# CMake and Make



In the previous notebook, you saw how example code could be split into multiple `.h` and `.cpp` files, and you used `g++` to build all of the files together. For small projects with a handful of files, this works well. But what would happen if there were hundreds, or even thousands, of files in the project? You could type the names of the files at the command line each time, but there tools to make this easier.

Many larger C++ projects use a build system (https://en.wikipedia.org/wiki/List_of_build_automation_software) to manage all the files during the build process. The build system allows for large projects to be compiled with a few commands, and build systems are able to do this in an efficient way by only recompiling files that have been changed.

In this workspace you will learn about

- Object files: what actually happens when you run `g++`.
- How to use object files to compile only a single file at a time. If you have many files in a project, this will allow you can compile only files that have changed and need to be re-compiled.
- How to use `cmake` (and `make`), a build system which is popular in large C++ projects (https://cmake.org/success/).CMake will simplify the process of building project and re-compiling only the changed files.

Click to go to the next page to see how this works!

---

## Object Files

When you compile a project with `g++`, `g++` actually performs several distinct tasks:

1. The preprocessor runs and executes any statement beginning with a hash symbol: `#`, such as `#include` statements. This ensures all code is in the correct location and ready to compile.
2. Each file in the source code is compiled into an "object file" (a `.o` file). Object files are platform-specific machine code that will be used to create an executable.
3. The object files are "linked" together to make a single executable. In the examples you have seen so far, this executable is `a.out`, but you can specify whatever name you want.

It is possible to have `g++` perform each of the steps separately by using the `-c` flag. For example,

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

will produce a `main.o` file, and that file can be converted to an executable with

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

---

## Your Turn

Try to compile `main.cpp` to an object file using the commands from the previous page. In the terminal to the right:

1. Compile the `main.cpp` file to an object file using the `-c` flag. You can list the files in the directory with `ls`.
   After compiling, you should see a `main.o` in the directory (along with other notebook files).
2. Convert the file to an executable with `g++`.
3. Run the executable with `./a.out`.

---

## Compiling One File of Many, Step 1

In the previous example, you compiled a single source code file to an object file, and that object file was then converted into an executable.

Now you are going to try the same process with multiple files. Navigate to the `multiple_files_example` directory in the terminal to the right. This directory should have the `increment_and_sum` and `vect_add_one` files from a previous Notebook. Try compiling with the commands below:

```
root@abc123defg:/home/workspace/multiple_files_example# g++ -c *.cpp
root@abc123defg:/home/workspace/multiple_files_example# g++ *.o
root@abc123defg:/home/workspace/multiple_files_example# ./a.out
```

Here, the `*` operator is a wildcard, so any matching file is selected. If you compile and run these files together, the executable should print:

```
The total is: 14
```

---

## Compiling One File of Many, Step 2

But what if you make changes to your code and you need to re-compile? In that case, you can compile only the file that you changed, and you can use the existing object files from the unchanged source files for linking.

Try changing the code in `/multiple_files_example/main.cpp` to have different numbers in the vector and save the file with `CTRL-s`.

When you have done that, re-compile just `main.cpp` by running:

```
root@abc123defg:/home/workspace/multiple_files_example# g++ -c main.cpp
root@abc123defg:/home/workspace/multiple_files_example# g++ *.o
root@abc123defg:/home/workspace/multiple_files_example# ./a.out
```

Compiling just the file you have changed saves time if there are many files and compilation takes a long time. However, the process above is tedious when using many files, especially if you don't remember which ones you have modified.

For larger projects, it is helpful to use a build system which can compile exactly the right files for you and take care of linking.

On the next page, we'll introduce a cross-platform build system that you'll be using in several of the projects in this Nanodegree program.

---

## CMake and Make

CMake is an open-source, platform-independent build system. CMake uses text documents, denoted as `CMakeLists.txt` files, to manage build environments, like make (https://en.wikipedia.org/wiki/Make_(software)). A comprehensive tutorial on CMake would require an entire course, but you can learn the basics of CMake here, so you'll be ready to use it in the upcoming projects.

### CMakeLists.txt

`CMakeList.txt` files are simple text configuration files that tell CMake how to build your project. There can be multiple `CMakeLists.txt` files in a project. In fact, one `CMakeList.txt` file can be included in each directory of the project, indicating how the files in that directory should be built.

These files can be used to specify the locations of necessary packages, set build flags and environment variables, specify build target names and locations, and other actions.

In the next few pages of this workspace, you are going to create a basic `CMakeLists.txt` file to build a small project.

If you have trouble with any of these steps, see the file `SolutionCMakeLists.txt` in the tab on the right.

---

## CMake Step 1

In the terminal to the right, navigate to the `cmake_example` folder. This folder should contain a simple CMake project, including:

- An empty `CMakeLists.txt` file
- A `src` directory with the `increment_and_sum` and `vect_add_one` files from a previous Notebook

The `CMakeLists.txt` file should be open in the tabs on the right, along with the files from the `src` directory. You will write a basic `CMakeLists.txt` file to build all of these project files into an executable.

The first lines that you'll want in your `CMakeLists.txt` are lines that specifies the minimum versions of cmake and C++ required to build the project. Add the following lines to your `CMakeLists.txt` and save the file:

```
cmake_minimum_required(VERSION 3.5.1)

set(CMAKE_CXX_STANDARD 14)
```

These lines set the minimum cmake version required to 3.5.1 and set the environment variable `CMAKE_CXX_STANDARD` so CMake uses C++ 14. On your own computer, if you have a recent `g++` compiler, you could use C++ 17 instead.

---

## CMake Step 2

CMake requires that we name the project, so you should choose a name for the project and then add the following line to `CMakeLists.txt`:

```
project(<your_project_name>)
```

You can choose any name you want, but be sure to change `<your_project_name>` to the actual name of the project!

---

## CMake Step 3

Next, we want to add an executable to this project. You can do that with the `add_executable` command by specifying the executable name, along with the locations of all the source files that you will need. CMake has the ability to automatically find source files in a directory, but for now, you can just specify each file needed:

```
add_executable(your_executable_name  path_to_file_1  path_to_file_2 ...)
```

Hint: The source files you need are the *three* .cpp files in the `src/` directory. You can specify the path relative to the `CMakeLists.txt` file, so `src/main.cpp` would work, for example.

---

## CMake Step 4

A typical CMake project will have a `build` directory in the same place as the top-level `CMakeLists.txt`. Make a `build` directory in the `/home/workspace/cmake_example` folder:

```
root@abc123defg:/home/workspace/cmake_example# mkdir build
root@abc123defg:/home/workspace/cmake_example# cd build
```

From within the build directory, you can now run CMake as follows:

```
root@abc123defg:/home/workspace/cmake_example/build# cmake ..
root@abc123defg:/home/workspace/cmake_example/build# make
```

The first line directs the `cmake` command at the top-level `CMakeLists.txt` file with `..`. This command uses the `CMakeLists.txt` to configure the project and create a `Makefile` (https://en.wikipedia.org/wiki/Makefile) in the `build` directory.

In the second line, `make` finds the `Makefile` and uses the instructions in the `Makefile` to build the project.

If CMake and Make are successful, you should see something like the following output in the terminal:



---

## CMake Step 5

If everything has worked correctly, you should now be able to run your executable from the build folder:

```
root@abc123defg:/home/workspace/cmake_example/build# ./your_executable_name
```

This executable should print:

```
The total is: 14
```

just as it did in the previous workspace.

If you don't remember the name of your executable, the last line of the `make` output should tell you:

```
[100%] Built target <your_executable_name>
```

---

## CMake Step 6

Now that your project builds correctly, try modifying one of the files. When you are ready to run the project again, you'll only need to run the `make` command from the build folder, and only that file will be compiled again. Try it now!

In general, CMake only needs to be run once for a project, unless you are changing build options (e.g. using different build flags or changing where you store your files).

`Make` will be able to keep track of which files have changed and compile only those that need to be compiled before building.

**Note:** If you do re-run CMake, or if you are having problems with your build, *it can be helpful to delete your build directory and start from scratch. Otherwise, some environment variables may not be reset correctly.*

---

## CMake Review

Excellent work! You've now written a basic `CMakeLists.txt` file that builds a small project for you.

CMake is a build system that uses text files named `CMakeLists.txt` to configure and build your project. Once the `CMakeLists.txt` is written, you only need the `cmake` and `make` commands to build your project again and again, so it is very convenient to use!

Coming up, we will provide the `CMakeLists.txt` for your course project, and as you will see, it creates two executables and links several external libraries, so it will be a bit longer than the one you've just created. However, you should now have a better understanding of the mechanics of CMake, and you are ready to start experimenting with CMake on your own projects.