Include files : an e-mail exchange with Herb Sutter

After the first article on Include Files was published in Overload 39 (September 2000) I received an e-mail Herb Sutter, author of *Exceptional C++*; I replied to his mail and the exchange was published in Overload 40 (December 2000). I include it here for completeness.

From: "Herb Sutter" <hsutter@peerdirect.com>
To: <allan.kelly@bigfoot.com>
Cc: "'Francis Glassborow'" <francis@robinton.demon.co.uk>
Subject: "Include Files" article (Overload 39, Sep00)
Date: Sunday, November 19, 2000 5:02 PM

Hi Allan,

I enjoyed your "Include Files" article in Overload issue 39 (September 2000), and would like to share some comments.

Re Rule 2: You argue that forward-declaring the standard "string" typedef yourself is impractical, but it's much stronger than that: It's illegal. The only thing you can do yourself in namespace std is declare specializations of standard templates; you can't declare standard names like basic_string.

Re Rule 3: I agree with the rule, but I don't think your rationale is quite right. You write, "I can use forward declarations with pointers and references, but I[sic] not with [objects]," which is correct if you're talking about function definitions but isn't correct if you're just talking about function declarations (which is what you seem to be talking about in the context of the other Rules). For example, most people are surprised that the following code is a complete header and does not require a definition for class X, because you only need the definition when you need to generate code that must know the size of X (e.g., allocate stack space for a local object or a function value parameter) or call a member function of X (e.g., copy it):

```
// ...usual include guards here and at the end...

void g( X&, double );
void h( X*, double );
```

```
class Y {
public:
  X& f( X& ); // doesn't need definition of X
  X* f( X* ); // ditto
  X  f( X  ); // ditto
};

inline X& Y::f( X& r ) {
  g( r, 1.0 ); // use the reference: still don't need definition of X
}

inline X* Y::f( X* p ) {
  h( p, 1.0 ); // use the pointer: still don't need definition of X
}

// can't define Y::f(X) without X's definition, though, because that
// would actually ask the compiler to generate code to call the
// copy constructor and know the object's size
```

See also Item 26 in Exceptional C++.

Re Rule 9: I don't understand this rule. When you say, "Think about include guards, especially for libraries," I can't see how this can apply to non-header files, yet Rule 8 just said that every header should have include guards (and that's right). What is Rule 9 intended to apply to?

Incidentally, I strongly disagree with Lakos' external include guards on two grounds:

1. There's no benefit on most compilers. I admit that I haven't done measurements, as Lakos seems to have done back then, but as far as I know today's compilers already have smarts to avoid the build time reread overhead -- even MSVC does this optimization (although it requires you to say "#pragma once"), and it's the weakest compiler in many ways.

2. External include guards violate encapsulation because they require many/all callers to know about the internals of the header -- in particular, the special #define name used as a guard. They're also

fragile -- what if you get the name wrong? what if the name changes?

Re Rule 10: No, never! Even if you disagree with me above and go ahead with external guard, your objections really do apply. If you forget (or mistype) the external guards in even one place...

Re Rules 12-14: Agreed. It's also worth noting that header files should always be written such that they do not depend on #include file ordering (e.g., being included before vs. after any other header files). This actually impacts namespaces and it's a reason I argue that using declarations should never appear anywhere in a header file, not even in a namespace (see my article "Migrating To Namespaces" in Dr Dobbs Journal, 25(10), October 2000).

These aren't intended to be criticisms, just feedback on a nice piece.
Thanks again for the enjoyable article! Best wishes,

Herb

And my response....

Herb,

First, thanks for taking the time to read my thoughts and pass on your comments. Interestingly, John Merrells has part two of the piece in the works for Overload 40, in this I added a paragraph as a result of reading *Exceptional C++* to clarify rules 2-3. Yes, I agree completely with your comments on my rule 3 completely - forward declarations can be used as long as no code is generated; my writing was a little confused.

In rule 2 (concerning forward declarations in separate files) I was trying to use the string type as an example of how forward declarations could get complex. Unfortunately I chose a bad example. In fact I didn't know re-opening std namespace was illegal, I should have done but I didn't. I wonder how many developers actually know this? Now you've brought it to my attention it's obvious. Perhaps more importantly: do any compilers enforce this rule? I suspect few if any. I suggest we shout it loud: "don't add anything to std namespace."

(This is kind of unfortunate, to my mind, one of the strongest advantages of a namespace over a class is that it can be re-opened and items added but many novices first encounter with namespaces will be with std, one or both points may well be lost on them.)

Rule 9 : Reading it again I see I should have phrased it as “Think about external include guards, especially within libraries.” My objective was to explore the Lakos argument about external include guards. Like yourself I have problems with Lakos argument, while he presents a convincing case for improving compile times increasingly I agree with you. In the long run I think external guards make the source code more difficult to read and increase the coupling between a header file and the file’s clients. Yet the benefits, if any, will on be apparent on very large projects.

I deliberately said “think about” because I didn’t want to give a definite rule, I wanted to make people aware of the argument, to this end your comments are an excellent contribution to the case.

Rule 10 : I never intended to advocate the removal of internal include guards. We agree here: always leave them in even if you have external guards!

I find it very interesting that you agree with my rules 12 to 14 “order includes with most inflexible first, most flexible last.” These rules where the genesis of the article, they have become a fixture of my coding style. However, they go directly against Lakos advice which states that you should order from local to system. (Not only this, Kevlin’s article in the same issue of Overload upholds the Lakos rule.)

I’m hope we have started a debate here.

I hope you’ll forgive this novice-writer a few slips and enjoy my continuation of the subject in Overload 40.

allan