Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761=08-0271
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

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. Specific attributes are not introduced in this proposal. 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. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received acceptance in EWG to proceed to wording stage. This proposal integrates suggestions and comments from the Oxford presentation, and email conversations post-Oxford. It addresses many of the controversial aspects from the Oxford presentation and includes comprehensive Standard wordings. Specifically, it adds:

Sept 15, 2008, Revision 6

- Updated based on latest draft N2723
- Added support for late-specified return type attributes in 8.1p1 and 8.3.5p2
- Added support for enum-base attributes in 7.2p1
- added support for variadic template packed expansion through base-specifier in Clause 10
- Attribute-specification removed
- Updated with Issue 681
- After Core review on Sept 16,2008 addressed comment
  - Moved attribute to left of statement
  - Refined noreturn attribute
- After Core review on Sept 17, 2008 addressed comment

    - Redrafted noreturn statement semantics
    - Added lambda, and other syntax support
    - Removed throw support

June 10, 2008, Revision 5

- Minor fix-ups
- Post Core review and address core comments

Feb 28, 2008, Revision 4

- Expand grammar based on feedback from WG14 to accept attribute at the beginning of declarations binding to the list of declarator-ids.
- Added attributed bitfield support

Sept 10, Revision 3:

- Expand on C and C++ compatibility and simultaneously publish on WG14.

July 18, Revision 2:

- HTML cleanup
- allow empty attribute-lists
- renamed "attribute-parameter-clause" to "attribute-argument-clause", "attribute-parameter-list" to "attribute-argument-list", "attribute-parameter" to "attribute-argument"
- allow attribute on conversion-type-id's type-specifier-seq
- Toronto: introduce attribute-specification (similar to 'extern "C" { }')
- Toronto: integrate attributes into all keyword-based statements (except "break" and "continue"), plus "throw"
- Toronto: allow attributes on using-directives
- Toronto: add noreturn and final attributes

May 4, Revision 1:

- Empty attribute list
- Added Using for block scope attributes
- Added OpenMP control flow attribute syntax
- Removed support for the first attribute left class/enum/struct-key and the function return type

2 The Problem

In the pre-Oxford mailing, n2224 [n2224] makes a case for extensible syntax without overloading the keyword space. It references a large number of existing C++0x proposals that would benefit from such a proposal. This paper will examine the extensible syntax mechanism through the authors' experience with its implementation in an existing C++ compiler.

3 The industry's solution

Most compilers implement extensions on top of the C++ Standard [C++03]. 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. 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 sufficiently specific nor complete to clearly avoid ambiguity. It is also meant to bind to C-only. 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++ classes, here is some example 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 tye 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 behaviour documented in the GCC manual, which differs from the current GCC (3.0.1) behaviour:

  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 applies 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 even if one attribute is included. The attribute syntax is not mangled leading to possible type collision. This causes problems when attributed types are used in templates and overloading. In this paper, attributed types could be mangled, although this is strictly not part of the C++ Standard specification. But mangling will help to resolve the overloading problem.

The GNU syntax also does not distinguish between attributed types of a typeid reference. The original GNU syntax does not cover class and templates, but extension to classes as types is fairly straight forward. Templates will need some amount of work.

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 rule 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. We will get consensus from 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, class, union, enum declaration, it will not allow attribute placement in a class head, between the class 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]] class C [[attr2]] **{ }** [[attr3]] **c** [[attr4]]**, d** [[attr5]]**;**

attr1 applies to declarator-ids c, d
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

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

A constructor can be named as such, ignoring the arguments:

C::C [[attr1]] (…) [[attr2]];

attr1 applies to the name C
attr2 applies to the function C::C()

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:

using [[attr1]];

attr1 applies to the translation unit from this point onwards

For a block:

using [[attr1]] **{ }**

attr1 applies to the block in braces.

For a control construct, annotation can be added at the beginning:

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

attr1 applies to the control flow statement for.

After the meeting in Toronto where the proposal was very well accepted, additional syntax was asked for other control flow statements such as do, and while in addition to

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

This was 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-key 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 worst.

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 the variable 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 it 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, 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)
- How does this change Overload resolution?
- 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.

After the meeting in Toronto, we 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:

- C99 restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- C++ 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 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.

"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__]]

Note: Alisdaire Meredith supplied the finding that VA_ARGS is supported in clause 16.3p5 of the current draft.

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 ]
]
```

"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 C++1x effort. It should not be difficult to find a suitable underscore name if "std" doesn't work.

Compatibility with Existing practice and feedback from WG14

In summary, and after discussions with the C liaisons in the Bellevue meeting, the main differences in opinion between the two committees were:

1. 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 from C, I have compiled 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. I quote from our paper http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2418.pdf
e.g.:

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).

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

We discussed the merits of these issues. We feel obliged to explain that C++ chose this design deliberately and wish to consider 2 resolution:

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.

2. 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).

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.

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.

Vendor-specific extensions

Currently, vendor-specific extensions are added using the vendor name as a prefix and double colon followed by the attribute name. 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 author 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 asked for 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 shortly after the Oxford meeting, 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:

```
using [[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:

```
using [[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).

We may address these issue in the next revision of this paper.

10 Introduction of specific new attributes

This proposal will standardize the use of three good attributes and use the process to identify the reason why they are good candidates and add them to the C++ Standard.

- Align
  - This feature adds alignment support that overrides the natural alignment of the type. It gives more information to the compiler to align types, or functions more suitably for the optimizers. As such, it improves the program, but its absence does not necessarily make no sense. While it is true that incorrect alignment can cause bad behavior, the code presumably can still make sense without it.
- Noreturn

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

}

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.
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 because it gives additional information that can be used by the optimizer, but does not alter the semantics of the program if removed.

- Final
  - The final attribute to a class declaration and the virtual function declaration can prevent them from being further inherited. A class with the final attribute will not be allowed to be a base class for another class. A virtual function with the final attribute will not be overridden in a subclass. This is a good attribute because it allows the compiler to emit a message if the class or function is extended.

Examples

The specific attributes are shown for exposition only, since they do not form a part of this proposal. In particular, N2165 does not specify that alignment be part of the type, it is only an attribute of variables or class data members.

```
struct S [[ gnu::packed ]];   // avoid padding in this
structure

class C [[ wish::explicit_override ]]
  : public B { ... };

typedef struct [[ ibm::align(16) ]] { ... } T;

int x [[ ibm::library("hidden") ]];   // the name "x" is not
DLL-exported

int [[ ibm::align(16) ]] * f [[ ibm::library("export") ]]
(int, double);
           // exported function that returns a pointer to
aligned int

[[ ibm::align(16) ]] int i,j;      // variable i,j is
aligned to 16 bytes
```

```
static [[align(8)) ]] int k;    // ill-formed, attribute
must come before static
```

11 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 2.11 lex.key paragraph1 as indicated:

> The identifiers shown in Table 3 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) **except in an *attribute-token* (**7.6 dcl.attr):

Modify 3.3.1 basic.scope.pdecl paragraph 6 as indicated:

> The point of declaration of a class first declared in an *elaborated-type-specifier* is as follows:
>
> - for a declaration of the form
>
> `class-key identifier` ***attribute-specifier***$_{\texttt{opt}}$ `;`
>
> the *identifier* is declared to be a *class-name* in the scope that contains the declaration, otherwise
>
> - ...

Modify 3.4.4 basic.lookup.elab paragraph 2 as indicated:

> If the *elaborated-type-specifier* has no *nested-name-specifier*, and unless the *elaborated-type-specifier* appears in a declaration with the following form:
>
> `class-key identifier` ***attribute-specifier***$_{\texttt{opt}}$ `;`
>
> the *identifier* is looked up according to 3.4.1 but ignoring any non-type names that have been declared. ... If the *elaborated-type-specifier* is introduced by the *class-key* and this lookup does not find a previously declared *type-name*, or if the *elaborated-type-specifier* appears in a declaration with the form:
>
> `class-key identifier` ***attribute-specifier***$_{\texttt{opt}}$ `;`
>
> the *elaborated-type-specifier* is a declaration that introduces the *class-name* as described in 3.3.1 basic.scope.pdecl.

Modify 5.1.1 expr.prim.lambda paragraph 1 as indicated

lambda-parameter-declaration:

(*lambda-parameter-declaration-list*$_{opt}$) mutable$_{opt}$ ***attribute-specifier***$_{opt}$ *exception-specification*$_{opt}$ *lambda-return-type-clause*$_{opt}$

lambda-parameter-declaration-list:

lambda-parameter
lambda-parameter , lambda-parameter-declaration-list

lambda-parameter:

decl-specifier-seq ***attribute-specifier***$_{opt}$ *declarator*

lambda-return-type-clause:

-> ***attribute-specifier***$_{opt}$ *type-id*

In a *lambda-parameter-declaration*, the *attribute-specifier* appertains to the lambda. In a *lambda-return-type-clause*, the attribute appertains to the lambda return type.

Modify 6 stmt.stmt paragraph 1 as indicated

Except as indicated, statements are executed in sequence.

statement:

labeled-statement
attribute-specifier$_{opt}$ *expression-statement*
attribute-specifier$_{opt}$ *compound-statement*
attribute-specifier$_{opt}$ *selection-statement*
attribute-specifier$_{opt}$ *iteration-statement*
attribute-specifier$_{opt}$ *jump-statement*
declaration-statement
attribute-specifier$_{opt}$ *try-block*

The optional *attribute-specifier* appertains to the respective statement.

Modify 6.1 stmt.label paragraph 1 as indicated:

A statement can be labeled.

```
labeled-statement:
      attribute-specifier(opt) identifier : statement
```

```
            attribute-specifier-opt case constant-expression :
            statement
            attribute-specifier-opt default : statement
```

The optional *attribute-specifier* appertains to the label. An identifier label declares the identifier. ...

Modify 6.4 stmt.select paragraph 1 as indicated:

…

```
    condition:

      expression
      type-specifier-seq attribute-specifier-opt declarator =
      assignment-expression
```

***See* 8.3 dcl.meaning for the optional *attribute-specifier* in a condition.** In clause 6, the term *substatement* refers to the contained *statement* or *statements* that appear in the syntax notation. ...

Modify clause 7 dcl.dcl paragraph 1 as indicated:

...

```
declaration:
       block-declaration
       function-definition
       template-declaration
       explicit-instantiation
       explicit-specialization
       linkage-specification
       namespace-definition
       attribute-declaration

block-declaration:
       simple-declaration
       asm-definition
       namespace-alias-definition
       using-declaration
       using-directive
       static_assert-declaration

simple-declaration:
      attribute-specifier-opt decl-specifier-seq-opt attribute-
      specifier-opt init-declarator-list-opt ;
...

attribute-declaration:
       attribute-specifier ;
```

[*Note:* ...] The *simple-declaration*

***attribute-specifier*$_{opt}$** decl-specifier-seq$_{opt}$ ***attribute-specifier*$_{opt}$** init-declarator-list$_{opt}$;

is divided into ~~two~~ **four** parts: *decl-specifier*s, the components of a *decl-specifier-seq*, are described in 7.1**; the first optional *attribute-specifier,* the second optional *attribute-specifier,*** and declarators, the components of an *init-declarator-list*, are **all** described in clause 8.

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

Modify 7 dcl.dcl paragraph 8 as indicated:

Only in function declarations for constructors, destructors, and type conversions can the *decl-specifier-seq* be omitted. [Footnote: The "implicit int" rule of C is no longer supported.] **If it is omitted, no *attribute-specifier* may appear.**

Modify 7.1.6.3 dcl.type.elab paragraph 1 as indicated:

If an *elaborated-type-specifier* is the sole constituent of a declaration, the declaration is ill-formed unless it is an explicit specialization (14.7.3), an explicit instantiation (14.7.2) or it has one of the following forms:

class-key identifier ***attribute-specifier*$_{opt}$** ;
friend class-key ::$_{opt}$ identifier ;
friend class-key ::$_{opt}$ simple-template-id ;
friend class-key ::$_{opt}$ nested-name-specifier identifier ;
friend class-key ::$_{opt}$ nested-name-specifier template$_{opt}$ simple-template-id ;

In the first case, the *attribute-specifier*, if any, appertains to the class being declared; the attributes in the *attribute-specifier* are henceforth considered attributes of the class whenever it is named.

Modify 7.2 dcl.enum paragraph 1 as indicated:

Drafting note: this is based on updates from "N2764: Forward Declaration of Enumerations":

enum-specifier:

enum-head { enumerator-list opt }
enum-head { enumerator-list , }

enum-head:

enum-key identifier$_{opt}$ ***attribute-specifier*$_{opt}$** enum-base$_{opt}$ ***attribute-specifier*$_{opt}$**

```
  enum-key nested-name-specifier identifier attribute-specifier-seq_opt
  enum-base_opt attribute-specifier-seq_opt

opaque-enum-declaration:

  enum-key identifier_opt attribute-specifier-seq_opt enum-base_opt ;
```

The first optional *attribute-specifier* in the *enum-head* and *opaque-enum-declaration* 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-head* may only appear if the *enum-base* is present; it appertains to the *enum-base*.

Modify 7.3.4 namespace.udir paragraph 1 as indicated:

```
  using-directive:

      attribute-specifier-seq_opt using namespace  ::_opt nested-name-
      specifier_opt namespace-name ;
```

A using-directive shall not appear in class scope, but may appear in namespace scope or in block scope. [Note: when looking up a *namespace-name* in a *using-directive*, only namespace names are considered, see 3.4.6. -- end note]. **The optional *attribute-specifier* appertains to the *using-directive*.**

Add a new section 7.6 dcl.attr entitled "Attributes" (not shown in **bold** below):

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

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

attribute-list:
  attribute_opt
  attribute-list , attribute_opt

attribute:
  attribute-token attribute-argument-clause_opt

attribute-token:
  identifier
  attribute-scoped-token

attribute-scoped-token:
  attribute-namespace :: identifier

attribute-namespace:
  identifier

attribute-argument-clause:
  ( balanced-token-seq )
```

```
balanced-token-seq:
  balanced-token
  balanced-token-seq balanced-token

balanced-token:
  ( balanced-token-seq )
  [ balanced-token-seq ]
  { balanced-token-seq }
  any token other than a parenthesis, bracket or brace
```

For each individual attribute, the form of the *balanced-token-seq* 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 (2.11 lex.key) contained in an *attribute-token* is considered an identifier. No name lookup (3.4 basic.lookup) is performed on any of the identifiers contained in an *attribute-token*. The *attribute-token* determines additional requirements on the *attribute-argument-clause* (if any). The use of an *attribute-scoped-token* is conditionally-supported, with implementation-defined behavior. [*Note:* Each implementation should choose a distinctive name for the *attribute-namespace* in an *attribute-scoped-token*.]

Each *attribute-specifier* is said to *appertain* to some entity or statement, identified by the syntactic context where it appears (clause 7 dcl.dcl, clause 8 dcl.decl). If an *attribute-specifier* that appertains to some entity or statement contains an *attribute* that does not apply to that entity or statement, the program is ill-formed. If an *attribute-specifier* appertains to a friend declaration (11.4 class.friend), that declaration shall be a definition. No *attribute-specifier* shall appertain to an explicit instantiation (14.7.2 temp.explicit).

For an *attribute-token* not specified in this International Standard, the behavior is implementation-defined.

In 8 dcl.decl paragraph 4, modify the grammar:

```
declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers -> attribute-
  specifier_opt type-id

ptr-declarator:
  noptr-declarator
  ptr-operator ptr-declarator


noptr-declarator:
  declarator-id attribute-specifier_opt
  noptr-declarator parameters-and-qualifiers
  noptr-declarator [ constant-expression_opt ] attribute-specifier_opt
```

```
  ( ptr-declarator )

parameters-and-qualifiers:
  ( parameter-declaration-clause ) attribute-specifier_opt cv-
  qualifier-seq_opt ref-qualifier_opt exception-specification_opt

ptr-operator:
  * attribute-specifier_opt cv-qualifier-seq_opt
  &
  &&
  ::_opt nested-name-specifier * attribute-specifier_opt cv-qualifier-
  seq_opt
```

Drafting note: Attributes cannot appertain to references. This is an update based on Issue 681 "Restrictions on declarators with late-specified return type."

In 8.1 dcl.name paragraph 1, modify the grammar:

```
type-id:
  type-specifier-seq attribute-specifier_opt abstract-declarator_opt

abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator_opt parameters-and-qualifiers ->
  attribute-specifier_opt type-id
  ...

ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator_opt


noptr-abstract-declarator:
  noptr-abstract-declarator_opt parameters-and-qualifiers
  noptr-abstract-declarator_opt [ constant-expression_opt ] attribute-
  specifier_opt
  ( ptr-abstract-declarator )
```

Drafting note: This is an update based on Issue 681 "Restrictions on declarators with late-specified return type."

Add at the end of 8.3 dcl.meaning paragraph 1:

... When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers (or of an inline namespace within that scope (7.3.1)), and the member shall not have been introduced by a *using-declaration* in the scope of the class or namespace nominated by the *nested-name-specifier* of the *declarator-id*. [*Note:* if the qualifier is the global :: scope resolution operator, the *declarator-id* refers to a name declared in the global namespace scope. -- end note] **The optional**

***attribute-specifier* following a *declarator-id* appertains to the entity that is declared.**

Modify 8.3 dcl.meaning paragraph 3:

Thus, a declaration of a particular identifier has the form

```
T D
```

where T is **of the form *attribute-specifier*$_{opt}$** *decl-specifier-seq* ***attribute-specifier*$_{opt}$** and D is a declarator. ...

Modify 8.3 dcl.meaning paragraph 5:

In a declaration

***attribute-specifier*$_{opt}$** `T` ***attribute-specifier*$_{opt}$** `D`

where D is an unadorned identifier the type of this identifier is " T." **The first optional *attribute-specifier* appertains to the entity being declared. The second optional *attribute-specifier* appertains to the type T, but not to the class or enumeration declared in the *decl-specifier-seq*, if any.**

Modify 8.3.1 dcl.ptr paragraph 1 as indicated:

In a declaration T D where D has the form

`*` ***attribute-specifier*$_{opt}$** `cv-qualifier-seq`$_{opt}$ `D1`

and the type of the identifier in the declaration T D1 is "*derived-declarator-type-list* T," then the type of the identifier of D is "d*erived-declarator-type-list cv-qualifier-seq* pointer to T." The *cv-qualifiers* apply to the pointer and not to the object pointed to. **Similarly, the optional *attribute-specifier* (7.6 dcl.attr) appertains to the pointer and not to the object pointed to.**

Modify 8.3.3 dcl.mptr paragraph 1 as indicated:

In a declaration T D where D has the form

`::`$_{opt}$ `nested-name-specifier *` ***attribute-specifier*$_{opt}$** `cv-qualifier-seq`$_{opt}$ `D1`

and the *nested-name-specifier* names a class, and the type of the identifier in the declaration T D1 is "*derived-declarator-type-list* T," then the type of the identifier of D is "*derived-declarator-type-list cv-qualifier-seq* pointer to member of class *nested-name-specifier* of type T." **The optional *attribute-specifier* (7.6 dcl.attr) appertains to the pointer-to-member.**

Modify 8.3.4 dcl.array paragraph 1 as indicated:

> In a declaration T D where D has the form
>
> `D1 [ constant-expression`$_{opt}$ `] ` ***`attribute-specifier`***$_{opt}$
>
> and the type of the identifier in the declaration T D1 is "*derived-declarator-type-list* T," then the type of the identifier of D is an array type; if the type of the identifier of D contains the auto *type-specifier*, the program is ill-formed. ... If the value of the constant expression is N, the array has N elements numbered 0 to N-1, and the type of the identifier of D is "derived-declarator-type-list array of N T." ... If the constant expression is omitted, the type of the identifier of D is "*derived-declarator-type-list* array of unknown bound of T," an incomplete object type. ... The type "*derived-declarator-type-list* array of N T" is a different type from the type "*derived-declarator-type-list* array of unknown bound of T," see 3.9 basic.types. Any type of the form "*cv-qualifier-seq* array of N T" is adjusted to "array of N *cv-qualifier-seq* T," and similarly for "array of unknown bound of T." **The optional *attribute-specifier* appertains to the array.** ...

Modify 8.3.5 dcl.func paragraph 1 as indicated:

> In a declaration T D where D has the form
>
> `D1 ( parameter-declaration-clause ) `***`attribute-specifier`***$_{opt}$ `cv-qualifier-seq`$_{opt}$ `ref-qualifier`$_{opt}$ `exception-specification`$_{opt}$
>
> and the type of the contained *declarator-id* in the declaration T D1 is "*derived-declarator-type-list* T," the type of the *declarator-id* in D is "*derived-declarator-type-list* function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning T." **The optional *attribute-specifier* appertains to the function type.**

Modify clause 8.3.5 dcl.func paragraph 2 to

> In a declaration T D where D has the form
>
> `D1 ( parameter-declaration-clause ) `***`attribute-specifier`***$_{opt}$ `cv-qualifier-seq`$_{opt}$ `ref-qualifier`$_{opt}$ `exception-specification`$_{opt}$ `->` ***`attribute-specifier`***$_{opt}$ `type-id`
>
> and the type of the contained *declarator-id* in the declaration T D1 is "*derived-declarator-type-list* T," T shall be the single *type-specifier* auto. Then the type of the *declarator-id* in D is " function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning *type-id*." Such a function type has a *late-specified return type*. **The first optional *attribute-specifier* appertains to the function type. The second optional *attribute-specifier* appertains to the return type.**

Modify clause 8.3.5 dcl.func paragraph 4 to

A type of either form is a *function type*.[88]

parameter-declaration-clause:

parameter-declaration-list$_{opt}$ ...$_{opt}$
parameter-declaration-list , ...

parameter-declaration-list:

parameter-declaration
parameter-declaration-list , *parameter-declaration*

parameter-declaration:

decl-specifier-seq ***attribute-specifier*****$_{opt}$** *declarator*
decl-specifier-seq ***attribute-specifier*****$_{opt}$** *declarator* = *assignment-expression*
decl-specifier-seq ***attribute-specifier*****$_{opt}$** *abstract-declarator*$_{opt}$
decl-specifier-seq ***attribute-specifier*****$_{opt}$** *abstract-declarator*$_{opt}$ = *assignment-expression*

Modify clause 8.4 dcl.fct.def paragraph 1 to

Function definitions have the form

function-definition:

decl-specifier-seq$_{opt}$ ***attribute-specifier*****$_{opt}$** *declarator function-body*
decl-specifier-seq$_{opt}$ ***attribute-specifier*****$_{opt}$** *declarator* = default ;
decl-specifier-seq$_{opt}$ ***attribute-specifier*****$_{opt}$** *declarator* = delete ;

function-body:

ctor-initializer$_{opt}$ *compound-statement*
function-try-block

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal *function-body*.

Modify clause 8.4 dcl.fct.def paragraph 9 to

A function definition of the form:

decl-specifier-seq$_{opt}$ ***attribute-specifier*****$_{opt}$** *declarator* = default ;

…

Modify clause 8.4 dcl.fct.def paragraph 10 to

A function definition of the form:

decl-specifier-seq$_{opt}$ ***attribute-specifier***$_{opt}$ *declarator* = delete ;

…

In clause 9 class paragraph 1, modify the grammar:

class-head:
  class-key identifier$_{opt}$ ***attribute-specifier***$_{opt}$ base-clause$_{opt}$
  class-key nested-name-specifier identifier ***attribute-specifier***$_{opt}$ base-clause$_{opt}$
  class-key nested-name-specifier$_{opt}$ simple-template-id ***attribute-specifier***$_{opt}$ base-clause$_{opt}$

Add to clause 9 class paragraph 2 as indicated:

... A class is considered defined after the closing brace of its *class-specifier* has been seen even though its member functions are in general not yet defined. **The optional *attribute-specifier* appertains to the class; the attributes in the *attribute-specifier* are henceforth considered attributes of the class whenever it is named.**

In 9.2 class.mem paragraph 1, modify the grammar

member-declaration:

  decl-specifier-seq$_{opt}$ ***attribute-specifier***$_{opt}$ member-declarator-list$_{opt}$;
  function-definition ;$_{opt}$
  ::$_{opt}$ nested-name-specifier template$_{opt}$ unqualified-id ;
  using-declaration
  static_assert-declaration
  template-declaration

…

member-declarator:

  declarator pure-specifier$_{opt}$
  declarator constant-initializer$_{opt}$
  identifier$_{opt}$ ***attribute-specifier***$_{opt}$: constant-expression

In 9.6 class.bit paragraph 1, modify the grammar

A *member-declarator* of the form

identifier$_{opt}$ ***attribute-specifier*$_{opt}$** : constant-expression

specifies a bit-field; its length is set off from the bit-field name by a colon. **The optional *attribute-specifier* appertains to the entity being declared.**

Modify clause 10 class.derived paragraph 1 as indicated:

A list of base classes can be specified in a class definition using the notation:

base-specifier:

::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* ***attribute-specifier*$_{opt}$**

virtual *access-specifier*$_{opt}$ *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* ***attribute-specifier*$_{opt}$**

access-specifier virtual$_{opt}$ *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* ***attribute-specifier*$_{opt}$**

The optional *attribute-specifier* appertains to the *base-specifier*.

Modify 12.3.2 class.conv.fct paragraph 1 as indicated:

A member function of a class X with a name of the form
conversion-function-id:

operator conversion-type-id

conversion-type-id:

type-specifier-seq ***attribute-specifier*$_{opt}$** conversion-declarator$_{opt}$

conversion-declarator:

ptr-operator conversion-declarator$_{opt}$

specifies a conversion from X to the type specified by the *conversion-type-id*. ...

Modify 13.3.1.1.2 over.call.object paragraph 2 as indicated:

In addition, for each conversion function declared in T of the form
operator *conversion-type-id* ( ) ***attribute-specifier*$_{opt}$** *cv-qualifier ;*
where *cv-qualifier* is the same cv-qualification as, or a greater cv-qualification than, *cv*, and ...

11.1 Alignment attribute

Editorial note: Please consider collecting all the attributes as subsections under 7.6. This will include 7.1.7 through 7.1.9.

Remove the `alignas` as a keyword to 2.11 lex.key paragraph 1 Table 3: keywords.

Modify 3.2 basic.def.odr paragraph 4 Note

> ….
> — the type T is the subject of an `alignof` expression (5.3.6) ~~or an alignas specifier (7.1.7)~~.

Modify 3.11 basic.align paragraph 1:

> Object types have *alignment requirements* (3.9.1, 3.9.2) which place restrictions on the addressses at which an object of that type may be allocated. An *alignment* is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using ~~alignas~~ **the alignment attribute (7.1.7)**.

Modify 7.1.7 dcl.align, to 7.1.7 dcl.attr.align:

> 7.1.7 Alignment **Attribute** specifier [dcl.attr.align]

Delete 7.1.7 dcl.align paragraph 1 and replace with:

> **The *attribute-token* `align` specifies alignment. The *attribute* shall have one of the following forms:**
>
> ```
> align ( type-id )
> align ( assignment-expression )
> ```
>
> **The attribute applies to a variable that is neither a function parameter nor declared with the register storage class specifier or to a class data member that is not a bit-field.**

Modify 7.1.7 dcl.align paragraph 2:

> When the alignment ~~specifier~~ **attribute** is of the form `align~~as~~` (***assignment*~~*constant*~~*-expression***):
>
> — the *assignment-expression* shall be an integral constant expression

Modify 7.1.7 dcl.align paragraph 3:

> When the alignment ~~specifier~~ **attribute** is of the form `align~~as~~`(*type-id*), it shall have the same effect as `align~~as~~`(`alignof`(*type-id*)) (5.3.6).

Modify 7.1.7 dcl.align paragraph 4:

When multiple alignment **attributes** are specified for an object, the alignment requirement shall be set to the strictest specified alignment.

Modify 7.1.7 dcl.align paragraph 5:

The combined effect of all alignment **attributes** in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the object being declared.

Delete 7.1.7 dcl.align paragraph 6:

~~An alignment specifier shall not be specified in a declaration of a typedef, a bit-field, a reference, a function parameter or return type, or an object declared with the register storage-class specifier. [Note: in short, the specifier can be used on automatic variables, namespace-scope variables, and members of class types (as long as they are not bit-fields). In other words, it cannot be used in contexts where it would become part of a type so it would affect name mangling, name lookup, or ordering of function templates. —end note]~~

Modify 7.1.7 dcl.align paragraph 7:

If the defining declaration of an object has an alignment **attribute**, any non-defining declaration of that object shall either specify equivalent alignment or have no alignment **attribute**. No diagnostic is required if declarations of an object have different alignment **attributes** in different translation units.

Modify 7.1.7 dcl.align paragraph 8:

[Example: An aligned buffer with an alignment requirement of A and holding N elements of type `T` other than `char, signed char, or unsigned char` can be declared as:

T ~~alignas(T) alignas(A)~~ buffer [[align(T), align (A)]] [N] ;

Specifying `align`~~`as`~~`(T)` in the ~~alignment specifier list~~ ***attribute-list*** ensures that the final requested alignment will not be weaker than `alignof(T)`, and therefore the program will not be ill-formed. —end example]

Modify 7.1.7 dcl.align paragraph 9:

[Note: the alignment of a union type can be strengthened by applying the alignment **attribute** to any member of the union. —end note]

Add 7.1.7 dcl.align paragraph 10:

[Example:

```
void f [[ align(double) ]] ();
```

```
        // error: alignment applied to function
    unsigned char c [[ align(double) ]]  [sizeof(double)];
        // array of characters, suitably aligned for a double
    extern unsigned char c[sizeof(double)];
        // no "align" necessary
    extern unsigned char c [[ align(float) ]] [sizeof(double)];
        // error: different alignment in declaration
    ]
```

In 20.5.7 meta.trans.other paragraph 1, change the example to use the `align` attribute.

[Note: a typical implementation would define aligned_storage as:

```
template <std::size_t Len, std::size_t Alignment>
struct aligned_storage {
        typedef struct {
                ~~alignas(Alignment)~~ unsigned char __data
[[align(Alignment)]] [Len];
        } type;
};
```

—end note]

11.2 Noreturn attribute

Add a new section 7.1.8 dcl.attr.noreturn (not shown in **bold** below):

7.1.8 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.

If a function f is called where f was previously declared with the `noreturn` attribute, and f eventually returns, the behavior is undefined. [Note: The function may terminate by throwing an exception.]

[Example:

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

11.3 Final attribute

Add a new section 7.1.9 dcl.attr.final (not shown in **bold** below):

7.1.9 The `final` attribute

The *attribute-token* `final` specifies overriding semantics for a virtual function. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to class definitions and to virtual member functions being declared in a class definition. If the attribute is specified for a class definition, it is equivalent to being specified for each virtual member function of that class, including inherited member functions.

If a virtual member function f in some class B is marked `final` and in a class D derived from B, a function D::f overrides B::f, the program is ill-formed; no diagnostic required. [Footnote: If an implementation does not emit a diagnostic, it is encouraged to execute the program as if `final` were absent.]

[Example:

```
   struct B {
      virtual void f [[ final ]] ();
   };
   struct D : B {
      void f();   // ill-formed
   };
]
```

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

[C++03] ISO C++ 2003 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
[OpenMP] http://www.openmp.org/drupal/node/view/8