CMake: Public VS Private VS Interface

Introduction

CMake is one of the most convenient building tools for C/C++ projects. When it comes to `target_include_directories` and `target_link_libraries`, there are several keywords, `PUBLIC`, `PRIVATE`, and `INTERFACE`, that I got confused about from time to time even if I have read the related official documentations. So when I was building my C/C++ projects using CMake, I often just use `PUBLIC` everywhere or leave the keyword blank (CMake will then use `PUBLIC` by default), the libraries and executables built from the projects would work in most of the scenarios. However, it is certainly not best practice.

Today, I read Kuba Sejdak's blog post "Modern CMake Is Like Inheritance" and I found his interpretation on the CMake keywords `PUBLIC`, `PRIVATE`, and `INTERFACE` inspiring. So in this blog post, I would like to discuss some of my thoughts on these CMake keywords from the perspective of "inheritance".

C++ Inheritance

Access Specifiers

In C++ object oriented programming, there are three types of access specifiers for classes.

Access Specifier	Description
`public`	Members are accessible from outside the class.
`protected`	Members cannot be accessed from outside the class. However, they can be accessed in inherited classes.
`private`	Members cannot be accessed (or viewed) from outside the class.

Alternatively, this could be described using the following simplified table.

Access Specifier	Same Class	Derived Class	Outside Class
`public`	Yes	Yes	Yes
`protected`	Yes	Yes	No
`private`	Yes	No	No

Inheritance Types

When it comes to class inheritance, there are also three types of inheritances.

Inheritance Type	Description
`public`	Public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
`protected`	Public and protected members of the base class become protected members of the derived class.
`private`	Public and protected members of the base class become private members of the derived class.

Alternatively, this could be described using the following simplified table.

Inheritance Type	base: public member	base: protected member	base: private member
`public`	`derived`: `public` member	`derived`: `protected` member	-
`protected`	`derived`: `protected` member	`derived`: `protected` member	-
`private`	`derived`: `private` member	`derived`: `private` member	-

CMake Inheritance

CMake uses somewhat similar inheritance concepts to C++, especially for the C++ `public` and `private` access specifiers and inheritance types. The CMake keywords `PUBLIC`, `PRIVATE`, and `INTERFACE` used in `target_include_directories` and `target_link_libraries`, in my opinion, are mixtures of access specifier and inheritance type from C++.

Include Inheritance

In CMake, for any `target`, in the preprocessing stage, it comes with a `INCLUDE_DIRECTORIES` and a `INTERFACE_INCLUDE_DIRECTORIES` for searching the header files building. `target_include_directories` will populate all the directories to `INCLUDE_DIRECTORIES` and/or `INTERFACE_INCLUDE_DIRECTORIES` depending on the keyword `<PRIVATE|PUBLIC|INTERFACE>` we specified. The `INCLUDE_DIRECTORIES` will

be used for the current `target` only and the `INTERFACE_INCLUDE_DIRECTORIES` will be appended to the `INCLUDE_DIRECTORIES` of any other `target` which has dependencies on the current `target`. With such settings, the configurations of `INCLUDE_DIRECTORIES` and `INTERFACE_INCLUDE_DIRECTORIES` for all building targets are easy to compute and scale up even for multiple hierarchical layers of building dependencies and many building targets.

Include Inheritance	Description
`PUBLIC`	All the directories following `PUBLIC` will be used for the current target and the other targets that have dependencies on the current target, i.e., appending the directories to `INCLUDE_DIRECTORIES` and `INTERFACE_INCLUDE_DIRECTORIES`.
`PRIVATE`	All the include directories following `PRIVATE` will be used for the current target only, i.e., appending the directories to `INCLUDE_DIRECTORIES`.
`INTERFACE`	All the include directories following `INTERFACE` will NOT be used for the current target but will be accessible for the other targets that have dependencies on the current target, i.e., appending the directories to `INTERFACE_INCLUDE_DIRECTORIES`.

Note that when we do `target_link_libraries(<target> <PRIVATE|PUBLIC|INTERFACE> <item>)`, the dependent `<item>`, if built in the same CMake project, would append the `INTERFACE_INCLUDE_DIRECTORIES` of `<item>` to the `INCLUDE_DIRECTORIES` of `<target>`. By controlling the `INTERFACE_INCLUDE_DIRECTORIES`, we could eliminate some unwanted or conflicting declarations from `<item>` to the `<target>`.

For example, the `fruit` library has `INCLUDE_DIRECTORIES` of `fruit_h`, `tree_h`, and `INTERFACE_INCLUDE_DIRECTORIES` of `fruit_h`. If there is a `apple` library that is linked with the `fruit` library, the `apple` library would also have the `fruit_h` in its `INCLUDE_DIRECTORIES` as well. We could equivalently say, the `apple` library's include directory inherited the `fruit_h` of the `fruit` library.

Link Inheritance

Similarly, for any `target`, in the linking stage, we would need to decide, given the `item` to be linked, whether we have to put the `item` in the link dependencies, or the link interface, or both, in the compiled `target`. Here the link dependencies means the `item` has some implementations that the `target` would use, and it is linked to the `item`, so that whenever we call the functions or methods corresponding to those implementations it will always be mapped correctly to the implementations in `item` via the link, whereas the link interface means the `target` becomes an interface for linking the `item` for other targets which have dependencies on the `target`, and the `target` does not have to use `item` at all.

Link Type	Description
`PUBLIC`	All the objects following `PUBLIC` will be used for linking to the current target and providing the interface to the other targets that have dependencies on the current target.

Link Type	Description
`PRIVATE`	All the objects following `PRIVATE` will only be used for linking to the current target.
`INTERFACE`	All the objects following `INTERFACE` will only be used for providing the interface to the other targets that have dependencies on the current target.

For example, if the `fruit` library has the implementation of functions, such as `size` and `color`, and the `apple` library has a function `apple_size` which called the `size` from the `fruit` library and was `PRIVATE` linked with the `fruit` library. We could create an executable `eat_apple` that calls `apple_size` by `PUBLIC` or `PRIVATE` linking with the `apple` library. However, if we want to create an executable `eat_apple` that calls the `size` and `color` from the `fruit` library, only linking with the `apple` library will cause building error, since the `fruit` library was not part of the interface in the `apple` library, and is thus inaccessible to `eat_apple`. To make the `apple` library to inherit the `size` and `color` from the `fruit` library, we have to make the linking of the `apple` library to the the `fruit` library `PUBLIC` instead of `PRIVATE`.

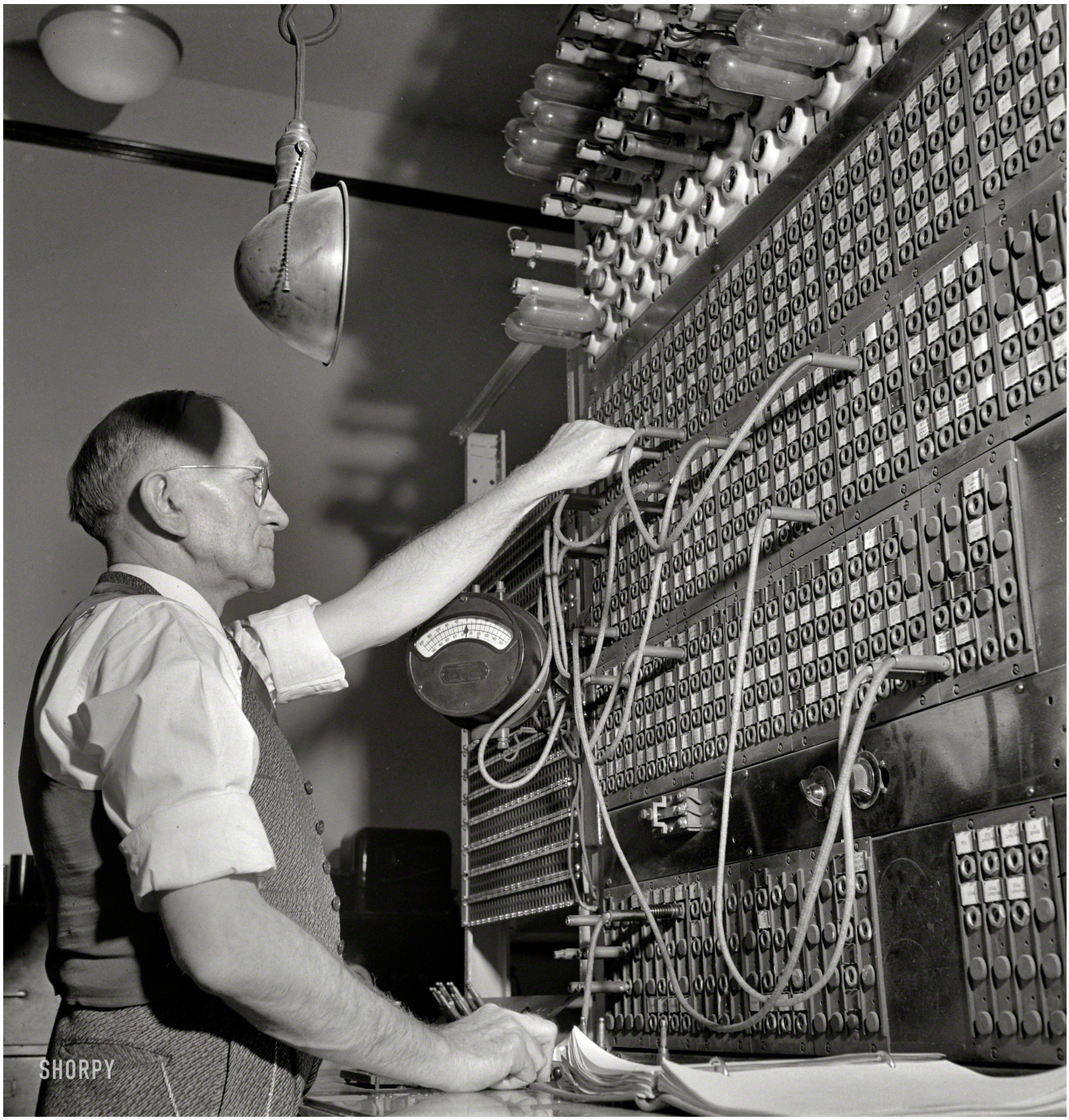
Conclusions

The CMake builds a hierarchical project via the include interface or link interface. The “inheritance” mechanism in C++ is built upon the include interface or link interface.

FAQ

How to Understand CMake Interface?

In my understanding, CMake interface is just like a telephone switch station in old times.



If `A` wants to call `B` and there is no direct telephone cable connection between `A` and `B`, `A` has to call a telephone switch station that has connection to `B` and the personal in the telephone switch station will connect `A` and `B` by jointing the cable of `A` and the cable of `B` together. If the telephone switch station does not know there is a `B`, it is impossible to get `A` and `B` connected. So CMake interface is simply a registration in the telephone switch station. When there is a dependency in CMake targets, targets from different levels of hierarchy are connected via interfaces, for both `include` and `link`.

How to Understand CMake Public, Private, and Interface Using Transitive and Non-Transitive Dependency?

In retrospect, it's much easier to understand CMake `PUBLIC`, `PRIVATE`, and `INTERFACE` using transitive and non-transitive dependency. Please read my new article "Transitive VS Non-Transitive Dependency In Build" for more details.

What are the Key Points for This Blog Post?

`PRIVATE` only cares about himself and does not allow inheritance. `INTERFACE` only cares about others and allows inheritance. `PUBLIC` cares about everyone and allows inheritance.

Is PUBLIC, PRIVATE, INTERFACE Part of the GCC/G++ Compiler?

No. Compilers, such as `gcc` and `g++`, do not have such mechanism. CMake invented those keywords for user to create a building graph that has very clear and explicit dependencies. The building graph translates to normal building commands using `gcc` and `g++`.

References