Tutorial: Easily supporting CMake install and find_package()

03 MAR 2016

As of version 0.5 my memory library now provides support for system-wide installation and CMake's `find_package()`.

Because I've spent hours of trial and error to come up with it, I'll document it here. In this post, I will show you how to install your library so that it can be used easily by other projects. In particular, the system will be able to handle multiple installed versions and multiple configurations.

Throughout this post, I'll be assuming a 3.x CMake version and an already existing CMake project.

The setup

For the scope of the tutorial, let's say we have a library that has the following CMake structure:

```
- include/
```

```
    - header-a.hpp
        - header-b.hpp
        - config.hpp
        - ...
- src/
  - source-a.cpp
  - source-b.cpp
  - config.hpp.in
  - ...
  - CMakeLists.txt
- example/
  - example-a.cpp
  - ...
  - CMakeLists.txt
- tool/
  - tool.cpp
  - CMakeLists.txt
- test/
  - test.cpp
  - CMakeLists.txt
- CMakeLists.txt
- ...
```

So we have a library consisting of various header and source files. It also comes with some examples, a tool and unit tests.

The library, the examples and the tool each has their own `CMakeLists.txt` defining the target and related code in their subdirectory. The root `CMakeLists.txt` defines configuration options and adds the subdirectories.

The configurations will be set in the file `config.hpp.in` which will

> *This allows a seperation of CMake's config files and other, unrelated configuration macros etc.*

The root CMakeLists.txt can look as follows:

```
cmake_minimum_required(VERSION 3.0)
project(MY_LIBRARY)

# define library version (update: apparently you can also do it in pr
set(MY_LIBRARY_VERSION_MAJOR 1 CACHE STRING "major version" FORCE)
set(MY_LIBRARY_VERSION_MINOR 0 CACHE STRING "minor version" FORCE)
set(MY_LIBRARY_VERSION ${MY_LIBRARY_VERSION_MAJOR}.${MY_LIBRARY_VERSI

# some options
option(MY_LIBRARY_USE_FANCY_NEW_CLASS "whether or not to use fancy ne
option(MY_LIBRARY_DEBUG_MODE "whether or not debug mode is activated'

# add subdiretories
add_subdirectory(src)
add_subdirectory(example)
add_subdirectory(tool)
add_subdirectory(test)
```

It defines a few options that can be used via `#cmakedefine01` or similar in the `config.hpp.in`.

> *Note that the library version is set via `FORCE`. This prevents users changing the value in the `CMakeCache.txt`.*

And the `src/CMakeLists.txt`:

```
# set headers
set(header_path "${MY_LIBRARY_SOURCE_DIR}/include/my_library")
set(header ${header_path}/header-a.hpp
                  ${header_path}/header-b.hpp
                  ${header_path}/config.hpp
                  ...)

# set source files
set(src source-a.cpp
                source-b.cpp
                ...)

# configure config.hpp.in
configure_file("config.hpp.in" "${CMAKE_CURRENT_BINARY_DIR}/config_im

# define library target
add_library(my_library ${header} ${src})
target_include_directories(my_library PUBLIC ${MY_LIBRARY_SOURCE_DIR}
```

First we define a list of all headers and source files in variables. This will be useful later on.

> *Note that the header path has to be given for the headers, since they are in a different subdirectory. The source files can be named directly.*

It also generates the `config_impl.hpp` that can be included inside the `config.hpp` in the current binary dir and defines the library with the given files. Its `PUBLIC` include directory is *both* the `included/` subfolder and the current binary dir. The latter is

The other `CMakeLists.txt` are simpler and I will not go over them here.

Client code can now call `add_subdirectory()` of the library folder and call `target_link_libraries(my_target PUBLIC my_library)`. This will also set up the include path, allowing `#include <my_library/header-a.hpp>` directly.

But we want to install it and support `find_package()`.

Installation

We only need to install the following in order to use the library: the header files, the tool executable and the built library. This can be done in a very straightforward way with the `install()` command. It will simply copy the files into the `${CMAKE_INSTALL_PREFIX}` (`/usr/local/` under Linux) when entering the `cmake install` command in a terminal.

First, we define the locations as variables in the root CMakeLists.txt:

```
set(tool_dest "bin")
set(include_dest "include/my_library-${MY_LIBRARY_VERSION}")
set(main_lib_dest "lib/my_library-${MY_LIBRARY_VERSION}")
```

Then we add the `install()` commands:

```
# in tool/CMakeLists.txt
install(TARGETS my_library_tool DESTINATION "${tool_dest}")
```

```
install(TARGETS my_library DESTINATION "${main_lib_dest}")
install(FILES ${header} DESTINATION "${include_dest}")
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/config_impl.hpp DESTINATION
```

This will install the tool executable under `${CMAKE_INSTALL_PREFIX}/bin`, the headers under `${CMAKE_INSTALL_PREFIX}/include/my_library-1.0` and the library itself under `${CMAKE_INSTALL_PREFIX}/lib/my_library-1.0`. It already satifies one of the goals I've set above: Different library versions do not run into conflicts since they will be installed under different destinations; the version is part of their folder.

> *Except for tool. I've avoided it here because I assume it will always provide full compatibility. Putting it under `bin/` directly makes it available in the terminal automatically. But you can of course adopt that easily.*

But this does not handle different configurations of the library: only one can exist in the location. We can of course prevent that by adding a unique identifier for each configuration like we did for the version, but this is unnecessary for most files.

Again ignoring the tool, there are only two files that depend on the configuration: the built library and the generated `config_impl.hpp` since it will have macros set that correspond to the library options. So we need to put only those two files in a different location depending on the configuration.

But what do we use as identifier?

I've chosen the value of the `${CMAKE_BUILD_TYPE}`. It already selects the compiler flags for value of `Debug`, `Release`, `MinSizeRel` and

`RelWithDebInfo`. It makes sense to couple all other options to it as well.

> *You can also define your own build types with a corresponding set of flags to allow an unlimited number of installed configurations.*

We thus add a new variable `lib_dest` in the root CMakeLists.txt:

```
set(lib_dest ${main_lib_dest}/${CMAKE_BUILD_TYPE}")
```

And also change the destination for `config_impl.hpp` and the `my_library` target to `${lib_dest}`. This will put those two files into different folders depending on the configuration to allow multiple configuration to be installed. So, for example, the `Debug` library will be installed under `${CMAKE_INSTALL_PREFIX}/lib/my_library-1.0/Debug` etc.

Exporting the target

The current setup already installs everything needed to use the library but it cannot be integrated into other CMake based projects. You'd have to manually specify the include directory and manually link to the native library.

This isn't comfortable.

CMake provides the ability to *export* targets though. Exporting a target allows reusing it in other CMake projects, just as if it were defined in the current project. To enable that, a file `my_library.cmake` will be created upon installation. It contains definitions of all the targets with references to the installed

build files and configuration. Users just need to `include()` that file and can use the target as usually.

To enable exporting for *my_library* we need to do two things:

- First, for each target specify that it will be added to an export group. This is accomplished by adding `EXPORT my_library` in the `install(TARGET)` command. For example, for the main library target, the target install command is now:

```
install(TARGETS my_library EXPORT my_library DESTINATION "${lib_dest}
```

- Then, the export group need to be installed as well. This can be done with the `install(EXPORT)` command called in the root `CMakeLists.txt`. Since the target references the build-type specific locations for the `config_impl.hpp` and library file it is build-type dependend and will be installed in the `${lib_dest}`:

```
install(EXPORT my_library DESTINATION "${lib_dest}")
```

There is still a minor problem though: The library has set the `target_include_diretories()` it will pass on to the linked targets to the directory the sources prior to the installation are stored! And we cannot change the directory because then the include directory for building is wrong.

An ugly feature named *generator expressions* help here though. It allows setting different include directories whether the library has been installed or is currently building. The call to `target_include_directories()` in the `src/CMakeLists.txt` needs to be

```
target_include_directories(my_library PUBLIC
                           $<BUILD_INTERFACE:${MY_LIBRARY_SOURCE_DIR}/inc


                           $<INSTALL_INTERFACE:${include_dest}> # for cli
                           $<INSTALL_INTERFACE:${lib_dest}> # for config_
```

Now we have a `my_library.cmake` type that just need to be included in order to use the library as destination in `target_link_libraries()` as usual. But before you go and add `include(/path/to/installation/my_library-1.0/Debug/my_library.cmake)` statements, let's automate that by enabling package support.

The final step: Packaging

CMake provides the `find_package()` command. I won't go into much detail here, but its basic form can help here.

> *I don't know the details either. The CMake documentation file is long and frightening.*

If we write `find_package(my_library ...)`, it will go and look for a file named `my_library-config.cmake` (among others) in a directory named `my_library*` under the `${CMAKE_INSTALL_PREFIX}/lib` (among *many* others).

And our installation directory name `lib/my_library-[major].[minor]` - the `${main_lib_dest}` - matches this expression.

> *What a coincidence.*

We just need to provide the `my_library-config.cmake` file. The

`find_package()`. It usually contains code defining the targets but we already have that code! It is in the `my_library.cmake` file created by the `install(EXPORT)`. We just need to `include()` that inside the `my_library-config.cmake` file.

Here we can also match the build-type. We include the export file version that matches the current build type:

```
# my_library-config.cmake - package configuration file

get_filename_component(SELF_DIR "${CMAKE_CURRENT_LIST_FILE}" PATH)
include(${SELF_DIR}/${CMAKE_BUILD_TYPE}/my_library.cmake)
```

This file can be stored inside your library repo, just remember to install it as well. It can be done right beside the `install(EXPORT)` command:

```
install(FILES my_library-config.cmake DESTINATION ${main_lib_dest})
install(EXPORT ...)
```

Now the client can call `find_package(my_library REQUIRED)` and the library will be searched, found (if the `${CMAKE_BUILD_TYPE}` is installed) and all exported targets made available allowing a simple `target_link_libraries(client_target PUBLIC my_library)`. This will link to the library version of matching build type.

Nice.

> *The client can also leave out the `REQUIRED` part if it is, well, not required. Then a variable needs to be queried before safely referring to the targets.*

Adding sugar: version control

One nice touch is version compatibility checks of the installed libraries. This is also supported by `find_package()`, you can give it a version as second argument.

The check is done by a file named `my_library-config-version.cmake` (or similar). Like `my_library-config.cmake`, you need to provide and install it under the current setup.

It gets the requested version in the form of `${PACKAGE_FIND_VERSION_MAJOR/MINOR}` and should set the variables `${PACKAGE_FIND_VERSION_EXACT/COMPATIBLE/UNSUITABLE}` as appropriate. It should also set the full version in `${PACKAGE_VERSION}`. One thing it does *not* get though is the version of the library with which it is installed. For that reason, it needs to refer to the version variables defined in the root `CMakeLists.txt` and to be configured prior installation.

Here is a simple script that requires the a major version match and a higher or equal minor version:

```
# my_library-config-version.cmake - checks version: major must match,


set(PACKAGE_VERSION @MY_LIBRARY_VERSION@)


if("${PACKAGE_FIND_VERSION_MAJOR}" EQUAL "@MY_LIBRARY_VERSION_MAJOR@'
    if ("${PACKAGE_FIND_VERSION_MINOR}" EQUAL "@MY_LIBRARY_VERSION_MI
        set(PACKAGE_VERSION_EXACT TRUE)
    elseif("${PACKAGE_FIND_VERSION_MINOR}" LESS "@MY_LIBRARY_VERSION_
        set(PACKAGE_VERSION_COMPATIBLE TRUE)
    else()
```

```
    endif()
else()
    set(PACKAGE_VERSION_UNSUITABLE TRUE)
endif()
```

Configuration (to replace the `@`-variables with the right version) and installation is done in the root CMakeLists.txt:

```
configure_file(my_library-config-version.cmake.in ${CMAKE_CURRENT_BIN

install(FILES my_library-config.cmake ${CMAKE_CURRENT_BINARY_DIR}/my_
install(EXPORT ...)
```

> *Note the `@ONLY` in order to not substitute the "normal" CMake variables in the script.*

Now a `find_package()` call of the form `find_package(my_library 1.0 REQUIRED)` will look for the 1.0 or a compatible (as you defined "compatible") library version.

Summary

So to sum it up, in order to support installation and `find_package()` in CMake you need to:

- Change the call to `target_include_directories()` so that it uses the `$<BUILD_INTERFACE:>` and `$<INSTALL_INTERFACE:>` generator expressions to set the right include directory. In installation mode this is the location where the header files will be installed (see directly below).

- Install the header files to `include/my_library-[major].[minor]` via `install(FILES)`.
- Install the configured header file (or all other header files depending on the configuration/build type) to `lib/my_library-[major].[minor]/${CMAKE_BUILD_TYPE}/` via `install(FILES)`.
- Install the library target to `lib/my_library-[major].[minor]/${CMAKE_BUILD_TYPE}/` via `install(TARGET target EXPORT my_library ...)`. This will also add it to the export group.
- Define a file named `my_library-config.cmake` that just includes the corresponding `my_library.cmake` file (see above, just copy-paste that). Also define a `my_library-config-version.cmake.in` similar to above for version compatiblity checks.
- Configure the version install file so that it uses the right version via `configure_file(...)` and install the configured version install file and the `my_library-config.cmake` file to `lib/my_library-[major].[minor]/` via `install(FILES)`.
- Install the export group to `lib/my_library-[major].[minor]/${CMAKE_BUILD_TYPE}/` via `install(EXPORT)`.

Now a client just need to write:

```
find_package(my_library 1.0 REQUIRED)
target_link_libraries(client_target PUBLIC my_library)
```

And it will automatically find an appropriate library version and link to a library of matching build type.

For an actual complete example, look at the source code of memory itself. It provides a similar directory structure but note that CMake dependent stuff is put into a `cmake` subdirectory.

> *Also check out the other half: my tutorial on dependency managment*

This post was made possible by my Patreon supporters. If you'd like to support me as well, please head over to my Patreon and do so! One dollar per month can make all the difference.

Jonathan

Nerd and C++ enthusiast. I write libraries for real-time applications.

Share this post

« Memory 0.5: Better build system, Low-level Allocators, BlockAllocator & Your Feedback is needed

Performing arbitrary calculations with the Concept TS »