2.12 — Header guards

ALEX[1] JULY 20, 2022

The duplicate definition problem

In lesson 2.7 -- Forward declarations and definitions (https://www.learncpp.com/cpp-tutorial/forward-declarations/)[2], we noted that a variable or function identifier can only have one definition (the one definition rule). Thus, a program that defines a variable identifier more than once will cause a compile error:

```
int main()
{
    int x; // this is a definition for variable x
    int x; // compile error: duplicate definition

    return 0;
}
```

Similarly, programs that define a function more than once will also cause a compile error:

```
#include <iostream>

int foo() // this is a definition for function foo
{
    return 5;
}

int foo() // compile error: duplicate definition
{
    return 5;
}

int main()
{
    std::cout << foo();
    return 0;
}
```

While these programs are easy to fix (remove the duplicate definition), with header files, it's quite easy to end up in a situation where a definition in a header file gets included more than once. This can happen when a header file #includes another header file (which is common).

Consider the following academic example:

square.h:

```
// We shouldn't be including function definitions in header files
// But for the sake of this example, we will
int getSquareSides()
{
    return 4;
}
```

geometry.h:

```
#include "square.h"
```

main.cpp:

```
#include "square.h"
#include "geometry.h"

int main()
{
    return 0;
}
```

This seemingly innocent looking program won't compile! Here's what's happening. First, *main.cpp* #includes *square.h*, which copies the definition for function *getSquareSides* into *main.cpp*. Then *main.cpp* #includes *geometry.h*, which #includes *square.h* itself. This copies contents of *square.h* (including the definition for function *getSquareSides*) into *geometry.h*, which then gets copied into *main.cpp*.

Thus, after resolving all of the #includes, *main.cpp* ends up looking like this:

```
int getSquareSides()  // from square.h
{
    return 4;
}

int getSquareSides() // from geometry.h (via square.h)
{
    return 4;
}

int main()
{
    return 0;
}
```

Duplicate definitions and a compile error. Each file, individually, is fine. However, because *main.cpp* ends up #including the content of *square.h* twice, we've run into problems. If *geometry.h* needs

getSquareSides(), and *main.cpp* needs both *geometry.h* and *square.h*, how would you resolve this issue?

Header guards

The good news is that we can avoid the above problem via a mechanism called a **header guard** (also called an **include guard**). Header guards are conditional compilation directives that take the following form:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations (and certain types of definitions) here

#endif
```

When this header is #included, the preprocessor checks whether *SOME_UNIQUE_NAME_HERE* has been previously defined. If this is the first time we're including the header, *SOME_UNIQUE_NAME_HERE* will not have been defined. Consequently, it #defines *SOME_UNIQUE_NAME_HERE* and includes the contents of the file. If the header is included again into the same file, *SOME_UNIQUE_NAME_HERE* will already have been defined from the first time the contents of the header were included, and the contents of the header will be ignored (thanks to the #ifndef).

All of your header files should have header guards on them. *SOME_UNIQUE_NAME_HERE* can be any name you want, but by convention is set to the full filename of the header file, typed in all caps, using underscores for spaces or punctuation. For example, *square.h* would have the header guard:

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

Even the standard library headers use header guards. If you were to take a look at the iostream header file from Visual Studio, you would see:

```
#ifndef _IOSTREAM_
#define _IOSTREAM_

// content here

#endif
```

For advanced readers

In large programs, it's possible to have two separate header files (included from different directories) that end up having the same filename (e.g. directoryA\config.h and directoryB\config.h). If only the filename is used for the include guard (e.g. CONFIG_H), these two files may end up using the same guard name. If that happens, any file that includes (directly or indirectly) both config.h files will not receive the contents of the include file to be included second. This will probably cause a compilation error.

Because of this possibility for guard name conflicts, many developers recommend using a more complex/unique name in your header guards. Some good suggestions are a naming convention of <PROJECT>_<PATH>_<FILE>_H , <FILE>_<LARGE RANDOM NUMBER>_H, or <FILE>_<CREATION DATE>_H

Updating our previous example with header guards

Let's return to the *square.h* example, using the *square.h* with header guards. For good form, we'll also add header guards to *geometry.h*.

square.h

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

geometry.h:

```
#ifndef GEOMETRY_H
#define GEOMETRY_H

#include "square.h"

#endif
```

main.cpp:

```
#include "square.h"
#include "geometry.h"

int main()
{
    return 0;
}
```

After the preprocessor resolves all of the #include directives, this program looks like this:

main.cpp:

```
// Square.h included from main.cpp
#ifndef SQUARE_H // square.h included from main.cpp
#define SQUARE_H // SQUARE_H gets defined here

// and all this content gets included
int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H

#ifndef GEOMETRY_H // geometry.h included from main.cpp
#define GEOMETRY_H
#ifndef SQUARE_H // square.h included from geometry.h, SQUARE_H is already
defined from above
#define SQUARE_H // so none of this content gets included

int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H
#endif // GEOMETRY_H

int main()
{
    return 0;
}
```

As you can see from the example, the second inclusion of the contents of *square.h* (from *geometry.h*) gets ignored because *SQUARE_H* was already defined from the first inclusion. Therefore, function *getSquareSides* only gets included once.

Header guards do not prevent a header from being included once into different code files

Note that the goal of header guards is to prevent a code file from receiving more than one copy of a guarded header. By design, header guards do *not* prevent a given header file from being included (once) into separate code files. This can also cause unexpected problems. Consider:

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

int getSquarePerimeter(int sideLength); // forward declaration for
getSquarePerimeter

#endif
```

square.cpp:

```
#include "square.h"  // square.h is included once here

int getSquarePerimeter(int sideLength)
{
    return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h is also included once here
#include <iostream>

int main()
{
    std::cout << "a square has " << getSquareSides() << " sides\n";
    std::cout << "a square of length 5 has perimeter length " <<
getSquarePerimeter(5) << '\n';

    return 0;
}
```

Note that *square.h* is included from both *main.cpp* and *square.cpp*. This means the contents of *square.h* will be included once into *square.cpp* and once into *main.cpp*.

Let's examine why this happens in more detail. When *square.h* is included from *square.cpp*, *SQUARE_H* is defined until the end of *square.cpp*. This define prevents *square.h* from being included into *square.cpp* a second time (which is the point of header guards). However, once *square.cpp* is finished, *SQUARE_H* is no longer considered defined. This means that when the preprocessor runs on *main.cpp*, *SQUARE_H* is not initially defined in *main.cpp*.

The end result is that both *square.cpp* and *main.cpp* get a copy of the definition of *getSquareSides*. This program will compile, but the linker will complain about your program having multiple definitions for identifier *getSquareSides*!

The best way to work around this issue is simply to put the function definition in one of the .cpp files so that the header just contains a forward declaration:

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides(); // forward declaration for getSquareSides
int getSquarePerimeter(int sideLength); // forward declaration for
getSquarePerimeter

#endif
```

square.cpp:

```
#include "square.h"

int getSquareSides() // actual definition for getSquareSides
{
    return 4;
}

int getSquarePerimeter(int sideLength)
{
    return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h is also included once here
#include <iostream>

int main()
{
    std::cout << "a square has " << getSquareSides() << "sides\n";
    std::cout << "a square of length 5 has perimeter length " <<
getSquarePerimeter(5) << '\n';

    return 0;
}
```

Now when the program is compiled, function *getSquareSides* will have just one definition (via *square.cpp*), so the linker is happy. File *main.cpp* is able to call this function (even though it lives in *square.cpp*) because it includes *square.h*, which has a forward declaration for the function (the linker

will connect the call to *getSquareSides* from *main.cpp* to the definition of *getSquareSides* in *square.cpp*).

Can't we just avoid definitions in header files?

We've generally told you not to include function definitions in your headers. So you may be wondering why you should include header guards if they protect you from something you shouldn't do.

There are quite a few cases we'll show you in the future where it's necessary to put non-function definitions in a header file. For example, C++ will let you create your own types. These user-defined types are typically defined in header files, so the type definitions can be propagated out to the code files that need to use them. Without a header guard, a code file could end up with multiple (identical) copies of a given type definition, which the compiler will flag as an error.

So even though it's not strictly necessary to have header guards at this point in the tutorial series, we're establishing good habits now, so you don't have to unlearn bad habits later.

#pragma once

Modern compilers support a simpler, alternate form of header guards using the *#pragma* directive:

```
1 | #pragma once
2 |
3 | // your code here
```

`#pragma once` serves the same purpose as header guards, and has the added benefit of being shorter and less error-prone. For most projects, `#pragma once` works fine, and many developers prefer to use them over header guards. However, `#pragma once` is not an official part of the C++ language (and probably will never be, because it can't be implemented in a way that works reliably in all cases).

For maximum compatibility, we recommend sticking to traditional header guards. They aren't much more work and they're guaranteed to be supported on all compilers.

Best practice

Favor header guards over `#pragma once` for maximum portability.

Summary

Header guards are designed to ensure that the contents of a given header file are not copied more than once into any single file, in order to prevent duplicate definitions.

Note that duplicate *declarations* are fine, since a declaration can be declared multiple times without incident -- but even if your header file is composed of all declarations (no definitions) it's still a best

practice to include header guards.

Note that header guards do *not* prevent the contents of a header file from being copied (once) into separate project files. This is a good thing, because we often need to reference the contents of a given header from different project files.

Quiz time

Question #1

Add header guards to this header file:

add.h:

```
1 | int add(int x, int y);
```

Show Solution (javascript:void(0))[3]

Next lesson

2.13 How to design your first programs

4

Back to table of contents

5

Previous lesson

2.11 Header files

6

7

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

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment!

Avatars from https://gravatar.com/[10] are connected to your provided email address.

Notify me about replies:

POST COMMENT

413 COMMENTS

Newest

R. S.

October 28, 2022 12:26 pm

>Let’s examine why this happens in more detail. When square.h is included from square.cpp, SQUARE_H is defined until the end of square.cpp. This define prevents square.h from being included into square.cpp a second time (which is the point of header guards). However, once square.cpp is finished, SQUARE_H is no longer considered defined. This means that when the preprocessor runs on main.cpp, SQUARE_H is not initially defined in main.cpp.

From this piece of text, I've come to two conclusions:

1. The preprocessor scans the files independently, without a specific order.
2. Preprocessor directives, such as these object-like macros, are only relevant within the files they are defined in.

Am I right?

Last edited 1 day ago by R. S.

0 Reply

Shah

September 21, 2022 6:31 am

I found the language in this article really confusing and hard to understand.
'When this header is #included, the preprocessor checks whether SOME_UNIQUE_NAME_HERE has been previously defined. If this is the first time we’re including the header, SOME_UNIQUE_NAME_HERE will not have been defined. Consequently, it #defines SOME_UNIQUE_NAME_HERE and includes the contents of the file. If the header is included again into the same file, SOME_UNIQUE_NAME_HERE will already have been defined from the first time

the contents of the header were included, and the contents of the header will be ignored (thanks to the #ifndef).'

Okay, what do you mean by header? Do you mean header file or the content of a header file?

Also, what's the difference between ifndef and define? Also I was going to say in this case how the prevention of the double definition work? Let's say in `square.h` the math function is defined. Okay, even though the function inside `square.h` gets copied into `geommetry.h` and they are equivalent to the same code, the header guard DEFINITIONS are different. So why would it prevent duplication of the header code being included if the two header guards are different for each header file?

0 Reply

LUKE

Reply to Shah [11] October 16, 2022 5:46 am

These who have hard time understanding these concept as myself go to https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/ read it again and than try to read this current page one more time, it was hard to grasp at first, Good luck!

0 Reply

Alex Author

Reply to Shah [11] September 22, 2022 1:40 pm

I mean the content of the header file that we defined in the prior code snippet:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations (and certain types of definitions) here

#endif
```

`#define X` defines a new preprocessor macro named `X`. `#ifndef X` checks if `X` has been previously defined, and if not, compiles in the code from the start of the `#ifndef` to the subsequent `#endif`.

The `main.cpp` code example in the lesson shows what the preprocessor/compiler sees when compiling a code file that has had `square.h` included twice:

```cpp
// Square.h included from main.cpp
#ifndef SQUARE_H // square.h included from main.cpp
#define SQUARE_H // SQUARE_H gets defined here

// and all this content gets included
int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H

#ifndef GEOMETRY_H // geometry.h included from main.cpp
#define GEOMETRY_H
#ifndef SQUARE_H // square.h included from geometry.h, SQUARE_H is already defined from above
#define SQUARE_H // so none of this content gets included

int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H
#endif // GEOMETRY_H

int main()
{
    return 0;
}
```

Yes, square.h gets included into geometry.h, but that geometry.h (which includes the content of square.h) is subsequently included into main.cpp. So we end up with two copies of square.h in main.cpp -- one included directly, and one transitively included when we included geometry.h.

Let's look at how this evaluates:

First, the preprocessor evaluates `#ifndef SQUARE_H`. `SQUARE_H` has *not* been defined yet, so the code from the #ifndef to the subsequent #endif is included for compilation. This code defines SQUARE_H, and has the definition for the `getSquareSides()` function.

Later, the next `#ifndef SQUARE_H` is evaluated. This time, `SQUARE_H` is defined (because it got defined above), so the code from the #ifndef to the subsequent #endif is excluded from compilation.

That's really it. Header guards prevent double inclusion because the first time a guard is encountered, the guard macro isn't defined, so the guarded content is included. Past that point, the guard macro is defined, so any subsequent copies of the guarded content is excluded.

Shah

Reply to Alex [12] September 25, 2022 5:44 pm

You are an absolutely amazing person. Thanks a billion.

0 Reply

MS Shohan

September 13, 2022 3:12 am

Hi,

You said the #pragma directive is not an official part of C++. I think that may be wrong. I checked the iostream header file and found that the #pragma directive is used there. Not only "#pragma once" but there seems to be other #pragma directives like "#pragma warning", "#pragma push". Not sure what they are called nor what they do. It would be great if you could give an explanation. Here is the screenshot: https://prnt.sc/xXx_jpPMrTRY

Thank you for the great tutorial.

0 Reply

Alex Author

Reply to MS Shohan [13] September 15, 2022 3:11 pm

An implementation of the C++ standard library for a given compiler is free to use whatever compiler-specific extensions are available. That doesn't mean those extensions are an official part of C++.

0 Reply

Abdullah

September 13, 2022 2:56 am

I understand how **#pragma once** can be considered non-standardized but don't you think that majority of the compilers can use it and **include guards** mostly for legacy code? Most folks I think can do well with **#pagma once** in comparison to **include guards**.

1 Reply

Kurt

September 4, 2022 8:19 pm

so till to this day do you think header guards is still better to use than #pragma once?

1 Reply

Alex Author

Reply to Kurt [14] September 13, 2022 3:14 pm

I do, but I don't feel that strongly about it. Some people feel you should use both.

I'm looking forward to the day that modules become usable, at which point the question will become moot.

1 Reply

Nurly

September 3, 2022 11:04 am

I finally understood this lesson!

0 Reply

starriet

July 20, 2022 12:11 am

In the main.cpp code in the section "Updating our previous example with header guards", the lines 16-21 are **not** copied into main.cpp, right?

I think it would be better to make those lines be commented out so learners like me don't get confused :)

(this code snippet)

```
1  #define SQUARE_H // so none of this content gets included
2
3  int getSquareSides()
4  {
5      return 4;
6  }
```

0 Reply

starriet

Reply to starriet [15] July 20, 2022 12:34 am

(just FYI for learners including me)
Oh, and after the preprocessing, when the compiler sees the result, all the preprocessor directives such as `#ifndef`, `#define`, or `#endif` already have been removed, and only the normal source codes remain.

So, only the below codes remain when the compiler reads main.cpp:

```
int getSquareSides()
{
    return 4;
}

int main()
{
    return 0;
}
```

Last edited 3 months ago by starriet

0 Reply

chris
July 5, 2022 4:26 am

>These user-defined types are typically defined in header files, so the type definitions can be propagated out to the code files that need to use them. Without a header guard, a code file could end up with multiple (identical) copies of a given type definition, which the compiler will flag as an error.

Isn't that a problem? The type definitions are propagated out to the code files that need to use them and we end up with multiple definitions just as in the example where both main.cpp and square.cpp included the definition of getSquareSides.

Is it allowed for user-defined types to have one definition per file whereas functions only one definition per program?
I think that would make sense.

Did I ask this before? I am not sure.

0 Reply

Alex Author

Reply to chris [16] July 11, 2022 9:32 pm

Types can have one definition per file. Functions with external linkage (which is what functions default to) can have one definition per program.

Note that functions with internal linkage can have one definition per file since those definitions aren't seen by the linker.

1 Reply

chris

Reply to Alex [17] July 14, 2022 3:44 am

I find this chapter quite confusing even though I read it a few times by now and I may still come back for a few more...

Anyway, I think I understand what you said. So, these types that are defined in header files, do they have external linkage or internal linkage? I would have guessed external because they can be used by other files but linkage is another concept that I find a bit confusing.

0 Reply

Alex Author

Reply to chris [18] July 14, 2022 9:52 pm

I believe class type names have external linkage by default, but I've never had to use this knowledge anywhere.

1 Reply

chris

Reply to Alex [19] July 15, 2022 10:33 am

my best guess would be that it doesn't matter because these types can have multiple definitions in different files and their definition is essentially examined on a per file bases, one per file.
I can't be sure but that's what I see

0 Reply

Tuan

April 23, 2022 8:15 am

hello Alex!
please help me!

I use Visual Studio to code game projects using SDL.
The problem I have when I create 1 const.h header file:

```
#ifndef const_H
#define const_H
const char* title = "Flappy Bird";
#endif
```

and the file func.cpp I include "const.h"
file main.cpp I include "const.h" and also include "func.cpp"

when i run it it says defined, while i used header-guards. What should I do, please help me
Thank you So much!

0 Reply

Alex Author

Reply to Tuan [20] April 23, 2022 2:35 pm

The solution to this is covered in lesson https://www.learncpp.com/cpp-tutorial/sharing-global-constants-across-multiple-files-using-inline-variables/

1 Reply

Fred

April 23, 2022 3:12 am

Hei sr. Is it okay to define macro inside header file

0 Reply

Landen

Reply to Fred [21] April 28, 2022 2:24 am

> Is it okay ?

Opinion: No. Take a look at this:

Just because you can (https://stackoverflow.com/questions/54884274/can-i-define-a-macro-in-a-header-file)[22] does not mean you should (https://google.github.io/styleguide/cppguide.html#Preprocessor_Macros)[23].

0 Reply

Fred

Reply to Landen [24] May 8, 2022 5:19 am

Thanks for your opinion sr :>

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/forward-declarations/
3. javascript:void(0)
4. https://www.learncpp.com/cpp-tutorial/how-to-design-your-first-programs/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/header-files/
7. https://www.learncpp.com/header-guards/
8. https://www.learncpp.com/cpp-tutorial/chapter-13-comprehensive-quiz/
9. https://www.learncpp.com/cpp-tutorial/scope-duration-and-linkage-summary/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-573414
12. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-573467
13. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-573052
14. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-572719
15. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-570821
16. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-570331
17. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-570486
18. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-570583
19. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-570632
20. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-567974
21. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-567971
22. https://stackoverflow.com/questions/54884274/can-i-define-a-macro-in-a-header-file
23. https://google.github.io/styleguide/cppguide.html#Preprocessor_Macros
24. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-568193
25. https://www.ezoic.com/what-is-ezoic/