2.12 — Header guards

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).

Author’s note

In upcoming examples, we’ll define some functions inside header files. You generally shouldn’t do this.

We are doing so here because it is the most effective way to demonstrate some concepts using functionality we've already covered.

Consider the following academic example:

square.h:

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

wave.h:

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

main.cpp:

```
#include "square.h"
#include "wave.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 *wave.h*, which #includes *square.h* itself. This copies contents of *square.h* (including the definition for function *getSquareSides*) into *wave.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 wave.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 *wave.h* needs *getSquareSides()*, and *main.cpp* needs both *wave.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 *wave.h*.

square.h

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

wave.h:

```
#ifndef WAVE_H
#define WAVE_H

#include "square.h"

#endif
```

main.cpp:

```
#include "square.h"
#include "wave.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 WAVE_H // wave.h included from main.cpp
#define WAVE_H
#ifndef SQUARE_H // square.h included from wave.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 // WAVE_H

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

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.

Header guards prevent duplicate inclusions 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 are excluded.

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 custom 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` preprocessor directive:

```
#pragma once

// your code here
```

`#pragma once` serves the same purpose as header guards: to avoid a header file from being included multiple times. With traditional header guards, the developer is responsible for guarding the header (by using preprocessor directives `#ifndef`, `#define`, and `#endif`). With `#pragma once`, we're requesting that the compiler guard the header. How exactly it does this is an implementation-specific detail.

For advanced readers

There is one known case where `#pragma once` will typically fail. If a header file is copied so that it exists in multiple places on the file system, if somehow both copies of the header get included, header guards will successfully de-dupe the identical headers, but `#pragma once` won't (because the compiler won't realize they are actually identical content).

For most projects, `#pragma once` works fine, and many developers now prefer it because it is easier and less error-prone. Many IDEs will also auto-include `#pragma once` at the top of a new header file generated through the IDE.

Warning

The `#pragma` directive was designed for compiler implementers to use for whatever purposes they desire. As such, which pragmas are supported and what meaning those pragmas have is completely implementation-specific. With the exception of `#pragma once`, do not expect a pragma that works on one compiler to be supported by another.

Because `#pragma once` is not defined by the C++ standard, it is possible that some compilers may not implement it. For this reason, some development houses (such as Google) recommend using traditional header guards. In this tutorial series, we will favor header guards, as they are the most conventional way to guard headers. However, support for `#pragma once` is fairly ubiquitous at this point, and if you wish to use `#pragma once` instead, that is generally accepted in modern C++.

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.

Duplicate *declarations* are fine -- 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