Towards support for attributes in C

David Svoboda
svoboda@cert.org
Date: 2009-09-16

General Attributes for C

1 Overview

The idea is to be able to annotate some entities in C with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C syntax, and presents a general means for such annotations, including its integration into the C grammar. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language.

1.1 Changelog

- A compiler must issue a warning unless it is certain that a noreturn function never returns.
    - Added deprecated and warn_if_unused attributes

2 The Problem

At the Kona 2007 WG14 meeting, WG14 reviewed a number of papers outlining attributes for C. These included the proposals from C++ (WG 14 N1262 or WG 21 N2418), and a liaison statement from WG 14 to WG 21 (WG 14 N1273) was delivered and discussed at the WG 21 Bellevue 2008 meeting.
However, since that time, the WG21 committee has approved attributes into the current draft of C++. (WG21 document N2914). The document proposal that was accepted into C++ is N2761. Consequently, many open questions regarding current practice have become academic, as C++ will support attributes, and C++ compatibility is paramount.

2.1 Acknowledgements

This document is modeled after WG21 document N2761, written by Jens Maurer and Michael Wong, and contains much content from that document. Consequently that document's authors should be credited for most of the content of this document. In particular, this document deviates from N2761 in omitting some C++-specific sections. Also the specific proposals are for the C standard, not the C++ standard. In particular, the proposed wording changes are to be applied to N1362. Finally, the author wishes to credit Blaine Garst, David Keaton, Clark Nelson, Tom Plum, Robert Seacord and Nick Stoughton for reviewing this document and providing helpful suggestions.

3 The industry's solution

Most compilers implement extensions on top of the C Standard [C99]. In order to not invade Standard namespace, compilers have implemented double underscore keywords, __attribute__(( )) [GNU], or __declspec() [MS]syntax. C# [C#] implements a single bracket system.
This paper will study the __attribute__ and the __declspec syntax and make a recommendation on a specific syntax.

The following are C++ entities that could benefit from attributes:

- functions
  - variables
  - names of variables or functions
  - types
  - blocks
  - translation units
  - control-flow statements

4 GNU's attribute syntax

Although the exact syntax is described in the GNU [GNU] manuals, it is a verbal description with no grammar rules attached. This is a qualifier on type, variable, or function. It is assumed that the compiler knows based on the attribute as to which of those it belongs to and parse accordingly. This functionality has been implemented by GCC since 2.9.3 and various compilers which need to maintain GCC source-compatibility. The IBM compiler is one of those and has implementation experience since 2001. Other compiler experience includes EDG.

The description in the GCC manual is neither complete nor sufficiently specific to clearly avoid ambiguity. There are also somewhat incorrect implementations in existing GCC compilers. But the statement described in the GCC manual does describe an intended future direction. We suggest that we follow this future direction. In this paper, I will try to highlight those intended directions, describe any deviations and omissions from the manual descriptions, while giving sufficient feel for the syntax.

The general syntax is:

__attribute__((attribute-list))

and:

attribute-list
The format is able to apply to structures, unions, enums, variables, or functions. An undocumented keyword __attribute is equivalent to __attribute__ and is used in GCC system headers. The user can also use the __ prefixed to the attribute name instead of the general syntax above. For C structs and C++ classes, here are some examples of usage. First, an attribute can only be applied to fully defined type declaration with declarators and declarator-id.

```
__attribute__((aligned(16))) class Z {int i;} ;
__attribute__((aligned(16))) class Y ;
```

An attribute list placed at the beginning of a user-defined type applies to the variable of that type and not the type. This behavior is similar to __Declspec's behavior.

```
__attribute__((aligned(16))) class A {int i;} a ; // a has alignment of 16
class A a1; // a1 has alignment of 4
```

An attribute list placed after the class keyword will apply to the user-defined type. This is also __Declspec's behavior.

```
class __attribute__((aligned(16))) B {int i;} b ; // Class B has alignment of 16
class B b1; // b1 also has alignment of 16
```

Similarly, an attribute list placed before the declarator will apply to the user-defined type:

```
class C {int i;} __attribute__((aligned(16))) c ; // Class C has alignment 16
class C c1; //c1 also has alignment 16
```

But an attribute list placed after the declarator will apply to the declarator-id:

```
class D {int i;} d __attribute__((aligned(16))) ; //d has alignment 16
class D d1; // d1 has alignment 4
```

When all these attributes are present, the last one read for the class will dominate, but it could be overridden individually:

```
__attribute__((aligned(16))) class __attribute__((aligned(32))) E {int i;} __attribute__
((aligned(64))) e __attribute__((aligned(128))); // Class E has alignment 64
class E e1; // e1 also has alignment 64
class E e2 __attribute__((aligned(128))); // e2 has alignment 128
class E __attribute__((aligned(128))) e3 ; //e3 has alignment 64
class __attribute__((aligned(128))) E e4 ; //e4 has alignment 64
__attribute__((aligned(128))) class E e5 ; //e5 has alignment 128
```

While an attribute list is not allowed incomplete declaration without a declarator-id, it is allowed on a complete type declaration without a declarator-id. An attribute that is acceptable as a class attribute will be allowed for a type declaration:

```
class __attribute__((aligned(16))) X {int i; }; // class X has alignment 16
class X x; // x has alignment 16
class V {int i; } __attribute__((aligned(16))) ; // class V has alignment 16
class V v; //v has alignment 16
```

An attribute specifier list is silently ignored if the content of the union, struct, or enumerated type is not defined in the specifier in which the attribute specifier list is used.

```
struct __attribute__((alias("__foo"))) __attribute__((weak)) st1;
union __attribute__((unused)) __attribute__((weak)) un1;
enum __attribute__((unused)) __attribute__((weak)) enum1;
```

When an attribute does not apply to types, it is diagnosed. Where attribute specifiers follow the closing brace, they are considered to relate to the structure, union, or enumerated type defined, not to any enclosing declaration the type specifier appears in, and the type is not complete until after the attribute specifiers.

```
struct {} __attribute__((unused)) __attribute__((weak)) st4;
struct {int i;} __attribute__((unused)) __attribute__((weak)) st4a;
```

```
struct struct3 {int j;} __attribute__((alias("__foo"))) __attribute__((weak)) st5;
union {int i;} __attribute__((alias("__foo"))) __attribute__((weak)) un4;
union union3 {int j;} __attribute__((unused)) __attribute__((weak)) un5;
enum { } __attribute__((alias("__foo"))) __attribute__((weak));
enum {k};
enum {k1} __attribute__((unused)) __attribute__((weak));
enum enum3 {l} __attribute__((unused)) __attribute__((weak));
enum enum4 {m,};
enum enum5 {m1,} __attribute__((alias("__foo"))) __attribute__((weak));
```

Any list of qualifiers and specifiers at the start of a declaration may contain attribute specifiers, whether or not a list may in that context contain storage class specifiers. An attribute specifier list may appear immediately before the comma, =, or semicolon terminating a declaration of an identifier other than a function definition.

```
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();
```

An attribute specifier can appear as part of a declaration counting declarations of unnamed parameters and type names, and relates to that declaration (which may be nested in another declaration, for example in the case of a parameter declaration), or to a particular declarator within a declaration. Where an attribute specifier is applied to a parameter declared as a function or array, it should apply to the function or array rather then to the pointer to which the parameter is implicitly converted.

```
void func1(int __attribute__((weak, alias("__foo"))) name);
void func1(int __attribute__((weak, alias("__foo"))) name) {
  int i;
}
void func2(int __attribute__((noreturn)) array[]);
void funcptr(void);
void func3(int __attribute__((noreturn)) funcptr());
```

An attribute specifier list may appear after the colon following a label, other that a case or default label. The only attribute it makes sense to use is unused.

```
int main() {
  typedef int INT1; // INT1 is a <typedef name>
  typedef int INT2; // INT2 is a <typedef name>
  short i;

  // Syntactically an attribute specifier list can follow a label, but semantically the
only
  // attribute it makes sense to use is "unused" which we do not support (yet). So we
will
  // emit a warning here
  INT1: __attribute__((alias("oxford"))) __attribute__((unused)) __attribute__((weak))
  i = 3;
  LABEL1: __attribute__((unused)) __attribute__((weak))
  i = 4;
  // old behaviour still valid
  INT2:
  i = 3;
  LABEL2:
  i = 4;

  // attribute specifiers cannot appear after case and default labels
  switch(i) {
  case 0:
```

```
    i++;
    break;
  case 1: __attribute__((unused))
    i++;
    break;
  default: __attribute__((unused))
    break;
  }
  return 0;
}
```

4.1 Attribute specifiers as part of aggregate types, and enumerations

- An attribute specifier list is *silently* ignored if the content of the union, struct, or enumerated type is not defined in the specifier in which the attribute specifier list is used (same as GCC)
- A diagnostic message is emitted when attribute specifiers that do not apply to types are used on aggregate types and enums.

4.2 Attribute specifiers in comma separated list of declarations

- The first attribute specifier list applies to all the declarators, any other attributes specifier applies to the identifier declared, not to all the subsequent identifiers declared in the declaration. This is the intended future behavior documented in the GCC manual, which differs from the current GCC (3.0.1) behavior:

  Example:

```
int __attribute__((attr1)) foo1 __attribute__((attr2)),
__attribute__((attr3)) foo2 __attribute__((attr4)),
__attribute__((attr5)) foo3 __attribute__((attr6));
attr1 applies to foo1, foo2, foo3 because it is a declaration specifier
attr2 applies to foo1 because it is part of the foo1 declarator
attr3, attr4 apply to foo2 because they are part of the foo2 declarator
attr5, attr6 apply to foo3 because they are part of the foo3 declarator
```

4.3 Attribute specifiers immediately before a comma, = or semicolon

- the attribute specifier list should apply to the outermost adjacent declarator, not to the declared object or function. This is the intended future GCC behaviour, which differs from the current GCC behaviour.

  Example:

```
void (****f) (void) __attribute__((noreturn));
```

  "noreturn" should apply to the function ****f, but currently (for GCC) applies to the identifier f.

4.4 Attribute specifiers at the start of a nested declarator apply to the outermost adjacent declarator

- The GCC intended future semantics differs from the current behaviour.

Example:

```
void (__attribute__((noreturn)) ****f) (); // "noreturn" applies to the
function ****f, not to f
char* __attribute__((aligned(8))) *f; // "aligned" applies to char*, so f is a
pointer to 8-byte aligned pointer to char
```

- When an attribute specifier follows the * of a pointer declarator it should be a type attribute, and will be ignored with a silent informational message if it is not
- When an attribute specifier follows the * of a pointer declarator, it must follow any type qualifier present, and cannot be mixed with them.

```
void foo( int * const __stdcall __attribute__((weak)) i ); // allowed
void foo ( int * const __attribute__((weak)) __stdcall i ); // illegal
void foo ( int * __attribute__((weak)) const __stdcall i ); // illegal
```

4.5 Attribute specifiers list following a label

- An attribute specifier list following a *case* or *default* label will cause a syntax (parse) error (same as GCC)
- because the only attribute it makes sense to use after a label is "unused", an attribute specifier list following a label (other than *case* or *default*) will always be ignored
- A declaration starting with an attribute specifier that immediately follows a label is will be considered to apply to the label because this is consistent with what GCC (3.0.1) does. The attribute specifier can be applied to the declaration by inserting a semicolon between the colon that follows the label and the declaration:

```
L1: __attribute__((weak)) int i = 0; // weak applies to L1
L1: ; __attribute__((weak)) int i = 0; // weak applies to variable i
```

4.6 Problems with GNU __attribute__

There are some problems with this syntax through implementation experience. The syntax is long and ugly. It generally makes declarations unreadable if even one attribute is included.

The attribute syntax is not mangled leading to possible type collision. This causes problems when attributed types are used in C++ templates and overloading. Attributed types can be mangled, although this is strictly not part of the C++ Standard specification. But mangling will help to resolve the overloading problem. These issues do not apply to C.

The syntax as implemented differs from the manual, and is somewhat different from the standard C syntax. This proposal intends to correct most of these differences in favor of the C standard syntax, but largely maintains compatibility with GNU's intended future direction and therefore the large body of Open Source software.

We will use this syntax as guidance, but will try to obtain syntax rules that we feel makes more sense for readability.

5 Microsoft __DeclSpec syntax

The Microsoft __Declspec syntax [MS] is more precise and offers a grammar.

The **__declspec** keywords should be placed at the beginning of a simple declaration. The compiler ignores, without warning, any **__declspec** keywords placed after * or & and in front of the variable identifier in a declaration.
A **__declspec** attribute specified in the beginning of a user-defined type declaration applies to the variable of that type. For example:

```
__declspec(dllimport) class X {} varX;
```

In this case, the attribute applies to `varX`. A **__declspec** attribute placed after the **class** or **struct** keyword applies to the user-defined type. For example:

```
class __declspec(dllimport) X {};
```

In this case, the attribute applies to `X`.

This syntax is a subset of the more wild GNU attribute syntax, and actually offers no contradiction to the GNU syntax.

6 This Proposal

There are different designs on the syntactic construct of an attribute -- that is, the group of tokens which specify an attribute. There have been considerable discussions on this topic. We would like an approach which uses some aspect of the GNU syntax, but remove that which is deemed to be too controversial. We would also like to make it short (small number of characters) to facilitate readability. Summarizing the different opinions, we offer two suggestions in this paper. We will defer detailed discussion of them in section 8. Since this feature is likely to be used in header files which are shared between C and C++, we would like to obtain acceptance by both programming communities. The proposal process for N2761 sought a consensus from both WG14 and WG21.

With the exception of section 8, the discussion in this paper applies equally to both syntactic proposals. Without lost of generality, we will use the double-square bracket construction from here on in this paper.

For a general struct, union, or enum declaration, it will not allow attribute placement in a struct head, between the struct keyword, and the type declarator. Also, unlike GNU attribute and MS Declspec, attribute at the beginning will not apply to the declared variable, but to the type declarator. This will have the effect of losing GNU attribute's ability of declaring an attribute at the beginning of a declaration list, and having it apply to the entire declaration. We feel that this loss of convenience in favor of clearer understanding is desirable.

[[attr1]] struct S [[attr2]] { } [[attr3]] s [[attr4]], t [[attr5]];
attr1 applies to declarator-ids s, t
attr2 applies to the definition of struct S
attr3 applies to type S
attr4 applies to declarator-id s
attr5 applies to declarator-id t

A general function declaration can be decorated as follows. Only one attribute specifier is allowed in a decl-specifier seq, and it applies to the function return type.

[[attr1]] int [[attr2]] * [[attr3]] (* [[attr4]] * [[attr5]] f [[attr6]]) ( ) [[attr7]], e[[attr8]];
attr1 applies to the pointer-to-pointer to function f, and to e
attr2 applies to the return type of int
attr3 applies to the return type *
attr4 applies to the first * in the pointer-to-pointer to f
attr5 applies to the second * in the pointer-to-pointer to f
attr6 applies to the function variable f
attr7 applies to the function (**f)()
attr8 applies to e

Parameter declaration can also apply through a general type declaration.
An array declaration will apply as follows:

int [[attr2]] a [10] [[attr3]];
attr2 applies to type int.
attr3 applies to the array a.

For a global decoration or a basic statement:
[[attr1]];
attr1 applies to the translation unit from this point onwards

For a block:
[[attr1]] { }
attr1 applies to the block in braces.

For a control construct, annotation can be added at the beginning:
for [[attr1]] (i=0; i<num_elem; i++) {process (list_items[i]); }
attr1 applies to the control flow statement for.

After the WG21 C++ meeting in July 2007 in Toronto where the proposal was very well accepted, additional syntax was added for other control flow statements such as do, and while in addition to

- Case
  - Switch
  - Default
  - If
  - Else
  - Labels
  - Return
  - Goto
  - Throw
  - Using
  - bitfields

These were added for this paper.

All other positions are disallowed for attribute decorations.

Although this syntax is meant to be used for standard extensions, it could also be used for vendor-specific extensions. Vendor-specific extension will be required to use double-underscores for their attribute names. A good rule to follow may be to prefix the attribute with the vendor name such as:
[[ibm::align, noreturn, align(size_t), omp::for]]

6.1 Complex examples

Another issue is where to place the attribute when we wish to associate an attribute with the definition of a class or enum type. Currently it is placed after the class keyword and the declarator-id. Others have argued for its placement between the class-key and the declarator-id. This is referring to the problem that Lawrence Crowl brought up which involves placing the [[ ]] between the struct-key and the declarator-id, e.g.:

struct [[attr]] S s;

He argued that this would prevent having to clone S and then apply that cloned S with the attribute to s whereas a

struct S [[attr]] s;

would require cloning S with the attribute.

This is a kind of implementer complication. We argue that we already do that (cloning) when we have const/vol qualifiers anyway. This will be no worse.

A typedef will modify the cloned instance similar to a const

typedef struct foo [[attr]] foo;

Only in these two cases

struct S [[attr]] ;
struct S [[attr]] { … };

does the attr modify S such that all instance of struct S will have the attribute.

But

typefef struct S [[attr]] { … } S;

will modify the struct type S and not a copy of it.

7 Guidance on when to use/reuse a keyword and when to use an attribute

So what should be an attribute and what should be part of the language?

It was agreed that attributes would be something that helps but can be ignorable with little serious side-effects.

If you are proposing a new feature, the decision of when to use the attribute feature and when to overload or invent a new keyword should follow a clear guideline. At the Oxford presentation of this paper (C++ WG21 April 2007), we were asked to offer guidance in order to prevent wholesale dumping of extension keywords into the attribute extension. The converse is no one will use the attribute feature and all electing to create or reuse keywords in the belief that this elevates their feature in importance.

Certainly, we would advise anyone who propose an attribute to consider comments on the following area which will help guide them in making the decision of whether to use attributes or not:

- The feature is used in declarations or definitions only.
  - Is the feature is of use to a limited audience only (e.g., alignment)?
  - The feature does not modify the type system (e.g., thread_local) and hence does not require new mangling?
  - The feature is a "minor annotation" to a declaration that does not alter its semantics significantly. (Test: Take away the annotation. Does the remaining declaration still make sense?
  - Is it a vendor-specific extension?
  - Is it a language Bindings on C that has no other way of tying to a type or scope(e.g. OpenMP)
  - What is the effect in typedefs, will it require cloning?

Some guidance for when not to use an attribute and use/reuse a keyword

- The feature is used in expressions as opposed to declarations.
  - The feature is of use to a broad audience.
  - The feature is a central part of the declaration that significantly affects its requirements/semantics (e.g., constexpr).
  - The feature modifies the type system and/or overload resolution in a significant way (e.g., rvalue references). (However, something like near and far pointers should probably still be handled by attributes, although those do affect the type system.)
  - The feature is used everywhere on every instance of class, or statements

Where each vendor wishes to create a vendor-specific attribute, the use is conditionally-supported with implementation-defined behavior.

We have also added specific guidance on the choice of when to use an attribute to avoid misuse. There was general agreement that attributes should not affect the type system, and not change the meaning of a program regardless of whether the attribute is there or not. Attributes provide a way to give hint to the compiler, or can be used to drive out additional compiler messages that are attached to the type, or statement.

They provide a more scoped way of relating to C statements then what pragmas can do. As such, they can detect ODR violation more easily.

We created a list of good and bad attributes that can be used as guidelines.

Good choices in attributes include:

- align(unsigned int)
- pure (promise that a function always returns the same value)
- probably(unsigned int) (hint for if, switch, ...)
  - **if [[probably(true)]] (i == 42) { ... }**
- noreturn (the function never returns)
- deprecated (functions)
- noalias (promises no other path to the object)
- unused (parameter name)
- final on virtual function declaration and on a class
- not_hiding (name of function does not hide something in a base class)
- register (if we had a time machine)
- owner (a pointer is owned and it is the owner's duty to delete it)

Bad choices in attributes include:

- restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- const

For a particular interesting use of attributes, Michael Spertus has suggested an owner attribute with the following syntax:

```
char * [[owner]] strdup( char *[[not_owner]]);
int pthread_mutex_lock(pthread_mutex_t *[[not_owner]]);
```

Part of what makes memory management hard is that when you get a ptr from someone, you don't know if you are responsible for freeing it. For example, any user of strdup needs to know that they are responsible for freeing the pointer returned by strdup. Similarly, the caller of pthread_mutex_lock is not giving pthread_mutex_lock the responsibility for managing the lifetime of the pthread_mutex_t to pthread_mutex_lock.

The owner attribute says that the user of this pointer is responsible for managing the object's lifetime.

The not_owner attribute says that the user of this pointer has no responsibility for managing the object's lifetime.

Assigning an [[not_owner]] pointer to an [[owner]] pointer is not allowed because you can't give away something you don't own.

Not all pointers are suitable for this annotation. For example, one sometimes calls a function that may or may not save a pointer to one of its arguments. However, that does not reduce the usefulness of being able to notate that a function (e.g., a factory function) is returning a pointer that the caller needs to manage or the value of calling a function and knowing that it will not perturb the lifetime of its pointer arguments.

What makes this a good candidate for attributes is that code that runs with these attributes

also runs identically if the attributes are ignored, albeit with less type checking.

8 Alternative Syntax and controversial issues

Different syntax for specifying an attribute were discussed on the WG21 reflector, during private conversations and EWG presentations. For the purpose of this paper, we will summarize these discussions into two representative syntax below, and present them as proposals.

8.1 "Double-square" syntax

In this syntax, the specification of attributes begins with the characters "[[" and ends with "]]". There are variations where the two brackets are treated as one token or two tokens.

attribute-specifier : [[*attribute-list*]]

The idea is to find a (one) character or character pair which does not form the starting tokens in the right hand of existing production rules. An opening square bracket pair satisfies this requirement.
This syntax is succinct, concise, and short. The usual GNU attribute and MS declspec syntax is long and makes declarations difficult to read. The MS square bracket syntax, while even shorter can cause ambiguity for arrays, and may lead to difficulty with some parsers. So we have chosen to not duplicate it.

While reviewing this syntax, some WG14 members pointed out that the following syntax is preferable. We will call this the “function-like” syntax.

declarative_attribute(thread_local)

This allows it to be manipulated by the preprocessor. This syntax is even longer then the GNU syntax. We understand the desire to make it possible for preprocess manipulation such as to make the attribute disappear for compilers that don’t understand this. But we believe this is a different issue as every compiler must parse this as it is a standard-compliant feature.

The double-square syntax can provide for potential compatibility for GNU. It also provides a path for WG14 to adapt a similar but alternate attribute keyword for C1x. If this name is something like ATTRIBUTE(…), then a possible translation is:

#define ATTRIBUTE (…) [[__VA_ARGS__]]

We thought about having [[as a single token. We believe it helps the parser to disambiguate:

```
int a [10] [[thread_local ]];
int b[10];
```

where the parser only has to do a one-token look ahead to distinguish the two cases. Clark Nelson convinced us that there will always be a look-ahead issue. The difference is that in one case it is a one-character look-ahead if it is a token, or a one token look-ahead if it is a

double token. So we will not add [[as a new token and leave it as two tokens. We also do not want people to write:

int a [10][
// here comes an attribute
[adfalfdfhl]
]

8.2 "Function-like" syntax

attribute-specifier : std (*attribute-list*)

In this syntax, the attribute specification begins with the tokens "std(" and ends with ")". Instead of "std", we can use other variations of spelling. Underscore prefix can also be added. If ( ) is ambiguous, then we can also use (( )).

One key advantage of this syntax is that it follows the prior art in GCC. There are other compiler vendors supporting the GCC syntax, and the programming community is familiar with it. Existing code can readily adapt to this syntax.

Depending on what we choose as the "function name", function-like syntax can be short, addressing a concern expressed in the previous subsection. Also, square brackets are traditionally associated with arrays in the C family of languages. Double-square syntax can disappear in the middle of a complicated array declaration, and can be mistaken as part of a multidimensional array by a human reader. It therefore has its own share of readability issues. Double-square syntax is not necessarily better than function-like syntax in this regard.

One issue with function-like syntax is that the function name could collide with names in existing programs. Adding underscore prefix would not completely solve the problem as these names are reserved for the implementer. However, the problem may not be as severe as it seems. Given "std" is already used elsewhere in the language, it is unlikely that a compiler vendor would use names like "std" or "_Std" in an existing implementation. The same applies to the use of "std" in an existing program. Furthermore, we can assume that C compiler vendors are paying attention to the current C1x effort. It should not be difficult to find a suitable underscore name if "std" doesn't work.

8.3 Keyword syntax

attribute-specifier : *attribute-keyword*
attribute-specifier : *attribute-keyword* (*attribute-list*)
attribute-keyword : __STDC_NORETURN | __STDC_ALIGN | …

In this syntax, the attribute specification begins with an attribute keyword, which is unique for each attribute. The keyword begins with the prefix **__STDC_** and continues with the

attribute name. The parenthesized *attribute-list* may be omitted if it is empty. Instead of **__STDC_**, we can use other variations of spelling. Underscore prefix can also be added. If ( ) is ambiguous, then we can also use (( )).

One issue with this syntax is that it requires a new keyword for each attribute. However it is shorter and more concise than the previous two proposals. The requirement that __STDC_ be part of any attribute keyword serves to deter namespace pollution.

8.4 Compatibility with Existing practice and feedback from WG14

In summary, and after discussions with the C liaisons in the Bellevue meeting (WG21, February 2008), the main differences in opinion between the WG14 and WG21 committees were:

First, C places great emphasis on backwards compatibility with existing syntax, and in particular does not like the design choice in C++ where we break existing positional placement practices. While we do not have a specific list of these breakages, we present such a list in this paper (which applies to a specific release of GNU and note that GNU is also changing with each release). The leading issue of concern was felt to be the one of allowing attributes in the storage qualifier location, and applied to the declarator-ids at the far right of the declaration. To quote from paper WG14/N2418 (http://www.open-td.org/jtc1/sc22/wg21/docs/papers/2007/n2418.pdf):

An attribute list placed at the beginning of a user-defined type applies to the variable of that type and not the type. This behavior is similar to __Declspec's behavior.

__attribute__((aligned(16))) class A {int i;} a ; // a has alignment of 16
class A a1; // a1 has alignment of 4

This is a feature (or an accident) which C++ has deliberately avoided for a number of reasons. First, it breaks with the clean design in that an attribute is associated with an entity far from its position. Second, it leads to ambiguity because some future storage modifiers may demand that we look into the semantics of the modifier to determine which entity it would associate with (the declarator-id a or the type A).

Secondly, C prefers the function-like syntax (similar to GNU attributes) the bracketed-notation syntax chosen by C++. C generally favors standardizing existing practice rather then invent new syntax.

When N2761 was submitted to WG21, we discussed the merits of these issues. We feel obliged to explain that C++ chose this design deliberately and wish to consider 2 resolutions:

1. C++ wanted to create a consistent design where there is little doubt of what entity the attribute should be attached. We largely succeeded, we believe. In most cases, the attribute attaches to the entity to the left. In block scope cases, it does attach to the entity (the brace block) to the right, but where those cases exists, we feel it is the natural choice of programmers and is done also to avoid parsing ambiguity. However,

C++ does acknowledge the need to support existing practice and would consider updating our paper to accommodate the syntax above under strict conditions of positioning and binding.

1. On the delimiter issue, C++ chose new syntax because we deliberately chose semantics and syntax constraints different then the GNU syntax. Reusing an existing syntax under such circumstance would be incorrect. C++ members were almost in unanimous agreement that we would continue to strongly support the double-bracket syntax. The C liaisons present (Nick, Barry, Clark, Bill) felt that this is not nearly as big an issue as #1 for C, the positional placement issue and a compromise would be more likely especially since C++ is less likely to move (and a bridge between syntaxes exists).
2.

For N2761, we chose to adapt resolution 1: to extend C++ to allow attributes at the beginning of simple declarations and member declarations (not excluding function declarations)and have it applied only to the declarator-id (and it must be the first position and not be swapped with the static keyword.

For N2761, we also chose to adapt resolution 2 (that we would like to retain our delimiter choices) and urge C to adapt similar delimiter, and feel that in case C does not, some kind of macro magic will bridge the two syntaxes.

8.5 Vendor-specific extensions

Currently, vendor-specific extensions are added using the vendor name as a prefix and double colon followed by the attribute name (for attributes in C++). There is controversy on this as some opinions prefer double underscore prefix and postfix to the vendor name. The other controversial issue is the potential need for naming compiler vendor companies officially with a registered name to prevent name collisions. This would involve directly naming compiler vendors. This position remains controversial.

9 OpenMP binding to C

One serendipitous benefit of a feature design is if it can be used to solve an unexpected problem. This feature can be used to bind OpenMP [OpenMP] syntax more closely to C. OpenMP is an industry specification for loop parallelism with a common binding for Fortran, C and C++. It is popular with industry, research, and government. It describes syntax using pragmas for C and C++ for shared memory parallelism. One of the N2761 authors is a member of the OpenMP language committee, and the steering committee.

There are many problem with the pragma syntax including its inability to convey scope, error and type information. This has limited OpenMP's acceptance in C and C++. In Fortran, the binding is more natural. An alternate syntax that would work better with C/C++ has been requested by the OpenMP committee.

The attribute syntax while not perfect can be used to map almost every syntax construct in C. After discussion with Christian Terbiven, Dieter An Mey, and Bern Mohr in April of 2007,

they were very enthusiastic on the potential of this proposal to allow an augmented syntax for C++ and C if they also adapt this syntax.

The [ ] here has the usual meaning as optional element and should not be confused with the [[ ]] notation of the attribute syntax. It is not part of the syntax.

According to the current OpenMP 2.5 [OpenMP] specification, a parallel loop construct looks as follows:

#pragma omp for *[clause[[,] clause] ...] new-line*
for-loop

and is bound to a parallel region that looks as follows:

#pragma omp parallel *[clause[[,]clause] ...] new-line*
structured-block

while both constructs can be combined into the following:

#pragma omp parallel for *[clause[[,] clause] ...] new-line*
for-loop

These three code snippets could be written using the proposed attribute syntax as shown below:

```
for [[omp::for(clause, clause), … ]] (loop-head)
  loop-body
```

The enclosing parallel region would look like this:
`[[omp::parallel(clause,clause), … ]] { }`
When there are several clauses or the clauses contain a lot of variables, the `for` keyword and the actual loop can get quite far apart but this is normally the case when many attributes are used.

In OpenMP, a barrier is written as follows:
`#pragma omp barrier`

In the attribute syntax, this might look as follows:
`[[omp::barrier]] { }`

Everything in the structured block `{ }` will get executed by all threads in parallel, no worksharing constructs are allowed inside the block, the actual barrier is at the end of the block.

All other OpenMP 2.5 constructs and directives could be translated to `omp::clause` or `omp::directive` in the attribute syntax.

Here is a motivating example showing a clear advantage of the attribute syntax for OpenMP: Reductions in orphaned worksharing constructs. Assume the following program where we have a parallel region calling a subrouting containing a worksharing construct:

```
#pragma omp parallel
{
```

```
  double result = evaluate_my_function(…);
}

double evaluate_my_function(…)
{
  double sum;
#pragma omp for reduction(+:sum)
  for (int i = 0; i < something_large; i++)
  {
    sum += computation(i, …);
  }
  return sum;
}
```

As a reduction variable cannot be a private variable, the current solution is to declare `sum` static, which also alters the original program:

```
static double sum;
```

Using the attribute syntax with OpenMP, one could possibly write:

```
double sum [[omp::shared]];
```

The attribute syntax leaves several problems untouched and open, as the parallelization is still not really *in* the language. For example

- It is not possible for a function to determine if it is called inside of a worksharing construct.
- It is not possible to directly bind any information regarding the parallelization on a template type to allow for specialization (and thus optimization).

10 Proposed Wording Changes

General drafting note: These words introduce the term "appertains" for the syntactic relationship between the placement of an attribute-specifier and various source constructs such as labels or statement to which it applies. In contrast, the term "applies" is used to describe the semantic restrictions on an attribute.

Modify 6.2.3, paragraph 6 as indicated:

The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier extern; an identifier that appears in an attribute-token.

Modify 6.4.1, paragraph 2 as indicated:

> The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise **except in an *attribute-token* (6.9**). The keyword...

Modify 6.7 grammar as indicated:

```
declaration:
  simple-declaration
```

```
  static_assert-declaration
  attribute-declaration

simple-declaration:
  attribute-specifier-opt declaration-specifiers attribute-
specifier-opt init-declarator-list-opt ;

attribute-declaration:
  attribute-specifier ;
```

Modify 6.7 paragraph 6 as indicated:

> The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init- declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared. **In a *simple-declaration*, the first optional *attribute-specifier* appertains to every entity that is declared, and the second optional *attribute-specifier* appertains to the *declaration-specifiers*.**

Add the following paragraph to 6.7:

> Except where otherwise specified, the meaning of an *attribute-declaration* is implementation-defined.

Modify 6.7.2.1 grammar as indicated:

```
struct-or-union-specifier:
  struct-or-union identifier-opt attribute-specifier-opt { struct-
declaration-list }
  struct-or-union identifier attribute-specifier-opt
```

Modify 6.7.2.1 paragraph 6 as indicated:

> Structure and union specifiers have the same form. The keywords struct and union indicate that the type being specified is, respectively, a structure type or a union type. **The optional *attribute-specifier* appertains to the struct or union; the attributes in the *attribute-specifier* are henceforth considered attributes of the struct or union whenever it is named.**

Modify 6.7.2.2 grammar as indicated:

```
enum-specifier:
  enum identifier-opt attribute-specifier-opt { enumerator-list }
attribute-specifier-opt
  enum identifier-opt attribute-specifier-opt { enumerator-list , }
attribute-specifier-opt
  enum identifier attribute-specifier-opt
```

Add a new paragraph before 6.7.2.2 paragraph 5 (Example):

> **The first optional *attribute-specifier* in the *enum-specifier* appertains to the enumeration; the attributes in that *attribute-specifier* are henceforth considered attributes of the enumeration whenever it is named. The second optional**

attribute-specifier **in the** ***enum-specifier*** **may only appear if the** ***enumeration-list*** **is present; it appertains to the** ***enumeration-list.***

Modify 6.7.5 grammar as indicated:

```
declarator:
  pointeropt direct-declarator attribute-specifieropt
…
pointer:
 * type-qualifier-listopt attribute-specifieropt
        • * type-qualifier-listopt attribute-specifieropt
          pointer
```

Modify 6.7.5 paragraph 2 as indicated:

Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers. **The optional attribute-specifier following a declarator-id appertains to the entity that is declared.**

Modify 6.7.5 paragraph 4 as indicated:

In the following subclauses, consider a declaration

attribute-specifieropt T ***attribute-specifier***opt D1

where T contains the declaration specifiers that specify a type T (such as int) and D1 is declarator that contains an identifier ident. The type specified for the identifier ident in the various forms of declarator is described inductively using this notation. **The first optional** ***attribute-specifier*** **appertains to the entity being declared. The second optional** ***attribute-specifier*** **appertains to the type T.**

Modify 6.7.5.1, paragraph 1 as indicated:

If, in the declaration ‘‘T D1’’, D1 has the form

* ***attribute-specifier***opt type-qualifier-listopt D

and the type specified for ident in the declaration ‘‘T D’’ is ‘‘derived-declarator-type-list T ’’, then the type specified for ident is ‘‘derived-declarator-type-list type-qualifier-list pointer to T ’’. For each type qualifier in the list, ident is a so-qualified pointer. **The optional** ***attribute-specifier*** **appertains to the** ***type-qualifier-list*****.**

Modify 6.7.5.2, paragraph 3 as indicated:

If, in the declaration ‘‘T D1’’, D1 has one of the forms:

D[type-qualifier-listopt assignment-expressionopt] ***attribute-specifier***opt

D[static type-qualifier-listopt assignment-expression] ***attribute-specifier***opt

D[type-qualifier-list static assignment-expression] ***attribute-specifier***opt

D[type-qualifier-listopt *] ***attribute-specifier***opt

and the type specified for ident in the declaration ‘‘T D’’ is ‘‘derived-declarator-type-list T ’’, then the type specified for ident is ‘‘derived-declarator-type-list array of T ’’.126) (See 6.7.5.3 for the meaning of the optional type qualifiers and the keyword static.) **The optional** ***attribute-specifier*** **appertains to the array type.**

Modify 6.7.5.3, paragraph 5 as indicated:

If, in the declaration ‘‘T D1’’, D1 has the form

D(parameter-type-list) ***attribute-specifier***opt

or

D(identifier-listopt) ***attribute-specifier*****opt**

and the type specified for ident in the declaration ‘‘T D’’ is ‘‘derived-declarator-type-list T ’’, then the type specified for ident is ‘‘derived-declarator-type-list function returning T ’’. **The optional *attribute-specifier* appertains to the function type.**

Modify 6.7.6 grammar as indicated:

```
type-name:
  specifier-qualifier-list abstract-declaratoropt attribute-specifieropt
abstract-declarator:
  pointer attribute-specifieropt
  pointeropt attribute-specifieropt direct-abstract-declarator
```

Modify 6.8 grammar as indicated :

```
statement:
  labeled-statement
  attribute-specifieropt compound-statement
  attribute-specifieropt expression-statement
  attribute-specifieropt selection-statement
  attribute-specifieropt iteration-statement
  attribute-specifieropt jump-statement
```

Modify 6.8 paragraph 2 as indicated:

A statement specifies an action to be performed. Except as indicated, statements are executed in sequence. **The optional *attribute-specifier* appertains to the respective statement.**

Modify 6.8.1 grammar as indicated :

```
labeled-statement:
  attribute-specifieropt identifier : statement
  attribute-specifieropt case constant-expression : statement
  attribute-specifieropt default : statement
```

Modify 6.8.1 paragraph 4 as indicated :

Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control, which continues unimpeded across
them. **The optional *attribute-specifier* appertains to the label.**

Add a new section 6.9, before the “External Definitions” section entitled "Attributes". Its contents are:

Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

```
attribute-specifier:
   [ [ attribute-list ] ]
attribute-list:
```

```
   attributeopt
   attribute-list , attributeopt
attribute:
   attribute-token attribute-argument-clauseopt
attribute-token:
   identifier
attribute-argument-clause:
        ( attribute-argument-list )
      attribute-argument-list:
        attribute-argument
        attribute-argument-list, attribute-argument
      attribute-argument:
        assignment-expression
  type-id
```

For each individual attribute, the form of the *attribute-argument-clause* will be specified.
An *attribute-specifier* that contains no *attribute*s has no effect. The order in which the *attribute-token*s appear in an *attribute-list* is not significant. A keyword (6.4.1) contained in an *attribute-token* is considered an identifier. An identifier in an *attribute-token* has no linkage (6.2.2 Linkages of identifiers). The *attribute-token* determines additional requirements on the *attribute-argument-clause* (if any).
Each *attribute-specifier* is said to *appertain* to some entity or statement, identified by the syntactic context where it appears. If an *attribute-specifier* that appertains to some entity or statement contains an *attribute* that does not apply to that entity or statement, the behavior of the program is undefined. For an *attribute-token* not specified in this International Standard, the behavior is implementation-defined.

11 Specific attribute proposals:

This proposal will standardize the use of some attributes:

11.1 noreturn

This attribute is useful for a few library functions such as abort and exit which cannot return. The user can also define their own functions that never return using this attribute.

```
void fatal [[noreturn]] (void);
void fatal(…)
{
  …
  exit(1);
}
```

The noreturn keyword tells the compiler to assume that fatal() cannot return. It can then optimize without regard to what would happen if fatal ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables. You cannot assume that registers saved by the calling function are restored before calling the noreturn function. It does not make sense for a noreturn function to have a return type other than void.

This is a good attribute for standardization because it gives additional information that can be used by the optimizer, but does not alter the semantics of the program if removed. It is supported by both MSVC and GCC.

Standard functions that should have the noreturn attribute:

longjmp()
raise()
abort()
exit()
_Exit()

Proposed Changes:

Add a new subsection to the Attributes section:

The `noreturn` ***attribute***

The *attribute-token* `noreturn` specifies that a function does not return. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to the *declarator-id* in a function declaration. The first declaration of a function shall specify the `noreturn` attribute if any declaration of that function specifies the `noreturn` attribute. If a function is declared with the `noreturn` attribute in one translation unit and the same function is declared without the `noreturn` attribute in another translation unit, the program is ill-formed; no diagnostic required.

The implementation should produce a diagnostic message if a function f is declared with the `noreturn` attribute, but the implementation cannot be certain that the function does not indeed return. The implementation may then proceed with the translation of the program. If a function f is called where f was previously declared with the `noreturn` attribute, and f eventually returns, the behavior is undefined.

[Example:

```
void f [[ noreturn ]] () {
  abort(); // ok
}
void g [[ noreturn ]] (int i) { // ill-formed if called with i <=
0
  if (i > 0)
    abort();
}
```

]

11.2 deprecated

This attribute is useful for functions, global variables and global types.

It indicates that an object (function / variable / type) is deprecated, considered obsolete. A compiler should flag any usage of a deprecated object. The documentation for a deprecated object is encouraged to direct programmers to alternate objects to use instead. For instance, the gets() function documentation might direct readers to fgets().

The deprecated attribute is supported by GCC. It is also supported by MSVC, but only for functions, as MSVC's __declspec does not apply to types or variables. A similar mechanism for deprecated functions, variables and classes is widely used in Java.

Proposed Changes:
Add a new subsection to the Attributes section:

> ***The*** `deprecated` ***attribute***
>
> The *attribute-token* `deprecated` specifies that an object has been deprecated. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to the *declarator-id* in a function declaration. The `deprecated` attribute may also appear in a variable or type that has static or external linkage.
> The first declaration of an object shall specify the `deprecated` attribute if any declaration of that object specifies the `deprecated` attribute. If an object is declared with the `deprecated` attribute in one translation unit and the same object is declared without the `deprecated` attribute in another translation unit, the program is ill-formed; no diagnostic required.
> The implementation must issue a diagnostic message for any usage of a function, variable, or type with the `deprecated` attribute. The implementation may then proceed with the translation of the program.

11.3 warn_unused_result

This attribute is useful for any functions that return a useful result.

```
char* create_string [[warn_unused_result]] (void);
char* create_string(…)
{
  char* string = malloc(STRING_SIZE);
  …
  return string;
}
create_string(); // memory leak!
```

The warn_unused_result keyword instructs the compiler to issue a warning if the result of a function with this attribute is discarded. A function may have this keyword because discarding its result may produce a memory leak or resource leak. Likewise, a function may have this keyword because its return value serves as the only indicator of its success; hence discarding its return value destroys any evidence of its success or failure.

This attribute can also be used by programmers as they update APIs and deprecate their own obsolete objects. It is supported by GCC.

Standard functions that should have the warn_unused_result attribute:
sin(), cos(), and other mathematical functions
isalnum(), isdigit(), and other character-classification functions (including wide-char functions)
tmpfile(), tmpnam(), fopen(), freopen(),
feof(), ferror()

rand()
malloc(), calloc(), realloc()
getenv()
bsearch()
memcmp(), strcmp(), strcoll(), strncmp(), memchr(), strchr(), strrchr(), strspn(), strstr(), strlen(), and analogous wide-char functions

Proposed Changes:
Add a new subsection to the Attributes section:

> ***The*** `warn_if_unused` ***attribute***
>
> The *attribute-token* `warn_if_unused` specifies that a function produces a return value that should not be discarded. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to the *declarator-id* in a function declaration. The first declaration of a function shall specify the `warn_if_unused` attribute if any declaration of that function specifies the `warn_if_unused` attribute. If a function is declared with the `warn_if_unused` attribute in one translation unit and the same function is declared without the `warn_if_unused` attribute in another translation unit, the program is ill-formed; no diagnostic required.
> The implementation should produce a diagnostic message if a function f that has been declared with the `warn_if_unused` attribute is ever invoked in a context where its return value is discarded. That is, its value is not assigned to a variable, nor is its value used as part of an expression. The implementation may then proceed with the translation of the program.
> The `warn_if_unused` attribute shall not be applied to a function that returns `void`.

Acknowledgement

We would like to recognize the following people for their help in urging this work, their extended discussions and recommendations: Alisdair Meredith, Lawrence Crowl, Clark Nelson, Tom Plum, Attilla Feher, Ettore Tiotto, Sasha Kasapinovic, Yan Liu, Jeff Heath, Zbigniew Sarbinowski, Christopher Cambly, Sean Perry, Barry Hedquist, Francis Glassborow, Michael Spertus, Lois Goldthwaite, Bill Seymour, Walter Brown, Raymond Mak, Edison Kwok, Howard Nasgaard, Christian Terboven, Dieter An-Mey, Bern Mohr, Raul Silvera, Paul McKenney, Herb Sutter, Daveed Vandevoorde, Bjarne Stroustrup. We would also like to recognize WG14 members and the C/C++ liasions who contributed to improving this proposal for both languages. Daniel Krugler made valuable corrections and adaptations for the interaction between the new 0x features and this feature.

Reference

[C99] ISO/IEC 9899:201x, C Standard
[GNU] Section 5.25: Attribute Syntax, http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Attribute-Syntax.html#Attribute-Syntax
[MS] http://msdn2.microsoft.com/en-us/library/dabb5z75(VS.80).aspx

[C#] http://msdn2.microsoft.com/en-us/library/aa287992(VS.71).aspx
[n2224] **Seeking a Syntax for Attributes in C++09,** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2224.html
[n2761] **Towards support for attributes in C++,** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.html
[OpenMP] http://www.openmp.org/drupal/node/view/8