More Include file rules

Reprise

Last issue I presented 15 rule-of thumb which I keep in mind when working with include files. To be honest, I don't think of these as rule when I code, I just keep to them, only documenting them for Overload has forced me to codify them as 15 rules.

And I'm very keen that people don't follow these rules slavishly. That's my big problem with coding standards, I don't want to see rules people follow without thinking, I want people to understand what they do. Sometimes, it's important to break from the rules. That's why I was very happy the Kevlin's piece appeared in the same issue. Hopefully, between us we've got people thinking about include files.

To complete what I started, I'd like to present a few more rules and discuss some common problems and debugging techniques with include files.

Clarification

Before I get started I must apologise for a potential error in my last article. In rules 2-3 I said that only if a class was declared as a reference or pointer could it's include be replaced with a forward declaration. While reading Herb Sutter's *Exception C++* I came across his remarks on include file dependencies and he correctly points out, that as long as you are only mentioning a class name and not actually using it you may replace an include with a forward declaration.

To clarify my point, suppose we have class Widget and class Thingy, if Thingy contains a Widget, or derives from a Widget it must include the full declaration of Widget. However, if Thingy only mentions Widget in a method (e.g. Action(Widget), Action(const Widget&) or Action(Widget*)), or contains a reference or pointer to Widget, then, it does not need to include Widget's declaration.

What I was trying to say was: if Thingy must access a Widget, prefer a pointer or reference to a Widget rather than containing one. While this is a valid enough rule in isolation, in practice whether Thingy uses reference or value semantics for it's Widgets is probably a larger question.

The rules

Where are the files?

Rule 16 : Avoid paths in #include

File locations vary platform to platform: is socket.h in include/ or include/sys/ ? As a general rule it is wise to avoid include paths inside the source code. By placing part of the include path in the source code you are tying the physical locations of source files into the logical source code. If you choose to change your directory structure at a later date you will need to change your source file, which impacts reusablability: not only must the re-user have a copy of your file but they must follow your directory structure.

Any time you reference the directory layout of the system you are coupling your code to the physical directory structure, this is especially true when the parent directory is referenced: e.g. code such as #include "../../utils/helper.h" imposes a lot of directory structure on a project.

Include paths specified outside of the code, through configuration setting, command line options or environment variables break the dependency between file location and file content - it is sensible to use them. On occasions it is necessary to specify part of an include path (e.g. socket.h) but this shouldn't be more than two levels and certainly shouldn't reference the parent directory. (Microsoft Visual C++ 4.0 contained a bug which caused it to corrupt make-files which included .. paths.)

Rule 17 : Break rule 16 when using namespaces

Having said all the above I've been thinking about Kevlin's suggestion that namespaces be placed in a sub-directory and I think he's right. Classically, a project would place most of it's headers in a few directories: the system headers, the compiler's headers, third party libraries and one application include. Inevitable the application directory becomes crowded. Now that namespaces provide a way of subdividing the application modules I think echoing these modules in subdirectories is a good idea.

When a system is divided into modules, e.g. ProjectX is divided into gui and xml modules, it makes sense to sub-divide the interfaces (header files) into their own spaces, ProjectX/include/gui and ProjectX/include/xml rather than dump them all in one directory such as ProjectX/include.

Once we have done this we have a choice of either

1. adding all the subdirectories to our include path, hence making it long and complicated, or,
2. placing ProjectX/include in the path and prefixing the filename with the module name, e.g. #inlcude <xml/somefile.h>

Using option 1, if we where to add a new module we would need to change the include path and perhaps effect the entire system. Option 2 follows the both the "least modification" and "only add code" rules, because no global change would be needed and only the files which used the new module would be effected.

Rule 18 : Know the difference between local & remote includes

The #include is really two directives:: #include "foo.h" and #include <bar.h> : local and remote includes. Traditionally, local meant: search for includes in the same directory as the current compilation unit, i.e. "local is here"; remote meant: search for includes from some include-path – but all this is implementation defined (see K&R or Hatton.)

This could cause problems where the *current compilation unit* (say bar.cpp) included another file remotely (say <bar.h>) which itself, included a file locally (say,. "foo.h"). Two problems would occur:

- Where foo.h exists in the directory containing bar.h but not the one containing bar.cpp : developers can see the file, but the compiler can't! Hours of frustration.
- Two different foo.h's existed: one in the same directory as bar.cpp and one in the same directory as foo.h; naturally, one *works* while one doesn't. Again, hours of frustration!

To complicate things futher, not only is this compiler dependent, but all compilers allow multiple ways of setting include paths, e.g. Microsoft allows “additional include directories” as part of the pre-processor, set for the project (stored in the .DSP file) while “include directories” is set in options/tools and is the same for all project on this machine whether you are compiling the Nuclear Reactor control project, or you Lottery Random number generator.

Some things, like stdio.h are part of the compiler environment regardless of project, some things are always part of the project and some things depend: imagine two versions of the STL on your system: a small fast version (perfect for choosing lottery numbers) and a slow but bug-free version (perfect for fault-tolerant systems).

At some point in your system, the logic in the file must meet the physical file system and directory layout (a subject worthy of an article in it’s own right.) I’ve always viewed the glue that bound the two together to be: the make-file. The make-file knows how files fit together with each other and the standard gubbins.

My view on include files was very clear:

Rule 19 : only use “local include” where the file is local to the includer, chances are it will never been seen by anything else,

Rule 20 : use <remote include> for everything else, let the make-file ensure the right thing is seen.

In this world, the Microsoft “additional includes” are part of the make (DSP) file and hence make sense, but they use the “local” mechanism.

I can’t give you a solid rule here: in my mind the jury is still out. Given a blank sheet of paper I

- use “local” includes for files which are in the same directory and only visible within this module,
- use <remote> includes for files which form part of the interface of this or another module, and are contained in a separate directory
- use the environment include paths for system include files, e.g. stdio.h : if I’m using a standard library supplied with the compiler I’d list them here
- use project include paths for application files and third party files including a third party standard library
- pay special attention to the standard library include paths: I don’t want Plauger string and Rogue Wave’s list.

Common problems and debugging techniques

Porting

As mentioned above, the search for header files is implementation dependent. For example, looking for #include “fred.h” Microsoft C++ (5.0) searches:

1. Directory containing the source file
2. Directory containing the file that included the source file, and subsequent searches all the way up the include tree

These two rules are skipped for #include <fred.h>, rules 3 and 4 apply to both types of include:

3. Any directory specified with /I on the command line

4. Any directory specified with the INCLUDE environment variable : this corresponds to the options directory list in Developer Studio.

While Sun's C++ (4.2) compiler searches:

1. The directory containing the source file if the include is "local"

Rule 1 doesn't apply for <includes> but the following rules are common:

2. Any directory listed with the –I option on the command line
3. Standard directory for C++ header files; on my 4.2 compiler this is: /opt/SUNWspro/SC4.2/include/CC
4. Standard directory for C header files; again, on my system this is: /opt/SUNWspro/SC4.2/include/cc
5. In /usr/include

Obviously very different from Microsoft, however, GNU C++ (actually egcs 1.1.2 in this case) complicates things further with a multitude of extra options, e.g. two search path are allowed. Generally, GCC seems closer to Sun but the hard coded paths can be varied at compile time so the same version may actually search in different places.

While I always knew differences existed only when researching this article did I realise different they can be. Microsoft's upward-recursion algorithm seems particularly dangerous.

Microsoft's compiler offers nothing to help debug include problems, Sun provides an –H option which will produce a list of include files as they are included making it possible to follow the logic. If memory server GCC/egcs lists which files included which file when some compilation errors occur.

But it is included!

One of the more infuriating problems is when you can see something is included but the compiler won't recognise it. For example:

```
// file: fred.h
class Fred { ... }


// file: Flintstones.cpp
#include <willma.h>
#include <fred.h>
Fred fred;       // compiler errors : undeclared identifier
```

The most common reasons for this are:

- The pre-processor actually found a different file called fred.h which doesn't define Fred : e.g. you are using Microsoft pre-compiled headers and the Visual C++ Wizards, you will have multiple copies of stdafx.h around, something was added to one but it's not the one included here
- Macro-guards mixed up: typical when cut-and-paste programming is used, e.g. you wrote willma.h and knowing that fred.h was going to be very similar you pasted the code from willma.h into a blank file called fred.h and just made the changes. However, if you forgot to change the #if defined WILMA_H at the start of the file fred.h and hence the pre-processor ignores the contents of fred.h.

The quickest way to find out what is happening to your file is to add a message which identifies it at compile time, e.g.

```
#pragma message("This is fred.h")
```

These can be sprinked around liberally, but remember you will want to remove them before check in. The best places to put these are:

- Inside the file with the problem: before actually including fred.h, e.g. message("About to include fred.h.....") or immediately after including fred.h, e.g. message("..... fred.h included")
- Inside the offending file; before the include guard ("This is fred.h") and inside the include guard ("Inside fred.h include guard").

If you place a message inside the include guard remember that you should only see it once for each compilation unit, e.g. each .cpp file, but you will most likely see it multiple times during a complete build because multiple compilation units will probably include the file.

In practice the pragma message is the most powerful tool for debugging header file issues for most problems. If your compiler doesn't support pragma message, try #error. Or, add an actual syntax error to the file, these are seldom fatal to the compile but will show you what is happening

Variations on the "it is included" theme are:

- The file has changed: one of your fellow programmers has revised Fred.h to declare class Fred_Impl instead of Fred.
- Namespaces: these give us a whole new way to hide identifiers.

Cannot open include file: 'barny.h': No such file or directory

Failure to actually find a file is perhaps the most common error. However, it usually has but one course: include paths are wrong. Sometimes only leg work will pay here. Check each any every path individually:

- If your using Microsoft DSP files, the Settings/Additional include paths dialog box is simply too small to be sure you are not seeing an 1 instead of an l, or a 0 instead of an O. Copy the text and paste it into a bigger window, break the line up and check you can dir each directory at the command line.
- Check your debug and release settings are the same, are you are building the one you think your building? In the heat of battle it's not known to be looking at the Debug settings when you actually have an error in the Release build
- If your using makefile these can get quite complex with files including files which include files which all add to compiler and linker options. Look at the command line make is issuing and check the paths.
- If you rely on an INCLUDE environment variable check the shell your executing in actually has it set to what you think it is. If you've su'ed to a different user to build it may of been reset.

- Is your compiler looking for headers in the current directory? (i.e. where the makefile file is) or is it looking in the directory containing the directory containing the source file? Or for that matter, are you looking in the current directory when you should be looking in the source directory?

Sometimes there are other courses to this error:

- Your system has run out of file handles : unlikely for NT but some people still develop for MSDOS and even UNIX have a limit.
- Access writes deny access to the file: can you cat the file at the command line? If you get an access denied the compiler will too.

type redefinition

One common error is the type redefinition. This happens when you declare something, e.g. class Table, and something you have previously included declares typedef .... Table. Hopefully my rules on include file order should help avoid this. However, this is not always possible.

Developers don't often make this mistake in their own code, but third party code, be it OS headers, bought in libraries, or in house libraries usually contain the other copy. (Hopefully, with namespaces this kind of thing will become less frequent.)

If your lucky you can place your cursor on the offening definition, hit F1 and the on-line help will tell you about the OS type of the name – at which point you must give in gracefully and rename your. Unfortunately, much that is defined in OS headers is not documented, at which point grepping the code may help.

A quicker way to find the first declaration is too cheat. Add a quick declaration before all your includes, something obscure like: typedef short <Offending identifier>, and then recompile. Normally this will give you the same error, but this time on the first declaration. Once you've removed your extra declaration you will know which items conflict and resolve the conflict.

A particularly nasty form of this involves macros with the same name as types or members, e.g. Sun defines a macro called minor in sysmacros.h, once this file has been included somewhere you will have problems compiling code with CORBA exceptions as these define a member called minor.

unexpected end of file found : and other very strange errors

As I said right at the start of these articles, including a header file is very primitive. If you write:

```
// file: fred.h
class Fred {
     ....
// <eof>

// file: flintstones.h
#include <fred.h>
.....
// <eof>
```

You will at least get an unexpected end of file error, and possibly a lot of other strange syntax errors. It's obvious here but it is easy to miss because the file it shows up in, in this case, flintstones.h, is a long way from fred.h. Potentially, flintstones.h did not include fred.h directly, but included family.h, which included "characters.h".

Again cut and paste programming is a common cause of this error; and again, leg work, and a few pragma message's are the best attack.

One short cut which you can make it to comment out everything (#includes and source code) in the .cpp file and re-introduce the includes one at a time until the error appears. Check the file you just included, if your luck it's guilty, if not repeat the process with it's include's too.

Dependencies

The includes of the system map out the dependencies of the system, which files depend on which, which subsystem depends on which. This is one third of your physical design – another third is the library dependencies and the final third your directory structure and make-files (which are the most physical part of your system.)

When designing a system I find it useful to map out the dependencies before hand, find out which parts will depend on which, which are the low level parts of the system, which the high. and which are the same level but independent.

If your dependency strategy is clear, your include files will be clear and you should have no problem following rules such as "most inflexible first, most flexible last". If you find this isn't clear you have a potential problem.

Final words.....

I hope the fact that Kevlin and myself have managed to write around 10,000 words on the subject of include files convinces you that there is more here than meets the eye! The fact is, include files are the first point at which your logical design meets the physical build environment and therefore determines much of what is to come.

References

- Kevlin Henney : C++ Patterns, Source Cohesion and Decoupling, Overload 39, September 2000.
- Allan Kelly : Include files, Overload 39, September 2000.
- Herb Sutter : Exceptional C++, 2000, Addison Wesley
- Kernighan & Richie: The C Programming Language, 1988, Prentice Hall
- Les Hatton : Safer C, 1995, McGraw-Hill