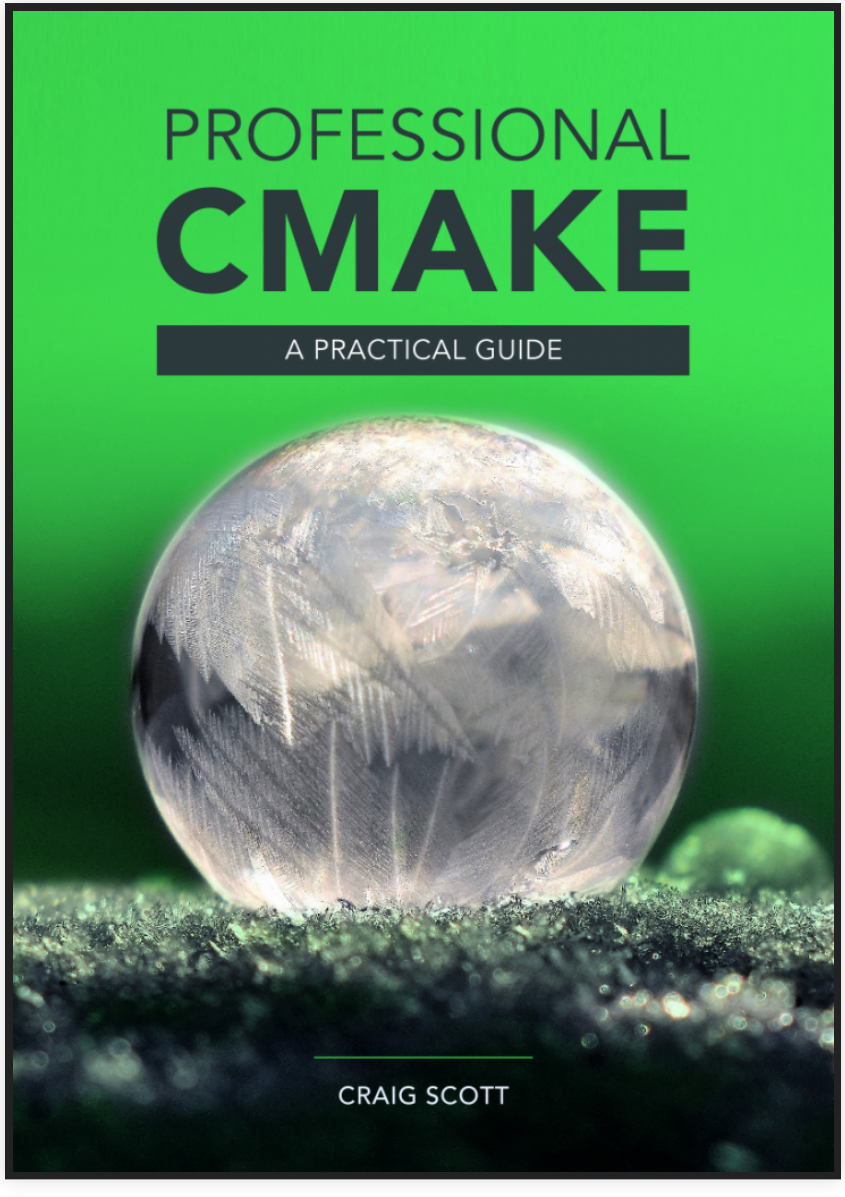
Modern CMake

An Introduction

CppMaryland

2020-03-05

Eric Richardson



Credits

Most of my references came from the book Professional CMake, by Craig Scott

Additional references are at the end of this presentation

Simple Examples

Add an executable

```
cmake_minimum_required(VERSION 3.15)
project(HelloWorld)

add_executable(App main.cpp)
```

- The top-level `CMakeLists.txt` **must** start with the `cmake_minimum_required(VERSION <X>)`
- The `project` should also be set
- The first argument to `add_executable` (`App`) is the **target** to build
- It is followed by the **source** file(s) to build from

main.cpp:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!\n";
}
```

Generating and running:

```
$ ls -1
  CMakeLists.txt
  main.cpp
$ mkdir build && cd build/
$ cmake .. -GNinja
$ ninja
$ ./App
Hello, world!
```

Directory created:

```
+build
 CMakeLists.txt
 main.cpp
```

cmake run:

```
+build/build.ninja
+build/CMakeCache.txt
+build/CMakeFiles
+build/CMakeFiles/3.17.20200216-g333a050
+build/CMakeFiles/3.17.20200216-g333a050/CMakeCXXCompiler.cmake
 ...
+build/CMakeFiles/3.17.20200216-g333a050/CompilerIdCXX/tmp
+build/CMakeFiles/App.dir
+build/CMakeFiles/cmake.check_cache
+build/CMakeFiles/CMakeOutput.log
+build/CMakeFiles/CMakeTmp
+build/CMakeFiles/rules.ninja
+build/CMakeFiles/TargetDirectories.txt
+build/cmake_install.cmake
```

`ninja` run:

```
+build/CMakeFiles/App.dir/main.cpp.o
+build/.ninja_log
+build/App
+build/.ninja_deps
```

All options to `add_executable`

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               [source1] [source2 ...])
```

- `<name>` : the name of the target to build into an executable
- `WIN32` and `MACOSX_BUNDLE` : OS-specific
- `EXCLUDE_FROM_ALL` : prevent a top-level `make all` (or similar) from building this target

Header files can be found automatically, and are not usually listed

The suffix of an executable is OS-specific:

- On Windows, target `App` will become `App.exe`
- On Linux, there is no suffix, so it's just `App`

This can be overridden with setting `CMAKE_EXECUTABLE_SUFFIX`:

```
set(CMAKE_EXECUTABLE_SUFFIX .bin)
```

Now target `App` will produce executable `App.bin`

You do not have to run `cmake` again, even though you changed the `CMakeLists.txt`:

```
$ ninja
[0/1] Re-running CMake...
-- Configuring done
-- Generating done
-- Build files have been written to: </build/dir>
[1/1] Linking CXX executable App.bin
```

The new application was built:

```
$ ls -1
App
App.bin
build.ninja
CMakeCache.txt
CMakeFiles
cmake_install.cmake
```

Add a library

The syntax to add a library is very similar to that used to add an executable:

```
add_library(Cat cat.cpp)
```

- The first argument (`Cat`) is still the name of the **target**
- The **source** file(s) follow

All options to `add_library`

```
add_library(<name> [STATIC | SHARED | MODULE | OBJECT]
            [EXCLUDE_FROM_ALL]
            [source1] [source2 ...])
```

- `<name>` : the name of the target
- The **Library Type** comes next:
  - `STATIC` : a static library
  - `SHARED` : a shared library
  - `MODULE` : a module (plug-in) library
  - If the type is omitted, the `BUILD_SHARED_LIBS` variable will be used to decide between `STATIC` and `SHARED`
- `EXCLUDE_FROM_ALL` : same as for `add_executable(...)`

Linking a library to an executable

If `App` depends on `Cat`:

```
target_link_libraries(App PUBLIC Cat)
```

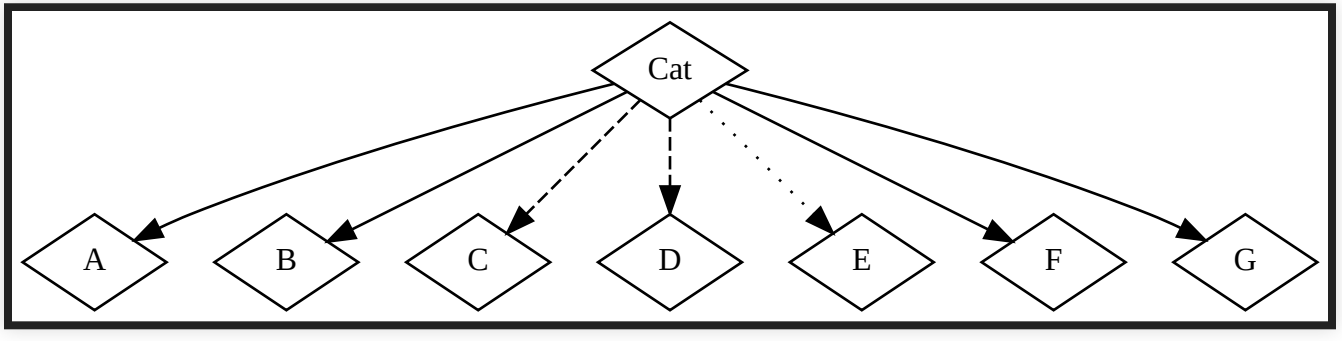
- `PUBLIC` : the symbols in `Cat`, **and** any of the symbols in libraries that `Cat` depends on, are visible to `App`
- `target_link_libraries` must come *after* the target is created
- Can be used for applications, libraries, or other custom targets

All options to `target_link_libraries`

```
target_link_libraries(<target>
                      <PRIVATE|PUBLIC|INTERFACE> <item>...
                     [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

A complex structure such as this is allowed:

```
target_link_libraries(Cat
    PUBLIC    A B
    PRIVATE   C D
    INTERFACE E
    PUBLIC    F G
    )
```



- `PUBLIC` produces solid lines
- `PRIVATE` produces dashed lines

Including files and directories

- Use the `include` command to add additional CMake scripts
- Use the `add_subdirectory` command to delve into directories

```
add_subdirectory(dir1)
include(dir2/HelperScript.cmake)
```

Options for add_subdirectory

```
add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
```

Options for `include`

```
include(<file|module> [OPTIONAL] [RESULT_VARIABLE <var>]
                      [NO_POLICY_SCOPE])
```

- If `OPTIONAL` is present, no error will be raised if the file or module is not found
- `RESULT_VARIABLE` will set `<var>` to the full filename, or `"NOTFOUND"`
- The `NO_POLICY_SCOPE` option is useful if you actually want to pull policies from an external file

Messages

One important way to debug `CMakeLists.txt` files is by using the `message` command:

```
message([<mode>] "message text" ...)
```

Available modes:

- TRACE
- DEBUG
- VERBOSE
- STATUS
- NOTICE
- DEPRECATION
- AUTHOR_WARNING
- WARNING
- SEND_ERROR
- FATAL_ERROR

Example:

```
message(STATUS      A)
message(NOTICE      B)
message(FATAL_ERROR C)
message(STATUS      D)
```

Produces output:

```
$ cmake -GNinja ..
...
-- A
B
CMake Error at CMakeLists.txt:5 (message):
  C
-- Configuring incomplete, errors occurred!
```

To make absolutely every message from CMake visible:

```
cmake --log-level=TRACE ...
```

At least on my Linux system:

- `STATUS` messages and below get sent to `stdout`
- `NOTICE` messages and above get sent to `stderr`

Variables

There are two kinds of variables:

- **Normal** variables : internal use only
- **Cache** variables : can be set from outside the project

The way they are set and used is very similar:

```
set(NormalVariable "A normal variable")
set(CacheVariable  "A cache variable" CACHE STRING "Some description"
```

Several options for set

```
# Set a normal variable
set(<variable> <value>... [PARENT_SCOPE])

# Set a cache variable (cache entry)
set(<variable> <value>... CACHE <type> <docstring> [FORCE])
# (Where <type> is one of BOOL, FILEPATH, PATH, STRING, or INTERNAL)

# Set an environment variable directly
set(ENV{<variable>} [<value>])
```

Running the previous `CMakeLists.txt`:

```
-- NormalVariable = A normal variable
-- CacheVariable = A cache variable
```

Only the **cache** variable is stored in `CMakeCache.txt`:

```
$ grep -IrE "Variable" -C1
CMakeCache.txt-//Some description
CMakeCache.txt:CacheVariable:STRING=A cache variable
CMakeCache.txt-
```

Here is the output from a few different builds

```
$ cmake .. -DCacheVariable:STRING=ShortStringNoSpaces
-- CacheVariable = ShortStringNoSpaces
$ grep Variable CMakeCache.txt
CacheVariable:STRING=ShortStringNoSpaces

$ cmake .. -DCacheVariable:STRING="A string with spaces"
-- CacheVariable = A string with spaces
$ grep Variable CMakeCache.txt
CacheVariable:STRING=A string with spaces

$ cmake .. -DCacheVariable:STRING=String\ with\ escaped\ spaces
-- CacheVariable = String with escaped spaces
$ grep Variable CMakeCache.txt
CacheVariable:STRING=String with escaped spaces
```

Note that once a cache variable is set, it never resets to what is in the `CMakeLists.txt`

The only way to get it back is to modify or delete `CMakeCache.txt`:

```
$ cmake .. -DCacheVariable:STRING="Permanent override"
-- CacheVariable = Permanent override

$ cmake ..
-- CacheVariable = Permanent override

$ rm CMakeCache.txt

$ cmake ..
-- CacheVariable = A cache variable
```

More about `CMakeCache.txt`

Sits at the top-level of the build directory, and contains all of the information about the build

Modifying (or even `touch`ing) this file will result in every `CMakeLists.txt` file being processed again the next time a configure or generate step runs

Setting variables

A variable can hold one of the following:

- A string
- A list
- A boolean

Setting `string` variables

```
# Manipulation
  string(TOLOWER     <string> <out-var>)
  string(TOUPPER     <string> <out-var>)
  string(LENGTH      <string> <out-var>)
  string(STRIP       <string> <out-var>)
  string(GENEX_STRIP <string> <out-var>)
  string(APPEND      <string-var> [<input>...])
  string(PREPEND     <string-var> [<input>...])
  string(CONCAT      <out-var> [<input>...])
  string(JOIN        <glue> <out-var> [<input>...])
  string(SUBSTRING   <string> <begin> <length> <out-var>)
  string(REPEAT      <string> <count> <out-var>)
```

More `string` operations:

```
# Search and Replace
  string(FIND    <string> <substring> <out-var> [...])
  string(REPLACE <match-string> <replace-string> <out-var> <input>...

# Regular Expressions
  string(REGEX MATCH    <match-regex> <out-var> <input>...)
  string(REGEX MATCHALL <match-regex> <out-var> <input>...)
  string(REGEX REPLACE  <match-regex> <replace-expr> <out-var> <input

# Comparison
  string(COMPARE <op> <string1> <string2> <out-var>)

# Hashing
  string(<HASH> <out-var> <input>)
```

Even more `string` operations:

```
# Generation
  string(ASCII <number>... <out-var>)
  string(CONFIGURE <string> <out-var> [...])
  string(MAKE_C_IDENTIFIER <string> <out-var>)
  string(RANDOM [<option>...] <out-var>)
  string(TIMESTAMP <out-var> [<format string>] [UTC])
  string(UUID <out-var> ...)
```

Setting `list` variables

Lists are separated by semicolons, and there are a few useful commands that you can use to operate on them:

```
Modification
  list(APPEND <list> [<element>...])
  list(PREPEND <list> [<element>...])
  list(POP_BACK <list> [<out-var>...])
  list(POP_FRONT <list> [<out-var>...])
  list(INSERT <list> <index> [<element>...])
  list(REMOVE_ITEM <list> <value>...)
  list(REMOVE_AT <list> <index>...)
  list(REMOVE_DUPLICATES <list>)
  list(TRANSFORM <list> <ACTION> [...])
  list(FILTER <list> {INCLUDE | EXCLUDE} REGEX <regex>)
```

More `list` operations:

```
Reading
  list(LENGTH <list> <out-var>)
  list(GET <list> <element index> [<index> ...] <out-var>)
  list(JOIN <list> <glue> <out-var>)
  list(SUBLIST <list> <begin> <length> <out-var>)

Search
  list(FIND <list> <value> <out-var>)

Ordering
  list(REVERSE <list>)
  list(SORT <list> [...])
```

Normal and cache variables can share names...

```
set(Variable "A normal variable")
set(Variable "A cache variable" CACHE STRING "Conflicts")
message(STATUS "CacheVariable = ${CacheVariable}")

if(Variable STREQUAL "A cache variable")
    message(STATUS "Variable (${Variable}) is Cache")
else()
    message(STATUS "Variable (${Variable}) is Normal")
endif()
```

...but they really shouldn't

```
$ cmake ..
-- CacheVariable = A cache variable
-- Variable (A cache variable) is Cache

$ cmake ..
-- CacheVariable = A cache variable
-- Variable (A normal variable) is Normal
```

Control flow

There aren't too many of these commands, but they all have a similar form. For a command `<x>`:

- Start the command with `x(...)`
- End the command with `endx()`

For example:

```
if(B)
# ...
endif()
```

List of flow control commands

```
if() ; else() ; elseif() ; endif()
while() ; endwhile()
foreach() ; endforeach()
function() ; endfunction()
macro() ; endmacro()
```

Also, there are normal loop control commands:

```
break() ; continue()
```

Using the `if` statement

Full usage:

```
if(<condition>)
  <commands>
elseif(<condition>) # optional block, can be repeated
  <commands>
else()              # optional block
  <commands>
endif()
```

Returns:

- **True** : `1`, `ON`, `YES`, `TRUE`, `Y`, or a non-zero number
- **False** : `0`, `OFF`, `NO`, `FALSE`, `N`, `IGNORE`, `NOTFOUND`, the empty string, or ends in the suffix `-NOTFOUND`

Commands can be joined with the normal boolean operators:

- AND
- OR
- NOT
- Parentheses

Example:

```
set(A1 FALSE)
if(((X EQUAL 3) OR (X EQUAL 7) OR (X EQUAL 14)) AND
    NOT ((Y EQUAL 4) OR (Y EQUAL 12) OR (Y EQUAL 18)))
    set(A1 TRUE)
endif()
```

Here is a much more readable way:

```
# More readable
set(A2 FALSE)
set(A2_X FALSE)
set(A2_Y FALSE)
if((X EQUAL 3) OR (X EQUAL 7) OR (X EQUAL 14))
    set(A2_X TRUE)
endif()
if(NOT ((Y EQUAL 4) OR (Y EQUAL 12) OR (Y EQUAL 18)))
    set(A2_Y TRUE)
endif()
if(A2_X AND A2_Y)
    set(A2 TRUE)
endif()
```

We can confirm that our last two examples produce the same result:

```
# Ensure that the variables match
if(NOT A1 STREQUAL A2)
    message(FATAL_ERROR "A1 (${A1}) != A2 (${A2}) [X=${X}, Y=${Y}]")
else()
```

Types of comparisons

Numbers	Strings	Versions
LESS	STRLESS	VERSION_LESS
LESS_EQUAL	STRLESS_EQUAL	VERSION_LESS_EQUAL
EQUAL	STREQUAL	VERSION_EQUAL
GREATER_EQUAL	STRGREATER_EQUAL	VERSION_GREATER_EQU
GREATER	STRGREATER	VERSION_GREATER

Example of version comparisons

```
set(versionA 1.2    1.2    1.2.3  2.0.1  1.8.2  1.7.3)
set(versionB 1.2.0  1.2.3  1.2    1.9.7  2      1.11.1)
list(LENGTH versionA listLength)
math(EXPR lastIndex "${listLength} - 1")
foreach(i RANGE 0 ${lastIndex})
    list(GET versionA ${i} a)
    list(GET versionB ${i} b)
    if(${a} VERSION_LESS ${b})
        message(NOTICE "  ${a} < ${b}")
    ...
    endif()
endforeach()
```

```
  1.2 == 1.2.0
  1.2 < 1.2.3
  1.2.3 > 1.2
  2.0.1 > 1.9.7
  1.8.2 < 2
  1.7.3 < 1.11.1
```

Using the `while` statement

```
while(<condition>)
  <commands>
endwhile()
```

- Takes the same arguments as `if`
- The loop can be interrupted by:
  - `break()` : breaks out of loop
  - `continue()` : jumps back to start of loop

Using the foreach statement

```
set(Out "RANGE 5 :")
foreach(Var RANGE 5)
    set(Out "${Out} ${Var}")
endforeach()
message(STATUS ${Out})

# Repeat for a few different signatures:
foreach(Var RANGE 11 15)
foreach(Var RANGE 21 25 2)
foreach(Var IN ITEMS 91 92 93 94 95)
```

Output

```
$ cmake ..
-- RANGE 5 : 0 1 2 3 4 5
-- RANGE 11 15 : 11 12 13 14 15
-- RANGE 21 25 2 : 21 23 25
-- IN ITEMS ... : 91 92 93 94 95
```

Functions and macros

The `function` and `macro` commands are very similar, but have one main difference:

- Functions run in their own scope
- Macros run in the caller's scope

Calling a macro is equivalent to copying its contents directly (much like C/C++ macros)

Example of a `function`:

```
function(F arg)
    if(DEFINED arg)
        message(STATUS "F : ${arg} is defined")
    else()
        message(STATUS "F : ${arg} is NOT defined")
    endif()
endfunction()
F("one")
```

Output:

```
-- F : one is defined
```

Example of a `macro`:

```
macro(M arg)
    if(DEFINED arg)
        message(STATUS "M : ${arg} is defined")
    else()
        message(STATUS "M : ${arg} is NOT defined")
    endif()
endmacro()
M("two")
```

Output:

```
-- M : two is NOT defined
```

Functions and macros *can* be overloaded, but it's not really recommended

When a function is overloaded, the original version of the function is still available, with a _ prefix

This means that a function can only be overloaded once, because the second time that it is overloaded the original version disappears

```
function(F arg1)
    message(STATUS "1 : ${arg1}")
endfunction()
F("A (1)" "arg2" "arg3")
function(F arg1 arg2)
    message(STATUS "2 : ${arg1}, ${arg2}")
endfunction()
_F("B (1)" "arg2" "arg3")
F("C (2)" "arg2" "arg3")
function(F arg1 arg2 arg3)
    message(STATUS "3 : ${arg1}, ${arg2}, ${arg3}")
endfunction()
_F("D (2)" "arg2" "arg3")
F("E (3)" "arg2" "arg3")
```

```
-- 1 : A (1)
-- 1 : B (1)
-- 2 : C (2), arg2
-- 2 : D (2), arg2
-- 3 : E (3), arg2, arg3
```

Build types

There are by default 4 build types:

Name	Description
`Debug`	All debugging information, including assertions
`RelWithDebInfo`	Some debug information, but no assertions
`Release`	Release optimized for speed
`MinSizeRel`	Release optimized for size

Here are the defaults on my Linux system using `g++` for `CMAKE_<LANG>_FLAGS_<TYPE>`:

```
$ cmake .. -GNinja
-- CMAKE_CXX_FLAGS =
-- CMAKE_CXX_FLAGS_DEBUG = -g
-- CMAKE_CXX_FLAGS_RELEASE = -O3 -DNDEBUG
-- CMAKE_CXX_FLAGS_RELWITHDEBINFO = -O2 -g -DNDEBUG
-- CMAKE_CXX_FLAGS_MINSIZEREL = -Os -DNDEBUG
-- CMAKE_C_FLAGS =
-- CMAKE_C_FLAGS_DEBUG = -g
-- CMAKE_C_FLAGS_RELEASE = -O3 -DNDEBUG
-- CMAKE_C_FLAGS_RELWITHDEBINFO = -O2 -g -DNDEBUG
-- CMAKE_C_FLAGS_MINSIZEREL = -Os -DNDEBUG
```

And here is the `CMakeLists.txt` that produced that output:

```
# Show the info for the different types
set(Types Debug Release RelWithDebInfo MinSizeRel)
foreach(LANG IN ITEMS CXX C)
    set(Flags CMAKE_${LANG}_FLAGS)
    message(STATUS "${Flags} = ${${Flags}}")
    foreach(Type IN LISTS Types)
        string(TOUPPER ${Type} TYPE)
        set(Flags CMAKE_${LANG}_FLAGS_${TYPE})
        message(STATUS "${Flags} = ${${Flags}}")
    endforeach()
endforeach()
```

Generators

There are two types of generators that CMake can use:

- Single-config (e.g. Ninja, GNU Makefiles)
- Multi-config (e.g. MSVC, Xcode)

The generator can be specified on the command-line by using the `-G` flag:

```
cmake -GNinja
# (or)
cmake -G"Unix Makefiles"
# (or)
cmake -G "Visual Studio 16 2019"
```

To see what generators are available on your system:

```
$ cmake --help
...
Options
  -G <generator-name>          = Specify a build system generator.
...
The following generators are available on this platform (* marks defa
* Unix Makefiles               = Generates standard UNIX makefiles.
  Ninja                        = Generates build.ninja files.
  CodeBlocks - Ninja           = Generates CodeBlocks project files.
  CodeLite - Unix Makefiles    = Generates CodeLite project files.
  Kate - Ninja                 = Generates Kate project files.
  Eclipse CDT4 - Ninja         = Generates Eclipse CDT 4.0 project fi
...
```

Single-configuration

This type of generator usually has a build directory for each build type:

```
$ ls -1F
build_make_Debug/
build_make_Release/
build_ninja_RelWithDebInfo/
build_ninja_Release/
src/
CMakeLists.txt
```

Multi-configuration

- These generators do *not* require a different build directory for each build type
- It doesn't really make sense to have a `build_MSVC_Debug` directory, because every build directory for MSVC is capable of switching build types

(The way it does this is to have a subdirectory for each build type that is used)

A brief word about build locations

- Back in the days of `make` and `autotools`, applications and binaries were often mixed in with source code
- This is referred to as an "**in-source**" build
- This required keeping a close eye on `.gitignore` files, and excluding things that were not really source code

CMake is a little different, as the most common method is using an "**out-of-source**" build directory

The normal use case looks something like this:

```
mkdir -p build && cd build
cmake -GNinja ..
```

This ensures that your build targets don't end up cluttering the source directory

Generator Expressions

There are two steps that CMake goes through:

1. **Configure** : all `CMakeLists.txt` files are processed
2. **Generate** : the appropriate `Makefile`s or other configuration files are generated

Examples

Here are few useful `generator expressions`:

```
$<BOOL:string>
$<AND:conditions> # comma-separated
$<OR:conditions>  # comma-separated
$<NOT:condition>
$<STREQUAL:string1,string2>
$<EQUAL:value1,value2>
$<IN_LIST:string,list>
$<VERSION_LESS:v1,v2> # and _GREATER, etc.
$<TARGET_EXISTS:target>
$<CONFIG:cfg>
$<CXX_COMPILER_VERSION:version>
```

You can string these together in a few ways:

```
set(X1 "$<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLD_COMPILER>
message(STATUS "X1 = ${X1}")

# OR

set(IsOldCompiler "$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>")
set(X2 "$<${IsOldCompiler}:OLD_COMPILER>")
message(STATUS "X2 = ${X1}")
```

Both of these produce the same output:

```
-- X1 = $<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLD_COMPILER>
-- X2 = $<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLD_COMPILER>
```

Symbol visiblity

Without going into too much detail:

- Symbols are **visible** by default in `gcc` and `clang`
- Symbols are **hidden** by default in `MSVC`

A symbol can be:

- A class
- A function outside of a class
- A global variable

To set the visibilty to the same defaults across all platforms:

```
set(CMAKE_CXX_VISIBILITY_PRESET hidden)
set(CMAKE_VISIBILITY_INLINES_HIDDEN YES)
...
generate_export_header(TargetName)
```

For more information, see Craig Scott's "Deep CMake for Library Authors" talk from CppCon 2019

Properties

There are a few per-target properties that you can set:

Target	Interface
`INCLUDE_DIRECTORIES`	`INTERFACE_INCLUDE_DIRECTORIES`
`COMPILE_DEFINITIONS`	`INTERFACE_COMPILE_DEFINITIONS`
`COMPILE_OPTIONS`	`INTERFACE_COMPILE_OPTIONS`

Custom Commands

- To run a custom command at configuration time, use `execute_process`
- To run a custom command at build time, use `add_custom_command` and `add_custom_target`

Options for execute_process

```
execute_process(COMMAND <cmd1> [<arguments>]
                [COMMAND <cmd2> [<arguments>]]...
                [WORKING_DIRECTORY <directory>]
                [TIMEOUT <seconds>]
                [RESULT_VARIABLE <variable>]
                ...
                [OUTPUT_QUIET]
                [ERROR_QUIET]
                [COMMAND_ECHO <where>]
                [OUTPUT_STRIP_TRAILING_WHITESPACE]
                [ERROR_STRIP_TRAILING_WHITESPACE]
                [ENCODING <name>])
```

I won't spend too much time on this, because most of the time you want to run commands at build time, but here's a good use case that we'll see again shortly:

```
execute_process(
    COMMAND ${CMAKE_COMMAND} -E touch ${FileName}
    COMMAND ${CMAKE_COMMAND} -E create_symlink ${FileName} ${LinkName
    )
```

This command runs at configure time, and it creates a files with a symbolic link to it

Options for add_custom_command

```
add_custom_command(OUTPUT output1 [output2 ...]
                   COMMAND command1 [ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [MAIN_DEPENDENCY depend]
                   [DEPENDS [depends...]]
                   [BYPRODUCTS [files...]]
                   [IMPLICIT_DEPENDS <lang1> depend1
                                     [<lang2> depend2] ...]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment]
                   [DEPFILE depfile]
                   [JOB_POOL job_pool]
                   [VERBATIM] [APPEND] [USES_TERMINAL]
                   [COMMAND_EXPAND_LISTS])
```

Options for add_custom_target

```
add_custom_target(Name [ALL] [command1 [args1...]]
                  [COMMAND command2 [args2...] ...]
                  [DEPENDS depend depend depend ... ]
                  [BYPRODUCTS [files...]]
                  [WORKING_DIRECTORY dir]
                  [COMMENT comment]
                  [JOB_POOL job_pool]
                  [VERBATIM] [USES_TERMINAL]
                  [COMMAND_EXPAND_LISTS]
                  [SOURCES src1 [src2...]])
```

Replacing bash/python scripts with CMake

The `cmake -E` command can be used in place of bash scripts

Even better, use `${CMAKE_COMMAND} -E`, so that whatever version of CMake is currently running will continue to be used.

```
execute_process(
    COMMAND ${CMAKE_COMMAND} -E touch ${FileName}
    COMMAND ${CMAKE_COMMAND} -E create_symlink ${LinkName}
    )
```

Useful commands:

- `touch` and `make_directory`
- `copy` and `copy_directory`
- `remove` and `remove_directory`
- `rename`
- `echo`
- `time`
- `copy_if_different`
- `compare_files`
- `md5sum`
- `tar`

For more information, see the CMake manual, or run `cmake -E`:

```
$ cmake -E
Usage: cmake -E <command> [arguments...]
Available commands:
  capabilities              - Report capabilities built into cmake in
  chdir dir cmd [args...]   - run command in a given directory
  compare_files [--ignore-eol] file1 file2
                            - check if file1 is same as file2
  copy <file>... destination  - copy files to destination (either fil
  copy_directory <dir>... destination   - copy content of <dir>... di
  copy_if_different <file>... destination  - copy files if it has cha
  echo [<string>...]        - displays arguments as text
  echo_append [<string>...] - displays arguments as text but no new l
  env [--unset=NAME]... [NAME=VALUE]... COMMAND [ARG]...
                            - run command in a modified environment
  environment               - display the current environment
```

Working with files

Easy to get parts of a file path

```
set(FullFilePath /tmp/cmake_test/file.x.y.z)
get_filename_component(FileDir ${FullFilePath} DIRECTORY)
get_filename_component(FileName ${FullFilePath} NAME)
get_filename_component(FileWithoutLongestExtension ${FullFilePath} NA
get_filename_component(FileExtension ${FullFilePath} EXT)
get_filename_component(FileWithoutLastExtension ${FullFilePath} NAME_
get_filename_component(FileLastExtension ${FullFilePath} LAST_EXT)
```

```
-- [/tmp/cmake_test/file.x.y.z]
--   DIRECTORY : /tmp/cmake_test
--   NAME      : file.x.y.z
--   NAME_WE   : file
--   EXT       : .x.y.z
--   NAME_WLE  : file.x.y
--   LAST_EXT  : .z
```

Problem

I want to create a library with a cyclic dependency

Here is a graph of the project that I created, generated with `cmake --graphviz=...`:

Top-level CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15)
project(TestCycle LANGUAGES CXX)

add_subdirectory(ExecutableApp)
add_subdirectory(LibraryA)
add_subdirectory(LibraryB)
add_subdirectory(LibraryC)
```

Add an application

```
add_executable(TestCycle test_cycle.cpp)

# TestCycle depends on A
target_link_libraries(TestCycle PUBLIC A)
```

Create a circular dependency

```
# A depends on B
add_library(A a.cpp)
target_link_libraries(A PUBLIC B)

# B depends on C
add_library(B b.cpp)
target_link_libraries(B PUBLIC C)

# C depends on A
add_library(C c.cpp)
target_link_libraries(C PUBLIC A)
```

First, here's an example of building with static libs, which works just fine:

```
$ mkdir -p build && cd build
$ cmake .. -GNinja -DBUILD_SHARED_LIBS:BOOL=OFF
...
-- Configuring done
-- Generating done
```

But if I try the same thing with shared libraries:

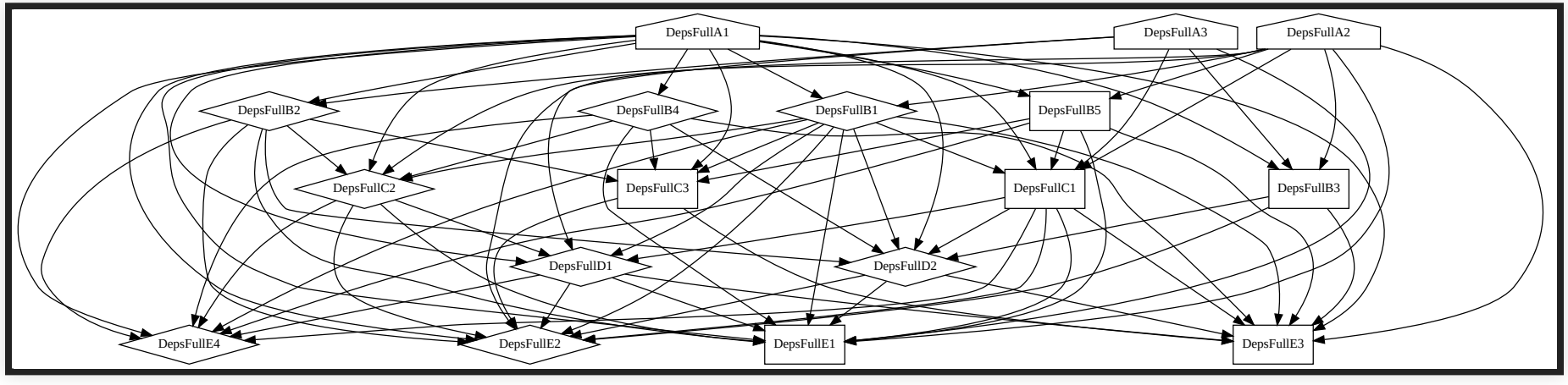
```
$ cmake .. -GNinja -DBUILD_SHARED_LIBS:BOOL=ON
...
CMake Error: The inter-target dependency graph contains the following
  "A" of type SHARED_LIBRARY
    depends on "B" (weak)
    depends on "C" (weak)
  "B" of type SHARED_LIBRARY
    depends on "C" (weak)
    depends on "A" (weak)
  "C" of type SHARED_LIBRARY
    depends on "A" (weak)
    depends on "B" (weak)
At least one of these targets is not a STATIC_LIBRARY.  Cyclic depend
CMake Generate step failed.  Build files cannot be regenerated correc
```

Dependencies

In his book, Craig Scott recommends that:

> *If a target uses something from a library, it should always link directly to that library. Even if the library is already a link dependency of something else the target links to, do not rely on an indirect link dependency for something a target uses directly.*

I think that so long as you are using `PUBLIC` linking anyway, this is unnecessary, as it can lead to dependency graphs that look like this:



By trimming the dependencies to only the necessary ones, you end up with a much cleaner graph:

Best practices

Do not define or call a function or macro with a name that starts with a single underscore

If a function is redefined, it will prefix an underscore to the name

Do not have variables and cache variables with the same name

As discussed before, it is too easy to mask one or the other

Define properties as locally as possible

Instead of setting global variables that will affect every target, use the `target_*(...)` commands:

```
target_compile_definitions(...)
target_compile_features(...)
target_compile_options(...)
target_include_directories(...)
target_link_libraries(...)
target_sources(...)
```

References

Books:

- Professional CMake, by Craig Scott
- CMake Cookbook, by Bast and Remigio

YouTube:

- vector-of-bool: How to CMake good
- Craig Scott “Deep CMake for Library Authors”
- Embracing Modern CMake - Stephen Kelly

Questions?