

CMake Hacking

sammietocat

2017-10-08

Contents

Foreword	5
About the version of CMake	5
1 Why CMake	7
1.1 The history of CMake	7
1.2 Why Not Others	8
1.2.1 Autoconf	8
1.2.2 JAM, qmake, SCons, or ANT	8
1.2.3 Script It Yourself	8
1.3 Platforms Requirement	8
2 Say Hello	9
2.1 Preparation—CMake Installation	9
2.1.1 3 Options	9
2.1.2 On UNIX and Mac	9
2.1.3 On Windows	9
2.1.4 Building from Source	9
2.2 Basic CMake Syntax	10
2.3 Hello World Example	10
2.3.1 Prepare the CMakeLists File	10
2.3.2 Build the Project	11

Foreword

About the version of CMake

If not stated explicitly, the version of CMake employed for demo will be 3.9.4.

Chapter 1

Why CMake

CMake is an open source build manager for software projects that allows developers to specify build parameters in a simple portable text file format. It can handle several difficult aspects of building software such as

- cross platform builds
- system introspection
- user customized builds

according to a user-friendly script file.

It's a unified build system, which helps to eliminate the need of maintaining platform-specific build systems, such as, the `Makefile` for UNIX and `workspace` for Microsoft Visual Studio.

CMake provides many benefits for single platform multi-machine development environments including:

- Search automatically for programs, libraries, and header files that may be required by the software being built. And also consider environment variables and Windows registry settings when searching.
- Enable building in a directory tree outside of the source tree.
- Extended by complex custom commands for automatically generated new source files during the build process and then are compiled into the software.
- Allow users to select optional components at configuration time.
- Generate workspaces and projects automatically from a simple text file.
- Switch easily between static and shared builds.
- Build up file dependencies automatically and support for parallel builds on most platforms.

When developing cross platform software, CMake provides a number of additional features:

- The ability to check for hardware specific characteristics like machine byte order.
- A single set of build configuration files that work on all platforms.
- Support for building shared libraries on all platforms that support it.
- The ability to configure files with system dependent information, such as the location of data files and other information (e.g., macro definitions by `#define` in C++).

1.1 The history of CMake

CMake development began in 1999 as part of the Insight Toolkit ([ITK](#)) funded by the US National Library of Medicine.

1.2 Why Not Others

1.2.1 Autoconf

Autoconf combined with automake provides some of the same functionality as CMake, but

- Requires the installation of many additional tools not found natively on a Windows box.
- Difficult to use or extend and impossible for some tasks that are easy in CMake.
- Generates `Makefiles` that will force users to the command line.
- Does not support dependent options where one option depends on some other property or selection.

More for UNIX users, CMake provides

- Automated dependency generation that is not done directly by autoconf.
- Simple input format is easier to read and maintain than a combination of `Makefile.in` and `configure.in` files.
- The ability to remember and chain library dependency information has no equivalent in autoconf/automake.

1.2.2 JAM, qmake, SCons, or ANT

Many of these tools require other tools such as Python or Java to be installed before they will work.

1.2.3 Script It Yourself

- Dependencies management is better done by CMake.
- CMake would require no more tools than
 - a C compiler, that compiler's native build tools
 - a CMake executable. CMake was written in C++, requires only a C++ compiler to build and precompiled binaries are available for most systems.
- Self Scripting typically means platform-dependence, limiting its application to building in Mac and Windows.

1.3 Platforms Requirement

Most OSs, including

- Microsoft Windows
- Apple Mac OS X
- Most UNIX or UNIX-like platforms

And most common compilers, such as

- GNU compilers
- Visual Studio

Chapter 2

Say Hello

Just like other programming books, we're going to start by a glimpse of the "Hello World" example of CMake.

2.1 Preparation—CMake Installation

2.1.1 3 Options

- CMake distributions
- Precompiled CMake at www.cmake.org/download/
- Build from source with a modern C++ compiler

2.1.2 On UNIX and Mac

If CMake is provided as one of standard packages in your system, follow your system's package installation instructions.

Otherwise (because of no CMake as standard package or out-of-date CMake), download the precompiled binaries from www.cmake.org/download/. Then extract all files from the compressed tar file downloaded, and place the extracted files into a destination directory (typically `/usr/local`) as you like.

2.1.3 On Windows

Download the Windows' installer or zip of CMake from www.cmake.org/download/, which are given one of following names for a specific version specified by tag `version` (which is evaluated as 3.9.4 throughout this book)

- `cmake-{version}-win64-x64.msi` as an installer to run as an executable
- `cmake-{version}-win64-x64.zip` as a zip archive

For the installer, just click it and follow the prompt to install CMake to somewhere in your Windows machine.

And for the zip archive, unzip it and place the files extracted to somewhere you like. Unlike the installation by means of installer, you need to append the absolute path of the `bin` directory under where you place the CMake folder to the system path.

2.1.4 Building from Source

The CMake source code can be obtained by from www.cmake.org/download/, which are typically named as

- `cmake-{version}.tar.gz`/`cmake-{version}.tar.Z` for Unix/Linux
- `cmake-{version}.zip` for Windows)

The source code can be built in 2 different ways as follows

- If a older version of CMake is available, build the new one with the old one
- Otherwise, CMake may be built by running its bootstrap build script.

1. Change directory into your CMake source directory

2. Execute 3 commands as listed in Listing 2.1

Listing 2.1: Install CMake by the bootstrap script

```
1 ./bootstrap
2 make
3 make install
```

The `make install` step is optional since CMake can run directly from the build directory if desired. On UNIX, if you are not using the GNU C++ compiler, you need to tell the bootstrap script which compiler you want to use. This is done by setting the environment variable `CXX` before running bootstrap. If you need to use any special flags with your compiler, set the `CXXFLAGS` environment variable.

2.2 Basic CMake Syntax

The build process is controlled by creating one or more `CMakeLists` files (the suffix `txt` is omitted for convenience) in each of the directories that make up a project. The `CMakeLists` files should contain the project description in CMake's simple **language**. The language is expressed as a series of **commands**. Each command is evaluated in the order that it appears in the `CMakeLists` file. The commands have the form as Listing 2.2

Listing 2.2: Command format in CMakeLists

```
1 command (args...)
```

where

- `command` is the name of the command, which is case insensitive. That's, `command`, `COMMAND` or `Command` means the same for CMake
- `args` is a white-space separated list of arguments. (Arguments with embedded white-space should be double quoted.)

2.3 Hello World Example

2.3.1 Prepare the CMakeLists File

Bla, bla, ..., it's time for the "Hello World" business.

Suppose we're going to build a `Hello` project written in C++ consisting only a single file `hello.cpp` as Listing 2.3

Listing 2.3: Hello World project in C++

```
1 #include <iostream>
2
3 int main(int argc, char *argv[]) {
4     std::cout << "Hello_World!" << std::endl;
5
6     return 0;
7 }
```

Before the compilation, we need to make up a `CMakeLists` file as Listing 2.4

Listing 2.4: CMakeLists for Hello World in C++

```
1 cmake_minimum_required(VERSION 3.9.4)
2 project (Hello CXX)
3 add_executable (hello hello.cpp)
```

where

- `cmake_minimum_required` command specifies the minimum version of CMake required by the project
- `project` command indicates
 - the name (`Hello`) of the resulting workspace
 - programming languages (`CXX` for C++) supported by the project

- `add_executable` command adds an executable target `hello` to build from the source file `hello.cpp`

With the `CMakeLists` file ready, build of the `hello` executable described in section 2.3.2 to generate the Makefiles or Microsoft project files.

2.3.2 Build the Project

When building a project, two main directories are involved, i.e., **the source directory** and **the binary directory**, where

- The source directory stores the source code for your project, and the `CMakeLists` files
- The binary directory is to store the resulting object files, libraries, and executables.

Typically CMake will not write any files to the source directory, only the binary directory.

Thanks to the separation of the source directory from the binary directory, CMake support 2 kinds of building

- **in-source build**: the source and binary directories are the same
- **out-of-source build**: otherwise

Having the build tree differ from the source tree also makes it easy to support having multiple builds of a single source tree.

Running from the Command Line

From the command line, CMake can be run as an interactive question and answer session (called **the interactive mode**) or as a non-interactive program (called **the non-interactive mode**).

- To run in interactive mode, just pass the `-i` option to CMake. CMake will ask you for some options/values set for the project, and provide reasonable defaults until no more questions is needed
- In non-interactive mode, CMake will run according to some specified setting, without any interaction with users

For starters, we'd to run CMake build our Hello World project in non-interactive mode as follows

1. we'd like to employ the out-of-source build, so we make an empty folder named `build` under current project directory
2. change the current working to `build` directory to where you want the binaries to be placed
3. run `cmake ..`, since the build directory is one level under the source directory
4. then compile the project by `make`

After all 3 steps above, we should get a `hello` executable in current binary directory where we invoke CMake.

That is all there is to installing and running CMake for simple projects. In the following chapters we will consider CMake in more detail and how to use it on more complex software projects.