2.11 — Header files

ALEX[1] AUGUST 2, 2022

Headers, and their purpose

As programs grow larger (and make use of more files), it becomes increasingly tedious to have to forward declare every function you want to use that is defined in a different file. Wouldn't it be nice if you could put all your forward declarations in one place and then import them when you need them?

C++ code files (with a .cpp extension) are not the only files commonly seen in C++ programs. The other type of file is called a **header file**. Header files usually have a .h extension, but you will occasionally see them with a .hpp extension or no extension at all. The primary purpose of a header file is to propagate declarations to code files.

Key insight

Header files allow us to put declarations in one location and then import them wherever we need them. This can save a lot of typing in multi-file programs.

Using standard library header files

Consider the following program:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!";
    return 0;
}
```

This program prints "Hello, world!" to the console using *std::cout*. However, this program never provided a definition or declaration for *std::cout*, so how does the compiler know what *std::cout* is?

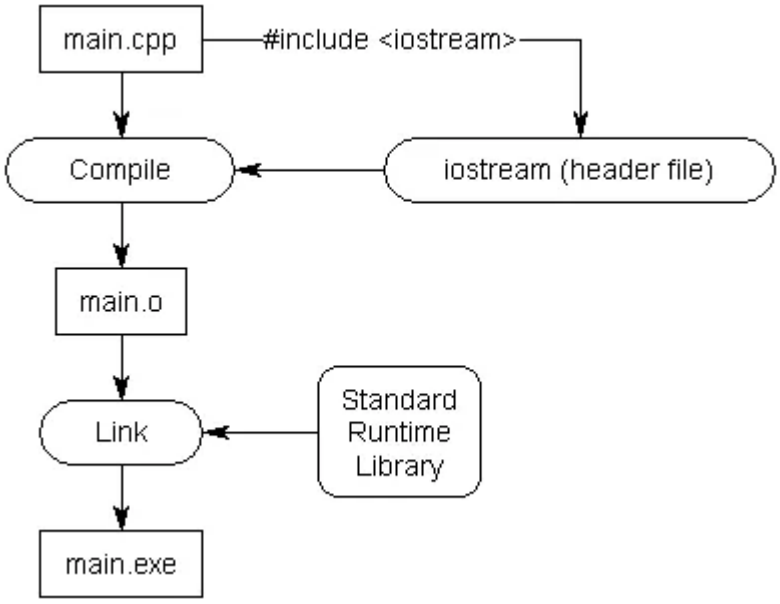
The answer is that *std::cout* has been forward declared in the "iostream" header file. When we `#include <iostream>`, we're requesting that the preprocessor copy all of the content (including forward declarations for std::cout) from the file named "iostream" into the file doing the #include.

Key insight

When you #include a file, the content of the included file is inserted at the point of inclusion. This provides a useful way to pull in declarations from another file.

Consider what would happen if the *iostream* header did not exist. Wherever you used *std::cout*, you would have to manually type or copy in all of the declarations related to *std::cout* into the top of each file that used *std::cout*! This would require a lot of knowledge about how *std::cout* was declared, and would be a ton of work. Even worse, if a function prototype was added or changed, we'd have to go manually update all of the forward declarations. It's much easier to just `#include <iostream>`!

When it comes to functions and variables, it's worth keeping in mind that header files typically only contain function and variable declarations, not function and variable definitions (otherwise a violation of the *one definition rule* could result). *std::cout* is forward declared in the iostream header, but defined as part of the C++ standard library, which is automatically linked into your program during the linker phase.



Best practice

Header files should generally not contain function and variable definitions, so as not to violate the one definition rule. An exception is made for symbolic constants (which we cover in lesson 4.13 -- Const variables and symbolic constants (https://www.learncpp.com/cpp-tutorial/const-variables-and-symbolic-constants/)[2]).

Writing your own header files

Now let's go back to the example we were discussing in a previous lesson. When we left off, we had two files, *add.cpp* and *main.cpp*, that looked like this:

add.cpp:

```
int add(int x, int y)
{
    return x + y;
}
```

main.cpp:

```
#include <iostream>

int add(int x, int y); // forward declaration using function prototype

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

(If you're recreating this example from scratch, don't forget to add *add.cpp* to your project so it gets compiled in).

In this example, we used a forward declaration so that the compiler will know what identifier *add* is when compiling *main.cpp*. As previously mentioned, manually adding forward declarations for every function you want to use that lives in another file can get tedious quickly.

Let's write a header file to relieve us of this burden. Writing a header file is surprisingly easy, as header files only consist of two parts:

1. A *header guard*, which we'll discuss in more detail in the next lesson (2.12 -- Header guards[3]).
2. The actual content of the header file, which should be the forward declarations for all of the identifiers we want other files to be able to see.

Adding a header file to a project works analogously to adding a source file (covered in lesson 2.8 -- Programs with multiple code files (https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/)[4]). If using an IDE, go through the same steps and choose "Header" instead of "Source" when asked. If using the command line, just create a new file in your favorite editor.

Best practice

Use a .h suffix when naming your header files.

Header files are often paired with code files, with the header file providing forward declarations for the corresponding code file. Since our header file will contain a forward declaration for functions defined in *add.cpp*, we'll call our new header file *add.h*.

Best practice

If a header file is paired with a code file (e.g. add.h with add.cpp), they should both have the same base name (add).

Here's our completed header file:

add.h:

```
// 1) We really should have a header guard here, but will omit it for simplicity (we'll cover header guards in the next lesson)

// 2) This is the content of the .h file, which is where the declarations go
int add(int x, int y); // function prototype for add.h -- don't forget the semicolon!
```

In order to use this header file in main.cpp, we have to #include it (using quotes, not angle brackets).

main.cpp:

```
#include "add.h" // Insert contents of add.h at this point.  Note use of double quotes here.
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

add.cpp:

```
#include "add.h" // Insert contents of add.h at this point.  Note use of double quotes here.

int add(int x, int y)
{
    return x + y;
}
```

When the preprocessor processes the `#include "add.h"` line, it copies the contents of add.h into the current file at that point. Because our *add.h* contains a forward declaration for function *add*, that forward declaration will be copied into *main.cpp*. The end result is a program that is functionally the same as the one where we manually added the forward declaration at the top of *main.cpp*.

Consequently, our program will compile and link correctly.

()Source files should include their paired header (#corresponding_include)[5]

In C++, it is a best practice for code files to #include their paired header file (if one exists). In the example above, *add.cpp* includes *add.h*.

This allows the compiler to catch certain kinds of errors at compile time instead of link time. For example:

something.h:

```
1 | int something(int); // return type of forward declaration is int
```

something.cpp:

```
1 | #include "something.h"
2 |
3 | void something(int) // error: wrong return type
4 | {
5 | }
```

Because *something.cpp* #includes *something.h*, the compiler will notice that function *something()* has a mismatched return type and give us a compile error. If *something.cpp* did not #include *something.h*, we'd have to wait until the linker discovered the discrepancy, which wastes time. For another example, see this comment (https://www.learncpp.com/cpp-tutorial/header-files/comment-page-8/#comment-398571)[6].

Best practice

Source files should #include their paired header file (if one exists).

Troubleshooting

If you get a compiler error indicating that *add.h* isn't found, make sure the file is really named *add.h*. Depending on how you created and named it, it's possible the file could have been named something like *add* (no extension) or *add.h.txt* or *add.hpp*. Also make sure it's sitting in the same directory as the rest of your code files.

If you get a linker error about function *add* not being defined, make sure you've added *add.cpp* in your project so the definition for function *add* can be linked into the program.

()Angled brackets vs double quotes (#includemethod)[7]

You're probably curious why we use angled brackets for `iostream`, and double quotes for `add.h`. It's possible that a header file with the same filename might exist in multiple directories. Our use of angled brackets vs double quotes helps give the preprocessor a clue as to where it should look for header files.

When we use angled brackets, we're telling the preprocessor that this is a header file we didn't write ourselves. The preprocessor will search for the header only in the directories specified by the `include directories`. The `include directories` are configured as part of your project/IDE settings/compiler settings, and typically default to the directories containing the header files that come with your compiler and/or OS. The preprocessor will not search for the header file in your project's source code directory.

When we use double-quotes, we're telling the preprocessor that this is a header file that we wrote. The preprocessor will first search for the header file in the current directory. If it can't find a matching header there, it will then search the `include directories`.

Rule

Use double quotes to include header files that you've written or are expected to be found in the current directory. Use angled brackets to include headers that come with your compiler, OS, or third-party libraries you've installed elsewhere on your system.

Why doesn't iostream have a .h extension?

Another commonly asked question is "why doesn't iostream (or any of the other standard library header files) have a .h extension?". The answer is that *iostream.h* is a different header file than *iostream*! To explain requires a short history lesson.

When C++ was first created, all of the files in the standard library ended in a *.h* suffix. Life was consistent, and it was good. The original version of *cout* and *cin* were declared in *iostream.h*. When the language was standardized by the ANSI committee, they decided to move all of the names used in the standard library into the *std* namespace to help avoid naming conflicts with user-defined identifiers. However, this presented a problem: if they moved all the names into the *std* namespace, none of the old programs (that included iostream.h) would work anymore!

To work around this issue, a new set of header files was introduced that lack the *.h* extension. These new header files define all names inside the *std* namespace. This way, older programs that include `#include <iostream.h>` do not need to be rewritten, and newer programs can `#include <iostream>`.

Key insight

The header files with the *.h* extension define their names in the global namespace, and may optionally define them in the *std* namespace as well.

The header files without the *.h* extension will define their names in the *std* namespace, and may optionally define them in the global namespace as well.

In addition, many of the libraries inherited from C that are still useful in C++ were given a *c* prefix (e.g. *stdlib.h* became *cstdlib*). The functionality from these libraries was also moved into the *std* namespace to help avoid naming collisions.

Best practice

When including a header file from the standard library, use the version without the .h extension if it exists. User-defined headers should still use a .h extension.

Including header files from other directories

Another common question involves how to include header files from other directories.

One (bad) way to do this is to include a relative path to the header file you want to include as part of the #include line. For example:

```
1 #include "headers/myHeader.h"
2 #include "../moreHeaders/myOtherHeader.h"
```

While this will compile (assuming the files exist in those relative directories), the downside of this approach is that it requires you to reflect your directory structure in your code. If you ever update your directory structure, your code won't work anymore.

A better method is to tell your compiler or IDE that you have a bunch of header files in some other location, so that it will look there when it can't find them in the current directory. This can generally be done by setting an *include path* or *search directory* in your IDE project settings.

For Visual Studio users

Right click on your project in the *Solution Explorer*, and choose *Properties*, then the *VC++ Directories* tab. From here, you will see a line called *Include Directories*. Add the directories you'd like the compiler to search for additional headers there.

For Code::Blocks users

In Code::Blocks, go to the *Project* menu and select *Build Options*, then the *Search directories* tab. Add the directories you'd like the compiler to search for additional headers there.

For GCC/G++ users

Using g++, you can use the -I option to specify an alternate include directory.

```
1 | g++ -o main -I/source/includes main.cpp
```

The nice thing about this approach is that if you ever change your directory structure, you only have to change a single compiler or IDE setting instead of every code file.

()Headers may include other headers (#transitive)[8]

It's common that a header file will need a declaration or definition that lives in a different header file. Because of this, header files will often #include other header files.

When your code file #includes the first header file, you'll also get any other header files that the first header file includes (and any header files those include, and so on). These additional header files are sometimes called **transitive includes**, as they're included implicitly rather than explicitly.

The content of these transitive includes are available for use in your code file. However, you should not rely on the content of headers that are included transitively. The implementation of header files may change over time, or be different across different systems. If that happens, your code may only compile on certain systems, or may compile now but not in the future. This is easily avoided by explicitly including all of the header files the content of your code file requires.

Best practice

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

Unfortunately, there is no easy way to detect when your code file is accidentally relying on content of a header file that has been included by another header file.

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

This is one of the most commonly asked questions on this site. The answer is: it's likely working, because you included some other header (e.g. <iostream>), which itself included <someheader>. Although your program will compile, per the best practice above, you should not rely on this. What compiles for you might not compile on a friend's machine.

The #include order of header files

If your header files are written properly and #include everything they need, the order of inclusion shouldn't matter.

Now consider the following scenario: let's say header A needs declarations from header B, but forgets to include it. In our code file, if we include header B before header A, our code will still compile! This is because the compiler will compile all the declarations from B before it compiles the code from A that depends on those declarations.

However, if we include header A first, then the compiler will complain because the code from A will be compiled before the compiler has seen the declarations from B. This is actually preferable, because the error has been surfaced, and we can then fix it.

Best practice

To maximize the chance that missing includes will be flagged by compiler, order your #includes as follows:

1. The paired header file
2. Other headers from your project
3. 3rd party library headers
4. Standard library headers

The headers for each grouping should be sorted alphabetically.

That way, if one of your user-defined headers is missing an #include for a 3rd party library or standard library header, it's more likely to cause a compile error so you can fix it.

Header file best practices

Here are a few more recommendations for creating and using header files.

- Always include header guards (we'll cover these next lesson).
- Do not define variables and functions in header files (global constants are an exception -- we'll cover these later)
- Give a header file the same name as the source file it's associated with (e.g. *grades.h* is paired with *grades.cpp*).
- Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
- Be mindful of which headers you need to explicitly include for the functionality that you are using in your code files
- Every header you write should compile on its own (it should #include every dependency it needs)
- Only #include what you need (don't include everything just because you can).
- Do not #include .cpp files.

Next lesson

2.12 Header guards

3

Back to table of contents

9

Previous lesson

2.10 Introduction to the preprocessor

10

11

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment!

Notify me about replies:

POST COMMENT

Avatars from https://gravatar.com/[12] are connected to your provided email address.

803 COMMENTS

Newest

boofar

October 4, 2022 5:39 am

Why is it best practice to use .h and not .hpp? I thought .hpp was made up specifically for C++ and is the counterpart to .cpp

I always thought:

C: .c and .h
C++: .cpp and .hpp

0 Reply

Alex Author

Reply to boofar [13] October 4, 2022 7:11 pm

Mainly because most tools and guides still recommend .h over .hpp (e.g. see http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rl-file-suffix). Inertia. You're welcome to use .hpp instead if you like.

0 Reply

Waldo

September 30, 2022 2:14 pm

Side note: I switched from using .h to using .hpp because some tools rely on the extension to distinguish between C and C++ header files.

I started splitting up my header files into tinier ones after seeing how many small header files some projects have. I've realized that doing so has a number of advantages:

1. It reduces compile times thanks to a smaller amount of transitive includes. The amount of global identifiers is also reduced. Consider the following example (header guards omitted for brevity):

```
1  // a.hpp
2  #include <istream>
3  #include <stdexcept>
4  #include <string>
5  // etc.
6
7  void useIstream(std::istream);
8  void useStdexcept(std::exception);
9  void useString(std::string);
10 // etc.
```

```
1  // main.cpp
2  #include "a.hpp" // also includes <istrem>, <stdexcept> and <string>
3  #include <string>
4
5  int main() { useString(std::string{}); }
```

Three standard library headers and their transitive includes get parsed and compiled. Now split up a.hpp:

```
1  // a.hpp
2  #include <istream>
3
4  void useIstream(std::istream);
```

```
1  // b.hpp
2  #include <stdexcept>
3
4  void useStdexcept(std::exception);
```

```
1  // c.hpp
2  #include <string>
3
4  void useString(std::string);
```

```cpp
// main.cpp
#include "c.hpp" // Only includes <string>
#include <string>

int main() { useString(std::string{}); }
```

Only <string> and its transitive includes get parsed and compiled.

This example is obviously exaggerated, but I'm sure my point applies to a lot of large projects.

2. Inspecting code to find what headers are needed is easier.

As a header file (or a code file) grows, the amount of #includes grows too. This can lead to files that start similarly to this:

```cpp
#include <algorithm>
#include <any>
#include <array>
#include <atomic>
#include <bit>
#include <bitset>
#include <cassert>
#include <cctype>
#include <cerrno>
#include <cfenv>
#include <cfloat>
#include <chrono>
#include <cinttypes>
#include <climits>
#include <cmath>
```

In such a file, making sure that nothing relies on transitive includes is challenging. Some #includes may also be unnecessary because the code that used them was removed, which wastes time when compiling. Having smaller files makes looking for these issues easier.

3. Code is easier to reason about.

Code can become confusing if you need to scroll around a lot. Scrolling also wastes time. Keeping files one screen height long removes the need to scroll. When working on code in multiple files in the same time, you can dock them side to side (screens are wider than they are tall).

4. Smaller header files can be given more specific names, which makes it easier to find things. For example, it's not currently obvious where std::stoi is, but if it were in a separate header from <string> called <string_conversions> or <numeric_conversions>, it would be more obvious.
5. A project with 50 code files looks more impressive than a project with 5.

Georgi
September 10, 2022 6:30 am

Hello, my files dont compile i cant figure out why i have added the header file to both of the cpp files and my header is with .h, but says no such file or directory, any idea what i can do to fix it?

Last edited 1 month ago by Georgi

0 Reply

> **Zak**
> Reply to Georgi [14] September 17, 2022 2:33 pm
>
> I will have to see more. Can you copy/paste your code?
>
> 0 Reply

Samira
September 7, 2022 7:16 pm

The following is the content of add.cpp and main.cpp

```
#include <iostream>
#include "add.h"

int main()
{
    std::cout << "The sum of " << 3 << " and " << 4 << " is: " << add(3,
4);
    return 0;
}
```

```
#include "add.h"
int  add(int x)
{

    return x;
}
```

In that example above, the compiler can't detect the issue despite the fact "add.h" is included in the add.cpp because it think of this function as an overloaded version and this can only be detected during the link time.

Last edited 1 month ago by Samira

0 Reply

Zak

Reply to Samira [15] September 17, 2022 2:45 pm

In main.cpp, there are two issues.

1. #include your header file before the standard library.
2. Your calling add() with two arguments, but it is initialized with one.

Please see the following.

in add.h

```
#ifndef ADD_H
#define ADD_H

int add(int a, int b);

#endif
```

in add.cpp

```
#include "add.h"

int add(int a, int b)
{
   return a + b;
}
```

int main.cpp

```
#include "add.h"
#include<iostream>

int main()
{
   std::cout<<"The numbers added together are "<<add(1,1)<<".\n";
   return 0;
}
```

1 Reply

Mikko Rintala

August 12, 2022 1:16 am

If the standard library is automatically linked, why couldn't all the headers be included automatically as well?

In terms of memory the definition is always at least as expensive as the declaration, right?

As a follow up question, shouldn't C++ provide us with "You only pay for what you use"? The automatic inclusion of the C++ Standard Library seems to go against that, athough on the other hand it could be viewed more as a feature of the compiler than the language?

Hopefully the above makes it easy to spot where my reasoning goes wrong and someone would be kind enough to point it to me also :)

1 Reply

Alex Author

Reply to Mikko Rintala [16] August 14, 2022 8:32 pm

There are at least a few good reasons not to include all header files:

1. Performance (would take longer to compile everything)
2. It hides the dependencies of what your code files are using

You can disable the inclusion of the standard library if you don't want any of it. Otherwise, most application dynamically link to the C++ runtimes, which means they load the library code at runtime from a system directory.

1 Reply

Percy

Reply to Mikko Rintala [16] August 12, 2022 10:56 pm

Same doubt. Why is the standard library being included automatically and header files have a choice. My best guess is it is to save the compilation time.

0 Reply

Anon

August 8, 2022 10:19 pm

Hello! This could be an obvious question, but what is the merit or upside of going the extra step and #including a .h file rather than just #including the c++ file itself? #include “add.cpp” seems to work okay for me but there could be a major difference I’m missing out on.

0 Reply

Ahuno

Reply to Anon [17] August 13, 2022 6:23 am

Non-trivially, using headers is a very standard procedure for software development projects. You might want to search for a certain function in a huge code base. The tools fetching those functions for you will first search for the header files since the amount of text in the headers will be much less than in the cpp files.

0 Reply

Alex Author

Reply to Anon [17] August 10, 2022 1:23 pm

.cpp files weren't designed to be #included, and will likely result in violations of the one-definition rule (at least in non-trivial cases, such as when a .cpp file gets included more than once).

1 Reply

ali

August 5, 2022 10:17 pm

Does the iostream contain both declaration & definition of functions like cout & cin or just a pure header containing declarations?

Last edited 2 months ago by ali

0 Reply

Alex Author

Reply to ali [18] August 6, 2022 11:00 pm

This is implementation dependent. But it's likely that the iostream header contains both declarations and definitions, since most reasonably complex headers have both.

std::cout and std::cin are objects, not functions.

0 Reply

Ahuno

Reply to Alex [19] August 13, 2022 6:26 am

Hi Alex. Isn't defining functions or objects within headers go against all the conventions such as One definition rule.

Last edited 2 months ago by Ahuno

0 Reply

Alex Author

Reply to Ahuno [20] August 15, 2022 11:19 am

Note that I said headers contain both declarations and definitions, but I did not say definitions of what. Header often contain definitions for things such as types and templates, which are not subject to the ODR in the same way that functions and objects are.

Second, objects and functions *can* be defined in headers if they are defined as inline. We cover this (including why you might want to do so) in future chapters.

0 Reply

J34NP3T3R

July 26, 2022 5:03 am

how come header files understands some types that we have yet to include ?

for example

main.cpp

```
#include <iostream>
#include "add.h"


int main()
{
    int num1{ getIntInput("Please enter a number : ") };
    int num2{ getIntInput("Please enter a second number : ") };

    std::cout << "\nThe sum of " << num1 << " and " << num2 << " is " << num1 +
num2 << "\n" ;

    return 0;
}
```

add.cpp

```cpp
#include <iostream>
#include <string>

int getIntInput(std::string msg)
{
    std::cout << msg;
    int num;
    std::cin >> num;

    return num;
}
```

but then in add.h

```cpp
#pragma once

int getIntInput(std::string);
```

it did not complain in add.h about std::string. <string> not being included

0 Reply

Alex Author

Reply to J34NP3T3R [21] July 27, 2022 12:57 pm

Because in main.cpp (the only place add.h was used) add.h was included AFTER iostream (which itself includes string). So the compiler has seen the definition for std::string before it parses the forward declaration for getIntInput().

This is why it is good to put your own includes before the standard library includes.

2 Reply

J34NP3T3R

Reply to Alex [22] July 27, 2022 3:07 pm

ohh i didn't know iostream already includes string. i used to include string even with iostream. thank you

0 Reply

Alex Author

Reply to J34NP3T3R [23] July 29, 2022 10:18 am

You should explicitly include string if you use it. Just because your iostream includes string doesn't mean an iostream on some other compiler (or a future version of your

compiler) will include string.

3 Reply

Krishnakumar
July 20, 2022 1:46 pm

To maximize the chance that missing includes will be flagged by compiler, order your #includes as follows:

The paired header file
Other headers from your project
3rd party library headers
Standard library headers

Hmm. All the `clang-format` styles I tried (Microsoft, Google, LLVM, Chromium, Webkit) use the exact opposite order for header-file inclusion. Is there any logical explanation for this?

0 Reply

Ahuno
Reply to Krishnakumar [24] August 13, 2022 6:33 am

I think it is for the opposite reason. # including standard libraries first would most likely put the required declarations (or definitions) for the use defined header files so that there will be as little possibility for having compilation errors.

0 Reply

Alex Author
Reply to Krishnakumar [24] July 21, 2022 11:50 am

Not that I'm aware of. If you find a good explanation somewhere, let me know.

0 Reply

Krishnakumar
July 20, 2022 1:38 pm

It's common that a header file will need a declaration or **definition** that lives in a different header file.

hmm..Wondering what is a scenario in which a header fill will need a definition from another header?

1 Reply

Alex Author

Reply to Krishnakumar [25] July 21, 2022 11:48 am

Types and templates are two examples. The compiler must be able to see the full definition of a type or template to instantiate it.

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/const-variables-and-symbolic-constants/
3. https://www.learncpp.com/cpp-tutorial/header-guards/
4. https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/
5. https://www.learncpp.com/cpp-tutorial/header-files/#corresponding_include
6. https://www.learncpp.com/cpp-tutorial/header-files/comment-page-8/#comment-398571
7. https://www.learncpp.com/cpp-tutorial/header-files/#includemethod
8. https://www.learncpp.com/cpp-tutorial/header-files/#transitive
9. https://www.learncpp.com/
10. https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/
11. https://www.learncpp.com/header-files/
12. https://gravatar.com/
13. https://www.learncpp.com/cpp-tutorial/header-files/#comment-573765
14. https://www.learncpp.com/cpp-tutorial/header-files/#comment-572968
15. https://www.learncpp.com/cpp-tutorial/header-files/#comment-572853
16. https://www.learncpp.com/cpp-tutorial/header-files/#comment-571950
17. https://www.learncpp.com/cpp-tutorial/header-files/#comment-571758
18. https://www.learncpp.com/cpp-tutorial/header-files/#comment-571563
19. https://www.learncpp.com/cpp-tutorial/header-files/#comment-571633
20. https://www.learncpp.com/cpp-tutorial/header-files/#comment-572026
21. https://www.learncpp.com/cpp-tutorial/header-files/#comment-571131
22. https://www.learncpp.com/cpp-tutorial/header-files/#comment-571199

23. https://www.learncpp.com/cpp-tutorial/header-files/#comment-571206
24. https://www.learncpp.com/cpp-tutorial/header-files/#comment-570872
25. https://www.learncpp.com/cpp-tutorial/header-files/#comment-570870
26. https://www.ezoic.com/what-is-ezoic/