C++ Header File Guidelines

David Kieras, EECS Dept., University of Michigan

Revised for EECS 381, April 2015

This document is similar to the corresponding document for C, but includes some material specific to C++.

What should be in the header files for a complex project? C and C++ programs normally take the form of a collection of separately compiled modules. Thanks to the separate compilation concept, as a big project is developed, an new executable can be built rapidly if only the changed modules need to be recompiled.

In C++, the contents of a module consist of structure type (struct) declarations, class declarations, global variables, and functions. The functions themselves are normally defined in a *source file* (a ".cpp" file). Each source (.cpp) file has a *header* file (a ".h" file) associated with it that provides the declarations needed by other modules to make use of this module. The idea is that other modules can access the functionality in module X simply by #including the "X.h" header file, and the linker will do the rest. The code in X.cpp needs to be compiled only the first time or if it is changed; the rest of the time, the linker will link X's code into the final executable without needing to recompile it, which enables the Unix make utility and IDEs to work very efficiently. Usually, the *main module* does not have a header file, since it normally uses functionality in the other modules, rather than providing functionality to them.

A well organized C++ program has a good choice of modules, and properly constructed header files that make it easy to understand and access the functionality in a module. They also help ensure that the program is using the same declarations and definitions of all of the program components. This is important because compilers and linkers need help in enforcing the One Definition Rule.

Furthermore, well-designed header files reduce the need to recompile the source files for components whenever changes to other components are made. The trick is reduce the amount of "coupling" between components by minimizing the number of header files that a module's header file itself #includes. On very large projects (where C++ is often used), minimizing coupling can make a huge difference in "build time" as well as simplifying the code organization and debugging.

The following guidelines summarize how to set up your header and source files for the greatest clarity and compilation convenience.

Guideline #1. Each module with its .h and .cpp file should correspond to a clear piece of functionality. Conceptually, a module is a group of declarations and functions can be developed and maintained separately from other modules, and perhaps even reused in entirely different projects. Don't force together into a module things that will be used or maintained separately, and don't separate things that will always be used and maintained together. The Standard Library modules <cmath> and <string> are good examples of clearly distinct modules.

Guideline #2. Always use "include guards" in a header file. The most compact form uses "ifndef". Choose a guard symbol based on the header file name, since these symbols are easy to think up and the header file names are almost always unique in a project. Follow the convention of making the symbol all-caps. For example "Geometry_base.h" would start with:

```
#ifndef GEOMETRY_BASE_H

#define GEOMETRY_BASE_H
```

and end with:

```
#endif
```

Guideline #3. All of the declarations needed to use a module must appear in its header file, and this file is always used to access the module. Thus #including the header file provides all the information necessary for code using the module to compile and link correctly; *the header file contains the public interface for the module*.

Furthermore, if module A needs to use module X's functionality, it should always #include "X.h", and never contain hard-coded declarations for structs, classes, globals, or functions that appear in module X. Why? If module X is changed, but you forget to change the hard-coded declarations in module A, module A could easily fail with subtle run-time errors that won't be detected by either the compiler or linker. This is a violation of the One-Definition Rule which C++ compilers and linkers can't detect. Always referring to a module through its header file ensures that only a single set of declarations needs to be maintained, and helps enforce the One-Definition Rule.

Guideline #4. No namespace using statements are allowed at the top level in a header file. See the Handout *Using "using"* for an explanation and guidelines. Usually you should be using fully qualified names in a header file, such as "std::ostream."

Guideline #5. The header file contains only declarations, templates, and inline function definitions, and is included by the .cpp file for the module. Put structure and class declarations, function prototypes, and global variable extern declarations, in the .h file; put the function definitions and global variable definitions and initializations in the .cpp file. The .cpp file for a module must include the .h file; the compiler will detect any discrepancies between the two, and thus help ensure consistency.

Note that for templates, unless you are using explicit instantiations (rare), the compiler must have the full definitions available in order to instantiate the template, and so all templated function definitions must appear in the header file. Similarly, the compiler must have the full definitions available for ordinary (nonmember) functions that need to be inlined, so the function definitions will also appear (declared inline) in the header file (this is unusual, and normally won't happen until late in project development during performance tuning).

Guideline #6. Set up global variables for a module with an *extern declaration* in the header file, and a *defining declaration* in the .cpp file. For global variables that will be known throughout the program, place an extern declaration in the .h file, as in:

```
extern int g_number_of_entities;
```

The other modules #include only the .h file. The .cpp file for the module must include this same .h file, and near the beginning of the file, a defining declaration should appear - this declaration both defines and initializes the global variables, as in:

```
int g_number_of_entities = 0;
```

Of course, some other value besides zero could be used as the initial value, and static/global variables are initialized to zero by default; but initializing explicitly to zero is customary because it marks this declaration as the *defining declaration*, meaning that this is the unique point of definition. Note that different C compilers and linkers will allow other ways of setting up global variables, but this is the accepted C++ method for defining global variables and it also works for C to ensure that the global variables obey the One Definition Rule.

Guideline #7. Keep a module's internal declarations out of the header file. Sometimes a module uses strictly internal components that are not supposed to be accessed by other modules, or are not needed by other modules. The header file is supposed to contain the public interface for the module, and everything else is supposed to be hidden from outside view and access. Thus, if you need class or struct declarations, global variables, templates, or functions that are used only in the code in the .cpp file, put their definitions or declarations at convenient points in the .cpp file and *do not mention them in the .h file*.

One common example is special-purpose function object class declarations for use with the Standard Library algorithms. Often these have absolutely no value outside the module, and so should not be simply tossed into the header file with other class declarations. It is better is to place their declarations in the .cpp file just ahead of the function that uses them.

Furthermore, declare these internal global variables and functions `static` in the .cpp file to give them internal linkage (or put them into the unnamed namespace). Constants declared as `const` variables with initialization automatically get internal linkage (even if they appear in a header file), so declaring these as `static` is redundant and should not be done.

This way, other modules do not (and can not) know about these declarations, globals, or functions that are internal to the module. The internal linkage will enable the linker to help you enforce your design decision.

Guideline #8. Put declarations in the narrowest scope possible in the header file. Double-check Guideline #5 and make sure that a declaration belongs in the header file rather than the .cpp file for a module. If it does belong in the header file, place the declaration in the private section of a class if possible, followed by the protected section, followed by the public section of a class. Do not make it top-level in the header file unless it really needs to be that way.

For example, a `typedef` or type alias declaration that is only used within a class's member functions should be declared in the private section of the class. If the client code needs to use the declaration, place the it in the public section of the class. Don't place it at the top-level of the header file unless both multiple classes and the client code needs to refer to it. A similar rule applies to enum types, functions, and function object classes such as ordering relations. These rarely need to be declared at the top level of a header file.

Guideline #9. Every header file A.h should #include every other header file that A.h requires to compile correctly, but no more. What is needed in A.h: If another class or structure type X is used as a member variable of a class or structure type A, then you must #include X.h in A.h so that the compiler knows how large the X member is. Similarly, if a class A inherits from class X which is declared in X.h, then you must #include X.h in A.h, so that the compiler knows the full contents of an A object. Do not include header files that only the .cpp file code needs. E.g. <cmath> or <algorithm> is usually needed only by the function definitions in the .cpp file - #include it in .cpp file, not in the .h file.

Guideline #10. If an incomplete declaration of a type X will do, use it instead of #including its header X.h. If another struct or class type X appears only as a pointer or reference type in the contents of a header file, or as a parameter type or returned value in a function *declaration*, then you should not #include X.h, but just place an incomplete declaration of X (also called a "forward" declaration) near the beginning of the header file, as in:

```
        class X;
```

See the handout *Incomplete Declarations* for more discussion of this powerful and valuable technique. Two important points:

- If your header file has a function definition that requires a complete declaration of X, you can almost always eliminate the need by moving that function *definition* to the .cpp file, so that the header file only has the function *declaration*. Now you can use only an incomplete declaration in the header file.
- The Standard library has a header of incomplete declarations that often suffices for the <iostream> library, named <iosfwd>. You should #include <iosfwd> whenever possible, because the <iostream> header file is extremely large (giant templates!), and unless you *must* have function definitions in the header file that use iostream operators, there is no reason to include <iostream>.

Guideline #11. The content of a header file should compile correctly by itself. A header file should explicitly #include or forward declare everything it needs. Failure to observe this rule can result in very puzzling errors when other header files or #includes in other files are changed. Check your headers by compiling (by itself) a `test.cpp` that contains nothing more than `#include "A.h"`. It should not produce any compilation errors.[1] If it does, then something has been left out - something else needs to be included or forward declared. Test all the headers in a project by starting at the bottom of the include hierarchy and work your way to the top. This will help to find and eliminate any accidental dependencies between header files.

Guideline #12. The A.cpp file should first #include its A.h file, and then any other headers required for its code. Always #include A.h first to avoid hiding anything it is missing that gets included by other .h files. Then, if A's implementation code uses X, explicitly #include X.h in A.cpp, so that A.cpp is not dependent on X.h accidentally being #included somewhere else.

There is no clear consensus on whether A.cpp should also #include header files that *A.h has already included*. Two suggestions:

- If the X.h file is a logically unavoidable requirement for the declaration in A.h to compile, then #including it in A.cpp is redundant, since it is guaranteed to be included by A.h. So it is OK to *not* #include X.h in A.cpp, and will save some compiler time (the compiler won't have to find and open the .h file twice).
- Always #including X.h in A.cpp is a way of making it clear to the reader that we are using X, and helps make sure that X's declarations are available even if the contents of A.h changes due to the design changes. E.g. maybe we had a Thing member of a class at first, then changed it to a Thing *, but still used members of Things in the implementation code. The #include of Thing.h saves us a compile failure. So it is OK to *redundantly* #include X.h in A.cpp. Of course, if X becomes completely unnecessary, all of the #includes of X.h should be removed.

Guideline #13. Explicitly #include the headers for all Standard Library facilities used in a .cpp file. The Standard does not say which Standard Library header files must be included by which other Standard Library headers, so if you leave one out, the code may compile successfully in one implementation, but then fail in another. This is a gap in the Standard, justified weakly as giving imple-

[1] If using gcc, take care not to compile the header file itself, which results in creating a *precompiled header* file. This can save compilation time, but has to be managed correctly to avoid confusing incorrect builds. If this happens, simply delete the precompiled header file and rebuild.

menters more freedom to optimize. Avoid future compile failures by always #including <algorithm>, <functional>, <iomanip>, etc. whenever your code uses these components of the Standard Library.

Guideline #14. Never #include a .cpp file for any reason! This happens occasionally and it is always a mess. Why does it happen? Sometimes you need to bring in a bunch of code that really should to be shared between .cpp files for ease of maintenance, so you put it in a file by itself. Because the code does not consist of "normal" declarations or definitions, you know that putting it in a .h file is misleading, so you are tempted to call it a ".cpp" file instead, and then write #include "stuff.cpp".

But this causes instant confusion for other programmers and interferes with convenience in using IDEs, because .cpp files are normally separately compiled, so you have to somehow tell people not to compile this one .cpp file out of all the others. Furthermore, if they miss this hard-to-document point, they get really confused because compiling this sort of odd file typically produces a million error messages, making people think something mysterious is fundamentally wrong with your code or how they installed it. Conclusion: If it can't be treated like a normal header or source file, don't name it like one!

If you think you need to do something like this, first make sure that there isn't a more normal way to share the code (such as simply creating another module). If not, then name the special #include file with a different extension like ".inc" or ".inl".