C/C++ project organization

24 Mar 2017, 12:30

In the past few years, I've written some very small C/C++ programs. Back then, I didn't pay much attention to project organization as one project is just a few source files, headers, and a Makefile. Recently I've been doing a lot of development with C++ in a larger project. This got me into thinking how a C/C++ project should be organized. In this blog post, I wanna talk about how a moderate size C/C++ project could be organized to keep things in order.

Overview

Let's have an overview of the project layout. The following directory layout is generated with the `tree` program.

```
project
├── CMakeLists.txt
├── cmake
│   └── find_xxx.cmake
├── doc
│   ├── CMakeLists.txt
│   └── Doxyfile
├── include
│   └── project
│       └── project.h
│   └── third-party-library
│       └── library.h
├── src
│   ├── CMakeLists.txt
│   ├── main.cpp
│   └── main.h
└── test
    ├── CMakeLists.txt
    └── test1.cpp
```

As you can see, in the top level, there is a `CMakeLists.txt` file. I'm assuming that you are using CMake as your build system. You could of course use something else like Autoconf or just plain Makefiles but the differences are not important here. This top-level `CMakeLists.txt` file contains general project settings such as compilers, compiler flags, pendencies, etc. Also, it will include `CMakeLists.txt` files in the subdirectories. The rpose of the subdirectories are explained in following sections.

src

The `src` folder is where you put all your source files, most header files, and the `CMakeLists.txt` file for compiling your source code. Notice that not all header files are in the `src` folder in the above example. Actually, you could also just put all the headers in `src`. However, putting some headers files in a separate directory has some advantages.

include

The `include` folder is where you put some of your header files. What header files should you put inside exactly? Well, first of all, you could put the header files used for installation inside (if you are developing a library). This allows you to easily install all headers files here with a glob command in CMake. Second, you could put header-only libraries here. This separates any third-party code with your own code residing in `src`.

doc

Assuming you are using Doxygen to generate your documentation, you could put your Doxygen configuration file here. Together with CMake, you can build your documentation easily with `make doc`. I plan to talk about how to combine Doxygen and CMake in another blog post.

test

This is where all the testing goes. By separating tests and source files, you could easily set up CMake so that all tests could be compiled optionally when you run `make test`.

cmake

This folder is optional. If you don't have complicated CMake code, you could just put all code inside `CMakeLists.txt` files. However, if you do have a lot of CMake code for configuring the project, you could put some of the code here and include them in `CMakeLists.txt` files.

Dependencies

Something I didn't talk about is dependencies of the project. Usually a large project would depend on several other projects. One way of solving the dependency issue is obviously to let the users install the dependencies by themselves. However, sometimes there might be too many dependencies or some of the dependencies are not easy to setup. In such a case, you could consider adding another folder named `third-party` and use CMake to wnload and compile these dependencies in the `third-party` folder. I've seen some ›jects taking this approach such as the Paraview Superbuild and PaddlePaddle. In the end, it's really up to you whether you wanna make the build self-contained or let the users handle dependency issues.

Conclusion

I hope you learnt something regarding C/C++ project organization and that you could utilize it in your own projects to keep them organized. Have fun coding, cheers!

‹ How to use CMake

Initialization forms in C++11 ›