Introduction to (modern) CMake

Pasquale Africa

Build systems

Purposes

Build systems are a way to deploy software.

They are used to

1. provide others a way to configure **your** own project;
2. configure and install third-party software on your system.

To configure means to

- meet dependencies
- build
- test

Build systems available

- **CMake**[1]
  - ✓ Easy to learn, great support for multiple IDEs, cross-platform
  - ✗ Sometimes more complicated than necessary. Does not perform compilation test for met dependencies.
- **GNU Autotools**[2]
  - ✓ Excellent support for legacy Unix platforms, large selection of existing modules.
  - ✗ Slow, hard to use correctly, painful to debug, poor support for non-Unix platforms.
- **Meson**[3], **Bazel**[4], **SCons**[5] …

[1] https://cmake.org/
[2] https://www.gnu.org/software/automake/manual/
[3] https://mesonbuild.com/
[4] https://bazel.build/
[5] https://scons.org/

Why CMake?

- More packages use CMake than any other system
- every IDE supports CMake (or vice-versa)
- really cross-platform, no better choices for Windows
- extensible, modular design

Why CMake?

- More packages use CMake than any other system
- every IDE supports CMake (or vice-versa)
- really cross-platform, no better choices for Windows
- extensible, modular design

Who else is using CMake?

- Netflix
- HDF Group, ITK, VTK, Paraview (visualization tools)
- Armadillo, CGAL, LAPACK, Trilinos (linear algebra and algorithms)
- deal.II, Gmsh (FEM analysis)
- KDE, Qt, ReactOS (user interfaces and operating systems)

Resources

- Official documentation
  https://cmake.org/cmake/help/latest/
- Modern CMake
  https://cliutils.gitlab.io/modern-cmake/
- It's time to do CMake right
  https://pabloariasal.github.io/2018/02/19/its-time-to-do-cmake-right/
- Effective Modern CMake
  https://gist.github.com/mbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1
- More Modern CMake
  https://www.youtube.com/watch?v=y7ndUhdQuU8&feature=youtu.be

CMake Basics

The root of a project using CMake needs to contain a **CMakeLists.txt** file.

```
cmake_minimum_required(VERSION 3.5)

# This is a comment.
project(MyProject VERSION 1.0
                  DESCRIPTION "Very nice project"
                  LANGUAGES CXX)
```

Please use a CMake version more recent than your compiler (at least $\geq$ 3.0).

Command names are **case-insensitive**. Then:

```
mkdir build && cd build
cmake SOURCE_DIR
```

Targets

CMake is all about targets and properties. An executable is a target, a library is a target. Your application is built as a collection of targets depending on and using each other.

```
# Header files are optional.
add_executable(my_exec my_main.cpp my_header.h)

# Options are STATIC, SHARED (dynamic) or MODULE (plugins).
add_library(my_lib STATIC my_class.cpp my_class.h)
```

Target properties

Target can be associated properties:

```
add_library(my_lib STATIC my_class.cpp my_class.h)
target_include_directories(my_lib PUBLIC include_dir)
# "PUBLIC" propagates the property to
# other targets depending on "my_lib".
target_link_libraries(my_lib PUBLIC another_lib)

add_executable(my_exec my_main.cpp my_header.h)
target_link_libraries(my_exec my_lib)
target_compile_features(my_exec cxx_std_11)
# Last command is equivalent to
# set_target_properties(my_exec PROPERTIES CXX_STANDARD 11)
```

Communicate with the outside world

Local variables

```
set(LIB_NAME "my_lib")

# List items are space- or semicolon-separated.
set(INCLUDE_LIST "one;two")

add_library(${LIB_NAME} STATIC my_class.cpp my_class.h)
target_include_directories(${LIB_NAME} PUBLIC ${INCLUDE_LIST})

add_executable(my_exec my_main.cpp my_header.h)
target_link_libraries(my_exec ${LIB_NAME})
```

Cache variables

Cache variables are used to communicate with the command line:

```
# "VALUE" is just the default value.
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "Description")

# Boolean specialization:
option(MY_OPTION "This is settable from the command line" OFF)
```

Then:

```
cmake SOURCE_DIR -DMY_CACHE_VARIABLE="CUSTOM_VALUE"
```

Useful variables

CMAKE_SOURCE_DIR : top-level source directory
CMAKE_BINARY_DIR : top-level build directory

If the project is organized in sub-folders:
CMAKE_CURRENT_SOURCE_DIR : current source directory being processed
CMAKE_CURRENT_BINARY_DIR : current build directory

```
# Options are "Release", "Debug",
# "RelWithDebInfo", "MinSizeRel"
set(CMAKE_BUILD_TYPE Release)

set(CMAKE_CXX_COMPILER "/path/to/c++/compiler")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY lib)
```

Environment variables

```
# Read.
message("PATH is set to: $ENV{PATH}")

# Write.
set(ENV{variable_name} value)
```

(though it is generally a good idea to avoid them).

Control flow

```
if("${variable}")
    # True if variable is not false-like
else()
    # Note that undefined variables would be "" thus false
endif()
```

The following operators can be used.

Unary: NOT, TARGET, EXISTS (file), DEFINED, etc.
Binary: STREQUAL, AND, OR, MATCHES (regular expression), ...

Parentheses can be used to group.

Branch selection

Useful for switching among different implementations or version of any third-party library.

`my_main.cpp`:

```
#ifdef USE_ARRAY
    std::array<double, 100> my_array;
#else
    std::vector<double> my_array;
#endif
```

How to select the correct branch?

Pre-processor flags

CMakeLists.txt:

```
target_compile_definitions(my_exec PRIVATE USE_ARRAY=1)
```

Or let user set the desired flag:

```
option(WITH_ARRAY "Use std::array instead of std::vector" ON)

if(WITH_ARRAY)
    target_compile_definitions(my_exec PRIVATE USE_ARRAY=1)
endif()
```

Debug

Content of variables is printed with

```
message("MY_VAR is: ${MY_VAR}")
```

Commands being executed are printed with

```
make VERBOSE=1
```

Let's try...

Advanced CMake

Looking for third-party libraries

CMake looks for **module files** `FindPackage.cmake` in the directories specified in `CMAKE_PREFIX_PATH`.

```
set(CMAKE_PREFIX_PATH "${CMAKE_PREFIX_PATH} /path/to/module/")

# Specify "REQUIRED" if library is mandatory.
find_package(Boost 1.50 COMPONENTS graph)
```

If the library is not located in a system folder, often a hint can be provided:

```
cmake SOURCE_DIR -DBOOST_ROOT=/path/to/boost
```

Once the library is found, proper variables are populated.

```
if(${Boost_FOUND})
    target_include_directories(my_lib PUBLIC
                               ${Boost_INCLUDE_DIRS})

    target_link_directories(my_lib PUBLIC
                            ${Boost_LIBRARY_DIRS})
    # Old CMake versions:
    # link_directories(${Boost_LIBRARY_DIRS})

    target_link_libraries(my_lib ${Boost_LIBRARIES})
endif()
```

Organize a large project

```
cmake_minimum_required(VERSION 3.5)
project(ExampleProject VERSION 1.0 LANGUAGES CXX)

find_package(...)

add_subdirectory(src)
add_subdirectory(apps)
add_subdirectory(tests)
```

Compilation test

CMake can try to compile a source and save the exit status in a local variable.

```
try_compile(
    HAVE_ZIP
    "${CMAKE_BINARY_DIR}/temp"
    "${CMAKE_SOURCE_DIR}/tests/test_zip.cpp"
    LINK_LIBRARIES ${ZIP_LIBRARY}
    CMAKE_FLAGS
        "-DINCLUDE_DIRECTORIES=${ZIP_INCLUDE_PATH}"
        "-DLINK_DIRECTORIES=${ZIP_LIB_PATH}")

# See also
try_run(...)
```

Execution test

CMake can run specific executables and check their exit status to determine (un)successful runs.

```
include(CTest)
enable_testing()
add_test(NAME MyTest COMMAND my_test_executable)
```

How to structure a large project

```
./
  src/
    CMakeLists.txt
    my_lib.{hpp,cpp}
  apps/
    CMakeLists.txt
    my_app.cpp
  tests/
    CMakeLists.txt
    my_test.cpp
cmake/
  FindSomeLib.cmake
docs/
  Doxyfile.in
scripts/
  do_something.sh
.gitignore
README.md
LICENSE.md
CMakeLists.txt
```