# CMake: Best Practices

Henry Schreiner

Software & Computing Roundtable 2021-2-2

Links: 



- The book: cliutils.gitlab.io/modern-cmake
- My blog: iscinumpy.gitlab.io
- The workshop: hsf-training.github.io/hsf-training-cmake-webpage
- This talk: gitlab.com/CLIUtils/modern-cmake-interactive-talk

# Intro to CMake

In [ ]:
```
cmake --version
```

# What is CMake?

Is it a build system?

# Build system example (Rake):

```
# 01-rake/Rakefile

task default: [:hello_world] do                    # hello_world.c
    puts 'All built'                              #       ↓
end                                              # hello_world
                                                #       ↓
file hello_world: ['hello_world.c'] do |t|          # default task
   sh "gcc #{t.prerequisites.join(' ')} -o #{t.name}"
end
```

In [ ]:

```
(cd 01-rake && rake)
```

## Features:

- Understands when to build/rebuild
- Doesn't understand how to build
- Generic; can be used for anything

## Examples

- `make`: Classic, custom syntax
- `rake`: Ruby make
- `ninja`: Google's entry, not designed to be hand written

# Build system generator

- Understands the files you are building
- System independent
- You give relationships
- Can find libraries, etc

CMake is two-stage; the configuration step runs CMake, the build step runs a build-system (`make`, `ninja`, IDE, etc).

> *Aside: Modern CMake can run the install step directly without invoking the build system agian.*

# Build system generator example (CMake):

```
# 01-rake/CMakeLists.txt
cmake_minimum_required(VERSION 3.11)

project(HelloWorld)

add_executable(hello_world hello_world.c)
```

In [ ]:

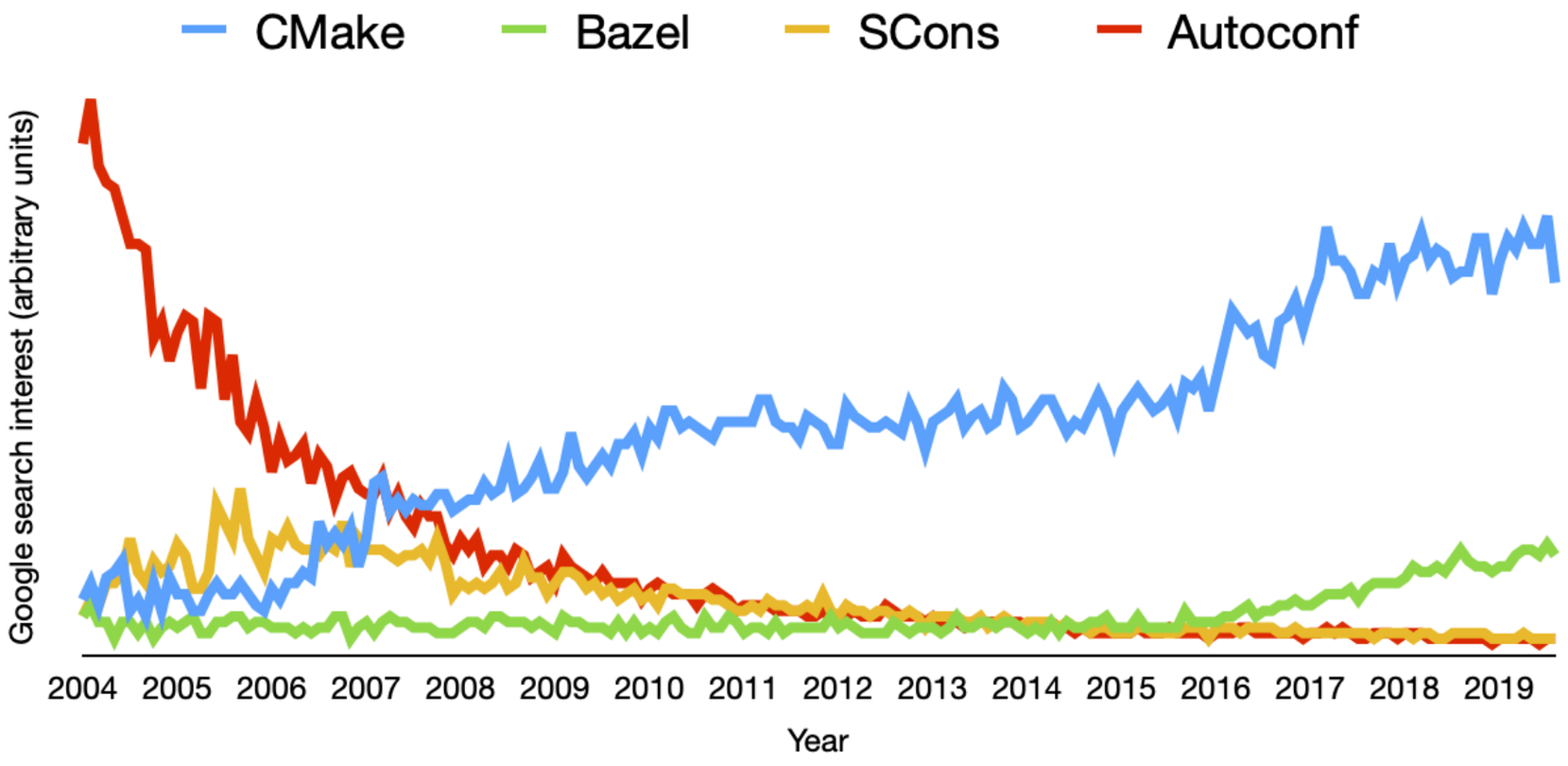
```
cmake -S 01-rake -B 01-build
cmake --build 01-build
```

## C/C++ Examples

- `cmake`: Cross-platform Make (also Fortran, CUDA, C#, Swift)
- `scons`: Software Carpentry Construction (Python)
- `meson`: Newer Python entry
- `basel`: Google's build system
- Other languages often have *their own build system* (Rust, Go, Python, ...)

> *We will follow common convention and call these "build-systems" for short from now on*

# Why CMake?




It has become a standard

- Approximately all IDEs support it
- Many libraries have built-in support
- Integrates with almost everything

## Custom Buildsystems are going away

- Qt 6 dropped QMake for CMake (note: C++17 only too)
- Boost is starting to support CMake at a reasonable level along with BJam
- Standout: Google is dual supporting Bazel and CMake

## Custom Buildsystems are going away

- Qt 6 dropped QMake for CMake (note: C++17 only too)
- Boost is starting to support CMake at a reasonable level along with BJam
- Standout: Google is dual supporting Bazel and CMake

## Recent highlights

- Thrust just received a major CMake overhaul
- TBB / Intel PSTL nicely support CMake in recent years
- Pybind11's CMake support was ramped up in 2.6

# (More) Modern CMake

- CMake is a new language to learn (and is a bit odd)
- Classic CMake (CMake 2.8, from 2009) was ugly and had problems, but that's not Modern CMake!

- *Modern CMake* and *More Modern CMake*!
    - CMake 3.0 in 2014: Modern CMake begins
    - CMake 3.1-3.4 had important additions/fixes
    - CMake 3.12 in mid 2018 completed the "More Modern CMake" phase
    - Current CMake is 3.19 (2.20 in rc2 phase)
- Eras of CMake
    - Classic CMake: Directory based
    - Modern CMake: Target based
    - More Modern CMake: Unified target behavior
    - CURRENT: Powerful CLI

# Best Practice: Minimum Version is important!

CMake has a (AFAIK) unique version system.

If a file start with this:

```
cmake_minimum_required(VERSION 3.0)
```

Then CMake will set all *policies* (which cover all *behavior* changes) to their 3.0 settings. This in theory means it is extremely backward compatible; upgrading CMake will not break or change anything at all. In practice, all the behavior changes had very good reasons to change, so this will be much buggier and less useful than if you set it higher!

You can also do this:

```
cmake_minimum_required(VERSION 3.4...3.14)
```

Then

- CMake < 3.4 will be an error
- CMake 3.4 -- 3.11 will set 3.4 policies (feature was introduced in 3.12, but syntax is valid)
- CMake 3.12 -- 3.14 will set current policies
- CMake 3.15+ will set 3.14 policies

- Don't: set this low without a range - it will harm users. Setting < 3.9 will break IPO, for example. Often in a way that can't be fixed by superprojects.
- Do: set the highest minimum you can (build systems are hard/ugly enough as it is)
- Do: test with the lowest version version you support in at least one job.
- Don't: expect a CMake version significanly *older* than your compiler to work with it (expecially macOS/Windows, or CUDA).

## What minimum to choose - OS support:

- 3.4: The bare minimum. Never set less.
- 3.7: Debian old-stable.
- 3.10: Ubuntu 18.04.
- 3.11: CentOS 8 (use EPEL or AppSteams, though)
- 3.13: Debian stable.
- 3.16: Ubuntu 20.04.
- 3.18: pip
- 3.19: conda-forge/chocolaty/direct download, etc. First to support Apple Silicon.

## What minimum to choose - Features:

- 3.8: C++ meta features, CUDA, lots more
- 3.11: IMPORTED INTERFACE setting, faster, FetchContent, COMPILE_LANGUAGE in IDEs
- 3.12: C++20, `cmake --build build -j N`, `SHELL:`, FindPython
- 3.14/3.15: CLI, FindPython updates
- 3.16: Unity builds / precompiled headers, CUDA meta features
- 3.17/3.18: Lots more CUDA, metaprogramming

# Best Practice: Running CMake

The classic method:

```
mkdir build
cd build
cmake ..
make
```

The modern method is cleaner and more cross-platform-friendly:

```
cmake -S . -B build
cmake --build build
```

(CMake 3.14/3.15) supports `-v` (verbose), `-j N` (threads), `-t target`, and more

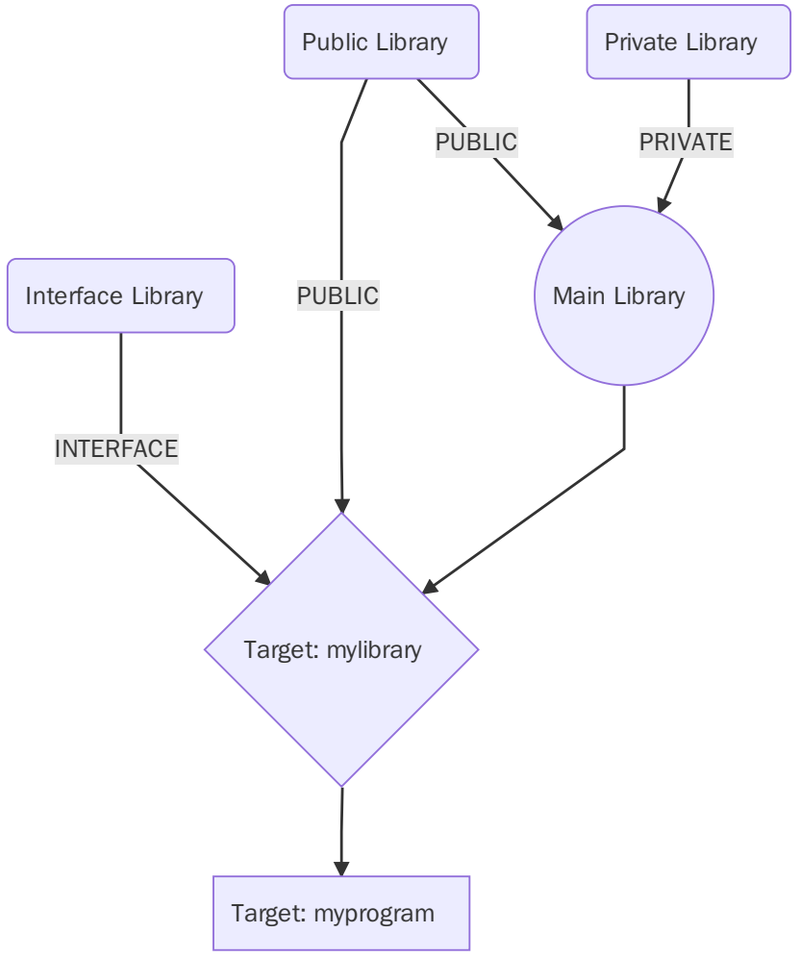
## Example options:

In [ ]:

```
cmake --build 01-build -v
```

# Best Practice: Use Targets

Excutables and libraries are *targets*



Properties include:

- Header include directories
- Compile flags and definitions
- Link flags
- C++ standard and/or compiler features required
- Linked libraries

Note that other things include properties, like files (such as LANGUAGE), directories, and global

## Tips for packaging

- Do: use targets.
- Don't: use common names for targets if you want to be used in subdirectory mode.
- Don't: write a FindPackage for your own package. Always provide a PackageConfig instead.

Export your targets to create a `PackageTargets.cmake` file. You can write a `PackageConfig.cmake.in`; you can recreate / reimport package there (generally use a shared `X.cmake` file instead of doing it twice).

- Don't: hardcode any system/compiler/config details into exported targets. Only from shared code in Config, or use Generator expressions.

Never allow your package to be distributed without the config files!

## Using other projects

1. See if the project provides a `<name>Config.cmake` file. If it does, you are good to go!
2. If not, see if CMake provides a Find package for it built-in. If it does, you are good to go!
3. If not, see if the authors provide a `Find<name>.cmake` file. If they do, complain loudly that they don't follow CMake best practices.
4. Write your own `Find<name>.cmake` and include in in a helper directory. You'll need to build IMPORTED targets and all that.
5. You can also use `FindPkgConfig` to piggy back on classic pkg-config.

# Best Practice: Handling remote dependencies

CMake can download your dependencies for you, and can integrate with files. It supports composable (sub-)projects: One project can include another

Does not have namespaces, can cause target collisions!

- Build time data and project downloads: `ExternalProject` (classic)
- Configure time downloads `FetchContent` (new in 3.11+)
- You can also use submodules (one of my favorite methods, but use with care)
- You can also use Conan.io's CMake integration

FetchContent

# FetchContent

`02-fetch/hello_fmt.cpp`

```cpp
#include <fmt/format.h>

int main() {
    fmt::print("The answer is {}\n", 42);
    return 0;
}
```

`02-fetch/CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.14)
project(HelloWorld LANGUAGES CXX)

include(FetchContent)
FetchContent_Declare(
  fmt
  GIT_REPOSITORY https://github.com/fmtlib/fmt.git
  GIT_TAG        5.3.0)
FetchContent_MakeAvailable(fmt) # Shortcut from CMake 3.14

add_executable(hello_world hello_fmt.cpp)
target_link_libraries(hello_world PRIVATE fmt::fmt)
target_compile_features(hello_world PRIVATE cxx_std_11)
```

In [ ]:

```
cmake -S 02-fetch -B 02-build
cmake --build 02-build
```

## Conan.io's conan-cmake

Conan.io has a nice CMake integration tool. It *should* support binaries too, since Conan.io supports them! You must have conan installed, though - I'm using conan from conda-forge. It works with old versions of CMake, as well.

```
cmake_minimum_required(VERSION 3.14)

project(HelloWorld LANGUAGES CXX)
```

```
# Conan bootstrap
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
  message(
    STATUS
      "Downloading conan.cmake from https://github.com/conan-io/cmake-conan")
  file(DOWNLOAD "https://github.com/conan-io/cmake-conan/raw/v0.16.1/conan.cmake"
       "${CMAKE_BINARY_DIR}/conan.cmake")
endif()

include("${CMAKE_BINARY_DIR}/conan.cmake")
conan_check(REQUIRED)

conan_cmake_run(
  REQUIRES fmt/6.1.2
  BASIC_SETUP CMAKE_TARGETS
  BUILD missing)
```

```
add_executable(hello_world hello_fmt.cpp)
target_link_libraries(hello_world PRIVATE CONAN_PKG::fmt)
target_compile_features(hello_world PRIVATE cxx_std_11)
```

You can also make a `conanfile.txt` and manage all your dependencies there.

In [ ]:

```
cmake -S 02b-conan -B 02b-build -DCMAKE_BUILD_TYPE=Release
cmake --build 02b-build
```

# Best Practice: Use IMPORTED targets for things you don't build

- Now (3.11+) can be built with standard CMake commands!
- Can now be global with `IMPORTED_GLOBAL`
- You'll need to set them back up in your Config file (place in one common location for both CMakeLists and Config).

```
add_library(ExternLib IMPORTED INTERFACE)

# Classic                    # Modern
set_property(                 target_include_directories(
  TARGET                        ExternLib INTERFACE /my/inc
    ExternLib                 )
  APPEND
  PROPERTY
    INTERFACE_INCLUDE_DIRECTORIES
      /my/inc
)
```

# Best Practice: Use CUDA as a language

Cuda is now a first-class language in CMake! (3.9+) Replaces FindCuda.

```
project(MY_PROJECT LANGUAGES CUDA CXX) # Super project might need matching? (3.14 at least)

# Or for optional CUDA support
project(MY_PROJECT LANGUAGES CXX)
include(CheckLanguage)
check_language(CUDA)
if(CMAKE_CUDA_COMPILER)
    enable_language(CUDA)
endif()
```

Much like you can set C++ standards, you can set CUDA standards too:

```
if(NOT DEFINED CMAKE_CUDA_STANDARD)
    set(CMAKE_CUDA_STANDARD 11) # Probably should be cached!
    set(CMAKE_CUDA_STANDARD_REQUIRED ON)
endif()
```

Much like you can set C++ standards, you can set CUDA standards too:

```
if(NOT DEFINED CMAKE_CUDA_STANDARD)
    set(CMAKE_CUDA_STANDARD 11) # Probably should be cached!
    set(CMAKE_CUDA_STANDARD_REQUIRED ON)
endif()
```

You can add files with `.cu` extensions and they compile with nvcc. (You can always set the LANGUAGE property on a file, too). Separable compilation is a property:

```
set_target_properties(mylib PROPERTIES
                      CUDA_SEPERABLE_COMPILATION ON)
```

## New for CUDA in CMake 3.18 and 3.17!

- A new `CUDA_ARCHITECTURES` property
- You can now use Clang as a CUDA compiler
- `CUDA_RUNTIME_LIBRARY` can be set to shared
- `FindCUDAToolkit` is finally available
- I rewrote the CUDA versions support in 3.20 to be more maintainable and more accurate.

# Best Practice: Check for Integrated Tools First

CMake has a *lot* of great tools. When you need something, see if it is built-in first!

Useful properties (with `CMAKE_*` variables, great from the command line_:

- `INTERPROCEDURAL_OPTIMIZATION`: Add IPO
- `POSITION_INDEPENDENT_CODE`: Add `-fPIC`
- `<LANG>_COMPILER_LAUNCHER`: Add `ccache`
- `<LANG>_CLANG_TIDY`
- `<LANG>_CPPCHECK`
- `<LANG>_CPPLINT`
- `<LANG>_INCLUDE_WHAT_YOU_USE`

Useful modules:

- `CheckIPOSupported`: See if IPO is supported by your compiler
- `CMakeDependentOption`: Make one option depend on another
- `CMakePrintHelpers`: Handy debug printing
- `FeatureSummary`: Record or printout enabled features and found packages

## Using clang-tidy in GitHub Actions example `.github/workflow/format.yml`:

```
on:
  pull_request:
  push:

jobs:
  clang-tidy:
    runs-on: ubuntu-latest
    container: silkeh/clang:10
    steps:
    - uses: actions/checkout@v2

- name: Configure
      run: cmake -S . -B build -DCMAKE_CXX_CLANG_TIDY="$(which clang-tidy);--warnings-as-errors=*"
    - name: Run
      run: cmake --build build -j 2
```

# Best Practice: Use CMake-Format

There is a CMake formatting tool now too, called `cmake-format`! You should use it to keep your CMake code from becoming messy and keeping it easy to merge conflicts.

Since you should *always* use pre-commit for formatting and style checking, here's the `.pre-commit-config.yaml`:

```yaml
# CMake formatting
- repo: https://github.com/cheshirekow/cmake-format-precommit
  rev: v0.6.13
  hooks:
  - id: cmake-format
    additional_dependencies: [pyyaml]
    types: [file]
    files: (\.cmake|CMakeLists.txt)(.in)?$
```

And here's the GitHub Actions job:

```
jobs:
  pre-commit:
    name: Format
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - uses: actions/setup-python@v2
    - uses: pre-commit/action@v2.0.0
```

## CompilerDetection and Flag checking

`try_compile` / `try_run` can tell you if a flag or file works. However, first check:

- `CheckCXXCompilerFlag`
- `CheckIncludeFileCXX`
- `CheckStructHasMember`
- `TestBigEndian`
- `CheckTypeSize`

## ~~Best~~ Bad Practice: `WriteCompilerDetectionHeader` will write out C/C++ macros for your compiler for you!

This has been removed in CMake 3.20. Feel free to set the minimum required to below 3.20, or find another tool, generate once, then keep the generated copy.

03-compiler/CMakeLists.txt

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

include(WriteCompilerDetectionHeader)

write_compiler_detection_header(
  FILE my_compiler_detection.h
  PREFIX MyPrefix
  COMPILERS
    GNU Clang MSVC Intel
  FEATURES
    cxx_variadic_templates
    cxx_nullptr
)
```

In [ ]:

```
cmake -S 03-compiler -B 03-build
```

# Best Practice: Use FindPython and the newest possible CMake (3.18.2+ best)

FindPython is an exciting new way to discover Python.

- venv/conda ready.
- Multiple runs (using unique caching system).
- Not very usable till 3.15, not very usable for PyPy until 3.18.2+ and PyPy 7.3.2 (about a week old)
    - But possibly vendorable to 3.7+

## Scikit-build and CMake wheels

Scikit-build is an adaptor for setuptools (not a true PEP 517 builder). Combined with `pyproject.toml` and Pip 10, it can be really useful, though!

- Still not quite as reliable as setuptools (distutils) in non-standard setups
- Doesn't work well without CMake wheels (but you can bypass PEP 518 if CMake is already installed)
- A little rough around some corners
- Development a bit stuck.

# Further investigation:

If we have some spare time, I can show you through the CMake systems I've helped design:

- github.com/pybind/pybind11
- github.com/pybind11/scikit_build_example
- github.com/cliutils/cli11
- github.com/scikit-hep/boost-histogram (Dual CMake / setuptools)
- github.com/goofit/goofit (CUDA and Scikit-Build)

I highly recommend my course hsf-training.github.io/hsf-training-cmake-webpage and book cliutils.gitlab.io/modern-cmake on CMake. Everything is linked from my blog, iscinumpy.gitlab.io.

Also, CMake's documentation is fantastic, if you already know what you are looking for, it is some of the best out there. The "new in" directives seem to be finally added in the 3.20 documentation! No more scrolling though old versions!

In [ ]: