Towards support for attributes in C++

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

Document number: N2236=07-0096
Date: 2007-05-04
Project: Programming Language C++, Evolution Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 1

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 t his 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:

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 C++ entities that could benefit from attributes:

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

3.1 Type Attributes

- alignment
- packing / padding
- deprecation

3.2 Function Attributes

- Aliasing
- forcing / prohibiting inlining
- optimization hints
- deprecation
- shared library visibility
- calling convention
- object code section
- identifying order-dependent functions for concurrency

3.3 Variable Attributes

- alignment
- object code section
- deprecation
- packing / padding

3.4 Name Attributes

- Shared library visibility

3.5 Block Attributes

- Garbage collection control

3.6 Translation Unit Attributes

- Garbage collection control

3.7 Control flow attributes

- OpenMP parallelization

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

This proposal will use some aspect of the GNU syntax, but remove that which is deemed to be too controversial. Instead of __attribute__ which is long and makes a declaration unreadable, we will use [[ ]] as delimiter for an attribute.

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.

class C [[attr2]] **{ }** [[attr3]] **c** [[attr4]]**, d** [[attr5]]**;**

attr2 applies to the definition of class C
attr3 applies to type C
attr4 applies to declarator-id c
attr5 applies to declarator-id d

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.

int [[attr2]] ***** [[attr3]] **(*** [[attr4]] ***** [[attr5]] **f** [[attr6]] **) ( )** [[attr7]]**, e**[[attr8]]**;**

attr2 applies to the return type of int
attr3 applies to the return type *
attr4 applies to the first *
attr5 applies to the second *
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.

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

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 strict S will have the attribute.

But

typdefef 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

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.)

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

8 Alternative Syntax and controversial issues

Other syntax was discussed on the reflector and during private conversations and EWG presentations. Our choice for this syntax is that it 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 WG14, they pointed out that they prefer the syntax as:

```
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.

We provide for potential compatibility for GNU. We also provide 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__]]

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

However, we would prefer that WG14 choose to adapt the same syntax.

We thought about having [[is currently 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 token. So we will not add [[as a new token and leave it as two tokens.

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 people 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.

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.

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

The enclosing parallel region would look like this:

```
using [[omp::parallel, omp::clause,  omp::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 Proposed Grammar change

General drafting note: These words introduce the term "appertains" for the syntactic relationship between the placement of an attribute-specifier and the entity to which it applies. In constrast, the term "applies" is used to describe the semantic restrictions on an attribute.

Drafting note: The closing item `]]` cannot be a single token because that would interfere with two-level array access: a[b[5]]

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`**$_{opt}$ `;` 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(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(opt) ;
```

the *elaborated-type-specifier* is a declaration that introduces the *class-name* as described in 3.3.1 basic.scope.pdecl.

Modify 6.5 stmt.iter paragraph 1 as indicated:

Iteration statements specify looping.

```
   iteration-statement:
        while ( condition ) statement
        do statement while ( expression ) ;
        for attribute-specifier(opt) ( for-init-statement
condition(opt) ; expression(opt) ) statement
   for-init-statement:
        expression-statement
        simple-declaration
```

[*Note:* a *for-init-statement* ends with a semicolon. -- end note]

Modify 6.5.3 stmt.for paragraph 1 as indicated:

The `for` statement

```
    for attribute-specifier(opt) ( for-init-statement
condition(opt) ; expression(opt) ) statement
```

is equivalent to ... [Note: ...] **The optional *attribute-specifier* appertains to the `for` statement.**

Modify clause 7 dcl.dcl paragraph 1 as indicated:

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

```
        static_assert-declaration
        attribute-declaration

simple-declaration:
        decl-specifier-seq_opt attribute-specifier_opt init-
declarator-list_opt ;
...

attribute-declaration:
        using attribute-specifier ;
```

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

```
        decl-specifier-seq_opt attribute-specifier_opt init-
declarator-list_opt ;
```

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

Add a new paragraph after 7 dcl.dcl paragraph 4:

In an *attribute-declaration* at namespace scope, the *attribute-specifier* appertains to its innermost enclosing namespace. An *attribute-declaration* at block scope shall appear as the first declaration of that block, it appertains to the block.

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 7.1.5.3 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 these cases, 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:

```
...
enum-specifier:
```

```
      enum identifier_opt attribute-specifier_opt {
enumerator-list_opt }
      enum identifier_opt attribute-specifier_opt {
enumerator-list , }
...
```

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

Add a new section 7.6 dcl.attr entitled "Attributes":

Attributes specify additional information for types, variables, names, blocks, or translation units.

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

attribute-list:
    attribute_opt
    attribute-list , attribute_opt

attribute:
    attribute-token attribute-parameter-clause_opt

attribute-token:
    identifier
    attribute-scoped-token

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

attribute-namespace:
    identifier

attribute-parameter-clause:
    ( attribute-parameter-list )

attribute-parameter-list:
    attribute-parameter
    attribute-parameter-list, attribute-parameter

attribute-parameter:
    assignment-expression
    type-id
```

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 insignificant. 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-parameter*s (if any), including their number and whether each is a *type-id* or an expression. Each *attribute-parameter* that is an expression is an

unevaluated operand (clause 5 expr). 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 appertains* to some entity, 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 contains an *attribute* that does not apply to that entity, 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).

Two attributes are the *same* if their *attribute-token*s are the same, either both have no *attribute-parameter-clause* or both have the same number of *attribute-parameter*s, each corresponding *attribute-parameter* is of the same kind (expression or *type-id*), each corresponding *attribute-parameter* that is a *type-id* refers to the same type, and each corresponding *attribute-parameter* that is an expression satisfies the requirements for multiple definitions of an entity (3.2 basic.def.odr).

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

```
direct-declarator:
    declarator-id attribute-specifier-seq[opt]
    direct-declarator ( parameter-declaration-clause )
attribute-specifier-seq[opt] cv-qualifier-seq[opt] exception-
specification[opt]
    direct-declarator [ constant-expression[opt] ]
attribute-specifier-seq[opt]
    ( declarator )

ptr-operator:
    * attribute-specifier-seq[opt] cv-qualifier-seq[opt]
    &
    &&
    ::[opt] nested-name-specifier * attribute-specifier-seq[opt]
cv-qualifier-seq[opt]
```

Drafting note: Attributes cannot appertain to references.

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

```
type-id:
       type-specifier-seq attribute-specifier-seq[opt]
abstract-declarator[opt]

...

direct-abstract-declarator:
```

```
    direct-abstract-declarator_opt ( parameter-
declaration-clause ) attribute-specifier_opt cv-
qualifier-seq_opt exception-specification_opt
    direct-abstract-declarator_opt [ constant-expression_opt
] attribute-specifier_opt
    ( abstract-declarator )
```

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, 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 and 5 as indicated:

Thus, a declaration of a particular identifier has the form

```
T D
```

where T is ~~a~~ **of the form** *decl-specifier-seq* ***attribute-specifier*_opt** and D is a declarator. ...

First, the *decl-specifier-seq* determines a type. In a declaration

```
T D
```

the *decl-specifier-seq* T determines the type T. [Example: ...]

In a declaration T ***attribute-specifier*_opt** D where D is an unadorned identifier the type of this identifier is "**attribute-specifier** T." **The 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 "derived-declarator-type-list cv-qualifier-seq **attribute-specifier** pointer to T." The cv-qualifiers apply to the pointer and not to the object pointed to. **Similarly, the *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* **attribute-specifier** pointer to member of class *nested-name-specifier* of type T." **The *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 deduction 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 **attribute-specifier** array of N T." ... If the constant expression is omitted, the type of the identifier of D is "derived-declarator-type-list **attribute-specifier** array of unknown bound of T," an incomplete object type. ... The type "derived-declarator-type-list **attribute-specifier** array of N T" is a different type from the type "derived-declarator-type-list **attribute-specifier** array of unknown bound of T," see 3.9 basic.types. Any type of the form "cv-qualifier-seq **attribute-specifier** array of N T" is adjusted to "**attribute-specifier** 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`$_{opt}$ `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* ***attribute-specifier*** function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ returning T"; a type of this form is a function type [Footnote: ...]. **The optional *attribute-specifier* appertains to the function.**

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 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
```

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;      // ill-formed
```

11 Modifications for existing papers

11.1 N2147 Thread-Local Storage

Drop the change to 2.11 lex.key (adding __thread as a keyword).

Instead of the proposed modification for 3.7 basic.stc paragraph 3, modify that paragraph as indicated:

The storage class specifiers `static` and `auto` **and the attribute `thread_local`** are related to storage duration as described below.

In the proposed new section 3.7.2(new) basic.stc.thread, modify the first sentence as indicated:

All objects declared with the ~~`__thread` keyword~~ **attribute `thread_local` (7.6.1 dcl.attr.thread)** have thread storage duration. ...

Add a new bullet to section 5.19 expr.const paragraph 2 as amended by N2235 "Generalized Constant Expresions --- Revision 5", as indicated:

- ...
- **a *unary-expression* with a & operator (5.3.1 expr.unary.op) unless it is applied to an lvalue that refers to a variable or data member with static storage duration;**
- a *new-expression* (5.3.4 expr.new);
- ...

Drop all changes to 7.1.1 dcl.stc, instead modify 7.1.1 dcl.stc paragraph 4 as indicated:

... A static specifier used in the declaration of an object declares the object to have static storage duration (3.7.1 basic.stc.static)**, unless the object is declared with the attribute thread_local (7.6.1 dcl.attr.thread)**. ...

Add a new section 7.6.1 dcl.attr.thread:

7.6.1 Thread-local storage [dcl.attr.thread]

The *attribute-token* thread_local specifies thread-local storage. It shall appear at most once in each *attribute-list* and no *attribute-parameter-clause* shall be present. The attribute applies to variables of block scope and class data members that are declared static and to variables of namespace scope (see 3.7.2(new) basic.stc.thread). [*Note:* The attribute does not apply to function parameters.] If the attribute appears in a declaration of a variable, it shall appear in all declarations of that same object, no diagnostic required.

[*Example:*

```
int i [[ thread_local ]] = 42;  // thread-local
namespace-scope variable "i"
void f() {
  static double v [[ thread_local ]] = 0.1;
              // "v" is a block-scope variable with
thread storage duration
}
```

extern int i; // error: redeclaration missing thread_local attribute

]

Instead of the proposed modification for 8.5 dcl.init paragraph 2, modify that paragraph as indicated:

> ~~Automatic, register, static, and external variables of namespace scope~~**Variables with static, thread, or automatic storage duration** can be initialized by arbitrary expressions involving literals and previously declared variables and functions. [Example: ...]

Drop the change to 9.2 class.mem paragraph 6.

Instead of the proposed modification for 9.4.2 class.static.data paragraph 1, modify that paragraph as indicated:

> A static data member is not part of the subobjects of a class. **If such a member is declared with the attribute thread_local (7.6.1 dcl.attr.thread), there is only one copy of the member per thread, otherwise there** ~~There~~ is only one copy of ~~a static data~~ **the** member shared by all the objects of the class.

11.2 N2165 Adding Alignment Support to the C++ Programming Language

Do not add alignas as a keyword to 2.11 lex.key.

Drop the change to 3.2 basic.def.odr (see N2253 "Extending sizeof to apply to non-static data members without an object (revision 1)").

Modify the added section 3.11 basic.align paragraph 2/3 as indicated:

> Fundamental alignments are
>
> - Alignments of fundamental types
> - Alignments of any type that is not affected by any ~~alignas alignment specifier~~ **align attribute** [Note: A type can only be affected by the ~~alignas alignment specifier~~ **align attribute** by applying it to non-static **class data** members ~~of class types or members of union types (8.3.7dcl.align)~~. - end note]
> - Alignments of any type that is affected by an ~~alignas specifier~~ **align attribute** that sets the alignment requirements to any of the previously listed fundamental alignments

Drop the change to 5.19 expr.const (see N2235 "Generalized Constant Expressions -- Revision 5").

Drop the change to 8 dcl.decl paragraph 4.

Drop the addition of 8.3.7 dcl.align, instead add a new section 7.6.2 dcl.attr.align:

7.6.2 Alignment

The *attribute-token* align specifies alignment; the *attribute-parameter-list* shall consist of exactly one *attribute-parameter* that is either a *type-id* (8.1 dcl.name) or an integral constant expression (5.19 expr.const). The attribute applies to a class data member and to a variable other than a function parameter or a variable declared register.

If the *attribute-parameter* is an integral constant expression, its value, if positive, specifies the alignment requirement of the declared object. If that value is zero, the attribute has no effect, if it is negative the program is ill-formed. If the *attribute-parameter* is a *type-id*, it is equivalent to the expression alignof(*type-id*) (5.3.6 expr.alignof).

If more than one align attribute is specified for an object, the alignment requirement for the object is the weakest alignment that meets all the alignment requirements specified by each attribute. If no such alignment exists, the program is ill-formed.

The combined effect of all align attributes shall not specify an alignment that is less strict than the alignment that would otherwise be required for the object being declared, or an alignment that is not compatible with the declared type.

If an align attribute appears in a declaration of an entity, the same attribute shall appear in all declarations of that same entity, except if a declaration is not a definition and no align attribute appears in that declaration; no diagnostic required.

[Examples:

```
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
```

]

(Add notes from the former 8.3.7 as desired.)

In 20.4.8 meta.trans.other paragraph 1, change the example to use the align attribute.

Drop all changes to appendix A, it's automatically generated anyway.

11.3 N2108 Explicit Virtual Overrides

Add a new section 7.6.3 dcl.attr.expl

7.6.3 Explicit class

The *attribute-token* explicit_override specifies that a class is explicit (10.3 class.virtual). It shall appear at most once in each *attribute-list* and no *attribute-parameter-clause* shall be present. The attribute applies to a class when it is defined (clause 9 class).

The example in 10.3 class.virtual paragraph 2 should be carefully amended to mention ill-formed cases using [[explicit_override]].

Acknowledgement

We would like to recognize the following people for their help in urging this work, their extended discussions and recommendations: Alisdaire Meredith, Lawrence Crawl, Clarke Nelson, Tom Plum, Attilla Feher, Ettore Tiotto, Sasha Kasapinovic, Yan Liu, Jeff Heath, Zbigniew Sarbinowski, Christopher Cambly, Walter Brown, Raymond Mak, Howard Nasgaard, Christain Terboven, Dieter An-Mey, Bern Mohr, Raul Silvera, Paul Mckenney, Herb Sutter, Daveed Vandevood, Bjarne Stroustrup.

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