CMake Hacking

sammietocat

2017-10-08

Contents

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 protable 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-specifc 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 onsider environment variables and Window's 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 fimctionality 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
- Precomipled 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

```
./bootstrap
make
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

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

where

- `command` is the name of the command, which is canse 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++

```
#include <iostream>

int main(int argc, char *argv[]) {
  std::cout << "Hello World!" << std::endl;

  return 0;
}
```

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

Listing 2.4: CMakeLists for Hello World in C++

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

where

- `cmake_minimum_required` command specify 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 of 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.

Chapter 3

Add a Header File

For better organization, a C++ project tends to put its interfaces into separate files. And these separate files usually take form of

- **header files** with .hpp suffix, is where interfaces are declared
- **source files** with .cpp suffix, to specify the detailed implementation of interfaces in the corresponding header files

In such structure, the compiler needs information of how to find those header files required by the project. One conventional way to do so is specifying paths (either relative or absolute) by means of compilation options.

In the world of `cmake`, these options can be defined in the `CMakeLists.txt`. As usual, we're going to explain how it's done by an example. For brevity, our demo will go by adding a header-only interface to `Hello` project from previous chapter.

About header-only interfaces

For a header-only interface, the declaration and implementation of it is put together in one header file, no corresponding source file

3.1 Make up the Hello Project

The file structure of the project is organized as follows

```
|-CMakeLists.txt
|-include
| |-greeter
| | |-about_time.hpp
|-src
| |-hello.cpp
```

3.1.1 C++ source codes

The source codes of the C++ part are respectively shown by listings 3.1 and 3.2.

Listing 3.1: Codes for `include/greeter/about_time.hpp`

```
#ifndef HELLO_ABOUT_TIME_HPP
#define HELLO_ABOUT_TIME_HPP

#include <iostream>

void sayGoodMorningTo(const std::string &who) {
    std::cout << "Good morning, " << who << std::endl;
}

#endif //HELLO_ABOUT_TIME_HPP
```

Listing 3.2: Codes for `src/hello.cpp`

```
#include "greeter/about_time.hpp"

int main(int argc, char *argv[]) {
    sayGoodMorningTo("CMake");

    return 0;
}
```

As indicated, the program invoke the the `sayGoodMorningTo()` function in `include/greeter/about_time.hpp` to print a `"Good morning, CMake"` to the standard output.

3.1.2 Especially CMakeLists Script

Our `CMakeLists` file goes as listings 3.3.

Listing 3.3: Codes for `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.9.4)
project(Hello CXX)

#[[ add the "include" directory under the source tree to the search path
  so that the '#include "greeter/about_time.hpp"' directive can be
  resolved ]]
include_directories("${PROJECT_SOURCE_DIR}/include")

# specify source files needed by the executable 'hello' to build
add_executable(hello src/hello.cpp)
```

Apart form the `cmake_minimum_required`, `project` and `add_executable` commands, we introduce 3 new features here

- Comments
  - **Bracket Comment**: start with `#[[` and end with `]]`, which can span across mutiple lines
  - **Line Comment**: start with `#` and run until the end of the line
- `include_directories` command: add the given directories to paths which compilers use to search for the include files. If the argument is specified as relative paths, it will be interpreted with respect to the current source directory.
- **Variable References**
  - format: `${variable_name}`
  - A variable reference will be dereferenced as the value of variable if the value is set, and an empty string otherwise.
  - Here, the variable in use is a built-in variable `CMAKE_SOURCE_DIR` which is predefined by the `cmake`. It refers to full path to the top level of the source tree. And its counterpart is the `CMAKE_BINARY_DIR` variable assuming the path to the top level of the binary tree.

3.2 Build the Project

So after horrible jargons, let's build the project to check if it's ok. Suppose we're in the source tree of the project now. Just execute following commands as listing 3.4 one by one, we will see it actually works!

Listing 3.4: Command to build the project

```
mkdir build
cd build
cmake ..
make
```

Which make a directory `build` as the binary tree and compile the project to generate the executable in it.

If nothing wrong, the output should be something similar to listing 3.5.

Listing 3.5: Successful build

```
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
```

Finally, find the generated `hello` executable, run it, and you should see "`Good morning, CMake`" in the standard output. Congratulations!

That's all for this example

Chapter 4

Tag the Version

4.1 Motivation

In an incremental development, a project is used to be built up feature by feature. To distinguish between these project with features added/removed/fixed, we usually tag these projects with special flags called version numbers. With each significant change, we assign the new project with a incremental version number.

For example, given a project `World`, it may called `World-0.1.2` at its initial launch, where

- 0 is the major version number, which will be incremented in case of a significant change, e.g., `World-1.1.2`
- 1 is the minor version number, and updated for a small change, e.g., `World-0.2.2`
- 2 is the revision number, signaling some revisions to users if any, e.g., `World-0.1.3`

One common usage of such version number is to tell users the version of software in use when given the `--version` or `-v` option.

Without `cmake`, these version numbers are usually hardcoded into the source codes. However, in our `cmake` style, we can specify them in the `CMakeLists` script, making them more configurable.

So, to keep up the convention, we'd like to tag the version of our `Hello` project, too. Again comes the example.

4.2 Make up the Toy Example

The file structure of our demo project goes as listing 4.1

Listing 4.1: File structure of the toy example

```
|-CMakeLists.txt
|-include
| |-config.hpp.in
|-src
| |-hello.cpp
```

4.2.1 The C++ Part

Source codes for C++ part are shown as listing 4.2 and 4.3.

Listing 4.2: `config.hpp.in`

```
#ifndef HELLO_CONFIG_HPP
#define HELLO_CONFIG_HPP

#cmakedefine HELLO_VERSION_MAJOR "@HELLO_VERSION_MAJOR@"
#cmakedefine HELLO_VERSION_MINOR "@HELLO_VERSION_MINOR@"
#define HELLO_REVISION "@HELLO_REVISION@"

#cmakedefine HELLO_FALSE_CONSTANT "@HELLO_FALSE_CONSTANT@"

#endif //HELLO_CONFIG_HPP
```

Listing 4.3: `hello.cpp`

```
#include <iostream>

#include "config.hpp"

int main(int argc, char *argv[]) {
    std::cout << argv[0] << "  " << HELLO_VERSION_MAJOR << "."
              << HELLO_VERSION_MINOR << "." << HELLO_REVISION << std::endl;

    return 0;
}
```

The program simply print the two version numbers defined in the `include/config.hpp` (still not generated yet!).

4.2.2 Especially the CMakeLists

The specification of the `CMakeLists` file to use is depicted as listing 4.4.

Listing 4.4: `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.9.4)
project(Hello CXX)

# specify the version number
set(HELLO_VERSION_MAJOR 3)
set(HELLO_VERSION_MINOR 1)
set(HELLO_REVISION 0)
set(HELLO_FALSE_CONSTANT 0)

# derive the "config.hpp" from the version numbers specified
# in this script, i.e., HELLO_VERSION_MAJOR and
# HELLO_VERSION_MINOR
configure_file("${PROJECT_SOURCE_DIR}/include/config.hpp.in"
        "${PROJECT_BINARY_DIR}/include/config.hpp" @ONLY)

#[[ add the "include" directory under the binary tree to the search path
  so that the '#include "config.hpp"' directive can be
  resolved ]]
include_directories("${PROJECT_BINARY_DIR}/include")

# specify source files needed by the executable 'hello' to build
add_executable(hello src/hello.cpp)
```

As you see, we can make cmake friends with several new commands agains. Let's introduce them one by one.

- `set` command for normal variables
  - Format: `set(<variable> <value>... [PARENT_SCOPE]`
    * `<varaible>` define the name of variable, which makes it dereferenceable by the `${variable}` syntax
    * `<value>` is a list of arguments (maybe none) as the value of the defined variables
    * As for the optional `PARENT_SCOPE`, we'd like to leave it for future
  - Last chapter, we just learn how to use variables pre-defined by the `cmake` system. The `set` command here enables us to define our own variables now.
  - Here, we defined 3 variables
    * `HELLO_VERSION_MAJOR` with value 3
    * `HELLO_VERSION_MINOR` with value 1
    * `HELLO_REVISION` with value 0

    . These variables are visible for the scope after their definitions.
- `configure_file` command for copying file `include/config.hpp.in` to the `include` folder in the binary tree and update its content with variables in the `CMakeLists`. As seen, mainly 2 macros involved: `#cmakedefine` and `#define`

- macro `#cmakedefine` is the recommanded one
  * It says that the `@HELLO_VERSION_MAJOR@`, `@HELLO_VERSION_MINOR@` and `@HELLO_REVISION@` should be replaced with values of the corresponding cmake variables assuming the same name with the `@` trimmed out. For example, `@HELLO_VERSION_MAJOR@` should be updated as the value of `HELLO_VERSION_MAJOR` in the `CMakeLists`.
  * As for `@HELLO_FALSE_CONSTANT@`, thing is a bit tricky. Since variable `HELLO_FALSE_CONSTANT` equals to 0 treated as a so-called **false constant** by `If` command (Sorry for another new jargon:(), macro declaration (`#cmakedefine HELLO_FALSE_CONSTANT @HELLO_FALSE_CONSTANT@`) will be replaced as (`/* #undef HELLO_FALSE_CONSTANT */`).
  * the `@...@` may be replaced with `${...}` if the `@ONLY` option is unspecified here.
- macro `#define` is the deprecated alternative to `#cmakedefine`. Unlike `#cmakedefine`, it doesn't suffer from the trouble of false constants.

4.2.3 Build it!

Bla, bla, ... Let's see if it's real.
Again, change the source tree of the project, and run following commands one by one as listing 4.5.

Listing 4.5: Commands to build the project

```
1 mkdir build
2 cd build
3 cmake ..
4 make
```

Check the `include` folder in the binary tree, you should see the generated `config.hpp` file. And run the `hello` executable, the expected output will printed

Haha, another example