**CMake**

**CMake** is a cross-platform free and open-source software tool for managing the build process of software using a compiler-independent method. It will help you to manage and build your source codes and deliveries effectively, used in conjunction with native build environments.

**CMake** :

* manages not only the build and packaging processes, but it can provide you also various IDE project files for development purpose. You don’t need to store these project files in a repository, they are (re)generated automatically by CMake after each configuration change.
* allows you to select various/desired compilers/build systems without duplicating your effort.
* It can be used for various programming languages.

**CMake** generates only the necessary build files requested by the native build systems in order to build from your source code the binaries.

# **Getting started with CMake**

**CMake** can be [installed](https://cmake.org/install/) on Windows, Linux, Mac OS operating systems.

For the majority of Linux distros, BSD variants have package managers supporting CMake installation with a simple command (ex: **apt-get install cmake** – by the Debian family).

*Just type* ***cmake –h*** *after your installation to verify whether CMake has been installed properly and which generators are seen by CMake on your system.*

CMake configuration files simple text files, placed in **a directory tree** and according to their names they can deserve the following scopes:

* + **CMakeLists.txt** – these fixed name files are seen by CMake as the entry CMake configuration file, used for managing the content of the current folder .
  + **<name>.cmake**– these files are used to extend the content of those CMakeList.txt files which are including them ex: *include (my.cmake)*. The content of the included <name>.cmake files are injected into the includers right after the inclusion point.

# **Creating an empty CMake solution**

1. Create a directory tree for the solution.
2. Define a project into CMakeLists.txt of the top level directory:

cmake\_minimum\_required(VERSION 3.2) # the minimal CMake version required

project(my\_solution) # the solution name

1. Link recursively the CMakeLists.txt files of the child directories:

add\_subdirectory(subfolder\_01) # linking in the CMakeLists.txt present in subfolder\_01

Create a build folder which is not included in any of the folders where a CMakeLists.txt file is present, go into the build folder and type:

cmake <folder\_of\_top\_level\_CMakeLists.txt>

or

cmake –G “<generator\_name>” <folder\_of\_top\_level\_CMakeLists.txt>

if you want to specify the generator, other than the default one.

# **Creating an executable**

Suppose we have in a folder, named **myfolder** the following C++ files:

* + myheader.hpp
  + mysource.cpp

Create a CMakeLists.txt in myfolder.

The first line is always the minimum required version:

**cmake\_minimum\_required**(VERSION 3.2) # the minimal CMake version required

Next we need to define the project name in the CMakeLists.txt:

**project**(myapp) # our solution/project name is myapp

Next we need to define the desired C++ language standard:

**set**(CMAKE\_CXX\_STANDARD 14) # we request C++14

**set**(CMAKE\_CXX\_STANDARD\_REQUIRED ON) # we request that CMake verifies the compiler

**set**(CMAKE\_CXX\_EXTENSIONS OFF) # we request -std=c++14 in the compiler flags instead of –std=gnu++14

Define the executable name and source file:

**add\_executable**(myapp mysource.cpp) # we specify the name of executable and source (cpp) files

**add\_executable**(myapp mysource.cpp myheader.hpp) #we can add headers into the executable definition

When the executable name is the same as the project name we can re-use the project name variable

**add\_executable**(${PROJECT\_NAME} mysource.cpp myheader.hpp) #where ${PROJECT\_NAME} contains the string what we added via the project() command.

Is possible to create new variables inside a CMakeLists.txt using the **set** command:

**set**(<variable\_name> <variable value>) # we store into <variable\_name> the <variable\_value>

We can define our executable by using a filenames-list variable also

**set**(myfiles mysource.cpp myheader.hpp)

**add\_executable**(${PROJECT\_NAME} ${myfiles})

To build the executable, we need to run, from the build folder for example:

cmake ../

make

The resulting executable (myapp) will be placed into the build folder. To run this application we need to run:

./myapp

# **Extending the executable**

Changing the executable definition (CMake searches the files relative to the current CMakeLists.txt folder). Let’s suppose we have an application that has the cpp file in the src directory and the header file in the inc.

**set**(*myfiles* src/mysource.cpp inc/myheader.hpp)

**add\_executable**(${PROJECT\_NAME} ${*myfiles*})

Defining the **search paths** – where the header files are relative to the current CMakeLists.txt:

**include\_directories**(<folder\_1> <flder\_2> .. )

**include\_directories**(inc) # for our example above

Suppose we want to create a header file dynamically (named **myconfig.hpp**), based on a configurable receipt, stored in the **myconfig.hpp.in** file, in the **cfg** sub-folder:

**configure\_file**(<source> <target>)

**configure\_file**(cfg/myconfig.hpp.in inc/myconfig.hpp)

This will require to adapt our file list by defining the executable, but we need also to tell to CMake which search paths to specify to the compiler for the #include C++ commands.

Suppose we want to send some C++ preprocessor commands to the compiler, which will set a macro for the source files (#define APP\_VERSION)

**add\_definitions**(-D<macro\_name>)

**add\_definitions**(-D<macro\_name>=<macro\_value>)

**add\_definitions**(-DAPP\_VERSION=“1.0”)

# **Creating a library**

Suppose we have in a folder, named **myfolder** the following C++ files:

* + inc/myheader.hpp
  + src/mysource.cpp
  + src/myconfig.hpp

and we want to generate from these files a static, or dynamic named **mylib**.

First we need to create a CMakeLists.txt just like we described above, in the myfolder dirrectory, after this we need to:

* set the minimum version of CMake
* define the project name
* define the desired C++ standard
* set the executable name

Now we define the library name and source files with the add\_library() command:

**set**(myfiles src/mysource.cpp inc/myheader.hpp)

* + For creating a static library

**add\_library**(${PROJECT\_NAME} **STATIC** ${myfiles})

* + For creating a dynamic library

**add\_library**(${PROJECT\_NAME} **SHARED** ${myfiles})

Specify with PUBLIC, or [INTERFACE](https://cmake.org/cmake/help/latest/prop_tgt/INTERFACE_INCLUDE_DIRECTORIES.html) keyword the paths which shall be used for build and export, or for build and install purposes also

**target\_include\_directories**(${**PROJECT\_NAME**}

**PRIVATE** src **INTERFACE**

$<**BUILD\_INTERFACE**:${**CMAKE\_CURRENT\_SOURCE\_DIR**}/inc>

$<**INSTALL\_INTERFACE**:inc>)

To build the library we follow the same steps as for an usual executable file.

If there is more than just a directory, is recommended to use the variables **PROJECT\_BINARY\_DIR** and **PROJECT\_SOURCE\_DIR**, to specify the directory that we refer to. These variables are set by the last **project**() call in the CMakeLists.txt.