Exercise session 08

Introduction to CMake.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

19 Nov 2025

Introduction to CMake

Traditional approach: Makefiles

Pros

- Direct control over build process
- Universal on Unix-like systems
- Simple syntax for small projects
- No additional tools required

Cons

- Platform-dependent (Unix vs Windows)
- Manual dependency tracking
- Tab/space syntax issues
- Difficult to scale for large projects
- No automatic detection of third-party libraries

Build systems

Purposes

Modern build systems (CMake, Meson, etc.) **generate** Makefiles automatically while providing cross-platform portability and automatic dependency management.

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**: Less automatic dependency resolution compared to Autotools; requires explicit `find_package()` calls.
- **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**, ...

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)
- ...

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>            # Replace <N> with the number of parallel jobs (e.g., -j4),
                      # or use -j$(nproc) to automatically use all available cores.
(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
```

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...]
# Modern approach (CMake ≥ 3.13):
cmake -S /path/to/src/ -B /path/to/build/ [options...]
```

- **Compile (and install)**

```
cd /path/to/build/; make -j<N>; (sudo) make install
# Modern approach (CMake ≥ 3.13):
cmake --build /path/to/build/ -j<N>
cmake --install /path/to/build/
```

- **List variable values**

```
cmake -S /path/to/src/ -B /path/to/build -L
```

(⚠ Advanced) Build generators: Make vs Ninja

CMake doesn't build your project directly: it **generates build files** for other tools.

GNU Make (default on Unix)

```
cmake -S . -B build     # Generates Makefiles.
cmake --build build -j4 # Runs: make -j4.
```

Ninja (faster alternative)

```
cmake -S . -B build -G Ninja # Generates build.ninja.
cmake --build build          # Runs: ninja (auto-detects cores).
```

Why Ninja?

- Faster parallel builds, especially for large projects (better dependency tracking).
- Cleaner output.
- Designed to be generated (not hand-written).

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, as they are automatically detected.
add_executable(my_exec my_main.cpp my_header.hpp)

# STATIC: Archive linked at compile time (.a, .lib).
# SHARED: Dynamic library loaded at runtime (.so, .dll, .dylib).
# MODULE: Plugin loaded explicitly by application.
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)
target_link_libraries(my_lib PUBLIC another_lib)

add_executable(my_exec my_main.cpp my_header.hpp)
target_link_libraries(my_exec my_lib)
target_compile_features(my_exec cxx_std_20)
# Or: set_target_properties(my_exec PROPERTIES CXX_STANDARD 20)

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

Visibility keywords:

- `PRIVATE` : Used only by this target.
- `PUBLIC` : Used by this target and propagated to dependents.
- `INTERFACE` : Not used by this target, only propagated to dependents.

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.hpp)
target_link_libraries(my_exec ${LIB_NAME})
```

Cache variables

Cache variables are used to interact with the outside world through 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 -S /path/to/src/ \
      -DMY_CACHE_VARIABLE="SOME_CUSTOM_VALUE" \
      -DMY_OPTION=OFF
```

Interacting with 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

You can evaluates the condition argument of the `if` clause according to the condition syntax described below. If the result is true, then the commands in the if block are executed. Otherwise, optional `elseif` blocks are processed in the same way. Finally, if no condition is true, commands in the optional `else` block are executed.

```
if("${variable}") # Or if("condition").
    # Do something if true.
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.

Example: Choose between `std::array` and `std::vector` at compile time.

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

Question: How to select the correct branch?

Answer: Use preprocessor flags (see next slide).

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

CMake can generate files from templates by substituting variable values at configuration time.

`print_version.hpp.in`:

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

`CMakeLists.txt`:

```
set(MY_PROJECT_VERSION 2.1.5)

configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/print_version.hpp.in"
  "${CMAKE_CURRENT_BINARY_DIR}/print_version.hpp"
  @ONLY) # Only substitute @VAR@, not ${VAR}.
```

Note: Variables use `@VAR@` syntax in `.in` files. The generated file goes in the build directory, not the source directory.

Alternative: `#cmakedefine` for conditional compilation.

Print messages and debug

Content of variables is printed with:

```
message(STATUS "MY_VAR is: ${MY_VAR}")  # Informational message.
message(WARNING "MY_VAR might be incorrect: ${MY_VAR}")  # Warning.
message(FATAL_ERROR "MY_VAR has the wrong value: ${MY_VAR}")  # Error, stops configuration.
```

Commands being executed are printed with:

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

# Or, with modern CMake:
cmake --build build --verbose
```

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)
```

Installation prefix (1/2)

Control where your project gets installed using `CMAKE_INSTALL_PREFIX`:

```
# In CMakeLists.txt (set default if not specified by user):
if(CMAKE_INSTALL_PREFIX_INITIALIZED_TO_DEFAULT)
    set(CMAKE_INSTALL_PREFIX "/opt/myproject" CACHE PATH "Install prefix" FORCE)
endif()

# Install targets:
install(TARGETS my_exec my_lib
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib)

# Install headers:
install(DIRECTORY include/ DESTINATION include)
```

Installation prefix (2/2)

Command line usage:

```
cmake -S . -B build -DCMAKE_INSTALL_PREFIX=/custom/path
cmake --build build
cmake --install build  # Or: cd build && (sudo) make install
```

Default prefix: `/usr/local` (Unix), `C:/Program Files/${PROJECT_NAME}` (Windows).

Looking for third-party libraries

CMake looks for packages in two ways:

1. **Find modules**: `FindPackageName.cmake` files in `CMAKE_MODULE_PATH`.
2. **Config files**: `PackageNameConfig.cmake` provided by the package itself.

```
# Add custom module path:
list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

# Modern CMake (≥ 3.12) uses imported targets.
find_package(Boost 1.50 REQUIRED COMPONENTS filesystem graph)

target_link_libraries(my_lib PUBLIC Boost::filesystem Boost::graph)
```

If the library is not in a system folder, you can provide a hint:

```
cmake -S /path/to/src/ -DBOOST_ROOT=/path/to/boost/installation/ # Or:
cmake -S /path/to/src/ -DCMAKE_PREFIX_PATH="/path/to/boost;/path/to/other"
```

Using third-party libraries

Once the library is found, proper variables are populated.

```
if(Boost_FOUND) # Always TRUE, as the library is marked as REQUIRED.
    # Link directly against imported targets.
    target_link_libraries(my_lib PUBLIC
                          Boost::filesystem
                          Boost::graph)

    # Include directories and compiler flags are handled automatically!
endif()
```

Legacy approach:

```
if(Boost_FOUND)
    target_include_directories(my_lib PUBLIC ${Boost_INCLUDE_DIRS})
    target_link_directories(my_lib PUBLIC ${Boost_LIBRARY_DIRS})
    target_link_libraries(my_lib PUBLIC ${Boost_LIBRARIES})
endif()
```

Compilation and execution tests

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`.

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

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

Tip: how to organize a large project

```
📂 project/
├── 📂 apps/
│   ├── 📝 CMakeLists.txt
│   └── 📄 my_app.cpp
├── 📂 cmake/
│   └── 📝 FindSomeLib.cmake
├── 📑 doc/
│   └── 📝 Doxyfile.in
├── 📂 scripts/
│   └── 📝 do_something.sh
├── 📂 src/
│   ├── 📝 CMakeLists.txt
│   └── 📄 my_lib.{hpp,tpl.hpp,cpp}
├── 📂 tests/
│   ├── 📝 CMakeLists.txt
│   └── 📄 my_test.cpp
├── 📒 .gitignore
├── 📝 CMakeLists.txt
├── 💼 LICENSE.md
└── 📰 README.md
```

Organizing a large project

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

set(CMAKE_CXX_STANDARD 17) # Set C++ standard.
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(...)

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

Further readings

- Official documentation
- Modern CMake
- 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.