2.11 — Header guards

ALEX DECEMBER 26, 2020

The duplicate definition problem

In lesson 2.6 -- 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'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 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 "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 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

Next lesson
2.12 How to design your first programs

Back to table of contents

Previous lesson
2.10 Header files

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

DP N N FOUT

Newest