Include files

Introduction

Although most of us work with the C/C++ #include mechanism every working day it is an area that has received surprisingly little coverage and analysis. To date I have regarded John Lakos's *Large Scale C++ Design* as the most authoritative work on the subject. Kevlin Henney's piece on *Source Cohesion and Decoupling* provides a solid basis for understanding the theory involved with include files.
For my part I'd like to share some of my rules-of-thumb. I've adopted these over the last few years, heavily influenced by Lakos, although this is my first attempt at codifying them as such. Having read Kevlin's piece I have a few more ideas to try out.
My emphasis tends to be on simplifying the work-a-day processes, improving re-usability and portability. For the most part I agree with Lakos and Henney – although not always!

Primitive mechanism

Perhaps one reason by #include has not received much analysis is because it is so primitive, other languages have similar constructs but few are as primitive. *#include "widget.h"* literally says "open the file widget.h and insert the code here as if it where part of this physical file." No syntax checking is provided, just about the only check that is provided is that the file widget.h exists, most compiler will get a little upset if it doesn't. It's often worth remembering just how primitive this mechanism is.
We could, for example, actually include the .cpp file into our file, this would remove the need to link in another object file - to the compilers its all, one big compilation unit. This technique can occasionally be useful but on the whole it's not advocated as a method of working. Instead, we use an include to simply declare prototypes (for functions), class declarations and other such non-procedural items. The actual source code for these is elsewhere and it is the linkers job to match it all up.
For all it's primitive-ness, #include is the basis of code reuse: at the simplest level, we have a set of common functions for which we provide prototypes in a .h file[1]. After including the file the functions are available for us. We may choose to compile the functions separately and package them as a library. Here we see another benefit of this approach: we are implementing abstraction, we can no longer see the source for a function only it's prototype. This breaks down very quickly, as often prototypes are not commented with the information we are looking for and we end up finding the source code and reading that.
At the next level, we implement class definitions in the header file and place the implementation in the hidden file. Now we are implementing data abstraction, or data hiding; although data hiding is one of the strong points of object oriented design, it's not the full story.

[1] Henney's advice on avoiding .h as an extension is new to me. He makes a good case which I look forward to trying out.

Rules of Inclusion

Basic rules

Rule 1 : Include everything you need in a file, no more, no less

When we re-use code at either the function, class or component level we are trying to save time and effort. If code is to be re-usable it must be self contained and presented in such a way that it can be used in our projects with minimal effort.

This rule means: include all the header files you need to ensure this header file will compile but don't include any more.

```
#include <string>
class Address;   // forward declaration
class Person {
     Person(const std::string& name, const Address& addr);
     ...
};
```

In this example, string is included because it is needed, if string is not declared before the compiler encounters this code it will error. Don't say "well the file that includes this file already includes string, so I don't need it here" : if you decide to reuse Person on the next project you don't have this guarantee, you'll either need to change this file, which you don't want to do because the code is shared between two projects now, hence you will have source control issues, or, the new file that includes this one will have to include string, which itself violates this rule.

The example does not include string_util.h, strstream.h, or anything else, it doesn't need them. If some other file in the system needs one of these the proper place to include it is in that file. Including stuff you don't need immediately in this file will (a) slow down the compile, (b) pollute the namespace and (c) introduce dependencies which may cause problems later.

This rule is basically a re-phrasing of Henney's *Minimal Header* and *Cohesive Header.*

Before moving on there is a school of thought which says: only include what you absolutely need, if someone else has already included it don't bother. This can reduce compile times, but it detracts from re-usability. I've heard, and seen, this approach taken but never seen a well argued case for it. The main case I can see is to reduce compile times.

Rule 2 : Use forward declarations wherever possible and practical

We could go further with the example above. Because string is not actually declared but only referenced, we could remove the include altogether and just present a forward declaration of string as I have done with Address.

If this where a Microsoft CString, I'd probably of done just that. But because string is in namespace std I'd need to open the namespace, this would probably double or even treble the number of lines concerned, then, because string is actually a typedef for basic_string, I'd have to work around this too, then basic_string is actually a template so more work. As the code stands it is obvious what is happening, if I where to add a full forward declaration of string the actual code would be obscured by

the declaration, hence I don't consider it practical. In addition, it's probably that any use of this class is going to include string anyway (I know, I said don't rely on this) so I'm more concerned with clarity of code.

However, I could provide a forward declaration in another file and include that – Henney's *Forward Declaring Header*. I'm happy to agree I should, I'm also happy not to get hung up on the issue. Balance the options, on a large project it is probably worth it, then once you have the forward reference file it there for small projects too. However, on a small project you have to balance this against the fact that you have added another file to the project which has to be managed and maintained, and potentially shared across projects.

Rule 3 : Use references wherever possible

This also leads to a more general point, because I can use forward declarations with pointers and references , but I not with actual instantiations of classes I prefer to use reference in my code where possible. This is also good advice because it cuts down on the number of copy constructors being called – although remember to accept parameters as const-references.

Rule 4 : Be consistent with your filename case – preferably always use lower-case

Is #include <Windows.h> the same as #include <windows.h>? On Microsoft systems, yes, on UNIX, no.

It is very easy on Windows systems to forget that filename case can be important. If your using Samba to share files things can get more complex as, depending on configuration, it can change the case of a filename. By far the simplest solution is to mandate the problem away. (I suggest lower-case over uppercase as most of standard header files are all lower case, and, I think it looks better.)

Sometimes it's beneficial to adopt conventions because, even if they aren't the best, it's what people expect. Going against a convention can cause a lot of convention – the rule of "least astonishment."

Rule 5 : Only have one copy of the file on your system

There should be no need in a build environment to have two copies of one file. This is unnecessary duplication. It also leads to the "Which file was changed?" scenario. You may think instance A is being included and change this one, but in reality it's instance B, so you keep getting the same error! (In truth, modern compilers and IDEs are better at describing exactly which file has changed so this problem arises less often than it did.)

This leads to.....

Rule 6 : File names should be unique

Even where files are separated in different sub-directories it makes sense to keep the actual file names unique. Now we are no longer restricted to 8.3 filenames there is little reason to re-use a name. Keeping filenames unique will not only rule out any chance that the compiler will include the wrong file but will also make your code more self documenting. After all, we are only human, if we see a file called StUtils.h we will first assume it is the same StUtils.h (for strings) that we have already seen, not some other StUtils.h (for streams) which is included because of magic in the make file.

(This is one of the reasons I don't like Microsoft's pre-compiled header system which places multiple files called stdafx.h all over my project.)

Rule 7 : only #include files at the top of your program

You can include files anywhere you like, remember, the compiler doesn't syntax check include files. However, with the exception of machine generated code, nobody expects to find:

```
// file: bar.cpp
vector<int>      wont_compile;
... loads of code ...
#include <vector>
... some more code ...
vector<char>     will_compile;
```

Include guards

The items listed here fall under Henney's *Indempotent Header* pattern.

Rule 8 : Guard the extremes of your file

Few C/C++ programmers don't place include guards around their header files:

```
#if !defined(_WIDGET_)
#define _WIDGET_
....
#endif
```

This is only sensible as we wish to avoid errors and warnings when a file is included for the second time. But even here there is room for differences. Some prefer the #if to come before the comments and some prefer it after the comments.

I don't see any point in placing the guards anywhere but on the first line and last line of the file. It may make little difference to the compiler, but the #if...endif has nothing to do with the source code in the file, the statements only exist because of deficiencies in C so keep them away from the guts.

Rule 9 : Think about include guards, especially for libraries

Lakos argues that we should actually place the check in the file which includes the header file, hence including the above would look like:

```
...
#if !defined(_WIDGET_)
#include <Widget.h>
#endif
```

Lakos makes a strong case for this. Simply put: even with guards within the file the compiler will still scan it looking for the end. This may be a trivial amount of time, but on a large project, with lots of files it mounts up and can effect compiler times.

Opposed to this, it makes the code more difficult to read and unusual. Both are legitimate points of view and I advise you to think about them.

(If your reaction to the above is to think "I could right a macro to remove the #if" think again, because this **is** the preprocessor you can't!)

On a library this advice goes twice, as you don't know how people will include your files you should cater for the worst case and, the potential speed improvement is multiplied across all projects which use the library.

Rule 10 : Keep internal guards

If you adopt the "external guard" rule you may also consider removing the internal guards. If you can guarantee all the files in your project will follow it you may be safe. But, there are two problems: new programmer will instinctively wonder where the include guards are and secondly your code will be less re-usable in systems which use "internal guards."

Rule 11 : Don't rely on *#pragma once* without thinking.

Microsoft provide the #pragma once to prevent header files being included more than once – I'm not aware of any other vendor who also support this, but that doesn't mean there aren't any. While this is a useful directive it has obvious cross platform implications and should not be used without consideration. If you do decide to us it you may wish to use regular include guards as well; while these would simplify porting you may still see warnings on other platforms.

Include ordering

Rule 12 : Organise the include order to recognise different type of files being included

The include statements are usually the first thing of significance we see in a file, in that way they help shape our expectations for what we are about to see – another good reason for including exactly what the files will need.

We could throw down our includes in what-ever random order we feel like, or the order we think we happen to need the files. I think there is good reason to always order the includes in a consistent fashion.

There are, to my mind, three classes of files we are including: system includes (those for the platform and standard libraries, e.g. stdio[.h], vector[.h], windows.h, socket.h, etc.); project specific files (usually libraries or objects that have been developed for this project, or are reusable within this organisation); and finally files specific to this file (e.g. the declaration of the class implemented in this file, or prototypes for functions implemented here.)

These categories are not water tight:

? "where does one include a third party library such as Rogue Wave?" : I normally put it in the second

? "where does one include the header file of the parent class I'm inheriting from?" : I normally put this in the third group (although it need only needs to be included in the header file declaring this class, because the implementation must include declaration header.)

Each of these three groups should be grouped together.

Rule 13 : Include system files, then project files and finally local includes

For example:

```
// file: Employee.cpp
```

```
// system includes ----------------------
#include <windows.h>
#include <vector>
// project includes --------------------
#include <string_utils.h>
#include <hr_dept_component.h>
#include <Address.h>
// local includes ----------------------
#include "Employee.h"
```

There are a few points that need to be observed here:

? These rules apply to both the .h files and the .cpp files: since the declarations (.h's) must be included in the implementations (.cpp's) for them to compile a group of includes are already given. These are normally the more visible dependencies, while the includes in the .cpp have only to do with implementation.

? It is quiet common for sections not to be used, if so comment then, e.g. // <none> this says a lot

Attentive readers will of noticed that I advocate the opposite of Lakos and Henney: both suggest placing the local or project includes first and the system includes later. I would argue that my ordering reduces work when a name clash is encountered. For example, suppose hr_dept_component.h defined a function to fire-people called FreeResource (sorry, I can't think of a better example at the moment!). This may work fine on the UNIX port but on the NT version it suddenly conflicts with a Microsoft function from Windows.h.

If we include hr_dept_component.h before Windows.h the compiler will generate an error on the line in Windows.h. Unfortunately, there is very little we can do: we can't change Windows.h, it works fine in every other program, it is us who have introduced the problem; the error tells us nothing about where the conflict was introduced in our own code.

However, if we include Windows.h before hr_dept_component.h the compiler will flag an error in hr_dept_component.h where we may be able to do something.

The general rule is to put the least changeable includes first: generally you do not want to change a system level file; project level files can be changed but we'd rather not (it may involve talking to another programmer!), but changing local includes, we can do that now.

Rule 14 : Place the most inflexible files first, the most flexible last

Even within these sub-groupings of system/project/include I tend to group the individual includes on a "least flexible first" strategy, although I don't get hung up on this unless it causes a problem: so for example, I would include a third party library above a locally developed one.

I find that the three groupings work well, however, you may wish to take it further to reflect the layering of your system more, or to fit IDL files in logically (I'd suggest between system and project files) although I don't think any purpose is served in codifying the rule to the n'th degree.

Rule 15 : Place any forward declarations after the local includes

For example:

```
// local includes ----------------------
#include "Employee.h"
// forward declarations ----------------
class Name;
```

Again, following the most easily changeable rule you can change these on the spot in this file – although this may have a ripple effect.

Break

These rules-of-thumb are the rules I apply everytime I code header files. Next month I hope to continue with a few more rules and say a few words about common problems with include files and debugging techniques specific to include problems.