Towards support for attribute-like syntax

Michael Wong, Jens Maurer, Alisdair Meredith
Howard Nasgaard, Sasha Kasapinovic, Yan Liu
michaelw@ca.ibm.com
jens.maurer@gmx.net
alisdair.meredith@uk.renaultf1.com
nasgaard@ca.ibm.com
skasapin@ca.ibm.com
yanliu@ca.ibm.com

Document number: D2236=07-0096
Date: 2007-04-11
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.

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__(( )), or __declspec() syntax..

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

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
- alignment

3.3 Variable Attributes

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

3.4 Name Attributes

- Shared library visibility

3.5 Block Attributes

- gc_strict, gc_relaxed

3.6 Translation Unit Attributes

- gc_forbidden, gc_required, gc_safe

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 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 are meant to 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.

[[attr1]] **class C** [[attr2]] **{ }** [[attr3]] **c** [[attr4]]**, d** [[attr5]]**;**

attr1, attr2, 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, attr2 applies to the return type of int
attr3 applies to the return type * int
attr4 applies to the first *
attr5 applies to the second *
attr6 applies to the function name f
attr7 applies to the function (**f)()
attr8 applies to e

Parameter declaration can also apply through a general type declaration.

An array declaration will apply as follows:

[[attr1]] **int** [[attr2]] **a** [[attr3]] **[10];**

attr1 and attr2 applies to type int
attr3 applies to the array a

For a global decoration:

register [[attr1]];

attr1 applies to the translation unit from this point onwards

For a block:

[[attr1]] **{ }**

attr1 applies to the block in braces.

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]]

This is not a necessary part of this proposal.

7 Proposed Grammar change

```
attribute-specification:
    [[ attribute-list ]]

attribute-list:
    attribute
    attribute-list , attribute

attribute:
    attribute-token attribute-parameter-clause_opt

attribute-token:
    identifier
```

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

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

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

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 declaraton of the form `class-key identifier attribute-specification`$_{opt}$ `;` 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-specification`**$_{opt}$ `;`

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-specification`**$_{opt}$ `;`

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

Modify 7.1.5 dcl.type paragraph 1 as indicated:

The type-specifiers are

```
type-specifier:
       simple-type-specifier
       class-specifier
       enum-specifier
       elaborated-type-specifier
       typename-specifier
       cv-qualifier
       attribute-specification
```

Modify the grammar before 7.1.5.3 dcl.type.elab paragraph 1 as indicated:

```
elaborated-type-specifier:
      class-key identifier attribute-specification
      class-key ::opt nested-name-specifieropt
identifier
```

class-key ::$_{opt}$ nested-name-specifier$_{opt}$
template$_{opt}$ simple-template-id
enum ::$_{opt}$ nested-name-specifier$_{opt}$ identifier

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-specification**$_{opt}$;
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 8 [dcl.decl] paragraph 4, modify the grammar:

direct-declarator:
declarator-id **attribute-specification**$_{opt}$
direct-declarator (parameter-declaration-
clause) **attribute-specification**$_{opt}$ cv-qualifier-
seq$_{opt}$ exception-specification$_{opt}$
direct-declarator [constant-expression$_{opt}$]
(declarator)

ptr-operator:
* **attribute-specification**$_{opt}$ cv-qualifier-seq$_{opt}$
&
&&
::$_{opt}$ nested-name-specifier * cv-qualifier-seq$_{opt}$

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

direct-abstract-declarator:
attribute-specification$_{opt}$
direct-abstract-declarator~~$_{opt}$~~ (parameter-
declaration-clause) **attribute-specification**$_{opt}$ cv-
qualifier-seq$_{opt}$ exception-specification$_{opt}$
direct-abstract-declarator~~$_{opt}$~~ [constant-
expression$_{opt}$]
(abstract-declarator)

In 9 class paragraph 1, modify the grammar:

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

```
            class-key nested-name-specifier_opt simple-
     template-id **attribute-specification_opt** base-clause_opt
```

In 7 dcl.dcl paragraph 1, modify the grammar:

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

In 6.3 stmt.block paragraph 1, modify the grammar:

```
     compound-statement:
          attribute-specification_opt { statement-seq_opt }
```

Examples

The specific attributes are shown for exposition only, since they do not form a part of this proposal.

```
struct S [[ packed ]];   // struct S is packed, avoid
padding in this structure

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

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

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

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

int [[ align(16) ]] * g(int, double) [[
library("export") ]];
           // same attributes as f
```

global: register [[gc_forbidden]];

block: [[gc_strict]] { ... };

Acknowledgement

I would like to recognize the following people for their help is urging this work, and their implementation experience discussions: Ettore Tiotto, Sasha Kasapinovic, Yan Liu, Jeff Heath, Zbigniew Sarbinowski, Christopher Cambly, Alistair Meredith, Walter Brown, Raymond Mak, Howard Nasgaard.

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
[n2224] **Seeking a Syntax for Attributes in C++09,** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2224.html