It's Time To Do CMake Right

19 Feb 2018 on Build Systems (/tag/build-systems/) , C++ (/tag/cpp/)

Not so long ago I got the task of rethinking our build system. The idea was to evaluate existing components, dependencies, but most importantly, to establish a superior design by making use of modern CMake features and paradigms. Most people I know would have avoided such enterprise at all costs, but there is something about writing find modules that makes my brain release endorphins. I thought I was up for an amusing ride. Boy was I wrong.

My excitement was soon shattered after discovering the lack of standard practices in CMake usage and specially the insufficient adoption of modern design patterns. This post explores the concepts of what is known as modern CMake, which advocates for abandoning a traditional variable-based approach for a more structured model based on so-called targets. My intention is to show how "new" (>= 3.0.0) features can be employed to reshape your CMake system into a more maintainable and intuitive alternative that actually makes sense.

Many of the concepts presented here find their roots in Daniel Pfeifer's masterpiece Effective CMake (https://youtu.be/rLopVhns4Zs) . Daniel has practically become the Messiah of the modern CMake church, preaching best practices and guidelines in a time when the only standard is to not have one. Daniel, I am your prophet.

Enough preambles. Does this look familiar to you?

```
find_package(Boost 1.55 COMPONENTS asio)
list(APPEND INCLUDE_DIRS ${BOOST_INCLUDE_DIRS})
list(APPEND LIBRARIES ${BOOST_LIBRARIES})

include_directories(${INCLUDE_DIRS})
link_libraries(${LIBRARIES})
```

Don't. Just don't. This is wrong in so many dimensions. You are just blindly throwing stuff into a pot of include directories and compiler flags. There is no structure. There is no transparency. Not to mention that functions like `include_directories` work at the directory level and apply to all entities defined in scope.

And this isn't even the real problem, what do you do with transitive dependencies? What about the order of linking? Yes, you need to take care about that yourself. The moment you need to deal with the dependencies of your dependencies is the moment your life needs to be reevaluated.

Targets and Properties

CMake developers saw the aforementioned problems and introduced language features that allow you to better structure your projects. Modern CMake is all about targets and properties. Conceptually this isn't complicated. Targets model the components of you application. An executable is a target, a library is a target. Your application is built as a collection of targets that depend on and use each other.

Targets have properties. Properties of a target are the source files it's built from, the compiler options it requires, the libraries it links against. In modern CMake you create a list of targets and define the necessary properties on them.

Build Requirements vs Usage Requirements

Target properties are defined in one of two scopes: **INTERFACE** and **PRIVATE**. Private properties are used *internally* to build the target, while interface properties are used *externally* by users of the target. In other words, interface properties model usage requirements, whereas private properties model build requirements of targets.

Interface properties have the prefix, wait for it, *INTERFACE_* prepended to them.

For example, the property **COMPILE_OPTIONS** encodes a list of options to be passed to the compiler when building the target. If a target must be built with all warnings enabled, for instance, this list should contain the option `-Wall`. This is a private property used only when building the target and won't affect its users in any way.

On the other hand, the property **INTERFACE_COMPILE_FEATURES** stores which features must be supported by the compiler when building *users* of the target. For instance, if the public header of a library contains a variadic function template, this property should contain the feature `cxx_variadic_templates`. This instructs CMake that applications including this header will have to be built by a compiler that understands variadic templates.

Properties can also be specified as **PUBLIC**. Public properties are defined in both **PRIVATE** and **INTERFACE** scopes.

All of this is better understood with an example.

libjsonutils

Imagine that you are writing a json utily library, *libjsonutils*, that parses json files from a provided location. Json files can be located on your local file system, as well as accessible via some URL.

The library has the following structure:

```
libjsonutils
├── CMakeLists.txt
├── include
│   └── jsonutils
│       └── json_utils.h
├── src
│   ├── file_utils.h
│   └── json_utils.cpp
└── test
    ├── CMakeLists.txt
    └── src
        └── test_main.cpp
```

We have a single public header, were we define the `loadJson()` function:

```
boost::optional<rapidjson::Document> loadJson(const std::string& url);
```

This function receives either a URL or a filepath to a json and loads it as a rapidjson object. If something goes wrong, `boost::none` will be returned instead.

Let's start writing jsonutil's `CMakeLists.txt` :

```
cmake_minimum_required(VERSION 3.5)
project(libjsonutils VERSION 1.0.0 LANGUAGES CXX)
```

Nothing surprising here. The first step is to create our library target:

```
add_library(JSONUtils src/json_utils.cpp)
```

Now let's define some properties on our target. Why not start with the include directories?

```
target_include_directories(JSONUtils
    PUBLIC
        $<INSTALL_INTERFACE:include>
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    PRIVATE
        ${CMAKE_CURRENT_SOURCE_DIR}/src
)
```

Our headers are located in two different places: inside `src/` , which contains a utility header called `file_utils.h` , and in `include/` , where our public header `json_utils.h` lives. For building our library we need all headers in both locations (`json_utils.cpp` includes both), so **INCLUDE_DIRS** must contain `src/` , as well as `include/` .

On the other hand, users of *jsonutils* only need to know about the location of the public header `json_utils.h`, so **INTERFACE_INCLUDE_DIRS** only needs to contain `include/`, but not `src/`.

There is still a problem, though. While building jsonutils, `include/` is at `/home/pablo/libjsonutils/include/`, but after installing our library, it will be under `${CMAKE_INSTALL_PREFIX}/include/`. Therefore, the location of this directory needs to be different depending on whether we are building or installing the library. To solve this problem, we use generator expressions (https://cmake.org/cmake/help/v3.5/manual/cmake-generator-expressions.7.html), which set the correct path depending on the situation.

Leave CMAKE_CXX_FLAGS Alone

We can now continue by defining extra properties on our target. For example, it could be beneficial to treat warnings as errors:

```
target_compile_options(JSONUtils PRIVATE -Werror)
```

Given that we use `constexpr` and `auto`, we could set the language standard to c++11:

```
target_compile_features(JSONUtils PRIVATE cxx_std_11)
```

Note that there is no reason to manually append `-std=c++11` to **CMAKE_CXX_FLAGS**, let CMake do that for you! Stay away from variable land, model your requirements via properties.

Model dependencies with `target_link_libraries`

Let's think about our dependencies. First off all, we need boost, as we use optional. Additionally, in order to figure out if the passed string is an URL, we have to evaluate it against some regex, so we need boost::regex (yes I know c++11 introduces regex utilities but bear with me). Second, we need rapidjson.

In CMake, `target_link_libraries` is used to model dependencies between targets.

```
find_package(Boost 1.55 REQUIRED COMPONENTS regex)
find_package(RapidJSON 1.0 REQUIRED MODULE)

target_link_libraries(JSONUtils
    PUBLIC
        Boost::boost RapidJSON::RapidJSON
    PRIVATE
        Boost::regex
)
```

Dependencies (a.k.a link libraries) of a target are just another property and are defined in an **INTERFACE** or **PRIVATE** scope. In our case, both rapidjson and boost optional (defined in the target `Boost::boost`) have to be interface dependencies and be propagated to users, as they are used in a public header that's imported by clients.

This means that users of `JSONUtils` don't just require `JSONUtil`'s interface properties, but also the interface properties of its interface dependencies (which define the public headers of boost and rapidjson in this case), and those of the dependencies of the dependencies, etc.

But how does CMake solve this problem? Easy, it adds all interface properties of `Boost::boost` and `RapidJSON::RapidJSON` to the corresponding `JSONUtil`'s own interface properties. This means that users of `JSONUtils` will transitively receive the interface properties of targets all up the dependency chain.

On the other hand, Boost::regex is only used internally and can be a private dependency. Here, `Boost::regexes` interface properties will be appended to the corresponding `JSONUtil`'s private properties, and won't be propagated to users.

Isn't this beautiful? Usage requirements are propagated and build requirements encapsulated. Welcome to modern CMake.

Sex, Drugs and Imported Targets

Note that `Boost::boost` and `RapidJSON::RapidJSON` are targets themselves. But where did they come from? I haven't told you the most breathtaking fact about targets yet: targets can be exported. Exported targets can be later imported into other projects.

When we call `find_package(Boost 1.55 REQUIRED COMPONENTS regex)`, CMake will execute `FindBoost.cmake`, where the targets `Boost::boost` and `Boost::regex` will be imported, allowing us to depend on them via `target_link_libraries()`.

Our projects have structure, as they are build as a collection of encapsulated targets, and CMake handles transitive requirements for us. You might be wondering with tears in your eyes how beautiful life can be, but boy are up for a revelation.

Let's try to build *jsonutils*:

```
CMake Error at CMakeLists.txt:9
    Target "JSONUtils" links to target "RapidJSON::RapidJSON" but the target was not found
```

Good boys export their targets

The imported target `RapidJSON::RapidJSON` could not be found, because `RapidJSONConfig.cmake` did not create it. Let's inspect what rapidjson does in the config installed on my arch linux system:

```
get_filename_component(RAPIDJSON_CMAKE_DIR "${CMAKE_CURRENT_LIST_FILE}" PATH)
set(RAPIDJSON_INCLUDE_DIRS "/usr/include")
message(STATUS "RapidJSON found. Headers: ${RAPIDJSON_INCLUDE_DIRS}")
```

Yes, welcome to hell. This is where the real pain begins: 3rdparty dependencies. In the case of rapidjson, a single variable is set to point to its include directories. This is exactly what we don't want, we don't want variables, we want targets!

In my case, 70% of my dependencies didn't define any targets in their find modules or configs. The reality is that CMake usage is an anarchy. There are few rules and too much flexibility. We need standard practices, we need guidelines. We have design patterns for C++, why not for CMake?

If you want it done right do it yourself

So what can you do in these cases?

Daniel Pfeifer advises to report such usage as a bug to the library developers. I agree. Upstreams should support downstream's modern target-based design. Ask yourself: do you really need this dependency? Are there alternatives that do support modern cmake usage?

In this case, however, there is no other option other taking matters into your own hands and write `FindRapidJSON.cmake` ourselves:

```
# FindRapidJSON.cmake
#
# Finds the rapidjson library
#
# This will define the following variables
#
#    RapidJSON_FOUND
#    RapidJSON_INCLUDE_DIRS
#
# and the following imported targets
#
#     RapidJSON::RapidJSON
#
# Author: Pablo Arias - pabloariasal@gmail.com

find_package(PkgConfig)
pkg_check_modules(PC_RapidJSON QUIET RapidJSON)

find_path(RapidJSON_INCLUDE_DIR
    NAMES rapidjson.h
    PATHS ${PC_RapidJSON_INCLUDE_DIRS}
    PATH_SUFFIXES rapidjson
)

set(RapidJSON_VERSION ${PC_RapidJSON_VERSION})

mark_as_advanced(RapidJSON_FOUND RapidJSON_INCLUDE_DIR RapidJSON_VERSION)

include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(RapidJSON
    REQUIRED_VARS RapidJSON_INCLUDE_DIR
    VERSION_VAR RapidJSON_VERSION
)

if(RapidJSON_FOUND)
    set(RapidJSON_INCLUDE_DIRS ${RapidJSON_INCLUDE_DIR})
endif()

if(RapidJSON_FOUND AND NOT TARGET RapidJSON::RapidJSON)
    add_library(RapidJSON::RapidJSON INTERFACE IMPORTED)
    set_target_properties(RapidJSON::RapidJSON PROPERTIES
        INTERFACE_INCLUDE_DIRECTORIES "${RapidJSON_INCLUDE_DIR}"
    )
endif()
```

This is a very simple find module that looks for rapidjson's headers in the system and creates the imported target `RapidJSON::RapidJSON` that we require. I use **INTERFACE** to indicate that this "library" isn't really a library, as there is no correponding .a or .so, but just defines usage requirements.

This is how you do CMake

We want jsonutils to integrate in a target-based build system of downstreams. This means that all they have to do to use jsonutils is this:

```
find_package(JSONUtils 1.0 REQUIRED)
target_link_libraries(example JSONUtils::JSONUtils)
```

To achieve this we need to do two things. First, we need to export the target `JSONUtils::JSONUtils`. And second, we need to import that target when downtreams call `find_package(JSONUtils)`, i.e. from inside our `JSONUtilsConfig.cmake`.

Let's start with exporting our target to a `JSONUtilsTargets.cmake` script that imports it. First of all, we need to install the library itself (the actual .a or .so file):

```
include(GNUInstallDirs)
install(TARGETS JSONUtils
    EXPORT jsonutils-export
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
)
```

In CMake, installed targets are registered to exports using the **EXPORT** argument. Exports are therefore just a set of targets that can be exported and installed. Here we just told CMake to install our library and to register the target in the export *jsonutils-export*.

Then we can go ahead and install the export that we defined above:

```
install(EXPORT jsonutils-targets
  FILE
    JSONUtilsTargets.cmake
  NAMESPACE
    JSONUtils::
  DESTINATION
    ${CMAKE_INSTALL_LIBDIR}/cmake/JSONUtils
)
```

This will install the import script `JSONUtilsTargets.cmake` that, when included in other scripts, will load the targets defined in the export *jsonutils-export*. By using the **NAMESPACE** argument, we tell CMake to prepend the prefix *JSONUtils::* to all targets imported.

Import your targets inside your Config.cmake

Remember, when clients call `find_package(JSONUtils)`, CMake will look for and execute a `JSONUtilsConfig.cmake`.

So that our target `JSONUtils::JSONUtils` is imported and can be used by clients, we need to load `JSONUtilsTargets.cmake` in our config file:

```
get_filename_component(JSONUtils_CMAKE_DIR "${CMAKE_CURRENT_LIST_FILE}" PATH)
include(CMakeFindDependencyMacro)

find_dependency(Boost 1.55 REQUIRED COMPONENTS regex)
find_dependency(RapidJSON 1.0 REQUIRED MODULE)

if(NOT TARGET JSONUtils::JSONUtils)
    include("${JSONUtils_CMAKE_DIR}/JSONUtilsTargets.cmake")
endif()
```

Be aware that `JSONUtilsTargets.cmake` contains code like:

```
add_library(JSONUtils::JSONUtils STATIC IMPORTED)
set_target_properties(JSONUtils::JSONUtils PROPERTIES
  INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"
  INTERFACE_LINK_LIBRARIES "Boost::boost;RapidJSON::RapidJSON;\$<LINK_ONLY:Boost::regex>"
)
```

Since this script references targets from boost and rapidjson, they need to be imported before including `JSONUtilsTargets.cmake` in `JSONUtilsConfig.cmake`.

This is why we need to call `find_dependency()` in `JSONUtilsConfig.cmake`: to make sure that downstreams have all required dependencies installed and the needed targets are imported before they are referenced in our `JSONUtilsTargets.cmake`.

That's all folks

You can refer to my github (https://github.com/pabloariasal/modern-cmake-sample) where I have uploaded the entire jsonutils project containing all the code shown in this post. There I also included examples on how to test the library using gtest, as well as how to export your targets from the build tree and register them in CMake's package registry.

Hopefully by now you were able to grasp how clean and structured a target-based CMake can be compared to a flag and variable based approach. Also, exporting your targets is something your grandma could do, so why not do it? I believe the reason is that CMake suffers from a syndrome of "if no one does it why should I?" We need to change this. We deserve to live in a better world. Export your targets goddammit.

About

Hola! My name is Pablo and I am a software engineer currently living in Munich, Germany. My first line of code was written 12 years ago while attempting to customize an add-on for my World of Warcraft UI in Lua. A passion for programming has been growning in me ever since and now, at 26 years old, I write software for a living. This blog explores a variety of topics that I've stumbled across in my journey as a software developer. Even though mostly written for my own understanding, my hope is that the curious programmer may find my writings entertaining.