2.12 — Header guards

BY ALEX ON APRIL 5TH, 2016 | LAST MODIFIED BY NASCARDRIVER ON DECEMBER 17TH, 2019

The duplicate definition problem

In lesson **2.7 -- Forward declarations and definitions**, 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've included 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 includes, this program looks like this:

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

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

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 definition can be propagated out to the code files that need to use them. Without a header guard, your code files can end up with multiple identical copies of these definitions, which will cause a duplicate definition compilation 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

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

```
#pragma once

// your code here
```

`#pragma once` serves the same purpose as header guards, and has the added benefit of being shorter and less error-prone.

However, `#pragma once` is not an official part of the C++ language, and not all compilers support it (although most modern compilers do).

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

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:

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

Show Solution

2.13 -- How to design your first programs

Index

2.11 -- Header files

C++ TUTORIAL | PRINT THIS POST

249 comments to 2.12 — Header guards

« Older Comments 1 2 3 4

Omran
January 29, 2020 at 6:53 am · Reply

here is my code , it's just an overview of what i learned so far :) , i just used char and if statement , float , i actually don't know a lot about them lol :D

MainSource.cpp

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

#define print(x) std::cout << x ;
#ifdef print(x)
```

```
float GetValueFromUser() {
    float input{ 0 };
    std::cin >> input;
    return input;
  }
#endif
void Operation(float x, float y, float z, float k, float o) {
    std::cout << x << " + " << y << " + " << z << " + " << k << " + " << o << " = " << add
(x, y, z, k , o);
}
int main() {
    print("hello user , i hope you are enjoying my little basic and small program here , l
et's begin our journey , :=)\n");
    print("enter a rational number :\n");
    float x = GetValueFromUser();
    print("enter another rational number :\n");
    float y = GetValueFromUser();
    print("enter one more :\n");
    float z = GetValueFromUser();
    print("okay i know this might sound crazy but enter one more please :\n");
    float k = GetValueFromUser();
    print("i bet you want to close this window but before you do that enter one more ratio
nal number please :\n ");
    float o = GetValueFromUser();
    print("enter a rational number : , ahahahahaha , i scammed you , here is your result ,
click the key = to get it :\n");
    char text;
    std::cin >> text;
    if (text == '=') {
    print("\n");
    print("it was nice playing with your anger haha :D \n");

    print('\n');
    Operation(x, y, z, k , o);
    print('\n');
    }
    else {
        print('\n');
        print("i said the key '=' \n");
        print("omg weren't you listening , by the way here is your result : \n");
        print('\n');
        Operation(x, y, z, k, o);
        print('\n');
        print("it was nice playing with your anger haha :D \n");

    }

    return 0;
}
```

PartnerSource.cpp

```
float add(float x, float y, float z, float k, float o) {
    return x + y + z + k + o;
}
```

Header.h

```
#pragma once
float add(float x, float y, float z, float k, float o);
```

Apaulture

January 16, 2020 at 12:27 pm · Reply

In main.cpp, square.h is #included and SQUARE_H is defined. Why would square.h be included in square.cpp when the function call from main.cpp to getSquarePerimeter(arg) doesn't take place until after getSquareSides() is called (at which point SQUARE_H is still defined)?

main.cpp

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

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;
}
```

square.h

```cpp
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

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

#endif
```

square.cpp

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

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

The only explanation I can think of is that the preprocessor resolves the directives in square.cpp and main.cpp simultaneously (not in order) before compiling.

nascardriver
January 17, 2020 at 1:23 am · Reply

Directives in one source file don't affect other source files. square.cpp doesn't know what happened in main.cpp and main.cpp doesn't know what happened in square.cpp.

David
January 12, 2020 at 1:43 am · Reply

Hello, thanks for these helpful lessons.

I have no idea why the following program can be compiled.
In my opinion ,in source.cpp , I have used the #define ADD_H,so the contents of "add.h" will not be imported because of #ifndef. That means ,in source.cpp, the compiler does not know what the add() is. However ,

compiler knows what the add() is.
Does my concept be wrong ? Thanks for replying!

add.h

```
#ifndef  ADD_H
#define ADD_H
int add(int x, int y);
#endif
```

add.cpp

```
#include"add.h"

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

source.cpp

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

using namespace std;

int main()
{
   int a = 5;
   int y = 6;
   cout<<add(a,y);

   return 0;
}
```

nascardriver
January 12, 2020 at 4:42 am · Reply

Please use code tags when posting code.

A define only affects the file it is in (Reminder: An #include copies a file's content to the including file), and is only visible _after_ its definition.
When you include "add.h" in "main.cpp", ADD_H is already defined (because "add.h" defines it and you included "add.h"). The #define in "main.cpp" has no effect.
If you defined ADD_H before you included "add.h", you wouldn't be able to use anything from "add.h", because "add.h" sees that ADD_H is already defined, and thus does nothing.
A #define in one .cpp file doesn't affect other .cpp files.

David
January 12, 2020 at 10:48 pm · Reply

I got it!
Also,could you tell me how to upload my program using code tags?
I have googled the "code tag",and it shows many methods,like Github Gist and Pastebin,but I do not know which one you used.

nascardriver

January 13, 2020 at 1:13 am · Reply

When you write a comment, there's a yellow message below the text field.
[-code]
// your code
[-/code]
without the -

```
// your code
```

David
January 13, 2020 at 3:02 am · Reply

Thank you! I know how to use it!

```
//Thanks
```

Giang from VN
December 19, 2019 at 6:42 am · Reply

Hello, thanks for these helpful lessons. I'm using DevC (cause my laptop can't install Visual Studio, and I've not tried to installed Code::Block yet) and sometimes I get an error "Id returned 1 exit status" and don't know how to fix it. In this Header Guards lesson, I tried some code like below:

getNum.h

```
#ifndef GET_NUM_H
#define GET_NUM_H

int getNum();
void print();

#endif
```

getNum.cpp

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

int getNum(){
std::cout<<"Enter a number: ";
int input;
std::cin>>input;
return input;
}

void print(){
std::cout<<"Done!!!\n";
}
```

main.cpp

```
 #include <iostream>
#include "getNum.h"
int main(){
getNum();
print();
return 0;
}
```

While compile "getNum.cpp", I got "undefined reference to 'WinMain@16'" and "Id returned 1 exit status" errors. While compile "main.cpp", I just got "Id returned 1 exit status", and I think the reason is the errors in "getNum.cpp" but I don't know how to fix this. Can anyone help me please

nascardriver
December 19, 2019 at 6:48 am · Reply

You're trying to compile "getNum.cpp" separately from "main.cpp". Since "getNum.cpp" doesn't have a `main` function, you get a linker error.
You need to compile all source files at the same time. I can't tell you how to do that in DevC++. DevC++ is outdated anyway, use a newer IDE and compiler (eg. Code::Blocks, so you can follow the tutorials on setting it up).

Chayim
December 17, 2019 at 10:27 pm · Reply

When I click on -NEXT- on the bottom it goes to page s.4.3a, and not to 2.13
https://www.learncpp.com/cpp-tutorial/4-3a-scope-duration-and-linkage-summary/

nascardriver
December 18, 2019 at 5:20 am · Reply

I can't reproduce it, is this still an issue for you?

Chayim
December 18, 2019 at 5:28 am · Reply

It’s not a problem because I usually use the green arrows that are always correct.

nascardriver
December 18, 2019 at 5:39 am · Reply

I though you were talking about the green arrow. Where's that -NEXT- you're talking about? I can't find it.

Chayim
December 18, 2019 at 5:46 am · Reply

It’s only on mobile, a NEXT button and a PREVIOUS button, at the end of the lesson after the arrows above the comments.

Also that on mobile the ability to add comments is above all comments, and on pc it’s below all comments on the bottom of the page.

nascardriver
December 18, 2019 at 5:52 am · Reply

Thanks for letting us know. We'll solve that :)

Chayim
December 17, 2019 at 5:24 am · Reply

In topic “Header Guards”:
-When this header is #included, the preprocessor check-
-check- missing -s- supposed to be -checks-

nascardriver
December 17, 2019 at 5:56 am · Reply

Lesson updated, thanks!

ntdong
December 16, 2019 at 12:49 am · Reply

In this lesson, the file square.cpp has bought the header "square.h". I think it does not need because function definition of both functions ware declared in "square.h"

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

Therefore, the square.cpp does not need header "square.h". I run these codes in Codebock. It is ok.

```
int getSquareside()
{
    return 4;
}
int getSquare(int length)
{
    return length*getSquareside();
}
```

nascardriver
December 16, 2019 at 4:11 am · Reply

Including it anyway can help detecting errors early.

Chayim
December 16, 2019 at 12:24 am · Reply

Even after this solution “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:”
This does not compile, receiving this error:

```
./Playground/file0.cpp:13:10: fatal error: square.h: No such file or directory
  13 | #include "square.h"
     |          ^~~~~~~~~~
compilation terminated.
```

nascardriver
December 16, 2019 at 4:10 am · Reply

You don't have a square.h file or you put it in the wrong directory.

Chayim
December 15, 2019 at 10:40 pm · Reply

In topic: "Header guards do not prevent a header from being included once into different code files"
How can it compile if “int main( )” is empty and has only “return 0” ?

nascardriver
December 16, 2019 at 4:09 am · Reply

The program won't do anything, but it can still be compiled.

Chayim
December 15, 2019 at 7:38 pm · Reply

Should be an option to delete own comment or to edit it even after an hour.

l1b3rator
October 4, 2019 at 7:55 am · Reply

This is square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

#endif

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

This is geometry.h:

```
#ifndef GEOMETRY_H
#define GEOMETRY_H

#endif

#include "square.h"
```

This is Project1:

```
#include <iostream>
#include "square.h"
#include "geometry.h"

int main()
{

   return 0;
}
```

and the error that i get is:

```
Error	C2084	function 'int getSquareSides(void)' already has a body	Project1	c:\c++ programs\visual studio\project1\project1\square.h	7
```

It seems like square.h gets copied twice, once from #include"square.h" and again from #include"geometry.h". I have no idea why the header guard is not working.

Thanks for help.

nascardriver
October 4, 2019 at 7:56 am · Reply

```cpp
1  #ifndef SQUARE_H
2  #define SQUARE_H
3
4  int getSquareSides()
5  {
6      return 4;
7  }
8
9  // has to be at the end of the file
10 #endif
```

Anderson
November 27, 2019 at 11:14 am · Reply

Sorry nascardriver, I didn't quite understand lines 9 and line 10 of your code snippet above.

```cpp
// has to be at the end of the file
#endif
```

Your comment mentioned that the #endif directive has to be at the end of the file but Im not entirely sure
if by 'end of the file' you meant relative to the other lines of code or at the very last line(line 10) in this case.
Please get back to me with a reply, I'll be sure to bookmark this page and come back at a later time to acknowledge your kind response.

By the way, I love this tutorial series. I am usually not into reading tutorials myself but I've learned far more from this series than any other course I have ever taken before.
I appreciate it nascardriver and Alex (and others who may have contributed...). Cheerio.. :)

nascardriver
November 28, 2019 at 2:23 am · Reply

There should be nothing (apart from empty lines or comments) after the `#endif`.

Anderson
November 28, 2019 at 8:08 am · Reply

nascardriver, thank you for your reply.
I understand now.

Mena
September 10, 2019 at 8:35 pm · Reply

#pragma once Is pretty legal and much more useful and even faster. I look forward to see a detailed account on this important c++ feature in the next update of these great tutorials.

nascardriver
September 11, 2019 at 3:14 am · Reply

It's not standard C++ and probably won't be described in any more detail than it is now.
C++20 adds modules, which remove the need for include guards.

Alex
September 16, 2019 at 12:35 pm · Reply

It's still not recommended as a best practice:
https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#sf8-use-include-guards-for-all-h-files

Maverick9107
August 21, 2019 at 4:37 am · Reply

So if I am correct, the tutorial (in this chapter) doesn't explain how to resolve the issue of functions being defined more than once, if there is a need to define one in the header file, am I right ? Will the solution to this problem be explained in later chapters ?

nascardriver
August 21, 2019 at 5:37 am · Reply

You don't need to define functions in headers (Unless they're part of something that allows multiple definitions).
Nonetheless, there is a way to define functions in headers, and it's shown later.

Maverick9107
August 21, 2019 at 8:18 pm · Reply

Thank you for replying

« Older Comments 1 2 3 4