CMake Build System

Introduction to the CMake build system

V2.1, SOTR 22/23

Paulo Pedreiras

Introduction

- CMake is a tool to manage building of source code – **build system generator**.
- CMake was originally designed as a generator for various variants of Makefile, but nowadays CMake generates modern build systems such as Ninja as well as project files for IDEs such as Visual Studio.
- CMake is normally used for C/C++ languages, but it can be used to build source code of other languages too.

- **Basically, CMake takes plain text files that describe a project as input and produces project files or makefiles for use with a wide variety of native development tools.**

Introduction

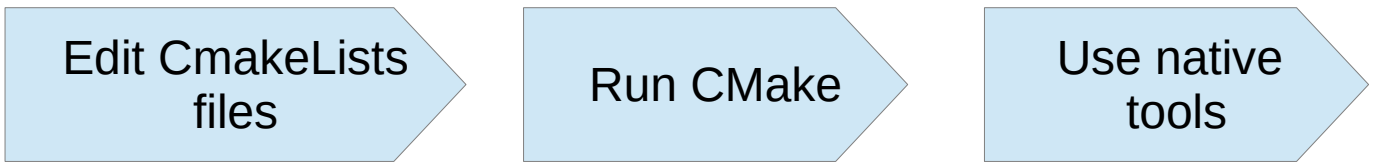
- Why CMake
  - Easy to use and works well
    - For large projects it is much simpler than typing Makefiles!
  - Cross-platform
    - Linus, Windows, OSx, …
    - Makefiles (Unix, Borland, ..), Ninja, MSBuild
    - IDEs: Code::Blocks, Eclipse CDT, Visual Studio, KDevelop, ...
  - Popular, both in scientific and industry communities
  - Allows building a directory tree outside of the source tree: “Out-of-source builds”
    - Source tree: project source code
    - Bin directory/Build directory: where CMake will put the resulting object files, libraries, and executables
    - CMake does not write any files to the source directory, only to the binary directory
  - Etc.

Installation

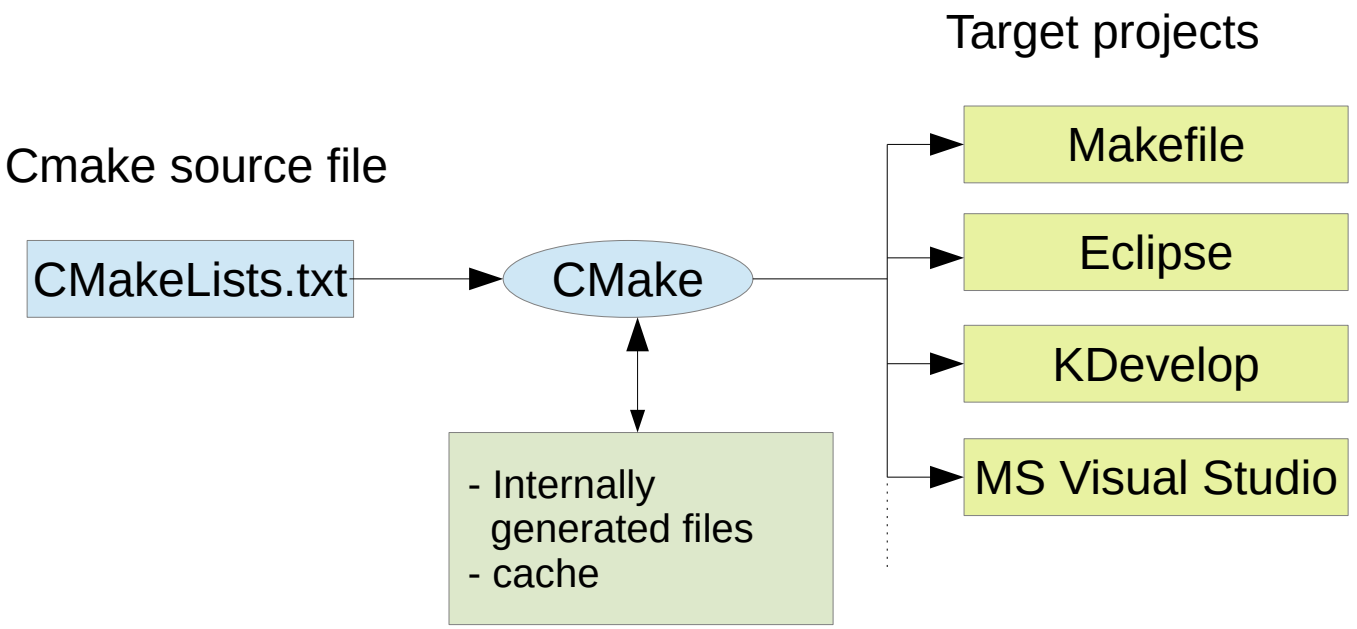
- Available at: https://cmake.org/download/
  - Source distributions for Unix/Linux and Windows
- Binary distributions for
  - Windows X64
  - Windows I386
  - Mac OS 10.10/10.13
  - Linux x86_64
- For Linux it is recommended to use the package manager of your distribution

Build process

- Workflow – 3 steps
  1) Define the project in one or more CMakeLists files
  2) Run cmake to generate/configure the native build system files
  3) Open the project files and use the native build tools



Build process



CmakeLists files

- CMakeLists files
  - Input plain text files that contain the project description in CMake's Language.
  - Contain project parameters and describe the flow control of the entire build process
  - CMake's language:
    - series of comments, commands and variables
  - Example: simplest possible "CMakeLists.txt" file

```
cmake_minimum_required(VERSION 3.20)

# set the project name
project(HelloWorld)

# add the executable
add_executable(Hello Hello.c)
```

Command that allows projects to require a minimum CMake version

Lines starting with "#" are (optional) comments

Command that sets the project name. May specify other options such as language or version. Causes CMake to generate a top-level Makefile or IDE project file

Command to generate executable (Hello) using a given source file (Hello.c)

Configuration and generation

- After having the source files and CmakeLists.txt set, it is time to configure and generate the build system
  - Via GUI
    - cmake-gui (requires QT)
      - Select the source code folder
      - Select the build directory (if it does not exist it is created automatically)
      - Then hit “Configure”, select the generator (e.g. Unix Makefiles) and the compiler/toolchain
      - Eventually adjust any variables
        - E.g. CMAKE_BUILD_TYPE can be “Debug”, “Release”, … (details latter on)
        - Configure again after changing a variable
      - Finally hit “Generate”
    - Go to the build dir and “make” the project

Configuration and generation

- Via curses interface
  - Most Linux/UNIX platforms support the curses library, enabling the use of ccmake.
    - Simple text-based interface
    - Go to the build dir
    - Run ccmake indicating as argument the directory where the sources are placed
    - Configure (“c”)
    - Edit the variables, if needed
    - Configure and generate (“c” then “g”)
    - “make” the project

Configuration and generation

- Via command line
  - cmake can be called form the command line to generate a project build system.
  - Best suited for projects with few or no options. For larger projects the use of ccmake or cmake-gui is recommended.
  - To build a project with cmake:
    - First create and change to the build dir
    - Run cmake specifying the path to the source tree and passing in any options using the -D flag.
      - Unlike ccmake, or the cmake-gui, the configure and generate steps are combined into one when using command line cmake.
      - E.g. 1: in build folder and assuming source is in the parent folder just type **`$cmake ..`**
      - E.g. 2: in the top level directory type **`$cmake -S . -B build`**

Configuration and generation

- A few important configuration options
  - Compiler
    - For Makefile-based generators, CMake tries a list of usual compilers until it finds a working one.
      - The desired compiler can be set by the user via environment variables
        - **CC** environment variable specifies the C compiler
        - **CXX** specifies the C++ compiler.
        - The user can specify the compiler directly on the command line by using “CMAKE_CXX_COMPILER”
          - E.g. “-DCMAKE_CXX_COMPILER=gcc”
      - Once cmake has been run and picked a compiler, if you wish to change the compiler, start over with an empty binary directory
    - The flags for the compiler and the linker can also be changed by setting environment variables:
      - Setting LDFLAGS initialize the cache values for link flags
      - CXXFLAGS and CFLAGS initialize CMAKE_CXX_FLAGS and CMAKE_C_FLAGS respectively.

Configuration and generation

- A few important configuration options (cont)
  - Build configurations: allow a project to be built in different ways.
  - CMake supports, by default:
    - **Debug** - basic debug flags turned on
    - **Release** - Release has the basic optimizations turned on
    - **MinSizeRel** - smallest object code
    - **RelWithDebInfo** - optimized build with debug information
  - For Makefile-based generators:
    - Only one configuration can be active at the time CMake
    - Configuration is specified with the CMAKE_BUILD_TYPE variable (can be empty)
    - Multiple configurations are managed by means of different build dirs
      - E.g. via command line: `$cmake .. -DCMAKE_BUILD_TYPE=Debug`

Writing CmakeLists files

- CMakeLists files are simple text files
  - Any editor can be used
  - Some do provide syntax highlight (e.g. vim, emacs, geany)
- CMakeLists files determine everything from which options to present to users, to which source files to compile.
- When the CMakeLists file is modified there are rules that automatically invoke CMake to update the generated files (e.g. Makefiles or project files)

Writing CmakeLists files

- The CMake language is composed of **comments**, **commands**, and **variables**.
- **Comments**:
  - start with # and run to the end of the line. Meaning and use are obvious ...
- **Variables**:
  - CMakeLists files use variables much like any programming language
  - Variable names are case sensitive and may only contain alphanumeric characters and underscores
  - There are several variables automatically defined by Cmake
    - These variables begin with “CMAKE_”. Don’t use this naming convention for variables specific to your project.
  - Variable contents are defined via the “set” command

Writing CmakeLists files

- **Variables (cont)**:
  - "set" command
    - The first argument to set is the name of the variable and the rest of the arguments are the values.
    - Multiple value arguments are packed into a semicolon-separated list and stored in the variable as a string.
    - Escape character is "\"

```
set(Foo "")           # 1 quoted arg -> value is ""
set(Foo a)            # 1 unquoted arg -> value is "a"
set(Foo "a b c") # 1 quoted arg -> value is "a b c"
set(Foo a b c)    # 3 unquoted args -> value is "a;b;c"
set(VAR "a\\b\\c and \"embedded quotes\"")
message(${VAR})       # "a\b\c and "embedded quotes""
```

Writing CmakeLists files

- **Variables (cont)**:
  - Referencing variables
    - Variables may be referenced in command arguments using syntax **${VAR}** where VAR is the variable name
    - Replacement is performed prior to the expansion of unquoted arguments

```
set(Foo a b c)    # 3 unquoted args -> value is "a;b;c"
command(${Foo})   # unquoted arg replaced by a;b;c
                  # and expands to three arguments
command("${Foo}") # quoted arg value is "a;b;c"
```

Writing CmakeLists files

- **Variables (cont)**:
  - Referencing variables (cont)
    - Environment variables can be accessed via the syntax **$ENV{VAR}**
  - Variable scope
    - When a variable is set it is visible:
      - In the current CMakeLists file or function and any subdirectory's CMakeLists files,
      - Any functions or macros that are invoked, and
      - Any files that are included using the include command.
    - Any new variables created in the child scope, or changes made to existing variables, will not impact the parent scope, unless explicitly stated (PARENT_SCOPE argument of set command, example in a latter slide).

Writing CmakeLists files

- **Variables (cont)**:
  - Variable scope: example

```
function(foo)
  message(${test}) # test is 1 here
  set(test 2)
  message(${test}) # test is 2 here, but only in this scope
endfunction()

set(test 1)
foo()
message(${test}) # test will still be 1 here
```

Writing CmakeLists files

- **Commands**:
  - A command consists of the command name, opening parenthesis, white space separated arguments, and a closing parenthesis: **cmd_name(arg1 arg2 ...)**
  - Each command is evaluated in the order that it appears in the CMakeLists file
  - Command names are not case-sensitive. Convention is using lowercase
  - Command arguments:
    - Are space separated and case sensitive
    - May be either quoted or unquoted.
    - A quoted argument starts and ends in a double quote (“) and always represents exactly one argument. Any double quotes contained inside the value must be escaped with a backslash.

Writing CmakeLists files

- **Commands (cont)**:
  - Example:

```
command("")          # 1 quoted argument
command("a b c")     # 1 quoted argument
command("a;b;c")     # 1 quoted argument
command("a" "b" "c") # 3 quoted arguments
command(a b c)       # 3 unquoted arguments
command(a;b;c)       # 1 unquoted argument expands to 3
```

Writing CmakeLists files

- Basic commands
  - **set** and **unset**: explicitly set or unset variables.
  - **string**, **list**, and **separate_arguments**: basic manipulation of strings and lists.
  - **add_executable** and **add_library**: main commands for defining the executables and libraries to build, and which source files comprise them
- **Flow Control**
  - Conditional statements, looping constructs and procedure definitions
  - **If/elseif**

```
if(MSVC80)
  # do something here
elseif(MSVC90)
  # do something else
elseif(APPLE)
  # do something else
endif()
```

Writing CmakeLists files

- **Flow Control (cont)**
  - **foreach** and **while**
    - These commands allow handling repetitive tasks that occur in sequence.
    - The break command breaks a loop before normal end.
    - The first argument of **foreach** is the name of the variable and the remaining arguments are the list of values over which to loop

```
foreach(tfile
        TestButterworthLowPass
        TestButterworthHighPass
        )
  add_test(${tfile}-image ${VTK_EXECUTABLE}
    ${VTK_SOURCE_DIR}/Tests/rtImageTest.tcl
    ….
    )
endforeach()
```

Writing CmakeLists files

- **Flow Control (cont)**
  - **foreach** and **while**
    - The **while** command provides looping based on a test condition.
    - The format for the test expression in the while command is the same as it is for the if command

```
#####################################################
# run paraview and ctest test dashboards for 6 hours
#
while(${CTEST_ELAPSED_TIME} LESS 36000)
  set(START_TIME ${CTEST_ELAPSED_TIME})
  ctest_run_script("dash1_ParaView_vs71continuous.cmake")
  ctest_run_script("dash1_cmake_vs71continuous.cmake")
endwhile()
```

Writing CmakeLists files

- **Procedure definitions**
  - Macro and function commands support repetitive tasks scattered throughout CMakeLists files.
  - Macro or function can be used by any CMakeLists files processed after its definition.
  - A Cmake **function** resembles a C/C++ function
  - Functions can have arguments that become variables within the function. Standard variables such as ARGC, ARGV, ARGN, and ARGV0, ARGV1 are also defined
  - Note the use of the PARENT_SCOPE keyword inside the function DetermineTime

```
function(DetermineTime _time)
  # pass the result up to whatever invoked this
  set(${_time} "1:23:45" PARENT_SCOPE)
endfunction()

# now use the function we just defined
DetermineTime(current_time)

if(DEFINED current_time)
  message(STATUS "The time is now: ${current_time}")
endif()
```

Writing CmakeLists files

- **Procedure definitions**
  - Macros are defined and called in the same manner as functions.
  - The main differences are that macros do not create a new variable scope and arguments to a macro are not treated as variables but as strings replaced prior to execution.
    - Very similar to the differences between a macro and a function in C or C++

```
# define a simple macro
macro(assert TEST COMMENT)
  if(NOT ${TEST})
    message("Assertion failed: ${COMMENT}")
  endif()
endmacro()

# use the macro
find_library(FOO_LIB foo /usr/local/lib)
assert(${FOO_LIB} "Unable to find library foo")
```

Cmake Cache

- The CMake cache may be thought of as a configuration file.
- The first time CMake is run on a project, it produces a CMakeCache.txt file in the top directory of the build tree.
- CMake uses this file to store a set of global cache variables, whose values persist across multiple runs within a project build tree.
- There are a few purposes of this cache:
  - **Store** user's **selections** and **choices**, so that if they should run CMake again they will not need to reenter that information.
    - For example, the option command creates a Boolean variable and stores it in the cache.
    - The user can set that variable from the user interface and its value will remain in case CMake is executed again in the future.

```
option(USE_JPEG "Do you want to use the jpeg library")
```

    - Variable in cache can also be created via the standard set command with the CACHE option

```
set(USE_JPEG ON CACHE BOOL "include jpeg support?")
```

Cmake Cache

- There are a few purposes of this cache (cont):
  - **Persistently store values** between CMake runs
    - These entries may not be visible or adjustable by the user. Typically, these values are system-dependent variables that require CMake to compile and run a program to determine their value.
    - Once these values have been determined, they are stored in the cache to avoid having to recompute them every time CMake is run.
    - **If you significantly change your computer (e.g. changing the OS or compiler) the cache file and likely all of your binary tree's object files, libraries, and executables must be deleted.**
- Restricting a cache entry to a limited set of predefined options.
  - Can be done by setting the STRINGS property on the cache entry. When cmake-gui is run and the user selects the variable cache entry from a pulldown list

```
set(CRYPTOBACKEND "OpenSSL" CACHE STRING "Select a cryptography backend")
set_property(CACHE CRYPTOBACKEND PROPERTY STRINGS
             "OpenSSL" "LibTomCrypt" "LibDES")
```

Basic usage

- The most basic project is an executable built from source code files.
- For simple projects, a three line CMakeLists.txt file is all that is required.

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(SimpleProj)

# set the executable and source file(s)
add_executable(simple src/main.c)
```

Basic usage – Structuring SW

- Projects should be structured using modules/libraries
- Lets assume that each library is stored into a sub-directory.
  - Each of these sub-directories contains the corresponding “.h” and “.c” files
  - Each of these sub-directories must have its own CmakeLists.txt defining the library name and the source file.
    - E.g. add_library(MovAvgLib movavg.c)
- Then it must be added a **add_subdirectory** call in the top-level CMakeLists.txt file, so that the library is built
- It is also necessary to add the new library to the include directory so that the “.h” header file can be found.
- CmakeLists example in next slide.
  - Adding a simple MovAvg library (“.c”, “.h” and library “CmakeLists.txt” in sub-directory MovAvgLib)

Basic usage – adding a library

Root CMakeLists.txt

```
...
# Add the MovAvgLib folder to the build. CMake will look for a
#   CMakeLists.txt in that folder and will process it
add_subdirectory(MovAvgLibFolder)


# Generate executable "libdemo" from source file "main.c"
add_executable(libdemo src/main.c)


# Generate variables with set of libs and include folders
# Dependencies are automatically set
list(APPEND EXTRA_LIBS MovAvgLib)
list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MovAvgLibFolder")


# Specify libraries or flags to use when linking a given target and/or its dependents
target_link_libraries(libdemo PUBLIC ${EXTRA_LIBS})


# Specifies include directories to use when compiling a given target
target_include_directories(libdemo PUBLIC ${EXTRA_INCLUDES})
```

Library CMakeLists.txt

```
# Set the source files that make up the library
set(MOVAVG_SRCS
   movavg.h movavg.c
)

# Set the library type (can be static or dynamic )
SET (LIB_TYPE STATIC)

# Create the library
add_library (MovAvgLib ${LIB_TYPE} ${MOVAVG_SRCS})
```

Basic usage – Generating documentation via Doxygen

Root CMakeLists.txt

```
...
project(LibExample)

# Ask if the user wants to generate documentation
option(GENERATE_DOC "Generate Doxygen documentation for the project?")
if(GENERATE_DOC)
	# Generate documentation if Doxygen is found
	find_package(Doxygen REQUIRED dot OPTIONAL_COMPONENTS mscgen dia)
	if(Doxygen_FOUND)
		message(STATUS "Doxygen was found, will build docs")
		add_subdirectory(docs)
	else()
		message(STATUS "Doxygen not found, not building docs")
	endif()
Endif()
...
add_subdirectory(MovAvgLibFolder)
...
```

CmakeLists.txt in “docs” folder

```
set(DOXYGEN_EXTRACT_ALL YES)
set(DOXYGEN_BUILTIN_STL_SUPPORT YES)

# set the documentation target name and files to include
doxygen_add_docs(LibDemoDoc
    "${PROJECT_SOURCE_DIR}/mainpage.md"
    "${PROJECT_SOURCE_DIR}/MovAvgLibFolder" )
```

Notes:

- Documentation files are created inside the build directory
- GENERATE_DOC must be set. E.g.:
  - **$cmake .. -DGENERATE_DOC=ON**
- Documentation must be explicitly generated **$make LibDemoDoc** in folder “build”
- See “mainpage.md” in the example for more detailed instructions

Basic usage – Unity Unit Test

Directory structure\

```
root---|
       |-- src             // Module to test (.c and .h)
       |-- unity_src       // Unity source files
       |-- test            // Test .c file

Note 1: this files are just examples. Have been created to
be generic, requiring minimal modifications.
Note 2: make test executes the test file
```

Root CMakeLists.txt

```
# Add CTest, which is the CMake test driver program
include(CTest)

# Add directories with the source code to test,
#    Unity and test code
add_subdirectory(src)
add_subdirectory(unity_src)
add_subdirectory(test)
```

“src” CMakeLists.txt

```
file(GLOB SOURCES ./*.c)

add_library(src STATIC ${SOURCES})

target_include_directories(src PUBLIC .)
```

“test” CMakeLists.txt

```
add_executable(testMyVectorLibApp testMyVectorLib.c)

target_link_libraries(testMyVectorLibApp src unity)

add_test(MyVectorLibTest testMyVectorLibApp)
```

“unity src” CMakeLists.txt

```
add_library(unity STATIC unity.c)

target_include_directories(unity PUBLIC .)
```

Finaly remarks and bibliography

- CMake is a complete, powerful, rich and complex tool!
- This presentation just scratches the surface. But it should allow you to understand and use CMake-based development frameworks such as ExpressIF (ESP32), Nordic nRF and several others, as well as to carry out some basic but common tasks
- There are many books and tutorials on CMake.
- The following are highly recommended (first to are part of the official documentation):
  - https://cmake.org/cmake/help/book/mastering-cmake/
  - https://cmake.org/cmake/help/book/mastering-cmake/cmake/Help/guide/tutorial/index.html#
  - https://cliutils.gitlab.io/modern-cmake/
  - https://github.com/toeb/moderncmake