CMake Cheatsheet (for Linux)

Arturo Roberti

February 4, 2025

1 Main Examples

1.1 File Structure

The desired structure is as follows:

```
my_package/
├── build/
├── cmake/
│   ├── script_1.cmake
│   └── ...
├── config/
│   ├── config_1.yaml
│   └── ...
├── externals/
│   ├── package_1/
│   └── ...
├── include/
│   ├── header_1.hpp
│   └── ...
├── src/
│   ├── source_1.cpp
│   └── ...
├── tests/
│   ├── CMakeLists.txt
│   ├── test_1.cpp
│   └── ...
├── CMakelists.txt
├── LICENSE
├── package.xml
├── pyproject.toml
└── README.md
```

Here, red is mandatory (i.e. necessary/extremely standard practice) and green is optional (i.e. standard practice). Additionally, the `package.xml` file is only used for *ROS* package management and the `pyproject.toml` file is only used in connection with *pybind*.

1.2 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)

project(example_project)

# Used to generate compile_commands.json
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Add custom library
add_library(example_library src/example.cpp)
target_include_directories(example_library PUBLIC include)

# Add the executable
add_executable(example_executable src/example_executable.cpp)
target_link_libraries(example_executable example_library)

# Include external library 'Eigen3'
find_package(Eigen3 3.4.0 QUIET)
if(Eigen3_FOUND)
        message(STATUS "Using system provided Eigen.")
else()
        message(STATUS "No system Eigen found. Auto-installing.")
        include(cmake/eigen.cmake)
endif()
target_include_directories(example_library PUBLIC ${EIGEN3_INCLUDE_DIR})

# Install the library to the system to /usr/local/lib
install(TARGETS example_library DESTINATION lib)

# Testing
option(BUILD_TESTING "Build the tests" ON)
if(BUILD_TESTING)
    find_package(GTest CONFIG REQUIRED)
    add_subdirectory(tests)
endif()

# Pybind
file(GLOB_RECURSE PYBIND_SRC src/*pybind*.cpp)
if(PYBIND_SRC)
    message(STATUS "Building Python bindings.")

    find_package(pybind11 2.9.1 QUIET)
    if(pybind11_FOUND)
        message(STATUS "Using system provided Pybind.")
    else()
        message(STATUS "No system Pybind found. Auto-installing.")
        include(cmake/pybind11.cmake)
    endif()
    # Enable PIC (necessary for dynamic libraries)
    set_target_properties(example_library PROPERTIES
                          POSITION_INDEPENDENT_CODE ON)

    # Add and link the pybind module
    pybind11_add_module(example_py MODULE ${PYBIND_SRC})
    target_link_libraries(example_py PRIVATE example_library)
endif()
```

2 General Commands

These commands and more can be found in the CMake documentation.

2.1 cmake_minimum_required

```
cmake_minimum_required(VERSION <version>)
```

Sets the minimum required version of cmake. Needs to be the first command in the `CMakeLists.txt` file.

2.2 project

```
project(<name> [VERSION <major>.<minor>.<patch>])
```

Sets the name of the project, and stores it in the variable `PROJECT_NAME`. Needs to be the second command in the `CMakeLists.txt` file.

2.3 message

```
message([<mode>] "<message>")
```

mode ∈ {FATAL_ERROR, SEND_ERROR, WARNING, AUTHOR_WARNING,
DEPRECATION, NOTICE, STATUS, VERBOSE, DEBUG, TRACE}

Default mode is `NOTICE`.

2.4 return

```
return()
```

Exits the current script file.

2.5 install

```
install(TARGETS <target> DESTINATION <dir>)
```

Allows installing the specified target to the specified system directory (usually `bin`, `lib`, `include`). The path is relative to the `CMAKE_INSTALL_PREFIX` variable (linux default `/usr/local/`).
For installation, later run `make install` (Section 9.2).

3 Include Libraries

3.1 Own library

3.1.1 add_library

```
add_library(<name> [STATIC | SHARED] <source1> <source2> ...)
```

Creates a library target with the given name and source files (.cpp; cannot run stand-alone).

- `STATIC` - Static library (.a, .lib) - copied into the target at build time.
- `SHARED` - Dynamic library (.so, .dll) - linked at runtime (i.e. can be updated without recompiling).

If neither is specified, the default depends on the `BUILD_SHARED_LIBS` variable.

3.1.2 include_directories

```
include_directories(<dir1> <dir2> ...)
```

Include directories containing header files for the entire project. Preferably, use `target_include_directories` instead.

3.1.3 target_include_directories

```
target_include_directories(<target> [PUBLIC | PRIVATE] <dir1> <dir2> ...)
```

Include directories containing header files for a specific target.

3.1.4 add_executable

```
add_executable(<name> <source1> <source2> ...)
```

Adds executables (.cpp; can run stand-alone) to a (new) target.

Here, the target is usually itself, not the overarching library target, as the library should be independent of the executable. Instead, the library is later linked to the executable target as if it was an external library.

3.1.5 include

```
include(<file_basename>.cmake)
```

Include another CMake script file in the current script location.

3.1.6 file

```
file([GLOB | GLOB_RECURSE] <variable> <pattern1> <pattern1> ...)
```

Saves a list of file paths into `<variable>` matching the specified patterns (recursively). Example:

```
file(GLOB_RECURSE FLIGHTLIB src/dir1/*.cpp src/dir2/*.cpp)
```

Later used to add the library, e.g.

```
add_library(${PROJECT_NAME} ${FLIGHTLIB})
```

3.2 External libraries

3.2.1 find_package

`find_package(<PackageName> [<version>] [QUIET] [REQUIRED] [NO_MODULE | CONFIG])`

Finds and loads a package on the system. Easiest way to include external libraries, if possible.

- `QUIET` suppresses info&warning output of the package search
- `REQUIRED` stops the configuration process if the package is not found. If not set, the package search can be checked with the boolean `<PackageName>_FOUND` (`find_package()` sometimes also sets other variables such as `<PackageName>_INCLUDE_DIR`, `<PackageName>_EXECUTABLE`, ...)
- `NO_MODULE` - Only search for a `Find<PackageName>.cmake` file to find the package
- `CONFIG` - Only search for a `<PackageName>Config.cmake` (or `<lowercasePackageName>-config.cmake`) file to find the package

If neither `NO_MODULE` nor `CONFIG` is specified, `find_package` will first search for a `Find<PackageName>.cmake` file and then for a `<PackageName>Config.cmake` file.

To find the exact package name (case-sensitive), use one of

- `sudo find /usr/ -type f -name "Find*.cmake"`
- `sudo find /usr/ -type f -name "*Config.cmake"`
- googling *~PackageName* cmake

From the first two, the package name can be inferred. I strongly recommend using `| grep -i <pkg>` to filter the output (`<pkg>` being a sure part of the package name).

Many packages can be installed via apt (see **Ubuntu Version ⇒ Library development** in Ubuntu Packages Repository, e.g. `sudo apt install libeigen3-dev`). In that case, the version can be found via `apt show <package-name> | grep Version`. Otherwise, refer to the *(auto)installation* section (Section 4). Keep in mind that the version is an opional specification of `find_package()`.

3.2.2 add_subdirectory

`add_subdirectory(<dir>)`

Add a subdirectory (external package with a `CMakeLists.txt` file) to the build. The `CMakeLists.txt` file in the specified directory will be processed.

3.2.3 link_libraries

`link_libraries(<library1>.end <library2>.end ...)`

Links libraries (end ∈ {.a, .lib, {}, .so, .dll}) to all future project targets. Preferably, use `target_link_libraries` instead.

3.2.4 target_link_libraries

`target_link_libraries(<target> [PUBLIC | PRIVATE] <library1>.end <library2>.end ...)`

Links libraries (end ∈ {.a, .lib, {}, .so, .dll}) to a specific target.

4 Auto Installation

For the auto-installation of external libraries, it best best to use two cmake scripts (names arbitrary). I can best explain it via the following eigen3 example files.

4.1 eigen.cmake

This file is used by the main `CMakeLists.txt` file to auto-intall Eigen3 if necessary. For this reason, it also sets `EIGEN3_INCLUDE_DIR` to the correct path at the end.

```
1   # Download and unpack eigen at configure time
2   message(STATUS "Getting Eigen...")
3
4   configure_file(
5       cmake/eigen_download.cmake
6       ${PROJECT_SOURCE_DIR}/externals/eigen/CMakeLists.txt)
7
8   execute_process(
9       COMMAND ${CMAKE_COMMAND} -G "${CMAKE_GENERATOR}" .
10      RESULT_VARIABLE result
11      WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/externals/eigen
12      OUTPUT_QUIET)
13  if(result)
14      message(FATAL_ERROR "Download of Eigen failed: ${result}")
15  endif()
16
17  execute_process(
18      COMMAND ${CMAKE_COMMAND} --build .
19      RESULT_VARIABLE result
20      WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/externals/eigen
21      OUTPUT_QUIET)
22  if(result)
23      message(FATAL_ERROR "Build step for eigen failed: ${result}")
24  endif()
25
26  message(STATUS "Eigen downloaded!")
27
28  set(EIGEN3_INCLUDE_DIR ${PROJECT_SOURCE_DIR}/externals/eigen/eigen3)
```

In essence, there are 3 steps:

1. Copy the `eigen_download.cmake` file to a useable location
2. Download Eigen3
3. Build Eigen3

In the latter two, the working directory is set and the (BOOL) success of the operation is saved in "result". Furthermore, `COMMAND ${CMAKE_COMMAND}` translates to running `cmake` in the terminal (e.g. `execute_process(COMMAND ${CMAKE_COMMAND} --build . ...)` corresponds to `cmake --build . ...` in a terminal). Lastly, `CMAKE_GENERATOR` is a default variable that specifies the build system to use (Linux/MacOS: `Unix Makefiles`).

4.2 eigen_download.cmake

```
cmake_minimum_required(VERSION 3.0)

project(eigen-external)

include(ExternalProject)
ExternalProject_Add(eigen
    GIT_REPOSITORY    https://gitlab.com/libeigen/eigen.git
    GIT_TAG           3.4.0
    SOURCE_DIR        "${PROJECT_SOURCE_DIR}/externals/eigen/eigen3"
    CONFIGURE_COMMAND ""
    BUILD_COMMAND     ""
    INSTALL_COMMAND   ""
    TEST_COMMAND      ""
    UPDATE_DISCONNECTED ON
)
```

This is later used as a script to download Eigen3. The external `ExternalProject` module is used to download repos from gitlab or girhub, depending on where it is hosted. For more infos and further ways of downloading exteral packages via cmake see the CMake documentation.

5 Variables

5.1 Base Types

The existing variable types are:

- `STRING` - Generic string value
- `BOOL` - Boolean ON/OFF value
- `PATH` - Path to a directory
- `FILEPATH` - Path to a file
- `INTERNAL` - Do not present in GUI at all
- `STATIC` - Value managed by CMake, do not change
- `UNINITIALIZED` - Type not yet specified

5.2 Lists

For the following and many other commands, see CMake documentation.

5.2.1 Populating lists

Lists are first defined via `set(<list_name>) ["<value1>" "<value2>" ...]` and later increased via `list(APPEND <list_name> "<value>")`.

Alternatively, for filenames, use `file([GLOB | GLOB_RECURSE] <list_name> <pattern1> <pattern2> ...)`, e.g. `file(GLOB_RECURSE <list_name> src/*.cpp)`.

5.2.2 Using lists

To access the elements of a list, use `${<list_name>_<index>}`, possibly within a for loop (Section 6.3).

Lists of filenames can be directly used in the `add_library()` or `add_executable()` commands, e.g. `add_library(${PROJECT_NAME} <list_name>)`.

5.3 Set Variables

5.3.1 option

`option(<variable> "<help text>" <value>)`

Defines a boolean variable as `ON` or `OFF` (default). If already defined, it will NOT be changed.

5.3.2 set

`set(<variable> [<value>])`

Assign a value to a variable. If `<value>` is not provided, the variable is unset (set to an empty string; equivalent to `unset(<variable>)`).

5.3.3 set

`set(<variable> <value>... CACHE <type> <docstring> [FORCE])`

Set a variable to a value and store it in the cache (i.e. is not re-built). `<type>` is one of the base types (Section 5.1). The `FORCE` option will overwrite the variable in the cache even if it is already defined.

Example: `set(TEST "Hello, World!" CACHE STRING "Test variable")`

6 Conditionals

6.1 If-Else Statements

```
if(<condition>)
    <commands>
elseif(<condition>)
    <commands>
else()
    <commands>
endif()
```

Conditions are case-insensitive and are evaluated as:

- **TRUE** if `1` (or non-zero number), `ON`, `True`, `YES` or `Y`
- **FALSE** if `0`, `OFF`, `False`, `NO`, `N`, `IGNORE`, `NOTFOUND`, `""` or ending in `-NOTFOUND`

6.2 While Loop

```
while(<condition>)
    <commands>
endwhile()
```

6.3 For Loop

```
foreach(<loop-variable> <values>)
    <commands>
endforeach()
```

Variations include (all values are positive integers):

- `foreach(<loop_var> RANGE <stop>)` - Loop from 0 to `<stop>`
- `foreach(<loop_var> RANGE <start> <stop> [<step>])` - Loop from `<start>` to `<stop>` with optional `<step>`
- `foreach(<loop_var> IN [<list>])` - Loop over the elements in the given list
- `foreach(<loop_var> IN ZIP_LISTS <lists>)` - Loop over the elements of multiple lists in parallel (access elements via `${<loop_var>_<list_index>}`, e.g. `${<loop_var>_0}`)

6.4 Existence Checks

`if(TARGET <target-name>)`

True if the given name is an existing logical target name created by a call to the `add_executable()`, `add_library()`, or `add_custom_target()` command that has already been invoked (in any directory).

`if(TEST <test-name>)`

True if the given name is an existing test name created by the `add_test()` command.

`if(DEFINED <name> [CACHE{<name>}|ENV{<name>}])`

True if a variable, cache variable or environment variable with given ¡name¿ is defined.

`if(<variable|string> IN_LIST <variable>)`

True if the given element is contained in the named list variable.

6.5 File Operations

File paths are always treated as full paths (and cannot include e.g. ∼). Relative paths are interpreted as relative to the current source directory.

`if(EXISTS <path-to-file-or-directory>)`

True if the named file or directory exists and is readable. False if the given path is an empty string.

`if(IS_xxx <path-to-file-or-directory>)`

xxx ∈ {READABLE, WRITABLE, EXECUTABLE, DIRECTORY, SYMLINK, ABSOLUTE}

True if the named file or directory has the specified property. False if the given path is an empty string.

`if(<file1> IS_NEWER_THAN <file2>)`

True if file1 is newer than file2 or if one of the two files doesn't exist.

6.6 Comparisons

`if(<variable|string> xxx <variable|string>)`

---

xxx ∈ {EQUAL, LESS, GREATER, LESS_EQUAL, GREATER_EQUAL}

True if the given string or variable's value parses as a real number (like a C double) and satisfies the specified comparison.

---

xxx ∈ {STREQUAL, STRLESS, STRGREATER, STRLESS_EQUAL, STRGREATER_EQUAL}

True if the given string or variable's value lexicographically satisfies the specified comparison.

---

xxx ∈ {VERSION_LESS, VERSION_GREATER, VERSION_EQUAL, VERSION_LESS_EQUAL, VERSION_GREATER_EQUAL}

Component-wise integer version number comparison (version format is major[.minor[.patch[.tweak]]], omitted components are treated as zero). Any non-integer version component or non-integer trailing part of a version component effectively truncates the string at that point.

`if(<variable|string> PATH_EQUAL <variable|string>)`

Compares the two paths component-by-component. Only if every component of both paths match will the two paths compare equal. Multiple path separators are effectively collapsed into a single separator, but note that backslashes are not converted to forward slashes. No other path normalization is performed.

7 Testing

To test features of a project, both CMake and C++ code are involved. Make sure to have the `GTest` library installed (`sudo apt install libgtest-dev`).

7.1 CMake

Add the following at the end of the main `CMakeLists.txt` file:

```
# Testing
find_package(GTest CONFIG REQUIRED)
add_subdirectory(tests)
```

Create a `tests` folder with its own `CMakeLists.txt` file:

```
include(GoogleTest)

add_executable(test_vector test_vector.cpp)

target_link_libraries(test_vector PRIVATE
    example_library
    GTest::gtest_main
)

gtest_discover_tests(test_vector)
```

The above example is an example of a single test using a *test_vector.cpp* file in the *tests* directory. The example can be repeated with further *.cpp* files.

7.2 C++

In each *test_¡feature¿.cpp* file, include a variation of the following example:

```cpp
#include <gtest/gtest.h>
#include "example.hpp"

TEST(VectorTests, TestSize) {
    example::ExampleClass ex;
    EXPECT_EQ(ex.getSize(), 3);
}

...
```

The first input to `TEST` is the testing group name, and the second is the specific test name. Multiple tests can be combined in the same group for readability purposes. In each test, multiple assertions can be made.

7.2.1 Available testing macros

In general, assertion macros start with `EXPECT_` or `ASSERT_`. The former continues testing even if the condition fails, while the latter stops the test. Either assertion then continues with:

Basic Assertions:

`TRUE(<condition>)`, `FALSE(<condition>)`, `EQ(<arg1>, <arg2>)`, `NE(<arg1>, <arg2>)`, `LT(<arg1>, <arg2>)`, `LE(<arg1>, <arg2>)`, `GT(<arg1>, <arg2>)`, `GE(<arg1>, <arg2>)`

Floating-Point Comparison Assertions:

`FLOAT_EQ(<arg1>, <arg2>)`, `DOUBLE_EQ(<arg1>, <arg2>)`, `NEAR(<arg1>, <arg2>, <tolerance>)`

String Assertions:

`STREQ(<arg1>, <arg2>)`, `STRNE(<arg1>, <arg2>)`

Container Assertions:

`THAT(<arg1>, testing::ElementsAre(<el1>, ...))`,
`THAT(<arg1>, testing::UnorderedElementsAre(<el1>, ...))`

For custom assertions (non-basic types; e.g. matrix comparison), use `ASSERT_EQ` and `EXPECT_EQ` with custom comparison functions.

7.3 Running

To run the tests, navigate to the build directory and run

- `ctest` to run all tests
- `ctest -R <test-group>` to run all tests in a specific group
- `ctest -R <test-group>.<test-name>` to run a specific test

8 Pybind11

Used to create a python package out of a C++ library and requires `CMake`, `C++` and a `pyproject.toml` file.

8.1 CMake

The CMake implementation example is given in the main example. The main steps include:

- `set_target_properties(example_library PROPERTIES POSITION_INDEPENDENT_CODE ON)` - Necessary, as the later built dynamic library (`.so`-file) will not have a permanent memory address
- `pybind11_add_module(example_py MODULE ${PYBIND_SRC})` - Custom Pybind command defining the new python module
- `target_link_libraries(example_py PRIVATE example_library)` - Link the custom library to the python module

Furthermore, the `pybind11.cmake` and `pybind11_download.cmake` files are similar to their *eigen* equivalents (Sections 4.1, 4.2). Additionally, *pybind* is linked to the main library via `add_subdirectory(${PROJECT_SOURCE_DIR}/externals/pybind/pybind11 EXCLUDE_FROM_ALL)`.

8.2 C++

Below an example of a `pybind_wrapper.cpp` file used in CMake to create a python module:

```cpp
#include <pybind11/eigen.h>
#include <pybind11/pybind11.h>
#include <sstream>

#include "example.hpp"

namespace py = pybind11;
using namespace example;

PYBIND11_MODULE(example_py, m) {
    py::class_<ExampleClass>(m, "ExampleClassPy")
        // Constructor(s)
        .def(py::init<>())

        // Singular methods
        .def("getSize", &ExampleClass::getSize)
        .def("getVector", &ExampleClass::getVector)

        // Overloaded method  - No input
        .def("printClassVector",
             static_cast<void (ExampleClass::*)()>
             (&ExampleClass::printVector))

        // Overloaded method - Vector input
        .def("printVector",
             static_cast<void (ExampleClass::*)(const Eigen::VectorXd&)>
             (&ExampleClass::printVector));

    // Further classes and functions...
}
```

For further information, see the pybind documentation.

8.3 Pip installation

A pyproject.toml file is necessary to install the python package via pip. An example is given below:

```
[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "example_py"
version = "0.1.0"
description = "Python bindings for the C++ example module"
authors = [{ name = "Arturo Roberti" }]
readme = "README.md"

[tool.setuptools]
packages = []
include-package-data = true

[tool.setuptools.package-data]
example_py = ["build/example_py*.so"]

[tool.setuptools.dynamic]
dependencies = { file = ["requirements.txt"] }  # Optional
```

This file is best added to the main directory and not the build directory to avoid accidental deletion. Then, the package can be installed from the main directory via `pip install -e .` (*editable mode* flag `-e` optional). However, as the shared library (`.so`-file) is built in the *build* directory, the `PYTHONPATH` variable must be extended to include the build directory. This can be done by adding the following line to the `.bashrc` file:

```
export PYTHONPATH=$PYTHONPATH:/path/to/build
```

9 Deploying

9.1 Configuring

To configure a project, run (bash) `cmake -S <source-dir> -B <build-dir>`. If currently in the build directory, `-S` and `-B <build-dir>` can be omitted, thus only `cmake <source-dir>` is necessary (usually `cmake ..`).

9.2 Building

To build the project, run (bash) `make` from the build directory. If `install`)) was used in the `CMakeLists.txt` file, run `make install` to install the project to the specified directory.

9.3 Running

To run an executable of the project (added via `add_executable()`), navigate to the build directory and run it ((bash) `./<executable>`). If the executable was previously installed to bin/, it can be run from anywhere ((bash) `<executable>`).