

Effective CMake

a random selection of best practices

Daniel Pfeifer

May 19, 2017

daniel@pfeifer-mail.de

Opening

Why?

The way you use CMake affects your users!

CMake's similarities with C++

- big userbase, industry dominance
- focus on backwards compatibility
- complex, feature rich, "multi paradigm"
- bad reputation, "bloated", "horrible syntax"
- some not very well known features

Standards

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)



CMake is code.

Use the same principles for `CMakeLists.txt`
and modules as for the rest of your codebase.

Language

Organization

Directories that contain a `CMakeLists.txt` are the entry point for the build system generator. Subdirectories may be added with `add_subdirectory()` and must contain a `CMakeLists.txt` too.

Scripts are `<script>.cmake` files that can be executed with `cmake -P <script>.cmake`.
Not all commands are supported.

Modules are `<script>.cmake` files located in the `CMAKE_MODULE_PATH`.
Modules can be loaded with the `include()` command.

Commands

1 `command_name(space separated list of strings)`

- Scripting commands change state of command processor
 - set variables
 - change behavior of other commands
- Project commands
 - create build targets
 - modify build targets
- Command invocations are not expressions.

Variables

```
1 set(hello world)
2 message(STATUS "hello, ${hello}")
```

- Set with the `set()` command.
- Expand with `${}`.
- Variables and values are strings.
- Lists are `;`-separated strings.
- CMake variables are not environment variables (unlike `Makefile`).
- Unset variable expands to `empty` string.

Comments

```
1 # a single line comment
2
3 #[=[
4   multi line comments
5
6   #[=[
7       may be nested
8   #]=]
9 #]=]
```

Generator expressions

```
1 target_compile_definitions(foo PRIVATE  
2     "VERBOSITY=$<IF:$<CONFIG:Debug>,30,10>"  
3 )
```

- Generator expressions use the `$<>` syntax.
- Not evaluated by command interpreter.
It is just a string with `$<>`.
- Evaluated during build system generation.
- Not supported in all commands (obviously).

Custom Commands

Two types of commands

- Commands can be added with `function()` or `macro()`.
- Difference is like in C++.
- When a new command replaces an existing command, the old one can be accessed with a `_` prefix.

Custom command: **Function**

```
1 function(my_command input output)
2   # ...
3   set(${output} ... PARENT_SCOPE)
4 endfunction()
5 my_command(foo bar)
```

- Variables are scoped to the function, unless set with `PARENT_SCOPE`.
- Available variables: `input`, `output`, `ARGC`, `ARGV`, `ARGN`, `ARG0`, `ARG1`, `ARG2`, ...
- Example: `${output}` expands to `bar`.

Custom command: **Macro**

```
1 macro(my_command input output)
2   # ...
3 endmacro()
4 my_command(foo bar)
```

- No extra scope.
- Text replacements: `${input}`, `${output}`, `${ARGC}`, `${ARGV}`, `${ARGN}`, `${ARG0}`, `${ARG1}`, `${ARG2}`, ...
- Example: `${output}` is replaced by `bar`.

Create macros to wrap commands
that have output parameters.

Otherwise, create a function.

Evolving CMake code

Deprecate CMake commands

```
1 macro(my_command)
2   message(DEPRECATION
3     "The my_command command is deprecated!")
4   _my_command(${ARGV})
5 endmacro()
```

Deprecate CMake variables

```
1 set(hello "hello world!")
2
3 function(__deprecated_var var access)
4     if(access STREQUAL "READ_ACCESS")
5         message(DEPRECATION
6             "The variable '${var}' is deprecated!")
7     endif()
8 endfunction()
9
10 variable_watch(hello __deprecated_var)
```

Variables are so CMake 2.8.12.

Modern CMake is about **Targets** and **Properties**!

Targets and Properties

Look Ma, no Variables!

```
1 add_library(Foo foo.cpp)
2 target_link_libraries(Foo PRIVATE Bar::Bar)
3
4 if(WIN32)
5     target_sources(Foo PRIVATE foo_win32.cpp)
6     target_link_libraries(Foo PRIVATE Bar::Win32Support)
7 endif()
```

Avoid **custom** variables
in the arguments of project commands.

Don't use `file(GLOB)` in projects.

Imagine Targets as Objects

- Constructors:
 - `add_executable()`
 - `add_library()`
- Member variables:
 - Target properties (too many to list here).
- Member functions:
 - `get_target_property()`
 - `set_target_properties()`
 - `get_property(TARGET)`
 - `set_property(TARGET)`
 - `target_compile_definitions()`
 - `target_compile_features()`
 - `target_compile_options()`
 - `target_include_directories()`
 - `target_link_libraries()`
 - `target_sources()`

Forget those commands:

```
add_compile_options()  
include_directories()  
    link_directories()  
    link_libraries()
```

Example:

```
1 target_compile_features(Foo
2     PUBLIC
3     cxx_strong_enums
4     PRIVATE
5     cxx_lambdas
6     cxx_range_for
7 )
```

- Adds `cxx_strong_enums` to the target properties `COMPILE_FEATURES` and `INTERFACE_COMPILE_FEATURES`.
- Adds `cxx_lambdas`; `cxx_range_for` to the target property `COMPILE_FEATURES`.

Get your **hands off** CMAKE_CXX_FLAGS!

Build Specification and Usage Requirements

- **Non-INTERFACE_** properties define the **build specification** of a target.
- **INTERFACE_** properties define the **usage requirements** of a target.

Build Specification and Usage Requirements

- PRIVATE populates the non-INTERFACE_ property.
- INTERFACE populates the INTERFACE_ property.
- PUBLIC populates both.

Use `target_link_libraries()`
to express **direct** dependencies!

Example:

```
1 target_link_libraries(Foo
2     PUBLIC  Bar::Bar
3     PRIVATE Cow::Cow
4 )
```

- Adds `Bar::Bar` to the target properties `LINK_LIBRARIES` and `INTERFACE_LINK_LIBRARIES`.
- Adds `Cow::Cow` to the target property `LINK_LIBRARIES`.

Example:

```
1 target_link_libraries(Foo
2     PUBLIC  Bar::Bar
3     PRIVATE Cow::Cow
4 )
```

- Adds `Bar::Bar` to the target properties `LINK_LIBRARIES` and `INTERFACE_LINK_LIBRARIES`.
- Adds `Cow::Cow` to the target property `LINK_LIBRARIES`.
- Effectively adds all `INTERFACE_<property>` of `Bar::Bar` to `<property>` and `INTERFACE_<property>`.
- Effectively adds all `INTERFACE_<property>` of `Cow::Cow` to `<property>`.

Example:

```
1 target_link_libraries(Foo
2     PUBLIC  Bar::Bar
3     PRIVATE Cow::Cow
4 )
```

- Adds `Bar::Bar` to the target properties `LINK_LIBRARIES` and `INTERFACE_LINK_LIBRARIES`.
- Adds `Cow::Cow` to the target property `LINK_LIBRARIES`.
- Effectively adds all `INTERFACE_<property>` of `Bar::Bar` to `<property>` and `INTERFACE_<property>`.
- Effectively adds all `INTERFACE_<property>` of `Cow::Cow` to `<property>`.
- Adds `$<LINK_ONLY:Cow::Cow>` to `INTERFACE_LINK_LIBRARIES`.

Pure usage requirements

```
1 add_library(Bar INTERFACE)
2 target_compile_definitions(Bar INTERFACE BAR=1)
```

- INTERFACE libraries have no build specification.
- They only have usage requirements.

Don't abuse requirements!

Eg: -Wall is not a requirement!

Project Boundaries

How to use external libraries

Always like this:

```
1 find_package(Foo 2.0 REQUIRED)
2 # ...
3 target_link_libraries(... Foo::Foo ...)
```

FindFoo.cmake

```
1 find_path(Foo_INCLUDE_DIR foo.h)
2 find_library(Foo_LIBRARY foo)
3 mark_as_advanced(Foo_INCLUDE_DIR Foo_LIBRARY)
4
5 include(FindPackageHandleStandardArgs)
6 find_package_handle_standard_args(Foo
7     REQUIRED_VARS Foo_LIBRARY Foo_INCLUDE_DIR
8     )
9
10 if(Foo_FOUND AND NOT TARGET Foo::Foo)
11     add_library(Foo::Foo UNKNOWN IMPORTED)
12     set_target_properties(Foo::Foo PROPERTIES
13         IMPORTED_LINK_INTERFACE_LANGUAGES "CXX"
14         IMPORTED_LOCATION "${Foo_LIBRARY}"
15         INTERFACE_INCLUDE_DIRECTORIES "${Foo_INCLUDE_DIR}"
16     )
17 endif()
```

FindPNG.cmake

[illegible]

Use a Find module for **third party** libraries
that are not **built with CMake**.

Use a Find module for **third party** libraries
~~that are not **built with CMake.**~~
that do not **support clients to use CMake.**

Use a Find module for **third party** libraries
~~that are not built with CMake.~~
that do not **support clients to use CMake.**
Also, report this as a bug to their authors.

Export your library interface!

```
1 find_package(Bar 2.0 REQUIRED)
2 add_library(Foo ...)
3 target_link_libraries(Foo PRIVATE Bar::Bar)
4
5 install(TARGETS Foo EXPORT FooTargets
6   LIBRARY DESTINATION lib
7   ARCHIVE DESTINATION lib
8   RUNTIME DESTINATION bin
9   INCLUDES DESTINATION include
10  )
11 install(EXPORT FooTargets
12   FILE FooTargets.cmake
13   NAMESPACE Foo::
14   DESTINATION lib/cmake/Foo
15  )
```

Export your library interface!

```
1 include(CMakePackageConfigHelpers)
2 write_basic_package_version_file("FooConfigVersion.cmake"
3   VERSION ${Foo_VERSION}
4   COMPATIBILITY SameMajorVersion
5 )
6 install(FILES "FooConfig.cmake" "FooConfigVersion.cmake"
7   DESTINATION lib/cmake/Foo
8 )
```

```
1 include(CMakeFindDependencyMacro)
2 find_dependency(Bar 2.0)
3 include("${CMAKE_CURRENT_LIST_DIR}/FooTargets.cmake")
```

Export the right information!

Warning:

The library interface may change during installation. Use the `BUILD_INTERFACE` and `INSTALL_INTERFACE` generator expressions as filters.

```
1 target_include_directories(Foo PUBLIC
2   $<BUILD_INTERFACE:${Foo_BINARY_DIR}/include>
3   $<BUILD_INTERFACE:${Foo_SOURCE_DIR}/include>
4   $<INSTALL_INTERFACE:include>
5   )
```

Creating Packages

- CPack is a packaging tool distributed with CMake.
- `set()` variables in `CPackConfig.cmake`, or
- `set()` variables in `CMakeLists.txt` and `include(CPack)`.

Write your own `CPackConfig.cmake`
and `include()` the one
that is generated by CMake.

The variable `CPACK_INSTALL_CMAKE_PROJECTS` is a list of quadruples:

1. Build directory
2. Project Name
3. Project Component
4. Directory

Packaging multiple configurations

1. Make sure different configurations don't collide:

```
1 set(CMAKE_DEBUG_POSTFIX "-d")
```

2. Create separate build directories for debug, release.
3. Use this CPackConfig.cmake:

```
1 include("release/CPackConfig.cmake")
2 set(CPACK_INSTALL_CMAKE_PROJECTS
3     "debug;Foo;ALL;/"
4     "release;Foo;ALL;/"
5     )
```

Package Management

My requirements for a package manager

- Support system packages
- Support prebuilt libraries
- Support building dependencies as subprojects
- Do not require any changes to my projects!

How to use external libraries

Always like this:

```
1 find_package(Foo 2.0 REQUIRED)
2 # ...
3 target_link_libraries(... Foo::Foo ...)
```

Do not require any changes to my projects!

- System packages ...

Do not require any changes to my projects!

- System packages ...
 - work out of the box.

Do not require any changes to my projects!

- System packages ...
 - work out of the box.
- Prebuilt libraries ...

Do not require any changes to my projects!

- System packages ...
 - work out of the box.
- Prebuilt libraries ...
 - need to be put into `CMAKE_PREFIX_PATH`.

Do not require any changes to my projects!

- System packages ...
 - work out of the box.
- Prebuilt libraries ...
 - need to be put into `CMAKE_PREFIX_PATH`.
- Subprojects ...
 - We need to turn `find_package(Foo)` into a no-op.
 - What about the imported target `Foo::Foo`?

Use the your public interface

When you export `Foo` in namespace `Foo::`,
also create an alias `Foo::Foo`.

```
1 add_library(Foo::Foo ALIAS Foo)
```

When you export **Foo** in namespace **Foo::**,
also create an alias **Foo::Foo**.

The toplevel super-project

```
1 set(CMAKE_PREFIX_PATH "/prefix")
2 set(as_subproject Foo)
3
4 macro(find_package)
5     if(NOT "${ARG0}" IN_LIST as_subproject)
6         _find_package(${ARGV})
7     endif()
8 endmacro()
9
10 add_subdirectory(Foo)
11 add_subdirectory(App)
```

How does that work?

If **Foo** is a ...

- system package:
 - `find_package(Foo)` either finds `FooConfig.cmake` in the system or uses `FindFoo.cmake` to find the library in the system. In either case, the target `Foo::Foo` is imported.
- prebuilt library:
 - `find_package(Foo)` either finds `FooConfig.cmake` in the `CMAKE_PREFIX_PATH` or uses `FindFoo.cmake` to find the library in the `CMAKE_PREFIX_PATH`. In either case, the target `Foo::Foo` is imported.
- subproject:
 - `find_package(Foo)` does nothing. The target `Foo::Foo` is part of the project.

CTest

Run with `ctest -S build.cmake`

```
1 set(CTEST_SOURCE_DIRECTORY "/source")
2 set(CTEST_BINARY_DIRECTORY "/binary")
3
4 set(ENV{CXXFLAGS} "--coverage")
5 set(CTEST_CMAKE_GENERATOR "Ninja")
6 set(CTEST_USE_LAUNCHERS 1)
7
8 set(CTEST_COVERAGE_COMMAND "gcov")
9 set(CTEST_MEMORYCHECK_COMMAND "valgrind")
10 #set(CTEST_MEMORYCHECK_TYPE "ThreadSanitizer")
11
12 ctest_start("Continuous")
13 ctest_configure()
14 ctest_build()
15 ctest_test()
16 ctest_coverage()
17 ctest_memcheck()
18 ctest_submit()
```

CTest scripts are the right place
for CI specific settings.

Keep that information out of the project.

Filtering tests by name

Define like this:

```
1 add_test(NAME Foo.Test
2     COMMAND foo_test --number 0
3     )
```

Run like this:

```
1 ctest -R 'Foo.' -j4 --output-on-failure
```

Follow a naming convention for test names.

This simplifies **filtering by regex**.

Fail to compile

```
1 add_library(foo_fail STATIC EXCLUDE_FROM_ALL
2     foo_fail.cpp
3 )
4 add_test(NAME Foo.Fail
5     COMMAND ${CMAKE_COMMAND}
6         --build ${CMAKE_BINARY_DIR}
7         --target foo_fail
8 )
9 set_property(TEST Foo.Fail PROPERTY
10     PASS_REGULAR_EXPRESSION "static assert message"
11 )
```

Running crosscompiled tests

- When the testing command is a build target, the command line is prefixed with `${CMAKE_CROSSCOMPILING_EMULATOR}`.
- When crosscompiling from Linux to Windows, set `CMAKE_CROSSCOMPILING_EMULATOR` to `wine`.
- When crosscompiling to ARM, set `CMAKE_CROSSCOMPILING_EMULATOR` to `qemu-arm`.
- To run tests on another machine, set `CMAKE_CROSSCOMPILING_EMULATOR` to a script that copies it over and executes it there.

Run tests on real hardware

```
1 #!/bin/bash
2 tester=$1
3 shift
4 # create temporary file
5 filename=$(ssh root@172.22.22.22 mktemp)
6 # copy the tester to temporary file
7 scp $tester root@172.22.22.22:$filename
8 # make test executable
9 ssh root@172.22.22.22 chmod +x $filename
10 # execute test
11 ssh root@172.22.22.22 $filename "$@"
12 # store success
13 success=$?
14 # cleanup
15 ssh root@172.22.22.22 rm $filename
16 exit $success
```

Cross Compiling

```
1 set(CMAKE_SYSTEM_NAME Windows)
2
3 set(CMAKE_C_COMPILER x86_64-w64-mingw32-gcc)
4 set(CMAKE_CXX_COMPILER x86_64-w64-mingw32-g++)
5 set(CMAKE_RC_COMPILER x86_64-w64-mingw32-windres)
6
7 set(CMAKE_FIND_ROOT_PATH /usr/x86_64-w64-mingw32)
8
9 set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
10 set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
11 set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
12
13 set(CMAKE_CROSSCOMPILING_EMULATOR wine64)
```

Don't put logic in toolchain files.

Static Analysis

Treat warnings as errors?

How do you treat build errors?

- You fix them.
- You reject pull requests.
- You hold off releases.

Treat warnings as errors!

Treat warnings as errors!

- You fix them.
- You reject pull requests.
- You hold off releases.

Treat warnings as errors!

*To treat warnings as errors, never pass **-Werror** to the compiler. If you do, your compiler treats warnings as errors. You can no longer treat warnings as errors, because you will no longer get any warnings. All you get is errors.*

-Werror causes pain

- You cannot enable **-Werror** unless you already reached zero warnings.
- You cannot increase the warning level unless you already fixed all warnings introduced by that level.
- You cannot upgrade your compiler unless you already fixed all new warnings that the compiler reports at your warning level.
- You cannot update your dependencies unless you already ported your code away from any symbols that are now `[[deprecated]]`.
- You cannot `[[deprecated]]` your internal code as long as it is still used. But once it is no longer used, you can as well just remove it...

Better: Treat new warnings as errors!

1. At the beginning of a development cycle (eg. sprint), allow new warnings to be introduced.
 - Increase warning level, enable new warnings explicitly.
 - Update the compiler.
 - Update dependencies.
 - Mark symbols as `[[deprecated]]`.
2. Then, burn down the number of warnings.
3. Repeat.

Pull out all the stops!

clang-tidy is a clang-based C++ “linter” tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis.

cpplint is automated checker to make sure a C++ file follows Google’s C++ style guide.

include-what-you-use analyzes `#includes` in C and C++ source files.

clazy is a clang wrapper that finds common C++/Qt antipatterns that decrease performance.

Target properties for static analysis

- `<lang>_CLANG_TIDY`
- `<lang>_CPPLINT`
- `<lang>_INCLUDE_WHAT_YOU_USE`
 - Runs the respective tool along the with compiler.
 - Diagnostics are visible in your IDE.
 - Diagnostics are visible on CDash.
- `LINK_WHAT_YOU_USE`
 - links with `-Wl,--no-as-needed`, then runs `ldd -r -u`.

`<lang>` is either C or CXX.

Each of those properties is initialized with `CMAKE_<property>`.

Scanning header files

- Most of those tools report diagnostics for the **current source file** plus the **associated header**.
- Header files with no associated source file will not be analyzed.
- You may be able to set a **custom header filter**, but then the headers may be analyzed multiple times.

For each header file,
there is an **associated source file**
that `#includes` this header file at the top.
Even if that source file would otherwise be empty.

Create associated source files

```
1 #!/usr/bin/env bash
2 for fn in `comm -23 \
3     <(ls *.h|cut -d '.' -f 1|sort) \
4     <(ls *.c *.cpp|cut -d '.' -f 1|sort)`
5 do
6     echo "#include \"$fn.h\"" >> $fn.cpp
7 done
```

Enable warnings from from outside the project

```
1 env CC=clang CXX=clazy cmake \  
2   -DCMAKE_CXX_CLANG_TIDY:STRING=\  
3   'clang-tidy;-checks=-*,readability-*' \  
4   -DCMAKE_CXX_INCLUDE_WHAT_YOU_USE:STRING=\  
5   'include-what-you-use;-Xiwyu;--mapping_file=/iwyu.imp' \  
6   ..
```

Supported by all IDEs

- Just setting `CMAKE_CXX_CLANG_TIDY` will make all `clang-tidy` diagnostics appear in your normal build output.
- No special IDE support needed.
- If IDE understands `fix-it hints` from `clang`, it will also understand the ones from `clang-tidy`.

Thank You!

Personal Wishlist

- For each of the following ideas, I have started a prototype.
- Contributions welcome!
- You can talk to me!

Disclaimer:

No guarantee that the following ideas
will ever be added to CMake.

PCH as usage requirements

PCH as usage requirements

```
1 target_precompile_headers(Foo
2     PUBLIC
3     "foo.h"
4     PRIVATE
5     <unordered_map>
6 )
```

PCH as usage requirements

- Calculate a list of headers per config and language for the **build specification** of each target.
- Generate a header file that **#includes** each of those headers.
- Tell the build system to **precompile** this header.
- Tell the build system to **force-include** this header.
- Require no changes to the code (No **#include "stdafx.h"**).

More Languages!

More Languages!

- CMake's core is language-agnostic.
- Language support is scripted in modules.
- Rules how to compile object files, link them together.
- The output of the compiler must be an object file.
- CMake can be used with **D** by putting necessary files in `CMAKE_MODULE_PATH`.¹

¹<https://github.com/dcarp/cmake-d>

Even more Languages!

- If we allow the output to be a source file of a known language, we would not need special handling for Protobuf, Qt-resources, or any other IDL.
- This would also allow using CMake for BASIC, BCX, Chapel, COBOL, Cython, Eiffel, Genie, Haxe, Java, Julia, Lisaac, Scheme, PHP, Python, X10, Nim, Vala.²

²https://en.wikipedia.org/wiki/Source-to-source_compiler

`find_package(Foo PKGCONF)`

`find_package(Foo PKGCONF)`

- `find_package()` has two modes: `PACKAGE` and `CONFIG`.
- Let's add a `PKGCONF` mode.
- In this mode, CMake parses `.pc` files and generates one `IMPORTED` library per package.

Declarative Frontend and Lua VM

- Execute CMake commands on Lua VM.³
- Allow CMake **modules** to be written in Lua.

³not the other way round. This failed before.

- For **directories**, use a declarative language that allows procedural subroutines.
- **libucl**⁴ is an interesting option.

⁴<https://github.com/vstakhov/libucl>

Tell me **your** ideas!