Modern CMake

Sean McDonough

Introduction

- Build systems
- What is Cmake?
- *Modern* Cmake
- A full fledged example project

Motivation

- C++ is a compiled language (can’t pip install antigravity)
- C++ does not have a standardized package manager
- Projects are getting larger, and are integrating more (open source) components
- Cmake is the defacto standard for the C++ community

Lamentations

1. Where is the source located?
2. How do I download the source?
3. Do I need to install a pre-built binary?
    1. What about ABI considerations?
    2. What if I'm cross-compiling?
    3. Even then, how do I resolve `#include` directives?
4. Do I need to be a privileged user to install?
5. After installing, how do I export it to my library?
6. If building from source, what other dependencies do I need to obtain first?
    1. (For each dependency, recurse on this list.)
7. Will I be able to install multiple versions of individual libraries on the same host?
8. Do any of the transitive dependencies require that I perform some wacky out-of-band step?
9. In some cases: How do I omit platform-specific dependencies?

vector<bool>

Lamentations


1. Where is the source located?


2. How do I download the source?

3. Do I need to install a pre-built binary?
    1. What about ABI considerations?
    2. What if I'm cross-compiling?
    3. Even then, how do I resolve `#include` directives?

4. Do I need to be a privileged user to install?


5. After installing, how do I export it to my library?

6. If building from source, what other dependencies do I need to obtain first?
    1. (For each dependency, recurse on this list.)

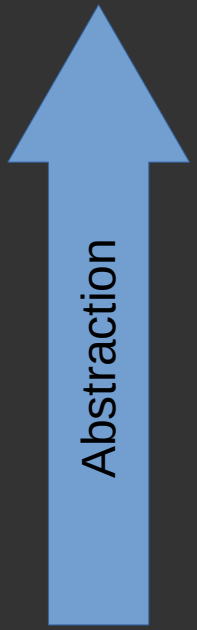

7. Will I be able to install multiple versions of individual libraries on the same host?

8. Do any of the transitive dependencies require that I perform some wacky out-of-band step?


9. In some cases: How do I omit platform-specific dependencies?

vector<bool>

Tool Time

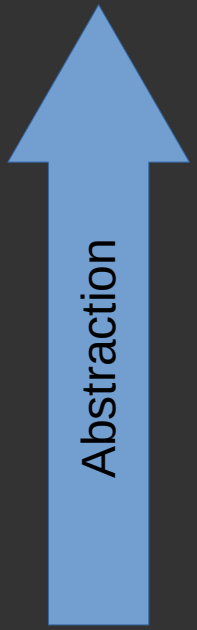


Package Manager

Meta Build System

Build System

Tool Time



Package Manager

Conan, vcpkg, dds

Meta Build System

Cmake, meson, premake

Build System

VC Proj, make, Ninja, Bazel

Meta Build System

- A tool which generates native build files by defining a more generic configuration
- Language agnostic
  - Cmake supports: C, C++, Fortran, CUDA, ObjC, ObjC++, ASM
  - Meson supports: C, C++, D, Fortran, Java, Rust
- Manage complex and cross platform build systems

Modern CMake

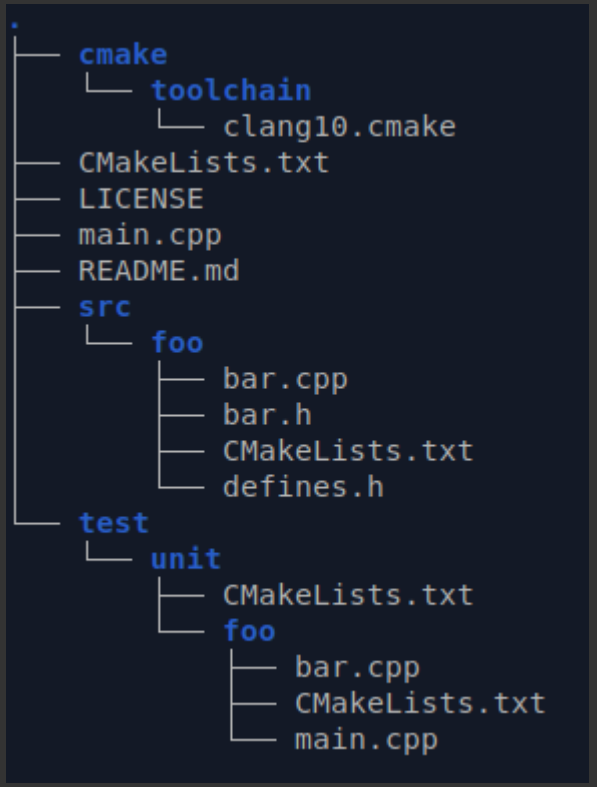
- Cmake has been in development since 2000
- Cmake vesion 3.0 introduced targets
  - The start of the “modern” era
- This talk is targeting versions >3.16

Usage

- Can be run from the command line or gui
  - Or curses using ccmake
- Out of source build(s)
  - Multiple configurations (e.g. Release, Debug, cross compiled)
- Point to the project root directory
- “-D<VAR>” appends a variable to the cache

Situational Awareness

- Root of the project contains the CmakeLists.txt entrypoint
- Properties propogate to all child subdirectories
- Targets are global in scoped
- Available on GitHub



Getting Started

- Very little is required to start a CMake project
  - cmake_minimum_required()
  - Project()
  - *something to build*

- Defined in CMakeLists.txt files

```
# We require a recent version of CMake to
# utilize modern features of the language
cmake_minimum_required(VERSION 3.16.0)
project(
  cmake-example
  LANGUAGES CXX
  VERSION 0.1.0)
```

```
.
├── cmake
│   └── toolchain
│       └── clang10.cmake
├── CMakeLists.txt
├── LICENSE
├── main.cpp
├── README.md
├── src
│   └── foo
│       ├── bar.cpp
│       ├── bar.h
│       ├── CMakeLists.txt
│       └── defines.h
└── test
    └── unit
        ├── CMakeLists.txt
        └── foo
            ├── bar.cpp
            ├── CMakeLists.txt
            └── main.cpp
```

Targets

- A target is any singular piece of code that can be built*
  - E.g. executables, static/shared libraries
- Targets are the fundemental unit of modern Cmake
- Analogous to objects (inheritance/composition)

```
add_library(foo SHARED)

# Set target properties
target_include_directories(foo PUBLIC .)
target_sources(foo PRIVATE bar.cpp)
target_compile_options(foo PUBLIC -Wconversion)
```

Targets

- A target is any singular piece of code that can be built*
  - E.g. executables, static/shared libraries

- To build we need source code
  - target_sources()
  - Relative pathing!

```
.
├── cmake
│   └── toolchain
│       └── clang10.cmake
├── CMakeLists.txt
├── LICENSE
├── main.cpp
├── README.md
├── src
│   └── foo
│       ├── bar.cpp
│       ├── bar.h
│       ├── CMakeLists.txt
│       └── defines.h
└── test
    └── unit
        ├── CMakeLists.txt
        └── foo
            ├── bar.cpp
            ├── CMakeLists.txt
            └── main.cpp
```

```cmake
add_library(foo SHARED)

# Set target properties
target_include_directories(foo PUBLIC .)
target_sources(foo PRIVATE bar.cpp)
target_compile_options(foo PUBLIC -Wconversion)
```

Targets

- Targets have properties
- Their properties have two lifetimes
  - Build requirements
    - E.g. sources, compiler flags, headers
  - Usage requirements

    E.g. headers

```
add_library(foo SHARED)

# Set target properties
target_include_directories(foo PUBLIC .)
target_sources(foo PRIVATE bar.cpp)
target_compile_options(foo PUBLIC -Wconversion)
```

```
.
├── cmake
│   └── toolchain
│       └── clang10.cmake
├── CMakeLists.txt
├── LICENSE
├── main.cpp
├── README.md
├── src
│   └── foo
│       ├── bar.cpp
│       ├── bar.h
│       ├── CMakeLists.txt
│       └── defines.h
└── test
    └── unit
        ├── CMakeLists.txt
        └── foo
            ├── bar.cpp
            ├── CMakeLists.txt
            └── main.cpp
```

Targets

- Targets have properties
- Their properties have two lifetimes
  - Build requirements (PRIVATE)
    - E.g. sources, compiler flags, headers
  - Usage requirements (INTERFACE)

    E.g. headers


```
add_library(foo SHARED)

# Set target properties
target_include_directories(foo PUBLIC .)
target_sources(foo PRIVATE bar.cpp)
target_compile_options(foo PUBLIC -Wconversion)
```

Targets

- Targets have properties
- Their properties have two scopes
  - Build requirements (PRIVATE)
    - E.g. sources, compiler flags, headers
  - Usage requirements (INTERFACE)

    E.g. headers


```
add_library(foo SHARED)

# Set target properties
target_include_directories(foo PUBLIC .)
target_sources(foo PRIVATE bar.cpp)
target_compile_options(foo PUBLIC -Wconversion)
```

Using Targets

- Motivating example: unit testing our library

- The unit test target (ut_foo) inherits from:
  - foo: code under test
  - Catch2: a unit testing framework

```
.
├── cmake
│   └── toolchain
│       └── clang10.cmake
├── CMakeLists.txt
├── LICENSE
├── main.cpp
├── README.md
├── src
│   └── foo
│       ├── bar.cpp
│       ├── bar.h
│       ├── CMakeLists.txt
│       └── defines.h
└── test
    └── unit
        ├── CMakeLists.txt
        └── foo
            ├── bar.cpp
            ├── CMakeLists.txt
            └── main.cpp
```

```cmake
add_executable(ut_foo)

target_sources(ut_foo PRIVATE main.cpp bar.cpp)
target_link_libraries(ut_foo PRIVATE Catch2::Catch2 foo)

# Links catch tests into CTest
catch_discover_tests(ut_foo)
```

Using Targets

- Motivating example: unit testing our library
- The unit test target (ut_foo) inherits fro
  - foo: code under test
  - Catch2: a unit testing framework

```
add_executable(ut_foo)

target_sources(ut_foo PRIVATE main.cpp bar.cpp)
target_link_libraries(ut_foo PRIVATE Catch2::Catch2 foo)

# Links catch tests into CTest
catch_discover_tests(ut_foo)
```

```
.
├── cmake
│   └── toolchain
│       └── clang10.cmake
├── CMakeLists.txt
├── LICENSE
├── main.cpp
├── README.md
├── src
│   └── foo
│       ├── bar.cpp
│       ├── bar.h
│       ├── CMakeLists.txt
│       └── defines.h
└── test
    └── unit
        ├── CMakeLists.txt
        └── foo
            ├── bar.cpp
            ├── CMakeLists.txt
            └── main.cpp
```

Fetch Content

- Cmake blurs the line between package manager and meta build system

- External library can be imported, built and used
  - Warning: library *must* be a cmake project

```
FetchContent_Declare(
  catch2
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG master)

# This line configures/builds the item specifed in _Declare and is done at
# configuration time. Targets in the Catch2 namespace are now avaible for use.
FetchContent_MakeAvailable(catch2)

# A slightly heavy handed approach to add a specific script into our cmake
# project. Used for catch_discover_tests()
include(${catch2_SOURCE_DIR}/contrib/Catch.cmake)

add_subdirectory(foo)
```

Fetch Content

- Some use cases:
  - Library is header only
  - Cross compiling or using custom compilation flags

```
FetchContent_Declare(
  catch2
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG master)

# This line configures/builds the item specifed in _Declare and is done at
# configuration time. Targets in the Catch2 namespace are now avaible for use.
FetchContent_MakeAvailable(catch2)

# A slightly heavy handed approach to add a specific script into our cmake
# project. Used for catch_discover_tests()
include(${catch2_SOURCE_DIR}/contrib/Catch.cmake)

add_subdirectory(foo)
```

Fetch Content

- What can be fetched?
  - Archives, via URL/Path
  - Repositories (e.g. Git/Subversion/Mercurial/CVS)
  - User command

```
FetchContent_Declare(
  catch2
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG master)

# This line configures/builds the item specifed in _Declare and is done at
# configuration time. Targets in the Catch2 namespace are now avaible for use.
FetchContent_MakeAvailable(catch2)

# A slightly heavy handed approach to add a specific script into our cmake
# project. Used for catch_discover_tests()
include(${catch2_SOURCE_DIR}/contrib/Catch.cmake)

add_subdirectory(foo)
```

Fetch Content

- WARNING: content is downloaded *per build*
  - May become unsutainable for larger projects with many dependencies
  - Package managers may be a better solution at dependency management

```
FetchContent_Declare(
  catch2
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG master)

# This line configures/builds the item specifed in _Declare and is done at
# configuration time. Targets in the Catch2 namespace are now avaible for use.
FetchContent_MakeAvailable(catch2)

# A slightly heavy handed approach to add a specific script into our cmake
# project. Used for catch_discover_tests()
include(${catch2_SOURCE_DIR}/contrib/Catch.cmake)

add_subdirectory(foo)
```

Find Package

- Useful if a package is installed on the system
  - E.g. large or expensive to compile libraries
- Can specify a minimum/exact version and components
- Modern Find* utilities will provide targets

```
find_package(Boost 1.45.0 REQUIRED COMPONENTS filesystem)
target_link_libraries(foo PRIVATE Boost::filesystem)
```

Find Package

- Can roll your own if one is not provided
- An exercise for the audience
- Please share!
  - There are several FindCosmic.cmake scripts

```
find_package(Boost 1.45.0 REQUIRED COMPONENTS filesystem)
target_link_libraries(foo PRIVATE Boost::filesystem)
```

Cross Compiling with Toolchains

- Each build can have a different compiler

- Specified at configuration time using CMAKE_TOOLCHAIN_FILE

```
sean@fi:~/devel/cmake-example/build_gcc$ cmake ..
-- The CXX compiler identification is GNU 9.3.0
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
```

```
sean@fi:~/devel/cmake-example/build_clang$ cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/toolchain/clang10.cmake ..
-- The CXX compiler identification is Clang 10.0.0
-- Check for working CXX compiler: /usr/bin/clang++-10
-- Check for working CXX compiler: /usr/bin/clang++-10 -- works
```

Cross Compiling with Toolchains

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR x86_64)

find_program(CMAKE_C_COMPILER clang-10 REQUIRED)
find_program(CMAKE_CXX_COMPILER clang++-10 REQUIRED)

# This needs to be done so the value is persistent
set(CMAKE_C_COMPILER
    "${CMAKE_C_COMPILER}"
    CACHE STRING "C compiler" FORCE)
set(CMAKE_CXX_COMPILER
    "${CMAKE_CXX_COMPILER}"
    CACHE STRING "C++ compiler" FORCE)
```

Generator Expressions

- A functional to express requirements at configuration
  - Akin to templates in C++

- Useful for setting architecture/build requirements

```
# Flags for all targets in this project
add_compile_options("$<$<CXX_COMPILER_ID:MSVC>:/W4>"
                    "$<$<CXX_COMPILER_ID:MSVC>:/WX>"
                    "$<$<NOT:$<CXX_COMPILER_ID:MSVC>>:-Wall>"
                    "$<$<NOT:$<CXX_COMPILER_ID:MSVC>>:-pedantic>"
                    "$<$<NOT:$<CXX_COMPILER_ID:MSVC>>:-Werror>")
```

Takeaway

- Use the latest version of Cmake possible
- ***Use targets!***
- External dependencies
  - If prebuilt use find_package
  - If building as part of your project use FetchContent

Questions?

https://www.newyorker.com/cartoons/bob-mankoff/the-creative-life-cycle-of-a-cartoonist

Bibliography

- Effective Modern CMake
- (Oh No!) More Modern CMake
- It's Time To Do CMake Right
- Effective Modern CMake
- Cmake Documentation