CMake Cheatsheet – A gentle introduction to CMake

This cheatsheet will give you an idea of how CMake works and how it can be used to configure software projects.
The document and the CMake examples are available at `https://github.com/mortennobel/CMake-Cheatsheet`.

CMake - Creating a simple C++ project

CMake is a tool for configuring how a cross-platform source code project should be built on a given platform.
A small project could be organized like this:

```
CMakeLists.txt
src/main.cpp
src/foo.cpp
src/foo.hpp
```

This project contains two source files located in the `src` directory and one header file in the same directory.
When running CMake on this project you are asked for a build directory. It is best practice to create a new directory since this directory will contain all files related to building the project. If something goes wrong, you can delete the folder and start over.
Running CMake will not create the final executable, but instead, it will generate project files for Visual Studio, XCode or makefiles. Use these tools to build the project.

Understanding CMakeLists.txt

Creating project files using CMake requires a `CMakeLists.txt` file, which describes how the project is structured and how it should be built.
For example 1 the file looks like this:

```
cmake_minimum_required (VERSION 2.9)

# Setup projectname
project (HelloProject)

# Compile and link main.cpp and foo.cpp
# into the executable Hello
add_executable(Hello src/main.cpp src/foo.cpp)
```

First, the minimum version of CMake is defined. Then the project name is defined using the command `project()`. A project can contain multiple targets (either executables or libraries). This project defines a single executable target called `Hello`, which is created by compiling and linking the two source files `main.cpp` and `foo.cpp`.
When the two source files are compiled the compiler will search for the header file `foo.hpp` since both source files depend on this using `#include "foo.hpp"`. Since the file is located in the same directory as the source file, the compiler will not have any problems finding it.

The CMake Scripting Language

The CMakeLists.txt file describes the build process using a command based programming language. The commands are case insensitive and take a list of arguments.

```
# This is a comment.
COMMAND( arguments go here )
ANOTHER_COMMAND() # this command has no arguments
YET_ANOTHER_COMMAND( these
  arguments are spread          # another comment
  over several lines )
```

CMake script also has variables. Variables can either be defined by CMake itself or can be defined in the CMake script. The command `set(parameter value)` sets the given parameter to the given value. The command `message(value)` prints out the value to the console. To get the value of a variable use `${varname}`, which substitutes the variable name with its value.

```
cmake_minimum_required (VERSION 2.9)

SET(x 3)      # x = "3"
SET(y 1)      # y = "1"
MESSAGE(x y) # displays "xy"
MESSAGE(${x}${y}) # displays "31"
```

All variable values are text strings. Text strings can be evaluated as boolean expressions (e.g. when used in `IF()` and `WHILE()`). The values "FALSE", "OFF", "NO", or any string ending in "-NOTFOUND" evaluate to false - everything else evaluates to true.
Text strings can represent multiple values as a list by separating entities using a semicolon.

```
cmake_minimum_required (VERSION 2.9)

SET(x 3 2) # x = "3;2"
SET(y hello world !) # y = "hello;world;!"
SET(z "hello world!") # y = "hello world!"
MESSAGE(${x}) # prints "32"
MESSAGE("y = ${y} z = ${z}")
# prints y = hello;world;! z = hello world!
```

Lists can be iterated using the command `FOREACH (var val)`:

```
cmake_minimum_required (VERSION 2.9)

SET(x 3 2) # x = "3;2"
FOREACH (val ${x})
  MESSAGE(${val})
ENDFOREACH(val)

# prints:
# 3
# 2
```

Exposing compile options

CMake allows the end user (who runs CMake) to modify some variables. This is usually used to defined properties of the build such as locations of files, machine architecture, and string values.
The command `set(<variable> <value> CACHE <type> <docstring>)` sets the variable to the given value - but it allows the value to be changed by the CMake user when configuring the build. The type should be one of the following:

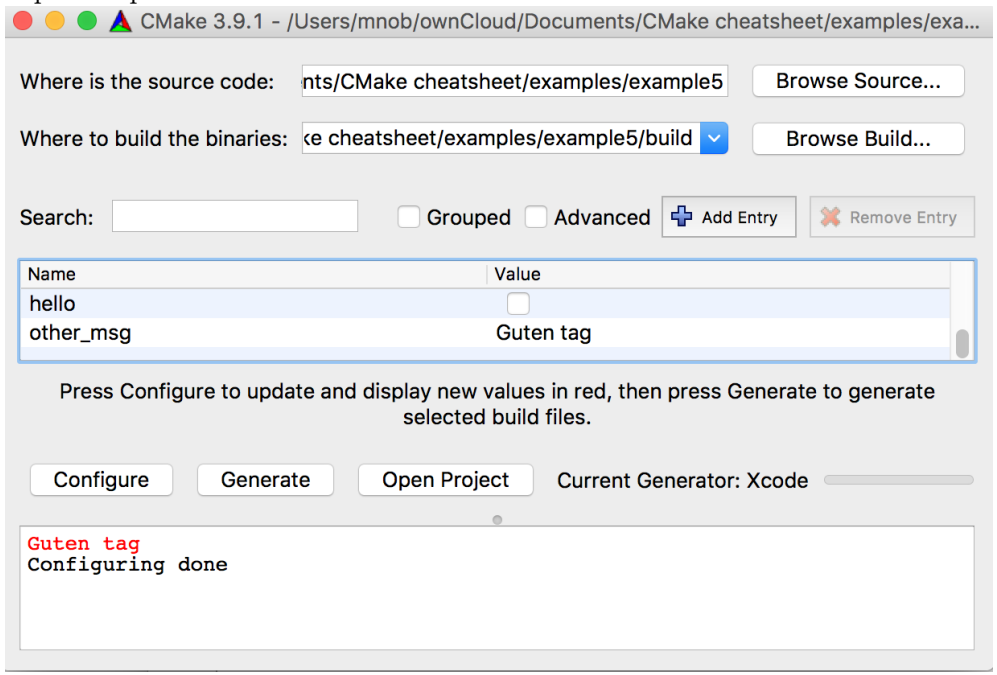
- FILEPATH = File chooser dialog.
- PATH = Directory chooser dialog.
- STRING = Arbitrary string.
- BOOL = Boolean ON/OFF checkbox.
- INTERNAL = No GUI entry (used for persistent variables).

In the following example, the user can configure whether "Hello" or an alternative string should be printed based on the configuration variables `hello` and `other\_msg`.

```
cmake_minimum_required (VERSION 2.9)

SET(hello true CACHE BOOL "If true write hello")
SET(other_msg "Hi" CACHE STRING "Not hello value")
IF (${hello})
        MESSAGE("Hello")
ELSE (${hello})
        MESSAGE(${other_msg})
ENDIF (${hello})
```

During configuration of the project, the CMake user gets prompted with the exposed options.



The values that the CMake user enters will be saved in the text file `CMakeCache.txt` as key-value pairs:

```
//...
//Print hello
hello:BOOL=OFF

//Not hello value
other_msg:STRING=Guten tag
//...
```

Complex projects

Some projects contain multiple executables and/or multiple libraries. For instance multiple executables for unit tests and programs. It is common to separate these subprojects into subfolders. Example:

```
CMakeLists.txt
somelib/CMakeLists.txt
somelib/foo.hpp
somelib/foo.cpp
someexe/CMakeLists.txt
someexe/main.cpp
```

The main `CMakeLists.txt` contains the basic project settings but then includes the subprojects:

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

project (HelloProject)

add_subdirectory(somelib)
add_subdirectory(someexe)
```

First the library `Foo` is compiled from the source in the `somelib` directory:

```
# somelib/CMakeLists.txt

# Compile and link foo.cpp
add_library(Foo STATIC foo.cpp)
```

Finally, the executable `Hello` is compiled and linked to the `Foo` library - note that the target name is used here - not the actual path. Since `main.cpp` references the header file `foo.hpp`, the `somelib` directory is added to the header search path:

```
# someexe/CMakeLists.txt

# add somelib to header search path
include_directories(../somelib/)

add_executable(Hello main.cpp)

# link to Foo library
target_link_libraries(Hello Foo)
```

Searching for source files

Use the `find(GLOB varname patterns)` to automatically search for files within a directory given one or more search patterns. Note that in the example below, both source files and header files are added to the project. This is not needed for compiling the project, but it is convenient when using an IDE since this also adds the header files to the project.

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

project (HelloProject)

file(GLOB sourcefiles
        "src/*.hpp"
        "src/*.cpp")

add_executable(Hello ${sourcefiles})
```

Runtime resources

Often runtime resources (such as DLLs, game-assets and text files) are read relative to the executable. One solution is to copy resources into the same directory as the executable. Example:

```
CMakeLists.txt
someexe/main.cpp
someexe/res.txt
```

In this project, the source files assume that the resource is located in the same directory as the executable:

```
// main.cpp
#include <iostream>
#include <fstream>

int main(){
    std::fstream f("res.txt");
    std::cout << f.rdbuf();
    return 0;
}
```

The CMakeLists.txt file makes sure to copy the resource file.

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

project (HelloProject)

add_executable(Hello someexe/main.cpp)

file(COPY someexe/res.txt DESTINATION Debug)
file(COPY someexe/res.txt DESTINATION Release)
```

Note: one problem with this approach is that if you modify the original resources, then you need to run CMake again.

External libraries

External libraries basically come in two flavors; dynamically linked libraries (DLLs) which are linked with the binary at runtime and statically linked libraries which are linked at compile time.

Static libraries have the simplest setup. To use one, the compiler needs to know the location of the header files and the linker needs to know the location of the actual library. Unless the external libraries are distributed along with the project it is usually not possible to know their location – for this reason, it is common to use cached variables, where the CMake user can change the location. Static libraries have the file extension `.lib` on Windows and `.a` on most other platforms.

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

project (HelloProject)

set(fooinclude "/usr/local/include"
        CACHE PATH "Location of foo header")
set(foolib "/usr/local/lib/foo.a"
        CACHE FILEPATH "Location of foo.a")

include_directories(${fooinclude})

add_executable(Hello someexe/main.cpp)
target_link_libraries(Hello ${foolib})
```

Dynamically linked libraries work similarly to statically linked libraries. On Windows, it is still needed to link to a library at compile time, but the actual linking to the DLL happens at runtime. The executable needs to be able to find the DLL file in the runtime linker's search path. If the DLL is not a system library, an easy solution is to copy the DLL in the same directory as the executable. Working with DLLs often requires platform specific actions, which CMake supports using the built-in variables `WIN32`, `APPLE`, and `UNIX`.

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

project (HelloProject)

set(fooinclude "/usr/local/include"
        CACHE PATH "Location of foo header")
set(foolib "/usr/local/lib/foo.lib"
        CACHE FILEPATH "Location of foo.lib")
set(foodll "/usr/local/lib/foo.dll"
```

```
        CACHE FILEPATH "Location of foo.dll")

include_directories(${fooinclude})

add_executable(Hello someexe/main.cpp)
target_link_libraries(Hello ${foolib})

IF (WIN32)
    file(COPY ${foodll} DESTINATION Debug)
    file(COPY ${foodll} DESTINATION Release)
ENDIF(WIN32)
```

Automatically locating libraries

CMake also contains a feature to automatically find libraries (based on a number of suggested locations) using the command `find_package()`. However, this feature works best on macOS and Linux.
`https://cmake.org/Wiki/CMake:How_To_Find_Libraries`.

C++ version

The C++ version can be set using the commands:

```
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

Defining preprocessor symbols

Use `add_definitions()` to add preprocessor symbols to the project.

```
# ...
add_definitions(-DFOO=\"XXX\")
add_definitions(-DBAR)
```

This will create the symbols `FOO` and `BAR`, which can be used in the source code:

```
#include <iostream>

using namespace std;

int main(){
#ifdef BAR
    cout << "Bar" << endl;
#endif
    cout << "Hello world " << FOO << endl;

    return 0;
}
```

Links and information

`https://cmake.org/Wiki/CMake/Language_Syntax`
`https://cmake.org/cmake/help/v3.0/command/set.html`

Created by Morten Nobel-Jørgensen, mnob@itu.dk, ITU, 2017

Latex template by John Smith, 2015
`http://johnsmith.com/`