**Header guards**

In C++, nothing prevents the programmer from including a header-file multiple times. This can cause a duplication of definitions, which is an error. Since it is difficult to ensure that a header-file is only included once, a common strategy is to make only the first include count. This can be done using an “include guard”, a small piece of preprocessor logic that looks like this:

Not only should you use a unique and predictable (internal) include guard but you should also consider using (external) include guards around each preprocessor include directive in header files.  
  
The following is a small example, both a.h and b.h include base.h but preprocessor will have information to not even visit base.h a second time. It makes little difference on a small project, but a large difference on big ones.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *// a.h*  *#ifndef INCLUDED\_A*  *#define INCLUDED\_A*  *#ifndef INCLUDED\_BASE*  *#include "base.h"*  *#endif*  *//code*  *#endif* |  |

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *// b.h*  *#ifndef INCLUDED\_B*  *#define INCLUDED\_B*  *#ifndef INCLUDED\_BASE*  *#include "base.h"*  *#endif*  *//code*  *#endif* |  |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | *// base.h*  *#ifndef INCLUDED\_BASE*  *#define INCLUDED\_BASE*  *//code*  *#endif* |

**The duplicate definition problem**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 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:

|  |  |
| --- | --- |
| 1 | #include "square.h" |

main.cpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #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 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 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #ifndef \_IOSTREAM\_  #define \_IOSTREAM\_    // content here    #endif |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #ifndef SQUARE\_H  #define SQUARE\_H    int getSquareSides()  {      return 4;  }    #endif |

geometry.h:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #ifndef GEOMETRY\_H  #define GEOMETRY\_H    #include "square.h"    #endif |

main.cpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #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.

Link: <https://www.learncpp.com/cpp-tutorial/header-guards/>

**#pragma once**

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

|  |  |
| --- | --- |
| 1  2  3 | #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.