Effective CMake

a random selection of best practices

Daniel Pfeifer

June 7, 2017

daniel@pfeifer-mail.de

Opening

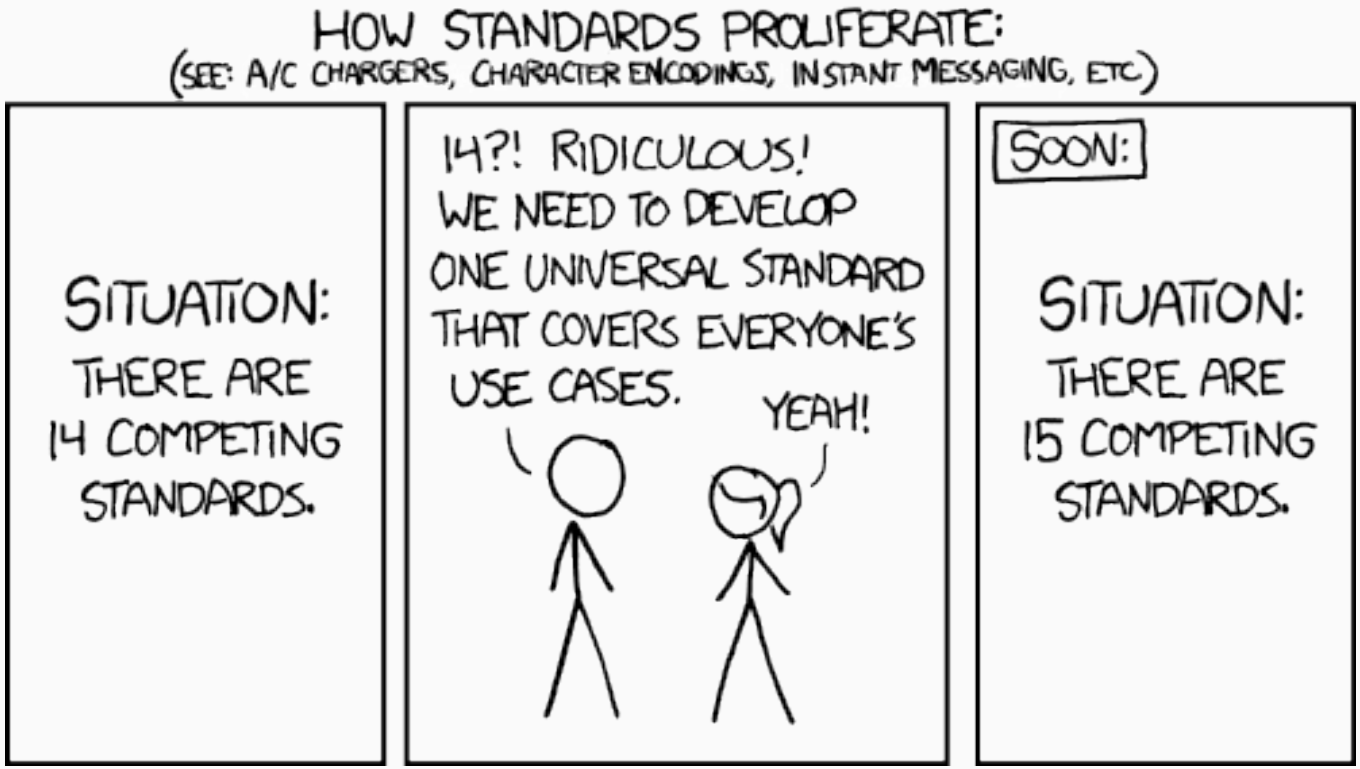
Why?

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



[0] https://xkcd.com/927/

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

```
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

```
# a single line comment

#[==[
   multi line comments

   #[=[
      may be nested
   #]=]
#]==]
```

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

```
function(my_command input output)
  # ...
  set(${output} ... PARENT_SCOPE)
endfunction()
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 parametes.

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:

```
target_link_libraries(Foo
  PUBLIC  Bar::Bar
  PRIVATE Cow::Cow
  )
```

- 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 reqiurements

```
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:

```
find_package(Foo 2.0 REQUIRED)
# ...
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

```
if(PNG_FIND_QUIETLY)
  set(_FIND_ZLIB_ARG QUIET)
endif()
find_package(ZLIB ${_FIND_ZLIB_ARG})

if(ZLIB_FOUND)
  find_path(PNG_PNG_INCLUDE_DIR png.h
  /usr/local/include/libpng             # OpenBSD
  )

  list(APPEND PNG_NAMES png libpng)
  unset(PNG_NAMES_DEBUG)
  set(_PNG_VERSION_SUFFIXES 17 16 15 14 12)
  if (PNG_FIND_VERSION MATCHES "^([0-9]+)\\.([0-9]+)(\\..*)?$")
    set(_PNG_VERSION_SUFFIX_MIN "${CMAKE_MATCH_1}${CMAKE_MATCH_2}")
    if (PNG_FIND_VERSION_EXACT)
      set(_PNG_VERSION_SUFFIXES ${_PNG_VERSION_SUFFIX_MIN})
    else ()
      string(REGEX REPLACE
          "${_PNG_VERSION_SUFFIX_MIN}.*" "${_PNG_VERSION_SUFFIX_MIN}"
          _PNG_VERSION_SUFFIXES "${_PNG_VERSION_SUFFIXES}")
    endif ()
    unset(_PNG_VERSION_SUFFIX_MIN)
  endif ()
  foreach(v IN LISTS _PNG_VERSION_SUFFIXES)
    list(APPEND PNG_NAMES png${v} libpng${v})
    list(APPEND PNG_NAMES_DEBUG png${v}d libpng${v}d)
  endforeach()
  unset(_PNG_VERSION_SUFFIXES)
  # For compatibility with versions prior to this multi-config search, honor
  # any PNG_LIBRARY that is already specified and skip the search.
  if(NOT PNG_LIBRARY)
    find_library(PNG_LIBRARY_RELEASE NAMES ${PNG_NAMES})
    find_library(PNG_LIBRARY_DEBUG NAMES ${PNG_NAMES_DEBUG})
    include(${CMAKE_CURRENT_LIST_DIR}/SelectLibraryConfigurations.cmake)
    select_library_configurations(PNG)
    mark_as_advanced(PNG_LIBRARY_RELEASE PNG_LIBRARY_DEBUG)
  endif()
  unset(PNG_NAMES)
  unset(PNG_NAMES_DEBUG)

  # Set by select_library_configurations(), but we want the one from
  # find_package_handle_standard_args() below.
  unset(PNG_FOUND)

  if (PNG_LIBRARY AND PNG_PNG_INCLUDE_DIR)
      # png.h includes zlib.h. Sigh.
      set(PNG_INCLUDE_DIRS ${PNG_PNG_INCLUDE_DIR} ${ZLIB_INCLUDE_DIR} )
      set(PNG_INCLUDE_DIR ${PNG_INCLUDE_DIRS} ) # for backward compatibility
      set(PNG_LIBRARIES ${PNG_LIBRARY} ${ZLIB_LIBRARY})

      if (CYGWIN)
        if(BUILD_SHARED_LIBS)
          # No need to define PNG_USE_DLL here, because it's default for Cygwin.
        else()
          set (PNG_DEFINITIONS -DPNG_STATIC)
        endif()
      endif ()

      if(NOT TARGET PNG::PNG)
        add_library(PNG::PNG UNKNOWN IMPORTED)
        set_target_properties(PNG::PNG PROPERTIES
          INTERFACE_COMPILE_DEFINITIONS "${PNG_DEFINITIONS}"
          INTERFACE_INCLUDE_DIRECTORIES "${PNG_INCLUDE_DIRS}"
          INTERFACE_LINK_LIBRARIES ZLIB::ZLIB)
        if(EXISTS "${PNG_LIBRARY}")
          set_target_properties(PNG::PNG PROPERTIES
            IMPORTED_LINK_INTERFACE_LANGUAGES "C"
            IMPORTED_LOCATION "${PNG_LIBRARY}")
        endif()
        if(EXISTS "${PNG_LIBRARY_RELEASE}")
          set_property(TARGET PNG::PNG APPEND PROPERTY
            IMPORTED_CONFIGURATIONS RELEASE)
          set_target_properties(PNG::PNG PROPERTIES
            IMPORTED_LINK_INTERFACE_LANGUAGES_RELEASE "C"
            IMPORTED_LOCATION_RELEASE "${PNG_LIBRARY_RELEASE}")
        endif()
        if(EXISTS "${PNG_LIBRARY_DEBUG}")
          set_property(TARGET PNG::PNG APPEND PROPERTY
            IMPORTED_CONFIGURATIONS DEBUG)
          set_target_properties(PNG::PNG PROPERTIES
            IMPORTED_LINK_INTERFACE_LANGUAGES_DEBUG "C"
            IMPORTED_LOCATION_DEBUG "${PNG_LIBRARY_DEBUG}")
        endif()
      endif()
  endif ()

  if (PNG_PNG_INCLUDE_DIR AND EXISTS "${PNG_PNG_INCLUDE_DIR}/png.h")
      file(STRINGS "${PNG_PNG_INCLUDE_DIR}/png.h" png_version_str REGEX "^#define[ \t]+PNG_LIBPNG_VER_STRING[ \t]+\".+\"")

      string(REGEX REPLACE "^#define[ \t]+PNG_LIBPNG_VER_STRING[ \t]+\"([^\"]+)\".*" "\\1" PNG_VERSION_STRING "${png_version_str}")
      unset(png_version_str)
  endif ()
endif()

include(${CMAKE_CURRENT_LIST_DIR}/FindPackageHandleStandardArgs.cmake)
find_package_handle_standard_args(PNG
                                  REQUIRED_VARS PNG_LIBRARY PNG_PNG_INCLUDE_DIR
                                  VERSION_VAR PNG_VERSION_STRING)

mark_as_advanced(PNG_PNG_INCLUDE_DIR PNG_LIBRARY )
```

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!

```
find_package(Bar 2.0 REQUIRED)
add_library(Foo ...)
target_link_libraries(Foo PRIVATE Bar::Bar)

install(TARGETS Foo EXPORT FooTargets
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib
  RUNTIME DESTINATION bin
  INCLUDES DESTINATION include
  )
install(EXPORT FooTargets
  FILE FooTargets.cmake
  NAMESPACE Foo::
  DESTINATION lib/cmake/Foo
  )
```

Export your library interface!

```
include(CMakePackageConfigHelpers)
write_basic_package_version_file("FooConfigVersion.cmake"
  VERSION ${Foo_VERSION}
  COMPATIBILITY SameMajorVersion
  )
install(FILES "FooConfig.cmake" "FooConfigVersion.cmake"
  DESTINATION lib/cmake/Foo
  )
```

```
include(CMakeFindDependencyMacro)
find_dependency(Bar 2.0)
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

- 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.

CPack secret

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:

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

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

```
include("release/CPackConfig.cmake")
set(CPACK_INSTALL_CMAKE_PROJECTS
  "debug;Foo;ALL;/"
  "release;Foo;ALL;/"
  )
```

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:

```
find_package(Foo 2.0 REQUIRED)
# ...
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

```
set(CMAKE_PREFIX_PATH "/prefix")
set(as_subproject Foo)

macro(find_package)
  if(NOT "${ARG0}" IN_LIST as_subproject)
    _find_package(${ARGV})
  endif()
endmacro()

add_subdirectory(Foo)
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`

```
set(CTEST_SOURCE_DIRECTORY "/source")
set(CTEST_BINARY_DIRECTORY "/binary")

set(ENV{CXXFLAGS} "--coverage")
set(CTEST_CMAKE_GENERATOR "Ninja")
set(CTEST_USE_LAUNCHERS 1)

set(CTEST_COVERAGE_COMMAND "gcov")
set(CTEST_MEMORYCHECK_COMMAND "valgrind")
#set(CTEST_MEMORYCHECK_TYPE "ThreadSanitizer")

ctest_start("Continuous")
ctest_configure()
ctest_build()
ctest_test()
ctest_coverage()
ctest_memcheck()
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:

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

Run like this:

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

Follow a naming convention for test names.

This simplifies filtering by regex.

Fail to compile

```
add_library(foo_fail STATIC EXCLUDE_FROM_ALL
  foo_fail.cpp
  )
add_test(NAME Foo.Fail
  COMMAND ${CMAKE_COMMAND}
    --build ${CMAKE_BINARY_DIR}
    --target foo_fail
  )
set_property(TEST Foo.Fail PROPERTY
  PASS_REGULAR_EXPRESSION "static assert message"
  )
```

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

```
#!/bin/bash
tester=$1
shift
# create temporary file
filename=$(ssh root@172.22.22.22 mktemp)
# copy the tester to temporary file
scp $tester root@172.22.22.22:$filename
# make test executable
ssh root@172.22.22.22 chmod +x $filename
# execute test
ssh root@172.22.22.22 $filename "$@"
# store success
success=$?
# cleanup
ssh root@172.22.22.22 rm $filename
exit $success
```

Cross Compiling

Toolchain.cmake

```
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 `#include`s 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 initialzied 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 assiciated 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

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

Enable warnings from outside the project

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

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

Personal Wishlist

- For each of the following ideas, I have started a prototype.
- Contibutions 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

```
target_precompile_headers(Foo
  PUBLIC
    "foo.h"
  PRIVATE
    <unordered_map>
  )
```

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 `#include`s 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]

[2] https://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

Lua VM

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

[3] not the other way round. This failed before.

Declarative Frontend

- 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!