Introduction to CMake

Jo-Frederik Krohn
5. April 2018

https://github.com/Norhk/cmake_demo

Why CMake

- Building C++ projects is horribly complicated because you have to specify all targets and their dependencies to the compiler

- Example:
  g++ target.cc -o exeName -I target.h dependency1.h […]
  -L /path/to/lib -INameOf.soFile(s) -Wall -Wextra -Wpedantic -O3
  -std=c++14 …

- Many compiler flags (architecture depending!!!)

- Includes can be thousands of files

- Library binaries/compilation units can be thousands (think of ATLAS, BELLE2 framework…)

Why CMake

- There is no way around specifying all of these files and options but there are build systems that can help to automate the process

- To name a few:
  - **Make**, Ninja, Automake …
  - SCons (python based, used to build Belle II framework)
- CMake

CMake

- OPEN SOURCE !
- CMake is a build system generator that builds **Make** files (by default)
- Industry standard (ROOT, Eigen, Qt, …)
- Many libraries offer CMake support (and should, if they don’t file a bug report to bully the developers to provide that)
- For the common ones that don’t offer direct support CMake has find scripts (OpenGL, Boost, …)
- CTest/CDash offer easy tools to test and log the results of your builds

Basic usage

- simple:
  >_ cmake . (creates makefile)
  >_ make (compiles)

- Options:
  >_ cmake -DCMAKE_CXX_COMPILER=clang++
  >_ make install (systemwide install, dangerous!)
  (default install path is /usr/local/)

- Configure via GUI:
  >_ ccmake

Targets and properties

- Binaries and executables are called target

- Create a target:
  add_executable(<targetname> <source>.cc)

- Set properties for a target:
  for example I want c++11 standards

```
set_property(TARGET <name> PROPERTY CXX_STANDARD 11)
```

- Set properties for all targets:
  set(CMAKE_CXX_FLAGS "-Wall -Wextra -Wpedantic”) #general flags
  set(CMAKE_CXX_FLAGS_DEBUG “-g") # for the debug build
  set(CMAKE_CXX_FLAGS_RELEASE “-O3") #“just for the release build”

If ...

```
# execute shell command and save output to file
EXECUTE_PROCESS( COMMAND uname -m OUTPUT_VARIABLE ARCHITECTURE )

# example if statement note that endif() is a function
if( ${ARCHITECTURE} STREQUAL "x86_64" )
  message("64bit")
elseif( ${ARCHITECTURE} STREQUAL "amd64")
  message("64bit")
else()
  message(“32bit")
endif()
```

CMake example structure

- CMake is a language steered by CMakeLists.txt-files and <name>.cmake scripts

- Each CMakeLists.txt defines a scope:

- /myProject/CMakeLists.txt
  /myProject/src/CMakeLists.txt
  /myProject/include/
  /myProject/build/
  /myProject/lib/CMakeLists.txt
  /myProject/lib/include/
  /myProject/lib/src/CMakeLists.txt
  /myProject/tests/CMakeLists.txt

- Main CMakeLists.txt specifies project properties and defines the hierarchy (/subfolder/CMakeLists.txt)

/myProject/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)
# define project name
project(test_cmake)

# define a variable in this (and its daughters) scope
# path to headers of my lib
set(MyLib_INCLUDES “${CMAKE_SOURCE_DIR}/lib/include/“)

# you can set environment vars like this
set(ENV{MY_ENV_VAR} “${CMAKE_SOURCE_DIR}/lib/include/“)

# adding the subdirectories
# if they have further subs dirs they have to be specified
# within the dirs cmakefile
add_subdirectory(src)
add_subdirectory(lib)
```

/lib/src/CMakelist.txt

```
# add the header dir so that the .cc can be compiled
include_directories(“${CMAKE_SOURCE_DIR}/lib/include”)

# define a variable (all compilation units I want in the lib)
set(MYLIB_SOURCES “${CMAKE_SOURCE_DIR}/lib/src/Logger.cc”)
```

```
# define the library
add_library(MyLib STATIC “${MYLIB_SOURCES}”)
```

STATIC -> lib<name>.a for archive (though you might expect .o)
SHARED -> lib<name>.dylib dynamic (though you might expect .so)

/src/CMakelist.txt

```
# path to your main project headers
include_directories(“${CMAKE_SOURCE_DIR}/include”)

# print to console, useful for debugging…
message(STATUS “CMAKE_SOURCE_DIR=${CMAKE_SOURCE_DIR}”)

add_executable(main main.cc) # meaning main is the “target”

# make will put the binary in the RUNTIME_OUTPUT_DIRECTORY
# which is the .cc location by default
set_target_properties(main PROPERTIES
RUNTIME_OUTPUT_DIRECTORY ${CMAKE_SOURCE_DIR}/build)

# add your on library located in ${CMAKE_SOURCE_DIR}/lib
include_directories(${MyLib_INCLUDES})
target_link_libraries(main MyLib)

# defines target for make install
INSTALL(TARGETS main DESTINATION ${test_cmake_SOURCE_DIR}/
build)
```

Linking ROOT

```
# Add ROOTSYS to the search path for make scripts
list(APPEND CMAKE_PREFIX_PATH $ENV{ROOTSYS})

# Find root package
# defines ROOT_LIBRARIES, ROOT_INCLUDE_DIRS (and much more)
find_package(ROOT REQUIRED COMPONENTS RIO Net MINUIT)
# in this case I just want the 3 libs above

# include the corresponding headers
include(${ROOT_USE_FILE})

# link libraries to my executable “main”
target_link_libraries(main ${ROOT_LIBRARIES})
```

Testing

- Write tests in /tests, use **gtest** for example

- Add the following to your project CMakeList.txt :

```
enable_testing()
add_subdirectory(tests)
```

- In tests/CMakeList.txt add

```
add_executable(test1 test_minuit.cc)
add_test(AllTests test1)
```

- Run “**ctest**” in the directory where you specified “enable testing”

Run test with each build

```
add_executable(test1 test_minuit.cc)
set(ALL_TESTS test1)
add_test(NAME ${ALL_TESTS} COMMAND ${ALL_TESTS})

add_custom_command(
     TARGET ${ALL_TESTS}
     COMMENT "Run tests"
     POST_BUILD                 <- define a post build command
     WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
     COMMAND ${CMAKE_CTEST_COMMAND} -C $<CONFIGURATION> -R "^$
{ALL_TESTS}$" --output-on-failures
)
```

Thank you

- Examples with make 3.1 as shipped with homebrew on osx

- For repo and slides:
https://github.com/Norhk/cmake_demo