C Header File Guidelines

David Kieras, EECS Dept., University of Michigan

December 19, 2012

This document is similar to the corresponding document for C++ but refers only 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, global variables, and functions. The functions themselves are normally defined in a *source file* (a ".c" file). Except for the main module, each source (.c) 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 `#include "X.h"` for the header file, and the linker will do the rest. The code in X.c 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.

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, minimizing coupling can make a huge difference in "build time" as well as simplifying the code organization and debugging.

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

Rule #1. Each module with its .h and .c 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 math.h and string.h are good examples of clearly distinct modules.

Rule #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
```

Note: Do not start the guard symbol with an underscore! Leading underscore names are reserved for internal use by the C implementation – the preprocessor, compiler, and Standard Library – breaking this rule can cause unnecessary and very puzzling errors. The complete rule for leading underscores is rather complex; but if you follow this simple form you'll stay out of trouble.

Rule #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. Furthermore, if module A needs to use module X's functionality, it should always `#include "X.h"`, and never contain hard-coded declarations for structure 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.

Rule #4. The header file contains only declarations, and is included by the .c file for the module. Put only structure type declarations, function prototypes, and global variable extern declarations, in the .h file; put the function definitions and global variable definitions and initializations in the .c file. The .c file for a module must include the .h file; the compiler can detect discrepancies between the two, and thus help ensure consistency.

Rule #5. Set up program-wide global variables with an *extern declaration* in the header file, and a *defining declaration* in the .c 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 .c 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 works for C as well to ensure that the global variables obey the One Definition Rule.

Rule #6. 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. If you need structure declarations, global variables, or functions that are used only in the code in the .c file, put their definitions or declarations near the top of the .c file and *do not mention them in the .h file*. Furthermore, declare globals and functions `static` in the .c file to give them internal linkage.

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 resulting from the `static` declaration will enable the linker to help you enforce your design decision.

Rule #7. 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 structure type X is used as a member variable of a structure type A, then you must `#include` X.h in A.h so that the compiler knows how large the X member is. Do not include header files that only the .c file code needs. E.g. `<math.h>` is usually needed only by the function definitions – `#include` it in .c file, not in the .h file.

Rule #8. If an incomplete declaration of a structure type X will do, use it instead of #including its header X.h. If a struct type X appears only as a pointer type in a structure declaration or its functions, and the code in the header file does not attempt to access any member variables of X, then you should not `#include` X.h, but instead make an *incomplete declaration* of X (also called a "forward" declaration) before the first use of X. Here is an example in which a structure type Thing refers to X by a pointer:

```
struct X;   /* incomplete ("forward") declaration */

struct Thing {
   int i;
   struct X* x_ptr;
   };
```

The compiler will be happy to accept code containing pointers to an incompletely known structure type, basically because pointers always have the same size and characteristics regardless of what they are pointing to. Typically, only the code in the .c file needs to access the members (or size) of X, so the .c file will `#include "X.h"`. This is a powerful technique for encapsulating a module and decoupling it from other modules.

Rule #9. 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.c that contains nothing more than `#include "A.h"`. It should not produce any compilation errors. 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.

Rule #10. The A.c 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.c, so that A.c is not dependent on X.h accidentally being `#include`d somewhere else.

There is no clear consensus on whether A.c 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.c is redundant, since it is guaranteed to be included by A.h. So it is OK to *not* `#include` X.h in A.c.
- Always #including X.h in A.c 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 struct Thing member of a struct at first, then got rid of it, but still used 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.c. Of course, if X becomes completely unnecessary, all of the `#include`s of X.h should be removed.

Rule #11. Never `#include` a .c 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 .c 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 ".c" file instead, and then write `#include "stuff.c"`.

But this causes instant confusion for other programmers and interferes with convenience in using IDEs, because .c files are normally separately compiled, so you have to somehow tell people not to compile this one .c 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".