
cmake --build .
```

This will create a directory called build, enter it, run cmake to generate the build files and then run the build system to compile the project. Building will typically make some executable available under build/bin or somewhere around.
It is also usual to "install" the project, which will copy the executable to some system folder where it can be run from anywhere. This is done by running the following command:

Source code

```
cmake --install .
```

Now any executable, shared library and/or header file that was installed can be used by any other project in the system as if it was a system library.

Advice

It is almost never a good idea to install your project in the system folders. Typically one works under a virtual environment (such as conda or a container) and installs the project there. CMake can be configured to install the project in a different folder:

Source code

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/install ..
```

If you are using conda, the prefix would be `$CONDA_PREFIX`.

4 Exercises

4.1 Multi-file manual compilation

Goal

Write a program that is defined in two separated components:

- `main.cpp` containing the main function and includes the header file `my_library.h`.
- `my_library.h` and `my_library.cpp` containing a function that is called from `main.cpp`.

Milestone

Create the files `main.cpp`, `my_library.h` and `my_library.cpp`. Place all three in the same directory.

Source code

```cpp
// main.cpp
#include "my_library.h"
int main() {
  my_function();
  return 0;
}
```

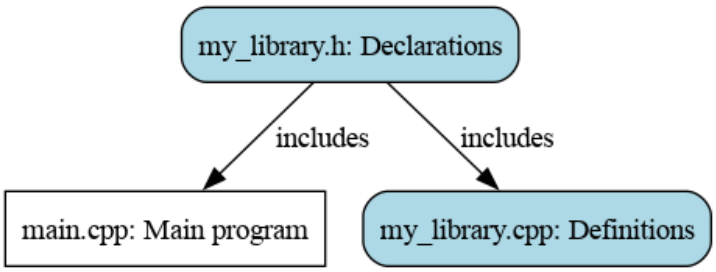
Source code

```cpp
// my_library.h
#pragma once
void my_function();
```

Source code

```cpp
// my_library.cpp
#include <iostream>
#include "my_library.h"
void my_function() {
  std::cout << "Hello world!" << std::endl;
}
```

Note how we separated the declaration of the function in the header file and the definition in the source file. The header is then included in both the main program and the library definitions.



Milestone

Compile the library and the main program separately. The library should be compiled into an object file and the main program into an executable.

Hint

Use the following commands:

- `g++ -c my_library.cpp -o my_library.o`
- `g++ -c main.cpp -o main.o`
  `g++ main.o my_library.o -o main`

4.2 CMake compilation

Goal

Reproduce the contents of this repository: `https://github.com/RaulPPelaez/cmake-compilation`

Milestone

Create the basic file structure of the project by placing all the sources in the src directory.
Make sure you are able to compile and run the project.
Do **not** clone the repository and copy-paste, I made it really short so that you can write it yourself file by file.

Milestone

Move this weeks homework here so that they can be compiled with CMake. Make sure you can build it and all the executable files are generated.
Put the homework executables in a new directory under the build directory.

Hint

- Move the sources to a new directory called homework.
- Create a CMakeLists.txt file in the homework directory.
- Add a new executable target for each source file.
- You can set the output directory for the executables using the RUNTIME_OUTPUT_DIRECTORY property of the target. Like this:

Source code

```
set_target_properties(executable_1 PROPERTIES
↪  RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/homework)
```

Advanced milestone

There is an alternative to manually adding an executable target for each source file. You can use the glob functionality combined with a foreach loop to do so.

Hint

- Check the documentation for the file(GLOB) command here.
- Check the documentation for the foreach command here.
- Use this function to extract the filename without the extension: get_filename_component.