NICK STOUGHTON

toward attributes

USENIX Standards Liaison

nick@usenix.org

BOTH THE C AND C++ STANDARDS ARE being revised at present, and one proposal the two revision projects have in common is to include syntax for attributes, a feature present as an extension in most modern C and C++ compilers.

Attributes allow the programmer to give additional hints to the compiler about how to generate code. They decorate variables, functions, and types. Both C and C++ have numerous places within their standards (and an enormous number, when one considers currently deployed applications) where attributes would help.

There are of course many ways to invent a syntax for a new language feature. One way is to invent new keywords in the language to represent the new feature. However, this robs from the end-user's name space and is generally regarded as a bad thing to do, unless the keyword uses an already reserved name space (which, in C, means it has to start with an underscore). Another alternative is to find some currently illegal combination of punctuation marks and make them a legal way of introducing the new feature. This cannot break existing programs . . . they wouldn't have compiled with older compilers. However, it does make it harder to use the preprocessor to mimic the new standard on an older compiler.

But, as I stated earlier, most modern compilers have *already* implemented attributes as an extension. GCC calls them attributes, whereas Microsoft's Visual C++ compiler calls them "declspec" (and almost every other compiler follows one or the other of these). In both cases, the existing practice has been, in fact, to use a new keyword. Both of them prefix their new keyword with two underscore characters, to put it into the name space reserved for the implementation.

Let's look at a trivial example of using attributes to decorate a function. I'm sure everyone who programs in C or C++ has at some time written a function something like the following:

```
void fatal(const char *msg)
{
    extern FILE *logfile;

    if (logfile) {
        fprintf(logfile, "Fatal: %s\n", msg);
        fclose(logfile);
    }
    fprintf(stderr, "Fatal: %s\n", msg);
    exit(1);
}
```

This simple function does some cleanup and exits the application on a fatal error. The function doesn't return; it calls exit(). There are a couple of things an optimizing compiler wants to be able to do with a function that doesn't return: remove dead code that follows a call to a nonreturning function and be able to notice that it doesn't need to worry about return paths following such a call. (Ever had that annoying error message "file.c:13: warning: control reaches end of non-void function"?) A function that doesn't return doesn't need to clean up the stack after itself, either.

Current existing practice in GCC allows you to add an attribute to the function prototype to indicate this:

```
__attribute__((noreturn)) void fatal(const char *);
```

The Microsoft compiler spells it slightly differently, but with the same effect:

```
__declspec(noreturn) void fatal(const char *);
```

The two committees, C and C++, are taking a very different approach to adding attributes.

The C Approach

The C committee wants to follow existing practice as much as possible; it is therefore looking at the __attribute__((xx)) and __declspec(xx) syntaxes closely. The committee will likely pick one rather than the other, and it may consider cleaning up the name a little. (All those underscores surely do look ugly!) They could go for new keywords for every attribute (e.g., noreturn) as a top-level keyword, but that would be very inflexible and hard to extend (al though there is precedent, since some of the current keywords, such as register, are really attributes). And remember what I was saying about keywords: Adding new keywords to the language is always going to be an uphill battle, as the users' name space is invaded. The syntax itself, however, is felt to be less important than the semantics of attributes. The intent of the committee is to select a common set of attributes that most vendors already support and to standardize what these attributes actually mean. To allow for further extension of this, the standardized attributes will have stdc_ prefixed to their name. The current proposal lists:

- stdc_noreturn: Applies to a function, indicating that the function does not return.
- stdc_pure: Applies to a function, indicating that the function has no side effects and will always return the same result for the same arguments (allowing the optimizer to possibly cache results).
- stdc_warn_unused_result: Applies to a function and will cause the compiler to issue a warning diagnostic if the result is not used (e.g., malloc() would be an example where this is appropriate).
- stdc_nonnull: Applies to a parameter to a function, indicating that the argument cannot be null.
- stdc_unused: Applies to a parameter to a function or to a variable, indicating that this parameter or variable is not used, but only required to ensure that the function has the correct signature.
- stdc_deprecated: Applies to a function, permitting the compiler to warn if the function is used.
- stdc_align: Applies to any variable, indicating the alignment of that variable.
- stdc_thread: Applies to any local variable, indicating that there should be a separate copy of the variable for each thread (GCC has a keyword, __thread, to do this).

- stdc_packed: Applies to a structure or union, indicating that no padding should be included, minimizing the amount of memory required to hold the type. It is also applicable to an enum type, indicating that the smallest integral type appropriate be used (e.g., a packed enum with fewer than 256 discrete values should be stored in a char).

Other attributes may yet be added to this list. In particular, the committee spent considerable time at its most recent meeting discussing the cleanup attribute from GCC, comparing it to the try {} finally {} construct added to Microsoft's compiler. A paper on this subject is expected at the next meeting, in April 2008.

The C++ Approach

The C++ committee, in contrast, *loves* to invent! If no new keywords are to be added to the language, why not invent a whole new syntax? Their proposal currently describes the syntax for adding attributes and only a few of the attributes themselves (noreturn, final, and align). The proposed syntax adds attributes surrounded by [[...]], *after* the definition. Currently both GCC and the Microsoft compiler expect attributes *before* the thing that they modify, though GCC can accept them after in some circumstances. So the fatal example above would become:

```
void fatal(const char *) [[noreturn]];
```

This syntax certainly doesn't suffer from the excess of underscores and general ugliness in the existing practice. It is certainly true that, by using the currently implemented extensions, the syntax can very rapidly get to be so opaque as to be almost unreadable:

```
int i __attribute__((unused));
static int __attribute__((weak)) const a5
      __attribute__((alias("__foo"))) __attribute__((unused));

// functions
__attribute__((weak)) __attribute__((unused)) foo()
      __attribute__((alias("__foo"))) __attribute__((unused));
__attribute__((unused)) __attribute__((weak)) int e();
```

The C++ proposal uses some aspects of the GCC syntax, but it removes that which the committee deems to be controversial. As stated, instead of __attribute__, which is long and makes a declaration unreadable, the proposal uses [[ ]] as delimiters for an attribute. For a general struct, class, union, or enum declaration, it will not allow attribute placement in a class head or between the class keyword, and the type declarator. Also, unlike the GCC attribute and Microsoft declspec, an attribute at the beginning will apply, not to the declared variable, but to the type declarator. This will have the effect of losing the GCC attribute's ability to declare an attribute at the beginning of a declaration list and have it apply to the entire declaration. The committee feels that this loss of convenience in favor of clearer understanding is desirable.

```
class C [[ attr2 ]] { } [[ attr3 ]] c [[ attr4 ]], d [[ attr5 ]];
attr2 applies to the definition of class C
attr3 applies to type C
attr4 applies to declarator-id c
attr5 applies to declarator-id d
```

Another aspect of the C++ proposal is to apply attributes to things other than simply variables, functions, and the like—for instance, to blocks and to translation units (or files). This aspect of attributes has no real implementa-

tion experience, although some compilers use the #pragma or _Pragma construct from C for something similar. So, for a global decoration or a basic statement, you might say:

```
using [[ attr1]];
```

to have attr1 apply to the translation unit from this point onward. Similarly, for a block, one might have:

```
using [[attr1]] { }
```

Now attr1 would apply to the block in braces. For a control construct, an annotation can be added at the beginning:

```
for [[ attr1 ]] (int i=0; i < num_elem; i++) {process (list_items[i]); }
```

where attr1 applies to the for control flow statement.

Conclusion

The C++ committee is also nearing the end of their revision process, whereas the C committee is just starting. If the C++ committee does indeed settle on the current proposed syntax, they will set new existing practice for the C committee to follow.

Several people have complained that recent changes to both C and C++ have led to divergence; neither committee appears to be able to follow the other's lead without making similar changes in an incompatible fashion. An example of this divergence was the introduction of variadic arguments to functions. C++ uses "..." following the last formal parameter, but in C there must also be a comma (", ..."). Indeed, some have noted that the *only* compatible extension that both languages have adopted is the // comment construct! So it will be interesting to see whether the introduction of attributes provides another place where the two languages diverge or a place where the two committees can actually work together for a change.

C is, after all, supposed to be a language compatible with C++. Once, C was a strict subset of C++, though it is no longer. But how far should they diverge? How much effort should we spend on maintaining the relationship between the languages?

I'm personally torn on the best way forward with attributes, in both languages, and would appreciate feedback.