CMake Tutorial

For practical use

Pierre-Alain Langlois
email : langloispierrealain@gmail.com

December 6, 2018

Contents

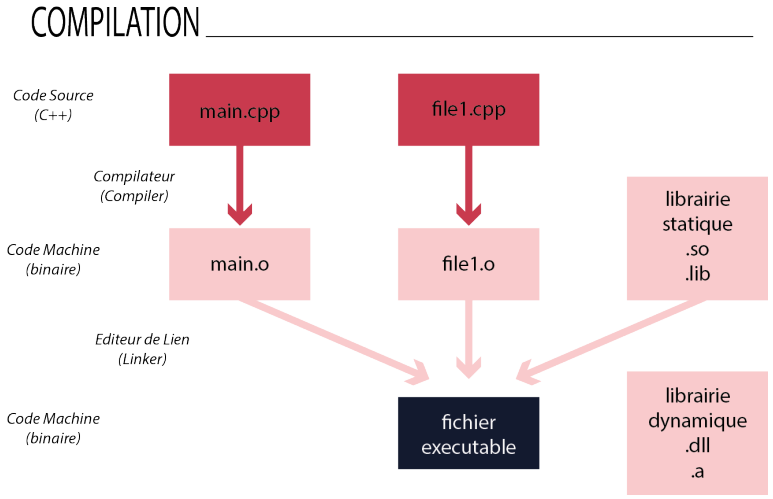

Figure 1: General C++ compilation scheme

1 Compilation in C++

C++ is a general-purpose programming language which provides both object-orientated and low level features. It is a compiled language, and thus understanding the process of machine code generation from C++ source code helps working efficiently on the project.

1.1 The compilation process

As explained on Figure 1, this process is divided into two main steps. The first step is called compiling. It consists in transforming each C++ source code file (.cpp) into a machine code object file (.o). During this step, each "include" statement is replaced by the content of the whole header file it refers to. The second step is called linking. It consists in taking every binary files which are part of the project (object files and external libraries), and performing link edition to obtain an executable file. Link edition means roughly that the calls between each elements of the code (function, class, global variables) are made explicit and embedded in an executable file.

1.2 The compiler softwares

Usually, these two kinds of software (compiler and linker) are embedded into a single tool which we just call a compiler. The most common compilers are GCC on Linux, CLANG on Mac OS, and Visual C++ on Windows. We can also mention MinGW, which is a GCC emulator on Windows. We can directly use these tools thanks to the console and we give them instructions by passing them extra-argument called flags. You can see an example of C++ code compilation with GCC in Listing 1.

Listing 1: Compilation and linkage example with GCC

```
g++ -c first.cpp
g++ -c second.cpp
g++ -c third.cpp
g++ -o myexecutable first.o second.o third.o [other dependencies]
```

The -c flag means "compiling the .cpp source code file to a .o object file".
The -o flag means "Linking the given .o object files into an executable file."

1.3 The integrated development environments

Because it would not be reasonable to directly call the compiler executable in the case of an important project, we use an IDE (integrated development environment) in order to delegate all the compilation steps to high-level tools and thus to reduce the whole process of going from the code to the executable by a single click.

However, IDEs are usually not cross-platform softwares. As a consequence, production motors were created in order to summarize the operations to do on the code and to translate them into a project file dedicated to your favorite IDE. CMake is widely used today because of its relative ease of use. Another advantage of using CMake is that it supports many Integrated Development Environments such as Visual Studio, KDevelop, XCode, Eclipse and many others. CMake also includes tools that are able to locate automatically where the external libraries are installed on the computer, and it manages the dependencies between each library of the program.

2 CMake's philosophy (yes, that much)

If you're willing to have a direct help regarding a project that won't configure/generate, please skip to the next section.
One important thing to keep in mind is that CMake's general goal is to be able to guarantee that the produced projects are cross-platform. This means that we want the same code to be produced as easily, be it on Mac, Linux or Windows platforms for instance. What is more, we want to produce the same compilation steps for different IDEs and different compilers. One has to always remember that these different systems are heterogeneous and even if CMake is a great tool (and if you don't think so, just try to build a cross-platform project without it ;)), you will always need to know how to pimp a little the things in order to have them working.
As a consequence, this tutorial's main purpose is to provide an operational help for the every day use of CMake.

3 CMake commands

In this section, we explain how CMake works in a project by detailing the use of each CMake command.
CMake can automatize the building process of libraries and/or executables. For more

convenience, we use the word *target* in order to arbitrarily make reference to an executable or a library. CMake is a script language whose commands are located in the CMakeLists.txt files. For simple projects, there is usually one single CMakeLists file, but there can be several of these files in a bigger project. Usually, there is a master CMakeLists file located at the root directory of the project. Its role is to find all the external libraries required by the project, to set all compiler/linker flags and to execute the other CMakeLists. Multiple CMakeLists files can be used when there are several targets to build in the project. The role of each CMakeLists file is to build the corresponding target, link the necessary libraries to the target and setting the specific required flags. Finally, there can be one special CMakeLists in the "external" folder which builds the external libraries whose sources have been added as git submodules.

Don't forget that these considerations only have a general value: you can imagine many ways to manage your building system.

In CMake everything starts with the

```
cmake_minimum_required()
```

command. With this line, we specify the minimum version of CMake which is required, and thus we ensure that every command used is available.
The next command is

```
project()
```

It defines the project, which correspond to the concept of solution in Visual Studio. The argument is a string of your choice which is your project's name.

Like every programming language, CMake makes use of variables. Every variable in CMake is shared among each CMakeLists file. We can set a variable thanks to the *set* command:

```
set(VARIABLE Value)
```

You can use a variable which is not defined; however, it will be considered as an empty string or a 0 integer depending on how you use it. In CMake, there is a difference between VARIABLE, which refers to the variable itself, and ${VARIABLE} which refers to the variable's content. You can print the content of variable VARIABLE thanks to the *message* command :

```
message(My_variable_s_content " ${VARIABLE}")
```

Let's assume that VARIABLE contains the integer "5". Then the previous command prints "My_variable_s_content 5" when CMake is executed.

Finding a library with CMake is possible with the command *find_package*:

```
find_package(<lib>)
```

Which means that CMake has to use a file called *Find<lib>.cmake*. CMake is embedded with many of these files for many popular libraries. Notice that you can make your own Find file, but this procedure is not detailed in this documentation. If you manage to get appropriate Find files for the library that you use, you can put them into a directory of your choice, let's say external/cmake_modules. In order for CMake to take these files into account, add the following command in your CMakeLists file:

```
list(APPEND CMAKE_MODULE_PATH
  "${CMAKE_CURRENT_LIST_DIR}/external/cmake_modules")
```

in the master CMakeLists.

Most of nowadays C++ projects make use of the C++ 11 standard. The standard cross-platform command to enable it in CMake is:

```
set(CMAKE_CXX_STANDARD 11)
```

A lot of compiler and linker flags are specified automatically. However, they can be customized thanks to the following CMake variables:

- CMAKE_CXX_FLAGS
- CMAKE_CXX_FLAGS_DEBUG
- CMAKE_CXX_FLAGS_RELSEASE
- CMAKE_LINKER_FLAGS
- CMAKE_SHARED_LINKER_FLAGS

Here is an example which enables the use of the greater optimization level in GCC :

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3")
```

Another very useful command allows one to gather a set of files into a variable :

```
file(GLOB VAR path)
```

Here is a practical example. If you put the following line in the main CMakeLists,

```
file(GLOB all_src_cpp_files ${CMAKE_CURRENT_SOURCE_DIRECTORY}/*.cpp)
```

then all the .cpp files in the CMakeLists' folder will be gathered in the all_src_cpp_files variable.

With all the commands that have been specified before, we are now able to build targets. As a reminder, a target can either be a library are an executable. We can create a library thanks to the following line :

```
add_library(libName TYPE srcFiles)
```

Where libName is the name of the created library ; TYPE is either SHARED are STATIC depending on the type of library you want to make, and srcFiles is a variable gathering all the source files that are part of your library. Notice that it is not required to put the header files in the srcFiles variable.

The corresponding command for making an executable is very similar:

```
add_executable(exeName srcFiles)
```

For each target, you can add specific properties such as specific flags thanks to the following command :

```
set_target_properties(targetName PROPERTIES prop1 val1
                                            prop2 val2
                                            ...   ...
                                            propn valn)
```

More details on this command are available in CMake documentation.

If you need to add specific instructions for specific platforms you can use the following structures :

```
if(WIN32)
  #Instructions...
endif(WIN32)
```

UNIX and APPLE variables are also available.
The command

```
include_directories(include_path)
```

allows you to add a directory where needed headers are located. In the case of an external library added with the Find<lib> command, the corresponding headers should automatically be added, but you can otherwise include_directories by yourself if you have a "header.h not found" error at compile time.
When linking a library to a target, you do not only need to specify the include directories to cmake ; you also need to specify the library's binary file to link. This is done thanks to the following command :

```
target_link_library(targetName library)
```

If the library's binary directory has been specified before with the command

```
link_directories(library_binary_dir)
```

then library is only the name of the library. Otherwise, CMake also allows the complete path to the binary file. In the case of a library built in the current project, just use the name you specified in the add_library command.
If you did add the correct include directories but you failed to link the library's binary, you will get an error at compile time which looks like "symbol void std::func(int,..) not found". If you get this kind of error, you can try to check that the library is correctly linked with the two previous commands.

The last command of this guide allows to execute CMakeLists files located in other directories. The corresponding command is

```
add_subdirectory(subdirectory_path)
```

4 Building a project with CMake

Assuming you just downloaded the source of a fresh CMake project, there is one rule to always remember.

Always execute CMake in an empty directory created specially for this purpose !

I usually create a build folder in the source code's root directory : /path/to/-source_code/build.

4.1 Using CMake from the command line

There are two main ways to build your project. Whether you want to use the command line : go to the build directory with the console and type

cmake ..

which means that CMake must execute the CMakeLists in the parent folder (..) since we are in source_code/build, and generate the files in the current folder (build). In this case you likely are using a UNIX-based system such as Linux or Mac OS. As a consequence CMake will build a Makefile that you can call in order to compile your program by just typing

make

in your build directory.

One important variation of this command is

make VERBOSE=1

which forces CMake to display every compiler command it executes. This is extremely useful in order to check whether include directories/external library paths are correctly set or not.

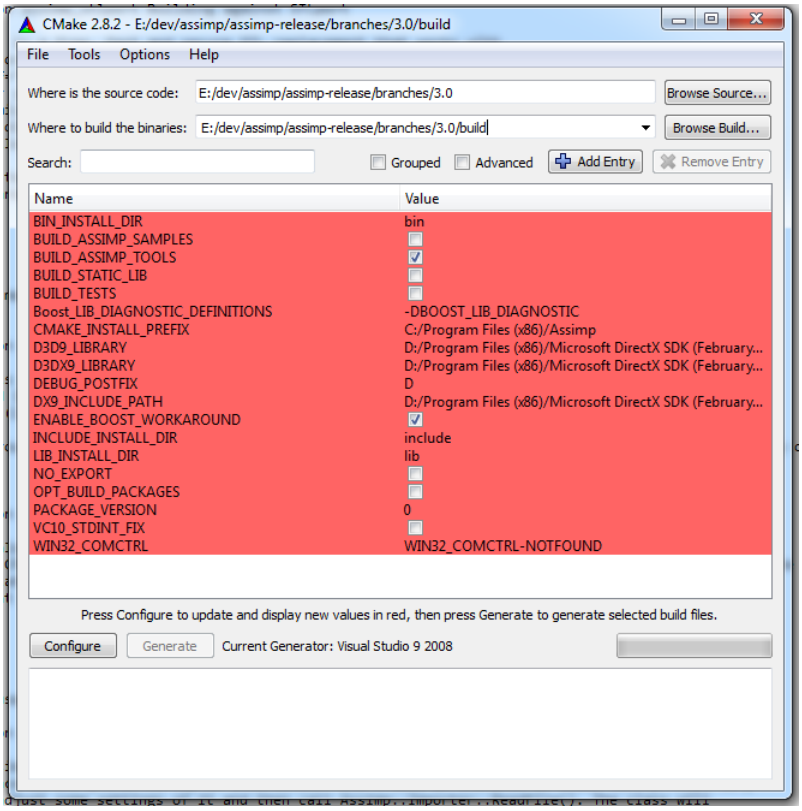

Figure 2: The CMake GUI

If you want to build a project for a particular IDE, just specify it to CMake thanks to the -G flag. The list of supported IDEs is given in CMake Documentation.
For example, if you want to build a project file for Eclipse with Ninja, type :

```
cmake -G"Eclipse CDT4 - Ninja" ..
```

4.2 Using the graphical interface

The other way is to make use of the CMake graphical user interface (see Figure 2) which might be more convenient if you are not used to the console. In this case, be sure to indicate the location of the main CMakeLists in *Where is the source code* and your newly created build directory in *Where to build the binaries*. First click on configure and specify which IDE you wish to use, and if everything went ok, click on generate.

5 Improving this tutorial

Feel free to e-mail me at langloispierrealain@gmail.com if you have any suggestion regarding this tutorial.