rix0r.nl

The Ultimate Guide to Modern CMake

13 August 2015

CMake is a great tool for managing a C++ system's build. It builds quickly, supports the major use cases, and is quite flexible. The problem is, it's *too* flexible, and for people used to writing Makefiles themselves, it's not always obvious what CMake commands and properties you should be using. If you're used to just splatting some compiler flags into the `CXX_FLAGS` environment variable, you might just do that in CMake as well, even though it supports better ways to manage your build.

It occurred to me recently that I need a gold standard reference for laying out a CMake project. There are a lot of "CMake tutorial" style articles out there, but they're of the form `add_executable`, `add_include_directories`, bam done. However, those projects are not easily portable, and their libraries are not easily reused in different contexts.

Instead, I want CMake to do proper dependency management for me. I don't want to be managing include directories (especially transitive include directories!), linker command lines and all that jazz. And I want to be able to import my libraries So I made this page to refer myself and others to in the future, so we can have a single point of truth and reference for these docs.

Most of this is gleaned from Daniel Pfeifer's presentation and the CMake docs.

Requirements

I have a number of requirements on my build:

- I want to define nothing more than modules, and dependencies between modules. CMake should automatically figure out transitive dependencies and set up the include paths and linker paths correctly. If I'm not using Boost, but my dependency is, and *I* have to think about that, I've lost.
- When I define my module, without extra effort I want it to:
  - Package correctly using CPack.
  - Install correctly into `/usr/local`.
  - Install 64-bit libraries into the correct directories on all platforms (be it `lib64` or `gnu-linux-x86_64`).
  - Be transitively linkable with nothing more than `target_link_libraries()` in the SAME CMake build.
  - Be transitively linkable with nothing more than and import command and `target_link_libraries()` when imported from an external CMake file.
- I don't want too many CMakeLists files; I'll lose track of them, and having a zillion open is just annoying. I'll prefer to have one CMakeLists file per component, not necessarily per directory.
- A library's public header files should be namespaced into a directory (i.e., `include <mylib/header.h>`–it's the only way to stay sane.

To achieve all this, we'll need to:

- Organize our source tree into a directory per module, with a CMakeLists per module. We'll use a parent CMakeLists file to tie all modules together, but this top-file should be OPTIONAL and only be taking care of bringing all required modules/dependencies into scope.
- Make a distinction between public and private headers for a library. We'll put the public headers in a directory called `include/mylib` such that we can just add that `include` directory to our search path to be able to include the headers as `<mylib/header.h>`.
- We're going to be using imported targets for libraries that don't have a CMake file. That means, no longer are we going to `find_package` a dependency and poke the include paths and library names directly into our own project.

Directory layout

Obviously we're going to be wanting an out-of-source build, so I'll start with a top-level `src` directory. That way I can make a `build` directory next to it.

```
/
    build/                          <-- out-of-source build
    src/
        CMakeLists.txt
        mylibrary/
            CMakeLists.txt
```

2016

» With Canon, you can't. A story of hor...

2015

» The Ultimate Guide to Modern CMake

2014

» Model-View-Whatever
» How I would design palm rejection

2013

» Conway's Game of Life in the Lambda C...
» The downside of being a generalist
» Turning 3 pallets into a table
» Lunch Boot Camp: Algorithmics
» Language Workbench Challenge 2013: Cl...
» Writing Tips

```
                    mylibrary.h
            src/
                lib.cpp
                frob.cpp
            test/
                testlib.cpp
        myapp/
            CMakeLists.txt
            src/
                myapp.cpp
                quux.cpp
        libs/
            (buildable 3rd party libs that you want to vend for
                                                    convenience)
            libfoo/
                CMakeLists.txt
                ...
```

Top-level CMakeLists.txt

The top-level CMake file is there to bring all modules into scope. That means, adding the subdirectories for all CMake projects in this tree, and finding external libraries and turning them into imported targets.

```
# At LEAST 2.8 but newer is better
cmake_minimum_required(VERSION 3.2 FATAL_ERROR)
project(myproject VERSION 0.1 LANGUAGES CXX)

# Must use GNUInstallDirs to install libraries into correct
# locations on all platforms.
include(GNUInstallDirs)

# Include Boost as an imported target
find_package(Boost REQUIRED)
add_library(boost INTERFACE IMPORTED)
set_property(TARGET boost PROPERTY
    INTERFACE_INCLUDE_DIRECTORIES ${Boost_INCLUDE_DIR})

# Some other library that we import that was also built using CMake
# and has an exported target.
find_package(MyOtherLibrary REQUIRED)

# Targets that we develop here
enable_testing()
add_subdirectory(liblib)
add_subdirectory(app)
```

Library

This file has the most going on, because it needs to be the most flexible.

```
# Define library. Only source files here!
project(liblib VERSION 0.1 LANGUAGES CXX)

add_library(lib
    src/lib.cpp
    src/frob.cpp)

# Define headers for this library. PUBLIC headers are used for
# compiling the library, and will be added to consumers' build
# paths.
target_include_directories(lib PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
    PRIVATE src)

# If we have compiler requirements for this library, list them
# here
target_compile_features(lib
    PUBLIC cxx_auto_type
    PRIVATE cxx_variadic_templates)
    boost
    MyOtherLibrary)

# 'make install' to the correct locations (provided by GNUInstallDirs)
install(TARGETS lib EXPORT MyLibraryConfig
    ARCHIVE  DESTINATION ${CMAKE_INSTALL_LIBDIR}
    LIBRARY  DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME  DESTINATION ${CMAKE_INSTALL_BINDIR})  # This is for Windo
install(DIRECTORY include/ DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})

# This makes the project importable from the install directory
# Put config file in per-project dir (name MUST match), can also
# just go into 'cmake'.
install(EXPORT MyLibraryConfig DESTINATION share/MyLibrary/cmake)

# This makes the project importable from the build directory
export(TARGETS lib FILE MyLibraryConfig.cmake)

# Every library has unit tests, of course
add_executable(testlib
    test/testlib.cpp)

target_link_libraries(testlib
    lib)

add_test(testlib testlib)
```

Program

```
# Define an executable
add_executable(app
    src/app.cpp
    src/quux.cpp)

# Define the libraries this project depends upon
target_link_libraries(app
    lib)
```

Commands to run

```
cd build

# Building
cmake ../src && make

# Testing
make test
ctest --output-on-failure

# Building to a different root directory
cmake -DCMAKE_INSTALL_PREFIX=/opt/mypackage ../src
```

Importing external libraries

If the library you're importing was not built with CMake, you'll define an imported target somewhere in your top-level CMake file (or perhaps in an included file if you have a lot of them). For example, this is how you import the Boost header library and a compiled Boost library:

```
find_package(Boost REQUIRED iostreams)

add_library(boost INTERFACE IMPORTED)
set_property(TARGET boost PROPERTY INTERFACE_INCLUDE_DIRECTORIES ${Boo

add_library(boost-iostreams SHARED IMPORTED)
set_property(TARGET boost-iostreams PROPERTY INTERFACE_INCLUDE_DIRECTO
set_property(TARGET boost-iostreams PROPERTY IMPORTED_LOCATION ${Boost
```

exported), then you can simply do this:

```
# CMakeLists
find_package(MyLibrary)

# Build like this:
cmake ../src -DMyLibrary_DIR=/path/to/mylibrary/build

# Passing the MyLibrary_DIR is not necessary if you
# 'make install'ed the project.
```

In case of an external target, obviously CMake can't track the binary to its sources so won't automatically rebuild your external project.

A short note on transitive dependencies: if the library you `EXPORT` depends on any targets, those targets will be recorded in the `MyLibraryConfig.cmake` file *by name*. This means that if you *import* this target into a separate project file, that project must have targets with the same name. So when importing an external target, you'll need to have `find_package()`d its dependencies already.

> I currently don't know of any way to hide these transitive dependencies from consumers. At work, we integrate CMake into a bigger build/dependency management system, and we post-process the generated `xxxConfig.cmake` files to insert additonal `find_package()` commands into the config files themselves. This works, but requires a postprocessing step that may not be easily achievable if you don't have a bigger build orchestration framework to hook into. Also, it requires that the target name and the config file name are exactly the same (whereas in my example up there the target name was `lib` but the config file name was `MyLibrary`). On the other hand, that seems like good practice anyway.

Integrating code generators

Goals:

- We want to rebuild the generation tool on-demand.
- Generated files go in the build tree, not the source tree.

Leading to something like this:

```
add_custom_command(
    OUTPUT file.output
    COMMAND tool --options "${CMAKE_CURRENT_SOURCE_DIR}/file.gen"
    MAIN_DEPENDENCY file.gen
    DEPENDS tool)

# Both for source files and header files
add_library(target
    ...
    file.output)

# For headers
target_include_directories(target
    ...
    PRIVATE ... ${CMAKE_CURRENT_BINARY_DIR})
```

The tool's cwd is `CMAKE_CURRENT_BINARY_DIR` so source file paths must be qualified with `${CMAKE_CURRENT_SOURCE_DIR}` in the command.

If you don't have an existing target to add the generated files to, create a custom target just for adding the dependency:

```
add_custom_target(flow-diagram ALL DEPENDS ${CMAKE_CURRENT_BINARY_DIR}
```

Packaging

I won't need to learn deeply about this part for my job (just yet), so I don't have a lot of tips here. Instead, I'll link to guides other people have written:

- Packaging Windows projects with an NSIS installer, by Paul Tsouchlos

← Previous | Blog | Next →

8 Comments rix0r.nl 1 Login

♡ Recommend Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

derp_cookies • 7 months ago

There's a number of things misleading or wrong about this article:

1. There is no add_include_directories. Perhaps you meant target_include_directories?

2. In you second example under "Importing external libraries", you should be doing: find_package(lib CONFIG). If you don't do this, CMake will first search for a module package (e.g. FindMyLibrary.cmake). If it can't find one, *then* it does config package mode.

3. When installing targets, you should specify `INCLUDES DESTINATION` so that the INTERFACE_INCLUDE_DIRECTORIES property is populated for the import targets.

4. You are explicitly specifying BUILD_INTERFACE and INSTALL_INTERFACE when you call target_include_directories(). IIRC, this is only required if your relative include path is actually different between building and installing. In your example, it's `include` for both. So you can just do:

see more

6 ^ | ˅ • Reply • Share ›

Michael Steffens → derp_cookies • 10 days ago

I'm really struggling to understand these improvements.

Regarding 3.: Works, but I don't notice any difference when importing the project. What is the purpose of INTERFACE_INCLUDE_DIRECTORIES? Why does it need to be populated?

Regarding 4.: If I follow your suggestion I end up like

https://stackoverflow.com/q...

and the suggestion to use the interface expressions rix0r.nl already did.

Regarding 5.: In what way is this idomatic and what is the benefit? It works, but I end up with two files in share/MyLibrary/cmake, one including the other. What do I gain this way? Could you point to any recommendation by CMake supporting your suggestion? At least

see more

^ | ˅ • Reply • Share ›

derp_cookies → Michael Steffens • 9 days ago

3.: If your library has an include directory, this specifies to downstream targets which paths to use for searching for your headers when included by those downstream projects. Read more here: https://cmake.org/cmake/hel...

4. In the SO post you linked, the OP is using an absolute path in `target_include_directories()`. In this article, only a relative path is needed (see my example from my first

`INSTALL_INTERFACE` need not be explicitly set.

5. Idiomatic here just means "The way CMake does things". The reason why it seems pointless in your case is because the Config.cmake file *only* includes the Targets.cmake file and doesn't do anything else. But the Config.cmake file allows you to perform additional logic beyond defining targets, such as fulfilling dependencies

see more

^ | v • Reply • Share ›

Michael Steffens ➦ derp_cookies • 9 days ago

Wow, thanks for this insightful and helpful response! Yes, you are absolutely right about the lack of consistent guidance, of course I upvoted your issue at Kitware.

To be honest, rix0r.nl's blogpost comes so close to what I was looking for and enabled a big leap in improving my CMake configurations. But there are gaps apparently, and I think with your latest response to 5. (to me, not to him) you actually solve his (and my) issue at the bottom of "Importing external libraries".

Some minor comments:

4.: I'm afraid, no. Even when using exactly your

target_include_directories(... PUBLIC include PRIVATE src)

see more

^ | v • Reply • Share ›

John Parker • 24 days ago

The title is somewhat misleading. It's probably an ultimate guide/placeholder just for you. May be it's just helpful if you can start the article with a disclaimer that it's not for beginners.

^ | v • Reply • Share ›

Michael Steffens ➦ John Parker • 9 days ago

I disagree. The scope of this guide is *exactly* what a beginner is looking for. Can't see any corner case requirement involved, nothing should be rocket science. Finding it well written and structured and superior to all official tutorials or "works-for-me" kind of guides I found elsewhere.

Has its shortcomings (see derp_cookies comments), but definitely the right direction. See also

https://gitlab.kitware.com/...

and consider an upvote, if you agree!

^ | v • Reply • Share ›

Evgeny Danilenko • 5 months ago

Hello again rix0r :D
I came to reference your CMakeLists structure to get boost unit tests working, but i kept getting undefined reference to main for the boost library. My CMakeLists.txt is not 1:1 with yours although the Boost parts were. Anyways the way i fixed my issue was by changing this part find_package(Boost REQUIRED) to find_package(Boost COMPONENTS unit_test_framework REQUIRED)

and then linking them with the lib target_link_libraries(lib boost ${Boost_LIBRARIES})
without Boost_LIBRARIES it wont work. at least for me.

Thanks again for this tutorial.

^ | v • Reply • Share ›

Hi, thanks for this great blog post, it's actually quite hard to find anything good about modern cmake out there. I do have one question, though: is it really good practice declaring external libs as imported targets in the top-most CMakeLists as you do for boost? By doing this, it makes "liblib" a CMake project that cannot be built on its own, which I thought was bad practice.
Shouldn't "liblib" use FindTarget or something to find boost, which perhaps is configured to export a cmake file? I've been trying to find more info on how to do this, but it's not easy.

^ | ⌄ • Reply • Share ›

ALSO ON **RIX0R.NL**

The downside of being a generalist

2 comments • 5 years ago

Avatar **Johan Johansson** — Being a generalist is very good for management positions, because …

Model-View-Whatever

1 comment • 4 years ago

Avatar **Egbert** — We coded www.izooble.com with Model-View-Nothing, using React. Most …

✉ Subscribe   **D** Add Disqus to your siteAdd DisqusAdd   🔒 Privacy   **DISQUS**