1.10a — Header guards

BY ALEX ON APRIL 5TH, 2016 | LAST MODIFIED BY ALEX ON FEBRUARY 15TH, 2018

The duplicate definition problem

In lesson **1.7 -- Forward declarations and definitions**, we noted that an identifier can only have one definition. 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 identifier 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()
{
    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 example:

math.h:

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

geometry.h:

```
#include "math.h"
```

main.cpp:

```
#include "math.h"
#include "geometry.h"

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

This seemingly innocent looking program won't compile! The root cause of the problem here is that math.h contains a definition. Here's what's actually happening. First, main.cpp #includes "math.h", which copies the definition for function getSquareSides into main.cpp. Then main.cpp #includes "geometry.h", which #includes "math.h" itself. This copies 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 math.h
{
    return 4;
}

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

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

Duplicate definitions and a compile error. Each file, individually, is fine. However, because main.cpp #includes two headers that #include the same definition, we've run into problems. If geometry.h needs getSquareSides(), and main.cpp needs both geometry.h and math.h, how would you resolve this issue?

Header guards

The good news is that this is actually easy to fix 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 definitions here

#endif
```

When this header is included, the first thing it does is check 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 we've included the header before, SOME_UNIQUE_NAME_HERE will already have been defined from the first time the contents of the header were included. Consequently, the entire header will be ignored.

All of your header files should have header guards on them. SOME_UNIQUE_NAME_HERE can be any name you want, but typically the name of the header file with a _H appended to it is used. For example, math.h would have the header guard:

math.h:

```
#ifndef MATH_H
#define MATH_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
```

Updating our previous example with header guards

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

math.h

```
#ifndef MATH_H
#define MATH_H

int getSquareSides()
{
    return 4;
}

#endif
```

geometry.h:

```
#ifndef GEOMETRY_H
#define GEOMETRY_H

#include "math.h"

#endif
```

main.cpp:

```
#include "math.h"
#include "geometry.h"

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

Now, when main.cpp #includes “math.h”, the preprocessor will see that MATH_H hasn’t been defined yet. The contents of math.h are copied into main.cpp, and MATH_H is defined. main.cpp then #includes “geometry.h”, which just #includes “math.h”. At this point, the preprocessor sees that MATH_H has previously been defined, and the contents between the header guards are skipped.

Can’t we just avoid definitions in header files?

We’ve generally told you not to include 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 desirable to put definitions in a header file -- for example, when it comes to user-defined types (such as structs and classes). We haven’t covered those topics yet, but we will. So even though it’s not strictly necessary to have header guards at this point, we’re establishing good habits now, so you don’t have to unlearn bad habits later.

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 different code files. This can 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" << std::endl;
    std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << std::endl;

    return 0;
}
```

Note that even though square.h has header guards, the contents of square.h are included once in square.cpp and once in 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!

There are multiple ways to work around this problem. One way is 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:

```
// It would be okay to #include square.h here if needed
// This program doesn't need to.

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

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

main.cpp:

```
#include <iostream>
```

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

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

    return 0;
}
```

Now function getSquareSides() has just one definition (in square.cpp), so the linker is happy. 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).

We'll explore other ways to solve this problem in future lessons.

#pragma once

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

```cpp
#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. The stdafx.h file that Visual Studio includes in projects that use precompiled headers makes use of this directive in place of header guards.

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 header guards.

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 don't cause the same kinds of problems 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 different project files. This is a good thing, because we often need to reference the contents of a given header from different project files.

Quiz

1) Using the math.h example above, add header guards to file geometry.h

Hide Solution

geometry.h:

```cpp
#ifndef GEOMETRY_H
#define GEOMETRY_H

#include "math.h"

#endif
```

1.10b -- How to design your first programs

Index