Exercise session 08

Introduction to CMake.

Advanced Programming - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa

19 Nov 2024

Introduction to CMake

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.

Configure means:

- Meet dependencies
- Build
- Test

Build systems available

- **CMake**
  - **Pros**: Easy to learn, great support for multiple IDEs, cross-platform
  - **Cons**: Does not perform automatic compilation test for met dependencies.
- **GNU Autotools**
  - **Pros**: Excellent support for legacy Unix platforms, large selection of existing modules.
  - **Cons**: Slow, hard to use correctly, painful to debug, poor support for non-Unix platforms.
- **Meson**, **Bazel**, **SCons**, ...

Package managers:

- **Conan**, **vcpkg**, ...

Let's try

Install dependencies, then compile and install.

Doxygen (CMake)

```
cd /path/to/doxygen/src/
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/opt/doxygen ../
make -j<N>
(sudo) make install
```

GNU Scientific Library (autotools)

```
cd /path/to/gsl/src/
./configure --prefix=/opt/gsl --enable-shared --disable-static
make -j<N>
(sudo) make install
```

Why CMake?

- More packages use CMake than any other system
- Almost 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)
- ...

CMake basics

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

```
cmake_minimum_required(VERSION 3.12)

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

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

Command names are **case insensitive**.

CMake 101

- **Configure**

```
cd /path/to/src/
mkdir build && cd build
cmake .. [options...]
# Or:
# cmake -S /path/to/src/ -B /path/to/build/ [options...]
```

- **Compile**

```
 cd /path/to/build/
 make -j<N>
```

- **List variable values**

```
cd /path/to/build/
cmake /path/to/src/ -L
```

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 each other.

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

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

Target properties

Targets can be associated with various properties :

```
add_library(my_lib STATIC my_class.cpp my_class.hpp)
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_20)
# Last command is equivalent to:
# set_target_properties(my_exec PROPERTIES CXX_STANDARD 20)

target_compile_options(my_exec PUBLIC -Wall -Wpedantic)
```

Interacting with the outside world: local variables

```
set(LIB_NAME "my_lib")

# List items are space- or semicolon-separated.
set(SRCS "my_class.cpp;my_main.cpp")
set(INCLUDE_DIRS "include_one;include_two")

add_library(${LIB_NAME} STATIC ${SRCS} my_class.hpp)
target_include_directories(${LIB_NAME} PUBLIC ${INCLUDE_DIRS})

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

Interacting with the outside world: cache variables

Cache variables are used to interact 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 /path/to/src/ \
  -DMY_CACHE_VARIABLE="SOME_CUSTOM_VALUE" \
  -DMY_OPTION=OFF
```

Interacting with the outside world: environment variables

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

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

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

Control flow

```
if("${variable}") # Or if("condition").
    #
else()
    # Undefined variables would be treated as empty strings, 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 versions of any third-party library.

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

How to select the correct branch?

Pre-processor flags

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

Or let the 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()
```

Modify files depending on variables

`print_version.hpp.in`:

```
void print_version() {
  std::cout << "Version number: " << @MY_PROJECT_VERSION@
            << std::endl;
}
```

`CMakeLists.txt`:

```
set(MY_PROJECT_VERSION 1.2.0)

configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/print_version.hpp.in"
  "${CMAKE_CURRENT_BINARY_DIR}/print_version.hpp")
```

See also: `#cmakedefine`.

Print messages and debug

Content of variables is printed with

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

Error messages can be printed with

```
message(FATAL_ERROR "MY_VAR has the wrong value: ${MY_VAR}")
```

Commands being executed are printed with

```
cmake /path/to/src/ -B build --trace-source=CMakeLists.txt
make VERBOSE=1
```

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++")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY lib)
```

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/modules/")

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

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

```
cmake /path/to/src/ -DBOOST_ROOT=/path/to/boost/installation/
```

Using third-party libraries

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()
```

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)
```

Organize a large project

```
cmake_minimum_required(VERSION 3.12)
project(ExampleProject VERSION 1.0 LANGUAGES CXX)

find_package(...)
find_package(...)

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

Tip: how to organize a large project

Further readings

- Official documentation
- Modern CMake
- It's time to do CMake right
- Effective modern CMake
- More modern CMake

Exercise 1

1. Following `exercises/07/solutions/ex1`, configure and install `muParserX` on your system using the builtin `CMake` configurator.
2. Write a `CMake` script to compile and link the test code `ex1.cpp` against it.

Exercise 2

Re-do `exercises/07/solutions/ex3` with the help of CMake.