**Header files**

Header files allow us to put declarations in one location and then import them wherever we need them. This can save a lot of typing in multi-file programs.

When you #include a file, the content of the included file is inserted at the point of inclusion. This provides a useful way to pull in declarations from another file.

When it comes to functions and variables, it’s worth keeping in mind that header files typically only contain function and variable declarations, not function and variable definitions (otherwise a violation of the one definition rule could result).

Header files only consist of two parts:

* A header guard
* The actual content of the header file, which should be the forward declarations for all of the identifiers we want other files to be able to see.

# #include <> and #include ""

When we use angled brackets, we’re telling the preprocessor that this is a header file we didn’t write ourselves. The compiler will search for the header only in the directories specified by the include directories. The include directories are configured as part of your project/IDE settings/compiler settings, and typically default to the directories containing the header files that come with your compiler and/or OS. The compiler will not search for the header file in your project’s source code directory.

When we use double-quotes, we’re telling the preprocessor that this is a header file that we wrote. The compiler will first search for the header file in the current directory. If it can’t find a matching header there, it will then search the include directories.

# Including header files from other directories

Bad practice - include a relative path to the header file you want to include

#include "headers/myHeader.h"

#include "../moreHeaders/myOtherHeader.h"

While this will compile (assuming the files exist in those relative directories), the downside of this approach is that it requires you to reflect your directory structure in your code. If you ever update your directory structure, your code won’t work anymore.

## For IDE(s)

A better method is to tell your compiler or IDE that you have a bunch of header files in some other location, so that it will look there when it can’t find them in the current directory. This can generally be done by setting an include path or search directory in your IDE project settings.

## For GCC/G++ users

Using g++, you can use the -I option to specify an alternate include directory.

g++ -o main -I/source/includes main.cpp

The nice thing about this approach is that if you ever change your directory structure, you only have to change a single compiler or IDE setting instead of every code file.

# Headers may include other headers

It’s common that a header file will need a declaration or definition that lives in a different header file. Because of this, header files will often #include other header files.

When your code file #includes the first header file, you’ll also get any other header files that the first header file includes (and any header files those include, and so on). These additional header files are sometimes called “transitive includes”, as they’re included implicitly rather than explicitly.

The content of these transitive includes are available for use in your code file. However, you should not rely on the content of headers that are included transitively. The implementation of header files may change over time, or be different across different systems. If that happens, your code may only compile on certain systems, or may compile now but not in the future. This is easily avoided by explicitly including all of the header files the content of your code file requires.

**Best practice**

*Each file should explicitly #include all of the header files it needs to compile. Do not rely on headers included transitively from other headers.*

# The #include order of header files

If your header files are written properly and #include everything they need, the order of inclusion shouldn’t matter.

Now consider the following scenario: let’s say header A needs declarations from header B, but forgets to include it. In our code file, if we include header B before header A, our code will still compile! This is because the compiler will compile all the declarations from B before it compiles the code from A that depends on those declarations.

However, if we include header A first, then the compiler will complain because the code from A will be compiled before the compiler has seen the declarations from B. This is actually preferable, because the error has been surfaced, and we can then fix it.

**Best practice**

To maximize the chance that missing includes will be flagged by compiler, order your #includes as follows:

* *The paired header file*
* *Other headers from your project*
* *3rd party library headers*
* *Standard library headers*
* *The headers for each grouping should be sorted alphabetically.*

# Header file best practices

Here are a few more recommendations for creating and using header files.

* Always include header guards (we’ll cover these next lesson).
* Do not define variables and functions in header files (global constants are an exception -- we’ll cover these later)
* Give your header files the same name as the source files they’re associated with (e.g. grades.h is paired with grades.cpp).
* Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
* Be mindful of which headers you need to explicitly include for the functionality that you are using in your code files
* Every header you write should compile on its own (it should #include every dependency it needs)
* Only #include what you need (don’t include everything just because you can).
* Do not #include .cpp files.

# Header guards

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

**CAUTION**

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

**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.

# #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.

# References

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

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

# END