C++ Libraries

More info at:

cplusplus.com
cppreference.com
isocpp.org
learncpp.com

by **NeptuneLuke**

Index

Standard Library

The **C Standard Library (**from now on **C SL)** is a collection of libraries (headers) that provides macros, type definitions and functions for various tasks such as string handling, mathematical operations, i/o handling, memory management, and several other services, only for the C language.
The **C++ Standard Template Library (**from now on **C++ STL)** is the Standard Library equivalent for C++. The **C++ STL** contains:

- C++ libraries
- C SL original libraries
- C++ headers for C library facilities

The **C SL headers** are included for compatibility, even though they are deprecated, and in most cases should be replaced by their C++ implementation.
The intended use of these headers is for interoperability only.

Question

How can I recognize a C SL header from a C++ STL header?

All the **C SL** headers are named following the format "**xxx.h**".
The **C++ STL** headers for C library facilities are named according to the format "**cxxx**". Except otherwise noted, the contents of each header "**cxxx**" is the same as that of the corresponding header "**xxx.h**".

The **C++ STL** headers are named according to the format "**yyy**". As you may have noticed, **C++ STL native header files don't require an extension** in the #include directive.

```
#include <string.h>  C SL header
#include <cstring>   C++ STL retrocompatible header

#include <time.h>    C SL header
#include <ctime>     C++ STL retrocompatible header

#include <algorithm> C++ STL header
#include <iostream>  C++ STL header
```

Standard libraries

What we talked about so far are C/C++ **standard header files** (part of ISO C/C++).
They come **included with the compiler/IDE**.
We can import them directly in a program using the preprocessor directive **`#include`**. When you #include a file, the content of the file is inserted at the point of inclusion, pulling in declarations of the source file.

Non-standard libraries

It is also possible to use **non-standard header files** (which are not part of ISO C/C++).
They are **defined by programmers for their convenience** and most of the time are only used by the ones that developed them, but can also be distributed like the **Boost Library** or bundled with a specific compiler/IDE.

Header files and Libraries

So far we used the terms headers and libraries interchangeably. There is a fundamental difference between them.
Header refers to the **header file** (.h or .hpp) that **defines an interface**.
Library refers to the **source file** (.c or .cpp) that **provides the implementation for that interface**.

The header of a library is going to tell the computer (the compiler to be more specific) the **signatures of functions**, the **names of variables**, **templating** and **class definitions** provided by the library.
The header file basically tells you **how to call some functionality** without the need to know how it works/is implemented.

The library properly so called is a source file which is compiled and is executed at run time (and the one that actually does the work).
The library file implements the interface defined by the header file.

With the division of interface and implementation we can have the same header file for different libraries (so the functionality needed is exactly called in the same way), and each library may implement the functionality in a different way. By keeping the same interface, we can replace the libraries without changing the code.

While there is nothing stopping us to implement directly into a header file, this approach is not favored as it can introduce extra coupling and dependencies in the code, and also negate all the advantages described before.

Tip

Generally speaking, you should put as many includes in your source file (.c or .cpp) as possible and only the ones that are needed by the header in the header file (.h or .hpp).

Question

What is the difference between .h, .hpp, .c, .cpp extensions? Can I use them interchangeably?

It depends if we are considering the header file or the source file, but as always we should follow the conventions:

- C header files must only use the **.h** extension.
- C++ header files can use
  - **.h**
  - **.hpp**
  - **.h++**
  - **.hh**
  - **.hxx**

  extensions **but they are not required** and could be omitted. If you choose to use one, it doesn't matter which it is. Most programmers use **.h** and **.hpp** extensions.
- C source files must only use the **.c** extension.
- C++ source files must use
  - **.cpp**
  - **.cxx**
  - **.cc**

  formatting. Which of those you choose doesn't matter. Most programmers use the **.cpp** extension.

Question

I didn't include <someheader> and my program worked anyway! Why?

You included some other header (let's say <iostream>), which itself included <someheader>. Although your program will compile, you should not rely on this. What compiles for you might not compile on another machine.

Each file should **explicitly** #include all of the header files it needs to compile. Do not rely on headers included transitively from other headers.

Include directives

As previously said, we can import header files by the preprocessor directive **`#include`**. There are two ways to use this instruction:

#include <header.h> - #include <header>

Is typically used with C SL or C++ STL headers (system-supplied, meaning files in system/compiler/IDE directory).
To be more specific, the preprocessor looks for the file in an implementation-dependent way, normally in directories pre-designed by the compiler/IDE where the standard library headers reside.

```
#include <algorithm>
#include <cstring>
#include <time.h>
```

#include "header.h" - #include "header"

Is typically used for user defined headers or for header files located in the same directory as the file which is including them. The preprocessor searches in the same directory as the file containing the directive.

```
main.cpp
   #include "my_library.h"  C header
   #include "my_library"    C++ header

my_library.h
```

```
   int mul(int a, int b);
```

my_library.c

```
   int mul(int a, int b) {
       return a*b;
   }
```

my_library

```
   int mul(int a, int b);
```

my_library.cpp

```
   int mul(int a, int b) {
       return a*b;
   }
```

It is good practice to include the header file in both the implementation and the main file.

main.cpp

```
   #include "my_library"    C++ header
```

my_library

```
   int mul(int a, int b);
```

my_library.cpp

```
   #include "my_library"    C++ header
   int mul(int a, int b) {
       return a*b;
   }
```

Including multiple headers

Problems may pop up if two (or more) include files contain the same third file. One solution is to avoid include files from including any other files, possibly requiring the programmer to manually add extra include directives to the original file.
When a header file is included n times within a program, the compiler includes the contents of that header files n times. This leads to an error in the program.

Header guards

To eliminate this error, the conditional preprocessor directives **`#ifndef #define #endif`**, called **header guards** are used.

```
main.cpp
   #ifndef HEADER_FILE
   #define HEADER_FILE
   ...
   here go the contents of the header
   ...
   #endif
```

When the header is included again, the conditional will become false, because **`HEADER_FILE`** is defined. The preprocessor will skip over the entire file contents, and the compiler will no longer see it n-times.

`#define, #ifndef` basically creates pre-processor variables, that (including names of header guards) must be unique throughout the program. To avoid name clashes with other entities in our programs, pre-processor variables usually are written in all uppercase.

Pragma once

An alternative is to use another preprocessor directive, called **`#pragma once`**.

```
grandparent.h
   #pragma once
   int mul(int a, int b);

parent.h
   #include "grandparent.h"
   int mul_3(int a, int b, int c);

child.c
   #include "grandparent.h"
   #include "parent.h"
```

It is a non-standard but widely supported preprocessor directive designed to cause the current source file to be included only once in a single compilation. Thus, **`#pragma once`** serves the same purpose as include guards, but with several advantages, including, less code, avoidance of name clashes, sometimes improvement in compilation speed.

Since the pre-processor itself is responsible for handling **`#pragma once`**, the programmer cannot make errors which cause name clashes, but how exactly **`#pragma once`** does it, is an implementation-specific detail. On the other hand, as we said **`#pragma once`** is not necessarily available in all compilers (note that practically every compiler nowadays supports **`#pragma once`**).

Including multiple mutually-exclusive headers

Sometimes it's essential to include several different header files based on the requirements of the program. To do this, the preprocessor directive **wrapper `#if`, `#elif`** is used.

```
main.cpp
   #if system_xxx
       #include "system_xxx"
   #elif system_yyy
       #include "system_yyy"
   #elif system_zzz
       #include "system_zzz"
   #endif
```

Compiler

The compiler sequentially goes through each source code (.cpp) file in your program and does two important tasks:

- checks your code to make sure it follows the rules of the C++ language. If it does not, the compiler will throw an error (at the corresponding line number) to help pinpoint what needs to be fixed. The compilation process will also be aborted until the error is fixed.

- translates your C++ code into machine language instructions. These instructions are stored in an intermediate file called an object file (.o or .obj). The object file also contains metadata that is required or useful in subsequent steps.

The compilation process is followed by the linking process.

Linker

The linker it's responsible for the task right after the compilation and does the following work:

- reads in each of the object files generated by the compiler and makes sure they are valid.

- ensures all cross-file dependencies are resolved properly. For example, if you define something in one .cpp file, and then use it in a different .cpp file, the linker connects the two together. If the linker is unable to connect a reference to something with its definition, you'll get a linker error, and the linking process will abort.

- links header files.

Finally, the linker links all the object files into a final file, the executable.

Static linking

The contents of that file are included at link time (basically a copy of that function is made). In other words, the contents of the file are physically inserted into the executable that you will run. Statically-linked files are 'locked' to the executable at link time so they never change.

Dynamic linking (DLL)

The linking is not made at link time but at execution time, the linker has only to insert in the object file the info needed to make the link at runtime. A dynamically linked file referenced by an executable can change just by replacing the file on the disk. This allows updates to functionality without having to re-link the code. The loader re-links every time you run it.