Introduction to CMake

Paolo Joseph Baioni

Politecnico di Milano

May 22, 2025

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 generators available

- **CMake**[1]
  - PRO: Easy to learn, great support for multiple IDEs, cross-platform
  - CON: Does not perform automatic compilation test for met dependencies.
- **GNU Autotools**[2]
  - PRO: Excellent support also for legacy Unix platforms, robust, large selection of existing modules.
  - CON: 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/

Leonardo Software Stack - 2024

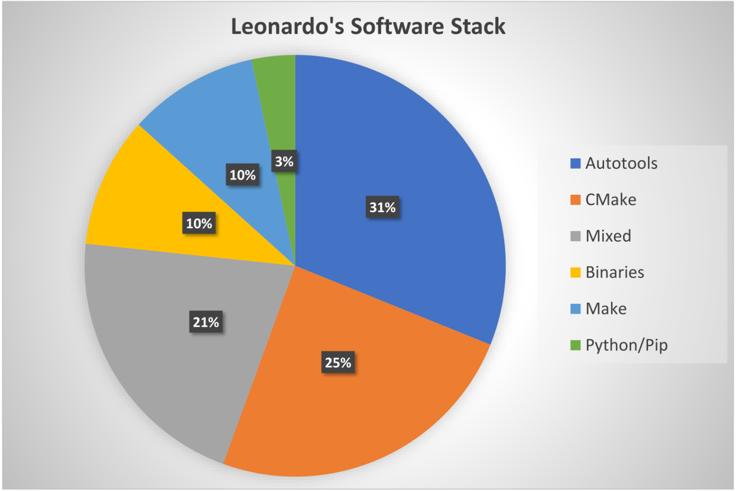

Figure: Leonardo@CINECA, top 9 hpc

Build Systems Google Trends - 2024

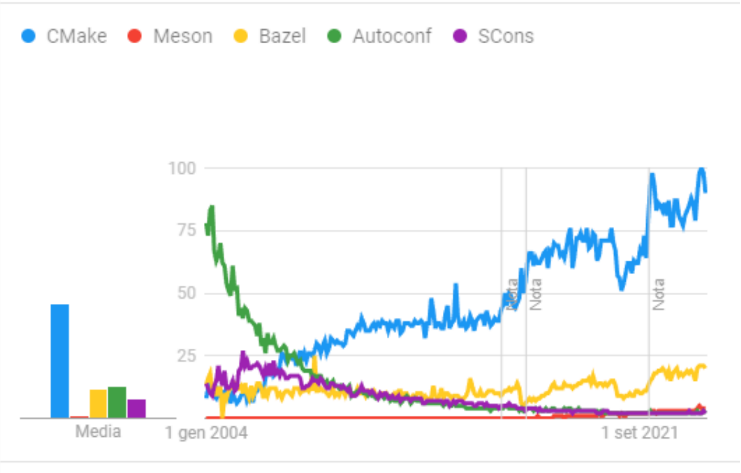

Figure: Google Trends as of 2024

Why CMake?

- Increasingly 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

Why CMake?

- Increasingly 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)
- . . .

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

Let's try

Unload the mk module system (module purge), install dependencies then compile and install.

{**fmt**} (https://github.com/fmtlib/fmt)

```
cd /path/to/fmt/src/
mkdir build && cd build
cmake ..
make -j<N>
make test
(sudo) make install
```

GNU Scientific Library (https://www.gnu.org/software/gsl/)

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

CMake 101

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 $\geq$ 3.0).

Command names are **case-insensitive**.

CMake 101

Configure:

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

Compile:

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

Remark: this works iff the generator is Unix Makefiles; otherwise, or for portability

```
cmake --build ./ build
```

To print a list of variable values:

```
cd 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.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 various properties[6]:

```
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_20)
# Last command is equivalent to
# set_target_properties(my_exec PROPERTIES CXX_STANDARD 20)
```

[6] https://cmake.org/cmake/help/latest/manual/cmake-properties.7.html

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 ${SRCS} my_class.h)
target_include_directories(${LIB_NAME} PUBLIC ${INCLUDE_DIRS})

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

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
```

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)
```

(although 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.

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 wrong value: ${MY_VAR}")
```

Commands being executed are printed with

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

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 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
```

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()
```

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()
```

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`.

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

```
# Set the minimum required version of CMake
cmake_minimum_required(VERSION 3.12)

# Set the project name and version
project(ScientificProject VERSION 1.0 LANGUAGES CXX)

# Define the source, include and test directories
set(SOURCE_DIR ${CMAKE_SOURCE_DIR}/src)
set(INCLUDE_DIR ${CMAKE_SOURCE_DIR}/include)
set(TEST_DIR ${CMAKE_SOURCE_DIR}/tests)

# Include directories
include_directories(${INCLUDE_DIR})

# Find and include external dependencies (e.g., Boost, Eigen)
find_package(...)
```

```
ScientificProject/
 CMakeLists.txt
 README.md
 LICENSE
 .gitignore
 cmake/
    FindSomeLib.cmake
 include/
    module1/
       module1.h
       module1_utils.h
    module2/
        module2.h
        module2_utils.h
 src/
    CMakeLists.txt
    main.cpp
    module1/
       CMakeLists.txt
       module1.cpp
       module1_utils.cpp
    module2/
       CMakeLists.txt
       module2.cpp
       module2_utils.cpp
 tests/
    CMakeLists.txt
    test_module1.cpp
    test_module2.cpp
    test_utils.cpp
 data/
    dataset1.csv
    dataset2.csv
 docs/
    API.md
    UserGuide.md
 scripts/
    do_something.sh
    another_script.py
```