C++ now

2025

Effective CTest

Daniel Pfeifer

CMake / CTest / CPack

Let’s go back in time.

C++ now 2017 MAY 15-20 Aspen, Colorado, USA

Daniel Pfeifer

Effective CMake

Variables are so CMake 2.8.12.

Modern CMake is about Targets and Properties!


Kitware
CMake / CTest / CPack
Open Source Tools to build, test, and install software
Bill Hoffman
bill.hoffman@kitware.com
BoostCon 2009

Why CMake

- A build system that just works
- A build system that is easy to use cross platform
- Typical Project without CMake (curl)

```
$ ls
CHANGES              RELEASE-NOTES  curl-config.in  missing
CMake                acinclude.m4   curl-style.el   mkinstalldirs
CMakeLists.txt       aclocal.m4     depcomp         notes
build        docs          notes~
COPYING              buildconf      include         packages
CVS                  buildconf.bat  install-sh      reconf
ChangeLog            compile        lib             sample.emacs
Makefile             config.guess   libcurl.pc.in   src
Makefile.am          config.sub     ltmain.sh       tests
Makefile.in          configure      m4              vc6curl.dsw
README               configure.ac   maketgz
$ ls src/
CMakeLists.txt   Makefile.riscos  curlsrc.dsp  hugehelp.h   version.h
CVS              Makefile.vc6     curlsrc.dsw  macos        writeenv.c
Makefile.Watcom  Makefile.vc8     curlutil.c   main.c       writeenv.h
Makefile.am      config-amigaos.h curlutil.h   makefile.amiga writeout.c
Makefile.b32     config-mac.h     getpass.c    makefile.dj  writeout.h
Makefile.in      config-riscos.h  getpass.h    mkhelp.pl
Makefile.inc     config-win32.h   homedir.c    setup.h
Makefile.m32     config.h.in      homedir.h    urlglob.c
Makefile.netware curl.rc          hugehelp.c   urlglob.h
```

Replacing all those files with a single `CMakeLists.txt`
“outsources the maintenance of the buildsystem to the CMake developers”.

Are we there yet?

CMake has become the de-facto standard build system for C++

- Pick a random C++ project on GitHub.
- See that there is a `CMakeLists.txt` file.
- See that there is no `Makefile`, or similar.
- Mission accomplished?

While you are looking at the project, what else do you see?

- Repeating patterns:
  - `.github/workflows/`, `.gitlab-ci.yml`, `.travis.yml`, `.appveyor.yml`, `.cirrus.yml`, `.circleci`, `.ci/`, **`contrib/`**, `CMakePresets.json`, `gcovr.cfg`, …
- Scripts in JavaScript, TypeScript, Python, Bash, PowerShell, etc. for:
  - building the project with different compilers
  - running static code analysis (clang-tidy, cppcheck, cpplint)
  - analyzing code coverage
  - performing and evaluating benchmarks
  - generating packages
- Thanks to CMake, those scripts are portable.
- And yet, they are duplicated across projects.

Mission Accomplished?

- There may be no platform-specific `Makefile`.
- But what about **other** platform-specific files?
- Maybe there is more that could be outsourced?

It is just sad

- Some projects advance their custom scripts so far, that they build web frontends for evaluating and visualizing their test measurements.
- But most projects use off-the-shelf solutions like Jenkins, GitLab CI, or GitHub Actions, which are so generic, that all they present is the plain command line output with no analysis at all.
- Your build failed. Go read “five pages of error messages” in a terminal view inside your web browser.

Introducing CTest

- You make know CTest as the tool for running tests in CMake projects.
- But like the `cmake` binary, the `ctest` binary has multiple modes.
- CTest can be used to script build pipelines and send the results to a dashboard.

beman.exemplar

		Configure		Build		Test				
Change ID	Jobs	Warnings	Errors	Warnings	Errors	Not Run	Failed	Passed	Coverage	Submit Time
0a275d	9	9	0	1	0	0	1	32	100.00%	20 hours ago
101f63	18	18	0	2	0	0	2	64	100.00%	1 days ago
e904ce	18	18	0	2	0	0	2	64	100.00%	3 days ago
fb698c	27	27	0	3	0	0	3	96	100.00%	5 days ago
ffc534	189	189	0	21	0	0	21	672	100.00%	8 days ago
e350d9	9	9	0	1	0	0	1	32	100.00%	29 days ago
81b08a	45	0	45	0	0	0	0	0		30 days ago
a78836	81	0	81	0	0	0	0	0		1 months ago
bdad9e	18	0	18	0	0	0	0	0		1 months ago
118e03	27	0	27	0	0	0	0	0		1 months ago
d09916	27	0	27	0	0	0	0	0		1 months ago
3ceff8	36	0	36	0	0	0	0	0		1 months ago
(unknown)	36	0	0	0	0	0	0	0		1 months ago
6674de	45	45	45	0	0	0	0	0		1 months ago
228080	45	45	18	0	0	0	0	0		1 months ago
f14457	9	9	9	0	0	0	0	0		1 months ago

Out of the box, CTest provides the following features

- Abstraction of version control system
- Build warnings / errors
- Test results
- Duration
- Resource usage
- Code coverage
- Memory Defects
- Custom metrics!
- file / image attachments

Why isn’t it widely used?

- Most features have been available for 20 years.
- Bill talked about Kitware’s test cycle at BoostCon 2009.
- Please tell me!

Whatever the reason, let's address it!

- "I did not know about any of those features!"
- "I heard about some of them, but I have no clue how to use them properly."
- "I know how to use them, but I consider them irrelevant."
- "I think they are relevant, but I have issues with the UX of CDash."

CMake's evolution rhymes with C++'s.

- Pre-standard C++
- ISO C++
- "Modern C++" (auto, lambda)
- "Post-Modern C++" (CppNow 2017)

- Legacy CMake (eg. `if()` command)
- Standard CMake
- "Modern CMake" (targets, properties)
- "Post-Modern CMake" (CppNow 2025)

CTest is still in the legacy, pre-modern era

- CMake is the de-facto standard for C++ projects.
- Paradigm shift from cmake 2 era to cmake 3 era. Post modern 4 era.
- Strong focus on **JSON** (eg file API).

- While CTest supports projects that are not build with CMake, it is not even used by the majority of CMake projects.
- Documentation is centered around a `DartConfiguration.tcl` file (Dart, TCL).
- Build results are submitted to dashboard as **XML**.
- Versioning control seems optimized for **CVS**.

Preparing your Project

Preparing your Project

- It is not required, but recommended to `include(CTest)` in your toplevel `CMakeLists.txt` file.
- Also in the top level directory, create a `CTestConfig.cmake` that defines:
  - The project's `Nightly` start time.
  - The `URL` of a server that accepts build results.
- Now, everybody can submit build and test results to that server.

```
set(CTEST_NIGHTLY_START_TIME "1:00:00 UTC")
set(CTEST_SUBMIT_URL
    "https://ci.purplekarrot.net/api/submit?project=beman.exemplar")
```

The existence and non-existence of files bear meaning.

There is a `CMakeLists.txt` file.

There is a `CMakeLists.txt` file.

“You are right to assume how to build that project.”

Next to the `CMakeLists.txt`, there is no `Makefile` or similar.

Next to the `CMakeLists.txt`, there is no `Makefile` or similar.

“We, the maintainers, use CMake to build that project.”

There is a `.github/workflows/ci.yml`, or similar.

There is a `.github/workflows/ci.yml`, or similar.

“My project satisfies the quality standards set by myself.”

There is a `.github/workflows/ci.yml`, or similar.

“My project satisfies the quality standards set by myself.”

“It works on my machine.”

There is a `CTestConfig.cmake`.

There is a `CTestConfig.cmake`.

“We challenge you to define a build pipeline that causes a failure.”

There is a `CTestConfig.cmake`, but NO build pipelines (also NO `CMakePresets.json`).

There is a `CTestConfig.cmake`, but NO build pipelines (also NO `CMakePresets.json`).

“We do not discriminate against any build pipeline.”

Make it a goal to have no build pipelines in your project.

The Challenge

The Challenge

- Let there be competition between maintainers of projects and build pipelines.
- Build pipeline maintainers try to cause build or test failures.
- Project maintainers try to avoid/fix them.
- Projects contain no code specific to build pipelines.
- Build pipelines contain no code specific to projects.

Share build pipelines

- Put build pipelines into a separate repository.
- Reuse build pipelines for a number of projects.
- You can do this with GitHub Actions:
  `https://github.com/purpleKarrot/nightly-builds`

Using CTest as a Dashboard Client

Legacy Configuration: **DartConfiguration.tcl**

```
Site: purplekarrot.net
BuildName: Linux-GCC
SourceDirectory: /home/daniel/Projects/beman-exemplar
BuildDirectory: /home/daniel/Projects/beman-exemplar/build
UpdateCommand: /usr/bin/git
ConfigureCommand: /usr/bin/cmake -GNinja ..
MakeCommand: /usr/bin/ninja
```

Run Legacy Dashboard Client

```
ctest --dashboard <dashboard>
ctest --test-model <model> --test-action <action>
```

- `<dashboard>` and `<model>` are one of `Experimental`, `Nightly`, `Continuous`.
- `<action>` is one of `Start`, `Update`, `Configure`, `Build`, `Test`, `Coverage`, `MemCheck`, `Submit`.
- `--test-action` can be provided multiple times.

Legacy Dashboard Client has hardcoded logic

- `--dashboard` performs actions `Start`, `Update`, `Configure`, `Build`, `Test`, `Coverage`, `Submit`
- `--dashboard Experimental` skips the `Update` action.
- `--dashboard NightlyMemoryCheck` performs `MemCheck` instead of `Test`.
- If the model is `Continuous` and the `Update` action does not find any changes, the build is cancelled.
- If the model is `Nightly`, the `Update` action tries to fetch the revision at `CTEST_NIGHTLY_START_TIME` defined in the project's `CTestConfig.cmake`. (This only works with `CVS`, `SVN`, `P4`).

The Effect of `include(CTest)`

- Defines the option `BUILD_TESTING`.
- Conditionally invokes `enable_testing()`.
- Writes a `DartConfiguration.tcl` file into the build directory.
- Defines build targets like `Experimental`,
  which performs `ctest --dashboard Experimental`.
- Another thing, that I will explain later.

To `include(CTest)` or not to `include(CTest)`

- Note that `include(CTest)` violates my guideline of "no build pipelines in your project".
- However, being able to rely on consistent behavior of `BUILD_TESTING` and especially *other thing* is valuable.
- If you don't mind too much about those build targets, my recommendation is to `include(CTest)`.

Make sure your project supports the default workflow.

Don’t break user assumptions. Allow knowledge transfer.

- Variables set by the user are respected (both env and `-D`).
- Testing is enabled by default, tests are built with the default build target.
- Chosing the build configuration is done the same way for any CMake project.

“Standard CMake” CTest Script

Legacy Configuration: **DartConfiguration.tcl**

```
Site: purplekarrot.net
BuildName: Linux-GCC
SourceDirectory: /home/daniel/Projects/beman-exemplar
BuildDirectory: /home/daniel/Projects/beman-exemplar/build
UpdateCommand: /usr/bin/git
ConfigureCommand: /usr/bin/cmake -GNinja ..
MakeCommand: /usr/bin/ninja


# Model and Actions are chosen via command line arguments.
# Logic is builtin.
```

Configuration: `CTestScript.cmake`

```
set(CTEST_SITE "purplekarrot.net")
set(CTEST_BUILD_NAME "Linux-GCC")
set(CTEST_SOURCE_DIRECTORY "/home/daniel/Projects/beman-exemplar")
set(CTEST_BINARY_DIRECTORY "${CTEST_SOURCE_DIRECTORY}/build")
set(CTEST_CMAKE_GENERATOR "Ninja")
find_program(CTEST_UPDATE_COMMAND "git")

ctest_start("Experimental")          # Model is set in the script.
ctest_update()
ctest_configure()                    # Actions are scripted.
ctest_build()                        # Custom conditions are possible.
ctest_test()
ctest_submit()
```

```
ctest --script CTestScript.cmake
```

Custom logic

```
ctest_update(RETURN_VALUE retval)
if(retval EQUAL 0)
  return()
elseif(retval LESS 0)
  message(FATAL_ERROR "Upgrade fails with exit code ${retval}")
endif()
```

Coverage and Dynamic Analysis

Coverage

```
set(ENV{CXXFLAGS} "--coverage")
set(CTEST_COVERAGE_COMMAND "/usr/bin/gcov")

# ...
ctest_test()
ctest_coverage()
```

Valgrind

```
set(ENV{CXXFLAGS} "-g")
set(CTEST_MEMORYCHECK_COMMAND "/usr/bin/valgrind")

# ...

ctest_memcheck()
```

Sanitizers

```
set(ENV{CXXFLAGS} "-fsanitize=thread -fno-omit-frame-pointer")
set(CTEST_MEMORYCHECK_TYPE "ThreadSanitizer")

# ...

ctest_memcheck()
```

Projects need zero configuration
to support coverage, valgrind, sanitizers.

Build and Test Parellelization

Use all available cores to build and test

```
cmake_host_system_information(RESULT nproc QUERY NUMBER_OF_LOGICAL_CORES)

ctest_build(PARALLEL_LEVEL ${nproc})

if(CTEST_MEMORYCHECK_COMMAND OR CTEST_MEMORYCHECK_TYPE)
  ctest_memcheck(PARALLEL_LEVEL ${nproc})
else()
  ctest_test(PARALLEL_LEVEL ${nproc})
endif()
```

Projects may control how their tests are parallelized.

Test properties for controlling how tests are parallelized

- If the test requires a known number of cores, set `PROCESSORS` and maybe also `PROCESSOR_AFFINITY`.
- If a test needs all system resources, set `RUN_SERIAL`.
- For fine grained resource allocation, set `RESOURCE_LOCK` or `RESOURCE_GROUPS`.

Compiler warnings

There is no case where enabling a warning
turns an unsuccessful build into a successful build.

Compiler warnings are not project requirements

- As a project maintainer,
  you may find it necessary to ***temprorarily disable*** individual compiler warnings.
  - At the directory level with `add_compile_options()`.
  - At the target level with `target_compile_options()`.
  - At the source file level with `set_source_files_properties(COMPILE_OPTIONS)`.
  - In source code regions with `#pragma`.
- It is a good thing that CMake **appends** compile options.
- Remove all code that enables compiler warnings from project files.
- Remove `-Werror` from project files (Using `CMAKE_COMPILE_WARNING_AS_ERROR` is ok).

Treat warnings as errors in your build pipeline

```
set(ENV{CXXFLAGS} "-Wall -Wextra")

ctest_configure(OPTIONS "--compile-no-warning-as-error")

ctest_build(NUMBER_ERRORS errors NUMBER_WARNINGS warnings)
if((errors GREATER 0) OR (warnings GREATER 0))
  ctest_submit()
  message(FATAL_ERROR "Build cancelled!")
endif()
```

When you return early in a build pipeline,
make sure all build results are submitted.

Allow warnings, as long as the number does not increase

```
file(STRINGS "~/warning_threshold.txt" threshold LIMIT_COUNT 1)

ctest_build(NUMBER_ERRORS errors NUMBER_WARNINGS warnings)
if((errors GREATER 0) OR (warnings GREATER threshold))
  ctest_submit()
  message(FATAL_ERROR "Build cancelled!")
endif()

file(WRITE "~/warning_threshold.txt" ${warnings})
```

More static code analysis tools

For users of Qt: Compile with Clazy

```
set(ENV{CC}              "/usr/bin/clang-18")
set(ENV{CXX}             "/usr/bin/clazy")
set(ENV{CLANGXX}         "/usr/bin/clang++-18")
set(ENV{CLAZY_CHECKS}    "level2")
```

- Project maintainers can
  - Disable clazy for a file with: `// clazy:skip`
  - Disable individual checks for the complete file: `// clazy:excludeall=check1,check2`
  - Disable individual checks for individual lines: `// clazy:exclude=check1,check2`

CMake has built-in support for many Linter tools

- Target properties:
  - `<LANG>_CLANG_TIDY`
  - `<LANG>_CPPCHECK`
  - `<LANG>_CPPLINT`
  - `<LANG>_ICSTAT` (new in 4.1)
  - `<LANG>_INCLUDE_WHAT_YOU_USE`
  - `LINK_WHAT_YOU_USE`
- Initialized by corresponding `CMAKE_<property>` variable.
- Source file property: `SKIP_LINTING`

Approach for static analysis tools

- Tools and checks are introduced in a build pipeline.
- Errors are fixed by the project maintainers.
- Project maintainers have the right to disable checks.

Build and Test Launchers

Launchers for Compiler, Linker, and Test

- Properties:
  - Target property: `<LANG>_COMPILER_LAUNCHER`
  - Target property: `<LANG>_LINKER_LAUNCHER`
  - Test property: `TEST_LAUNCHER`
- Initialized by corresponding `CMAKE_<property>` variable,
- which is initialized by corresponding environment variable.

Recommendations for Launchers

- Build pipeline: Define launchers via **environment variables**.
- Projects: Don't ever set them. Respect the build pipeline.
- Projects: Reset them to the default value in case of unfixable errors.

Launchers for Compiler, Linker, and Test

- Compiler launcher is a good place to set **ccache**, **sccache**, **distcc**.
- But launchers can also be used to inject custom static analysis tools.
- Your imagination is the limit!

Build Instrumentation

Build Output Scraping

- The `ctest_build()` command scrapes the build output for errors and warnings.
- Errors and warnings are sent to dashboard in `Build.xml`.
- What identifies an error or warning can be extended:
  - `CTEST_CUSTOM_ERROR_MATCH`, `CTEST_CUSTOM_ERROR_EXCEPTION`
  - `CTEST_CUSTOM_WARNING_MATCH`, `CTEST_CUSTOM_WARNING_EXCEPTION`
  - `CTEST_CUSTOM_ERROR_PRE_CONTEXT`, `CTEST_CUSTOM_ERROR_POST_CONTEXT`
  - `CTEST_CUSTOM_MAXIMUM_NUMBER_OF_ERRORS`, `CTEST_CUSTOM_MAXIMUM_NUMBER_OF_WARNINGS`

Why Build Instrumentation

- Some information is hard to retrieve from build log, even for verbose output.
- If there is an error in a header file, compiling which source file causes the error?
- What is the compiler command line? What is the exit code?
- What target does the file belong to?
- CMake, when it generates the build system, has all this information.
- It can wrap the compiler command line with a launcher tool.

Enable CTest Launchers

- Build pipeline: `set(CTEST_USE_LAUNCHERS ON)`
- Project: `include(CTest)` / `include(CTestUseLaunchers)`
- Project: Don't abuse `RULE_LAUNCH_COMPILE` for ccache!

Effect of CTest Launchers

- For each compilation that produces an error or warning, the following information is captured:
  - Target name, Language
  - Source file, Output file, Output type
  - Command line, Working directory
  - Stdout, Stderr, Exit code
  - Labels
- **But:** The output is not parsed into diagnostics.
- The number of warnings and errors reported depends on whether `CTEST_USE_LAUNCHERS` is used or not (this should be fixed).

Enable Experimental Instrumentation (CMake 4.0)

- Set environment variables:

```
CTEST_USE_INSTRUMENTATION=1
CTEST_USE_VERBOSE_INSTRUMENTATION=1
CTEST_EXPERIMENTAL_INSTRUMENTATION="a37d1069-1972-4901-b9c9-f194aaf2b6e0"
```

- Because CTest reads the environment variables early,
  they cannot be set in the build script.

Effect of Build Instrumentation

- All build commands are reported, not just failing ones.
- Compared to `CTEST_USE_LAUNCHERS`,
  the following additional information is captured:
  - Start timestamp
  - Duration
  - Configuration
  - Metrics: CPU load, memory usage
- But the following information is not captured:
  - Stdout
  - Stderr

Combining Launchers and Instrumentation

- Launchers and instrumentation can be enabled together.
- `cdash-proxy` combines both. Also parses diagnostics.
- See: `https://github.com/purpleKarrot/cdash-proxy`
- Goal: Become obsolete.

Feedback for Experimental Instrumentation

- I don't see the use case for `CTEST_USE_VERBOSE_INSTRUMENTATION`.
  You either want instrumentation or not.
- If instrumentation is extended to capture `stdout` and `stderr`,
  it can replace `CTEST_USE_LAUNCHERS`.
- Instrumentation should be allowed to be enabled from CTest script.
- Output should be parsed into diagnostics (using SARIF output).

Custom Metrics, Measurements

Experimental Instrumentation

Enabling instrumentation reports CPU load and memory usage as metrics not only for build commands, but also for tests.

Speaking of measurements, projects can add their own!

- Static measurement: Set the `MEASUREMENT` test property.
- Dynamic measurement: Output special `<CTestMeasurement>` XML.

```
std::cout << "<CTestMeasurement type=\"numeric/double\" name=\"score\">"
          << score
          << "</CTestMeasurement>\n";
```

- `numeric/double`: Displayed as a plot over time.
- `text/string`: Shown as text.
- `text/link`: Turned into hyperlink.
- `text/preformatted`: Newlines, whitespace, and ANSI color codes are preserved.

Attaching Files, Images

Attaching files to the build

- Files listed in `CTEST_NOTES_FILES` are encoded in `Notes.xml`.
- Files passed to `ctest_upload()` are encoded in `Upload.xml`.
- Files listed in `CTEST_EXTRA_SUBMIT_FILES` are submitted with the other XML files.
- `ctest_submit(CDASH_UPLOAD)` sends individual files after asking for permission.

Attaching files to the tests

- Test properties: `ATTACHED_FILES`, `ATTACHED_FILES_ON_FAIL`.
- `<CTestMeasurementFile>` XML in test output.

```
std::cout <<
  "<CTestMeasurementFile type=\"file\" name=\"TestData\">"        <<
  "/dir/to/data.csv</CTestMeasurementFile>\n"                     <<
  "<CTestMeasurementFile type=\"image/png\" name=\"TestImage\">"  <<
  "/dir/to/test_img.png</CTestMeasurementFile>\n";
```

File attachments on the dashboard

- Download link for non-image files.
- Images will be shown with `<img>`.
- Special case: An interactive image diff is shown when there are two or more of the following: `TestImage`, `ValidImage`, `BaselineImage`, `DifferenceImage2`.

Testing Frameworks

Use a testing framework!
Even if you think you don't need one.

Excuse Nr. 1

- “My tests are simple. Each test is its own executable.”
- OK, but when you have large, static libraries as dependencies, using a testing framework can save time and disk space.
- Alternative: `create_test_sourcelist()`

Excuse Nr. 2

- “`constexpr` all the things! I just define a library, if it builds, it is all good.”
- OK, but it would be good to have test failures be reported as test failures rather than build failures. Using a testing framework can produce more informative output, like the expected and actual values.
- Alternative:

```
add_library(project.test.static OBJECT EXCLUDE_FROM_ALL)
add_test(NAME project.static COMMAND ${CMAKE_COMMAND}
  --build ${CMAKE_BINARY_DIR}
  --target project.test.static
  )
```

Fine grained test registration

- Use a testing framework to link all tests to a single executable.
- In your `CMakeLists.txt`, still register tests individually.
- Allow CTest to schedule and parallelize tests.

```
include(GoogleTest)
gtest_discover_tests(beman.exemplar.tests.identity
  DISCOVERY_MODE PRE_TEST
  )
```

Test Framework Wishlist

- Someone please generalize `gtest_discover_tests()` so that it can be used with other test frameworks.
- `https://purplekarrot.net/blog/cmake-and-test-suites.html`

Test Framework Wishlist

- Someone please write a test framework that has **no builtin support for**:
  - parallelization
  - order randomization
  - test list formatting
- But with builtin support for CTest's dynamic **measurements** and **attachments**.

Testing Compilation Failures

```
add_library(project.test.failure OBJECT EXCLUDE_FROM_ALL)
add_test(NAME project.failure COMMAND ${CMAKE_COMMAND}
  --build ${CMAKE_BINARY_DIR}
  --target project.test.failure
  )
set_property(TEST project.failure PROPERTY WILL_FAIL TRUE)
```

- **Warning:** `WILL_FAIL` can turn your test into a watermelon (looks green from the outside, red inside).
- **Better:** Use `static_assert()` with a custom message. Set `PASS_REGULAR_EXPRESSION` property to match the error message.

When you test for failures,
make sure the test fails for the right reason.

Crosscompiling

Toolchain File

```
set(CMAKE_SYSTEM_NAME "Windows")
set(CMAKE_SYSTEM_PROCESSOR "x86_64")

set(__prefix "x86_64-w64-mingw32")

set(CMAKE_C_COMPILER   "/usr/bin/${__prefix}-gcc")
set(CMAKE_CXX_COMPILER "/usr/bin/${__prefix}-g++")
set(CMAKE_RC_COMPILER  "/usr/bin/${__prefix}-windres")

set(CMAKE_FIND_ROOT_PATH "/usr/${__prefix}/sys-root/mingw/")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)

set(CMAKE_CROSSCOMPILING_EMULATOR "/usr/bin/wine64")
```

Toolchain file is set by build pipeline

- Set platform.
- Set compilers.
- Set **crosscompiling emulator** (wine, qemu).

```
set(ENV{CMAKE_TOOLCHAIN_FILE} "mingw64.cmake")

ctest_configure()
```

How to define Tests in CMake

```
add_executable(foo.tests.bar)


target_sources(foo.tests.bar PRIVATE bar_test.cpp)

# The test command is a target.
# CMake generates a CTest test file with the absolute path,
# prefixed with the crosscompiling emulator.
add_test(NAME foo.bar COMMAND foo.tests.bar)
```

How to define Tests in CMake

```
add_executable(foo.tests.bar)


target_sources(foo.tests.bar PRIVATE bar_test.cpp)

# The test command is *NOT* a target (due to incomplete refactoring).
# CMake generates a CTest test file with the unmodified test command.
# CTest will try to locate an executable named `foo.testing.bar`.
add_test(NAME foo.bar COMMAND foo.testing.bar)
```

How to define Tests in CMake

```
add_executable(foo.tests.bar)


target_sources(foo.tests.bar PRIVATE bar_test.cpp)

# The test command is a generator expression.
# CMake reports typos early.
# The test will fail in the context of crosscompiling!
add_test(NAME foo.bar COMMAND $<TARGET_FILE:foo.tests.bar>)
```

How to define Tests in CMake

```
add_executable(foo.tests.bar)
add_executable(foo::tests::bar ALIAS foo.tests.bar)

target_sources(foo.tests.bar PRIVATE bar_test.cpp)

# The test command is an alias target.
# CMake reports typos early.
# The crosscompiling emulator is correctly applied.
add_test(NAME foo.bar COMMAND foo::tests::bar)
```

It is always a good idea to use alias targets.
Not limited to imported targets. Not limited to libraries.

Here be Dragons

Not Possible Yet

```
set(CTEST_CMAKE_GENERATOR "Ninja Multi-Config")
ctest_start()
ctest_configure()
ctest_build(CONFIG Debug)
ctest_build(CONFIG Release)
ctest_test(CONFIG Debug)                    # Not possible yet
ctest_test(CONFIG Release)                  # Not possible yet
ctest_package(CONFIGURATIONS Debug Release)  # Not possible yet
ctest_submit()
```

Summary

Reprise: The Challenge

- Let there be competition between maintainers of projects and build pipelines.
- Build pipeline maintainers try to cause build or test failures.
- Project maintainers try to avoid/fix them.
- Projects contain no code specific to build pipelines.
- Build pipelines contain no code specific to projects.

Thank you!