Proposal for C2x
WG14 N2335

Title:	Attributes in C
Author, affiliation:	Aaron Ballman, GrammaTech
Date:	2019-03-09
Proposal category:	New features
Target audience:	General developers, compiler/tooling developers

Abstract: Attributes are a mechanism by which the developer can attach extra information to language entities with a generalized syntax, instead of introducing new syntactic constructs or keywords for each feature. This information is intended to be used by an implementation in ways which have minimal semantic impact, such as improving the quality of diagnostics produced by an implementation or specifying platform-specific behavior. Attributes are intended for lightweight situations where keywords may be inappropriate, but are not intended to obviate the need or ability to add such keywords when appropriate.

Prior art: C++ has this feature using the same syntax. Various vendors have vendor-specific extensions such as `__attribute__` and `__declspec` that provide a similar mechanism.

Attributes in C

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2335
Revises Document No: N2269
Date: 2019-03-09

Summary of Changes

N2335

- Clarified 6.7.11p4; standards-based attributes should not have semantic impact on strictly conforming programs.
- Added 6.7.11p5 to ensure that `[[attr]]` and `[[__attr__]]` are equivalent for standards-based attributes. Hinted that implementations should do something similar for vendor attributes.
- Moved the *attribute-specifier-seq* out of the *enumeration-constant* production and into the enumerator production.
- Removed explicit mention of attributes appertaining to a translation unit from 6.7.11p3. They can appertain to a TU, but it needs to be specified under the semantics of an attribute because there is no syntactic location to specify an attribute on a TU explicitly.
- Consistently treat attribute appertainment as a semantic property (this moves some paragraphs from Syntax or Constraints to Semantics).
- Added information under the Proposal section to clarify that attribute support for compound literals is not being proposed at this time and why.
- Applied numerous editorial changes recommended by a project editor. Mostly relating to renaming grammar productions, using prose rather than grammar productions in the text, and rearranging some of the exposition. All of these changes were verified to be editorial in nature.

N2269

- Made it undefined behavior to `#define` or `#undef` a standard attribute.

N2165

- Added the appropriate cover page.
- Can no longer specify an attribute on a K&R C function.
- Created the *no-leading-attribute-declaration* production to ensure attributes do not ambiguously parse with declarations nesting a K&R C function.
- Moved a note out of the Constraints section in 6.7.11 Attributes.
- Rewrote 6.7.2.1p6-7 to clarify the wording intent.
- Added a new constraint to 6.7.2.3 for tag declarations with attributes.
- Added wording to 6.7.2.3p7-8 to clarify that the attributes appertain to the declared tag whenever it is named.
- Removed grammar production for adding attributes to a bit-field in 6.7.2.1 because it was already covered by the declarator grammar.

N2137

- Identified that it is common practice for header files to be written in C but consumed by a C or a C++ compiler.
- Removed an example from the Proposal section of code that was not conforming C code; also switched to some more C-like terms such as “identifier” rather than “name”.
- Added the Alternative Syntaxes section that discusses some alternative syntaxes and why they are not being proposed in this paper.
- Added Proposed Wording.

N2049

- Original proposal.

Introduction

Attributes are a mechanism by which the developer can attach extra information to language entities with a generalized syntax, instead of introducing new syntactic constructs or keywords for each feature. This information is intended to be used by an implementation in ways which have minimal semantic impact, such as improving the quality of diagnostics produced by an implementation or specifying platform-specific behavior. Attributes are intended for lightweight situations where keywords may be inappropriate, but are not intended to obviate the need or ability to add such keywords when appropriate. Attributes are not an inventive concept in C, as vendors have produced different language extensions covering this functionality in the past, as discussed in great detail in N1229, N1264, and N1403. Attributes can either be vendor-specific attributes, which are introduced by a vendor-supplied namespace, or standardized attributes, which are not.

```
// Standardized attributes.
[[something]] void f([[something_else]] int i);
// Vendor-specific attributes.
[[gnu::something]] void g([[clang::something_else]] int i);
```

Rationale

The C++ syntax was carefully designed to allow full generality, and is being proposed over Microsoft `__declspec` and GNU `__attribute__` syntaxes. For instance, `__declspec` attributes appertain only to declarations, and not to other syntactic constructs such as statements. The `__attribute__` syntax can appertain to a wider range of entities, but suffers from ambiguity (e.g., `void f(int (__attribute__((foo)) x));`).

The placement of the attributes in various syntactic constructs was determined by WG21 to eliminate ambiguity and provide a consistent design while covering all possible use cases. The general rule of thumb for attribute placement is that an attribute at the start of a declaration or statement appertains to everything to the right of the attribute, and an attribute elsewhere appertains to the syntactic element immediately preceding the attribute.

Use of the C++ syntax is also consistent with the WG14 charter principle to minimize incompatibilities with C++ [N2021]. The C++ syntax using double square brackets was introduced in C++11 and has gained wide vendor adoption (MSVC, GCC, Clang, EDG, et al) and considerable positive use from users in the

form of adding new, vendor-specific attributes in addition to standards-mandated attributes. Concerns were raised in the past about the inventiveness of using double square brackets, but their inclusion in the C++ standard for 5+ years and their implementation by major compiler vendors that also support C implementations suggests that this is no longer a truly inventive syntax.

Use of double colons to delineate vendor-specific attributes from standards-based attributes is similarly proposed to be consistent with the C++ syntax. While this construct is not currently found in the C programming language, deviation from this syntax causes a seemingly-gratuitous incompatibility with C++. A different syntax may be plausible, but it forces users desiring interoperability with C++ to make extended use of the preprocessor and increases the teaching burden for people learning about attributes in either language. While the syntax may be unfortunate for the C programming language, it is also not unduly egregious -- it poses no backwards compatibility issues nor an extra burden on implementers to support and is concise. Given the utility of vendor-specific attributes in practice and the extant syntax with C++, this proposal recommends use of double colons as a reasonable syntax for the feature.

Note that this proposal is not proposing to add attributes to C in the form proposed simply because C++ has them in that form, but instead due to the wide popularity vendor-specific attribute implementations in C have enjoyed over the past two decades. The choice of syntax is a pragmatic one, especially given the common practice of providing a header file written in C that is to be consumed by either a C or C++ compiler.

Previous proposals raised concerns about how double square brackets would interact with other C-like languages, such as Objective-C. Specifically, Objective-C uses square brackets for "message send" expressions. e.g., `[foo bar];` where `foo` is the recipient of the message and `bar` is the selector. There were concerns that using double square brackets would create parsing ambiguity, such as with an attributed *expression-statement* that was a complex message send expression. Objective-C has a sibling language called Objective-C++ (usually denoted with a .mm file extension instead of a .m file extension) that uses C++ instead of C as a foundation, and this language is implemented by the Clang open source compiler. In practice, there is no ambiguity between Objective-C++ and C++ attributes. Given that Clang does not currently implement any attributes that appertain to an *expression-statement*, I privately implemented an attribute named `foobar` and tested it with what could be an ambiguous parse to see whether the Clang parser could handle it without modification, and whether AST properly reflected the attribute.

```
@interface Base
@end
@interface S : Base
- (void) bar;
@end
@implementation S
- (void) bar {}
@end
@interface T : Base
- (S *) foo;
@end
@implementation T
- (S *) foo { return nullptr; }
@end
```

```
void func(T *t) {
  [[foobar]][[t foo] bar];
}
```

The above code was properly parsed and the `foobar` attribute was properly applied to the Objective-C message send expression, as shown by this AST dump of the `func()` function definition:

```
`-FunctionDecl 0x5c591327c0 <line:20:1, line:22:1> line:20:6 func 'void (T *)'
  |-ParmVarDecl 0x5c59132700 <col:8, col:11> col:11 used t 'T *'
  `-CompoundStmt 0x5c59132980 <col:14, line:22:1>
    `-AttributedStmt 0x5c59132960 <line:21:3, col:31>
      |-FoobarAttr 0x5c59132950 <col:5>
      `-ObjCMessageExpr 0x5c59132920 <col:19, col:31> 'void' selector=bar
        `-ObjCMessageExpr 0x5c591328f0 <col:20, col:26> 'S *' selector=foo
          `-ImplicitCastExpr 0x5c591328d8 <col:21> 'T *' <LValueToRValue>
            `-DeclRefExpr 0x5c591328b0 <col:21> 'T *' lvalue ParmVar 0x5c59132700 't' 'T *'
```

Because Objective-C++ is a superset of Objective-C, it is reasonable to conclude that possible ambiguous parses that could arise from adoption of the proposed attribute syntax in C can be overcome by vendors supporting the feature in Objective-C using similar implementation strategies.

Another possible ambiguity arises from the fact that WG21 chose to standardize the concept of a function that never returns by using the `[[noreturn]]` attribute, while WG14 chose to standardize the same concept by using the `_Noreturn` keyword. It is likely that with acceptance of this proposal users will attempt to use the following declaration in a header file shared by both C and C++ code: `[[noreturn]] void f(void);` However, this construct can be gracefully handled in one of two ways: a user concerned about code portability can define a macro to specify that the function never returns using the proper language-specific constructs, or the user's vendor can implement `[[noreturn]]` in C as a matter of QoI due to the fact that use of attribute tokens not specified by the C standard results in implementation-defined behavior.

Proposal

This document proposes to add support for attributes in C using the syntax introduced by WG21 for attributes in C++ [WG21 N2761, N1403]. This document also serves as background information on syntax for the following four, related WG14 proposals: N2051 (nodiscard), N2053 (maybe_unused), N2050 (deprecated), and N2052 (fallthrough).

Attributes appertain to a particular source construct, such as a variable, type, identifier, statement, etc. Concrete examples include:

```
[[attr1]] struct [[attr2]] S { } [[attr3]] s1 [[attr4]], s2 [[attr5]];
```

`attr1` appertains to the identifiers `s1` and `s2`, `attr2` appertains to the declaration of `struct S`, `attr3` appertains to the type `struct S`, `attr4` appertains to the identifier `s1`, and `attr5` appertains to the identifier `s2`.

```
 [[attr1]] int [[attr2]] * [[attr3]] f([[attr4]] float [[attr5]] f1 [[attr6]],
int i) [[attr7]];
```

`attr1` appertains to the function declaration `f()`, `attr2` appertains to the type `int`, `attr3` appertains to the type `int *`, `attr4` appertains to the function parameter `f1`, `attr5` appertains to the type `float`, `attr6` appertains to the identifier `f1`, and `attr7` appertains to the function declaration `f()`.

```
 [[attr1]] int [[attr2]] a[10] [[attr3]], b [[attr4]];
```

`attr1` appertains to the variable declarations `a` and `b`, `attr2` appertains to the type `int`, `attr3` appertains to the variable declaration `a`, and `attr4` appertains to the variable declaration `b`.

```
  [[attr1]] stmt;
```

`attr1` appertains to the entire statement, regardless of statement kind (including null statements, labels, and compound blocks).

Attributes can also appear in constructs that allow the declaration of an identifier.

```
  for ([[attr1]] int i = 0; i < 10; ++i)
    ;
```

`attr1` appertains to the variable declaration `i`.

```
  enum e { i [[attr1]] };
```

`attr1` appertains to the enumerator `i`.

```
  struct S {
    [[attr1]] int i, *j;
    int k [[attr2]];
    int l [[attr3]] : 10;
  };
```

`attr1` appertains to both `i` and `j` member declarations, `attr2` appertains to the member declaration `k`, and `attr3` appertains to the bit-field member declaration `l`.

In all cases, attributes are delimited by double square brackets. Between the square brackets is the (possibly empty) comma-separated list of attributes. If no attributes are present in the list, the attribute specifier is silently ignored. Attributes in the list that are not specified by the International Standard have implementation-defined behavior. The order of the attributes in the attribute list is not significant. The attribute identifier determines additional requirements for an optional attribute argument clause, allowing attributes to be parameterized; e.g., a hypothetical `deprecated` attribute may have an argument clause allowing an optional message for the compiler to use when emitting a diagnostic, but another attribute specification may disallow any arguments.

C++ supports vendor-specific attribute syntax, which is an integral component to the feature that has considerable popularity with vendors. For instance, to date, the Clang implementation supports 30 vendor-specific attributes under the `clang` attribute scope, and the GCC implementation supports all GNU-style `__attribute__` constructs under the `gnu` attribute scope (50+ unique attributes). This vendor-specific syntax uses the C++ nomenclature of double colons to separate the vendor name

component from the attribute name component, e.g., `clang::fallthrough` or `gnu::format`. It is worth noting that the notion of scoped attributes is separate from the notion of namespaces in C++. The name component does designate a namespace of sorts, but does not tie in to the namespace feature itself (attribute names do not participate in name lookup, scoped attribute tokens cannot be compounded to form a scope chain, etc). Attribute tokens (including scoped attribute tokens) that are unknown to the implementation are ignored. This allows vendors to implement an attribute without fear of conflicting with the International Standard (including future revisions) or other vendors, but still allows a vendor the latitude to implement attributes from other vendors. For instance, the Clang implementation also implements several attributes under the `gnu` scoped attribute name, as a matter of QoI.

Compound Literals

C allows a programmer to create a new temporary object of a specified type through use of compound literals and it could make sense to specify attributes on such an object. For example, alignment is attribute-like functionality that could be useful with compound literals. However, attributes do not appertain to any other subexpressions, so it is not clear that this functionality fits with the original design of attributes. Further, it may cause confusion for C to allow attributes to appertain to a compound literal when C++ does not allow attributes to appertain to initializer lists (which are a similar, but not identical feature in C++). While it may make sense to allow attributes to appertain to this unnamed object, that functionality is not being proposed as a part of this paper. Should this functionality be desired, care would need to be taken to ensure that the attribute syntax does not create parsing ambiguities and remains consistent with typical attribute placement.

Alternative Syntaxes

During the Oct 2016 Pittsburgh meeting, a few alternative syntaxes were discussed by the committee. The alternatives discussed were:

`_Pragma`

It was observed that C already has the ability to attach extra information to language constructs with the `_Pragma` preprocessor directive, and it was questioned whether an attribute syntax was required.

The `_Pragma` preprocessor directive is unfit as a replacement for an attribute syntax. The *string-literal* provided to the directive is processed through translation phase 3 as though it was a series of *pp-tokens*, which are limited in their capabilities.

`_Attribute`

As an alternative to using `[[]]` to denote an attribute list, it was questioned whether a function-like keyword would be more appropriate, such as `_Attribute`. Concerns were raised that the double square bracket syntax would disallow multiple attributes in a single attribute list from being used as a macro replacement list; e.g.,

```
#define M(x) x
M([[foo, bar]]) void f(void);
```

However, this code is ill-formed in C++ and would not be expected to successfully translate as C code under this proposal. Further, the Clang compiler implementation has not received any feature requests to allow such a construct in C++.

It was nonetheless observed that a function-like keyword would prevent such problems while still allowing common code to be shared between C and C++ through use of macros; e.g.,

```
#define _Attribute(…) [[__VA_ARGS__]]
```

A function-like keyword would work but is not being proposed due to it being divergent from the C++ syntax. One common approach to writing libraries is to provide a header file in C that is consumed by either a C++ compiler or a C compiler and the nature of attributes is that they frequently appertain to constructs in a header file (such as tag declarations, function declarations, and function parameters). While a macro like the one above could be used to support this case, there is a strong incentive to not diverge from the syntax of feature already implemented in C++ unless there is clearly specified rationale [SC22WG14.14310].

The implementation experience in C++, at least for the implementations with public bug trackers, is that the double square bracket syntax does not result in complications where users are asking for a function-like keyword syntax. Further, this approach would require users to write error-prone macros for a common use case in the field.

Proposed Wording

The wording proposed is a diff from ISO/IEC 9899-2018. Green text is new text, while ~~red~~ text is deleted text.

6.4 Lexical elements

Drafting notes:

Some attributes in the wild make use of keywords as part of the attribute identifier, such as `[[gnu::const]]`. In order to support that use case, we need to allow identifiers that could be either a keyword or an identifier to be an identifier for attribute tokens.

Additionally, in order to support vendor namespaces for attributes in the same manner as C++, `::` is added as a punctuator. However, this could potentially break conforming extensions. GCC has the `__asm__` extension, which uses colons to separate optional string literals. Code exists in the wild that looks like: `__asm__("..." ::: "memory");`, for which treating `::` as a single token might require GCC to alter the implementation of their extension. However, GCC already handles the above example in C++, so this change may or may not break user code, but using consecutive single colon tokens creates the possibility of users writing attributes accepted by C that are rejected by C++, such as `[[foo: :bar]]`. Due to this, the single token form is proposed, but if implementation experience suggests this breaks conforming extensions, the consecutive token form may be a viable alternative.

Modify 6.4.1p2:

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

Modify 6.4.2.1p4:

When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword except in an attribute token.

Modify 6.4.6p1:

punctuator: one of

[ ] ( ) { } . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : :: ; ...
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
<: :> <% %> %: %:%:

6.7 Declarations

Drafting notes:

The goal of these changes is to allow an attribute specifier to appear to the left of a declaration so that the attributes appertain to all of the declarators in the declaration list, or to appear to the right of all declaration specifiers so that the attributes appertain to the type determined by the specifier sequence. One divergence from C++ is with the alignment specifier. In C++, an alignment specifier is an attribute itself, and the remainder of the grammar falls out naturally from that. Further, in C++, the alignment specifier may only appear after the full sequence of declaration specifiers, not in the middle of the sequence. In this draft, I have left *alignment-specifier* as-is in order to reduce drafting churn or break existing code.

Similarly, an attribute specifier can appear to the right of a type in a declarator to appertain to the type, or to the right of an identifier in a declarator to appertain to the identifier declared.

There is a notion of an attribute declaration, which is a convenience production (rather than having a null attributed statement) that is used for attributes like `[[fallthrough]];`.

The no-leading-attribute-declaration production is required to disallow nested K&R functions, such as `int (*f(a, b))(int, int) [[something]] int a; int b; { return 0; },` where disallowing the attribute on the K&R declarator is insufficient to resolve the ambiguous parse. This also ensures that the function signature of a K&R function does not introduce attributes on the parameters only within the definition of the *declaration-list* and not on the function signature.

Finally, this adds a new subclause for the syntactic and semantic requirements for attributes themselves. Under this subclause is where the specific attribute definitions (`deprecated`, `nodiscard`, etc.) will be defined.

Modify 6.7p1:

no-leading-attribute-declaration:
declaration-specifiers init-declarator-list$_{opt}$ **;**
static_assert-declaration

declaration:
no-leading-attribute-declaration
attribute-specifier-sequence declaration-specifiers init-declarator-list **;**
attribute-declaration

declaration-specifiers:
~~*storage-class-specifier declaration-specifiers*$_{opt}$~~
~~*type-specifier declaration-specifiers*$_{opt}$~~
~~*type-qualifier declaration-specifiers*$_{opt}$~~
~~*function-specifier declaration-specifiers*$_{opt}$~~
~~*alignment-specifier declaration-specifiers*$_{opt}$~~
declaration-specifier attribute-specifier-sequence$_{opt}$
declaration-specifier declaration-specifiers

declaration-specifier:
storage-class-specifier
type-specifier-qualifier
function-specifier

init-declarator-list:
init-declarator
init-declarator-list **,** *init-declarator*

init-declarator:
declarator
declarator **=** *initializer*

attribute-declaration:
attribute-specifier-sequence **;**

Modify 6.7p2:

A declaration other than a static_assert or attribute declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

Editorially modify 6.7p5 for clarity:

A declaration specifies the interpretation and ~~attributes~~ properties of a set of identifiers. …

Modify 6.7p6:

The declaration specifiers consist of a sequence of specifiers, followed by an optional attribute specifier sequence, that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if

any) being declared. The optional attribute specifier sequence appertains to each of the entities declared by the declarators of the init declarator list.

Add new paragraphs after existing 6.7p7:

8 The optional attribute specifier sequence terminating a sequence of declaration specifiers appertains to the type determined by the preceding sequence of declaration specifiers. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

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

10 **Example 1** In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the identifier for that declaration.

```
[[deprecated]] void f [[deprecated]] (void); // valid
```

Modify 6.7.2.1p1:
Drafting notes: These changes are assuming DR 444 has been applied [DR 444].

struct-or-union-specifier:
struct-or-union *attribute-specifier-sequence*$_{opt}$ *identifier*$_{opt}$ **{** *struct-declaration-list* **}**
struct-or-union *attribute-specifier-sequence*$_{opt}$ *identifier*

struct-or-union:
struct
union

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
attribute-specifier-sequence$_{opt}$ *specifier-qualifier-list struct-declarator-list*$_{opt}$ **;**
static_assert-declaration

specifier-qualifier-list:
~~*type-specifier specifier-qualifier-list*$_{opt}$~~
~~*type-qualifier specifier-qualifier-list*$_{opt}$~~
~~*alignment-specifier specifier-qualifier-list*$_{opt}$~~
type-specifier-qualifier attribute-specifier-sequence$_{opt}$
type-specifier-qualifier specifier-qualifier-list

type-specifier-qualifier:
type-specifier
type-qualifier
alignment-specifier

struct-declarator-list:

struct-declarator

struct-declarator-list **,** *struct-declarator*

struct-declarator:

declarator

declarator$_{opt}$ **:** *constant-expression*

Add 6.7.2.1p6 to the Constraints section:

6 An attribute specifier sequence shall not appear in a struct-or-union specifier without a **struct** declaration list, except in a declaration of the form:

struct-or-union attribute-specifier-sequence identifier **;**

The attributes in the attribute specifier sequence, if any, are thereafter considered attributes of the **struct** or **union** whenever it is named.

Add 6.7.2.1p9 to the Semantics section:

9 The optional attribute specifier sequence in a struct-or-union specifier appertains to the structure or union type being declared. The optional attribute specifier sequence in a **struct** declaration appertains to each of the members declared by the **struct** declarator list; it shall not appear if the optional **struct** declarator list is omitted. The optional attribute specifier sequence in a specifier qualifier list appertains to the type denoted by the preceding type specifier qualifiers. The attribute specifier sequence affects the type only for the **struct** declaration or type name it appears in, not other types or declarations involving the same type.

Add 6.7.2.1p21:

21 Example 1 The following declarations illustrate the behavior when an attribute is written on a tag declaration:

```
struct [[deprecated]] S; // valid, [[deprecated]] appertains
                         // to struct S
void f(struct S *s);     // valid, the struct S type has
                         // the [[deprecated]] attribute
struct S {               // valid, struct S inherits the
  int a;                 // [[deprecated]] attribute from
};                       // the previous declaration
void g(struct [[deprecated]] S s); // invalid
```

Modify 6.7.2.2p1:

Drafting notes:

Because C and C++ do not allow the forward declaration of an enum type, the type specifier that does not define an enumeration is not allowed to specify any attributes. This is intentionally different than struct and union specifiers, which can be a forward declaration.

enum-specifier:

enum *attribute-specifier-sequence*$_{opt}$ *identifier*$_{opt}$ **{** *enumerator-list* **}**
enum *attribute-specifier-sequence*$_{opt}$ *identifier*$_{opt}$ **{** *enumerator-list* **,** **}**
enum *identifier*

enumerator-list:

enumerator
enumerator-list **,** *enumerator*

enumerator:

enumeration-constant *attribute-specifier-sequence*$_{opt}$
enumeration-constant *attribute-specifier-sequence*$_{opt}$ **=** *constant-expression*

Add 6.7.2.2p3 to the Semantics section:

3 The optional attribute specifier sequence in the **enum** specifier appertains to the enumeration; the attributes in that attribute specifier sequence are thereafter considered attributes of the enumeration whenever it is named. The optional attribute specifier sequence in the enumerator appertains to that enumerator.

Add 6.7.2.3p4 (to the Constraints section) and its accompanying footnote:

Drafting notes:

This constraint is intended to disallow type specifiers from specifying attributes while still allowing incomplete declarations to specify attributes. e.g.,

```
struct [[something]] x; /* valid */
void f(struct [[something]] s); /* invalid */
```

See 6.7.2.1p6 for similar specification.

4 A type specifier of the form

struct-or-union attribute-specifier-sequence$_{opt}$ *identifier*

shall not contain an attribute specifier sequence.[X)]

[X)] As specified in 6.7.2.1 above, the type specifier may be followed by a **;** or a **struct** declaration list.

Modify 6.7.2.3p7-10 (newly numbered):

7 A type specifier of the form

struct-or-union attribute-specifier-sequence$_{opt}$ *identifier*$_{opt}$ **{** *struct-declaration-list* **}**

or

enum *attribute-specifier-sequence*$_{opt}$ *identifier*$_{opt}$ **{** *enumerator-list* **}**

or

enum *attribute-specifier-sequence*$_{opt}$ *identifier*$_{opt}$ **{** *enumerator-list* **,** **}**

declares a structure, union, or enumerated type. The list defines the *structure content*, *union content*, or *enumeration content*. If an identifier is provided,[130] the type specifier also declares the identifier to be the tag of that type. The optional attribute specifier sequence appertains to the structure, union, or enumeration type being declared; the attributes in that attribute specifier sequence are thereafter considered attributes of the structure, union, or enumeration type whenever it is named.

8 A declaration of the form

struct-or-union attribute-specifier-sequence$_{opt}$ *identifier* **;**

specifies a structure or union type and declares the identifier as a tag of that type.[131] The optional attribute specifier sequence appertains to the structure or union type being declared; the attributes in that attribute specifier sequence are thereafter considered attributes of the structure or union type whenever it is named.

9 If a type specifier of the form

struct-or-union attribute-specifier-sequence$_{opt}$ *identifier*

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.[131]

10 If a type specifier of the form

struct-or-union attribute-specifier-sequence$_{opt}$ *identifier*

or

enum *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

6.7.4 and 6.7.5 Drafting notes:

`_Noreturn` and `_Alignas` are implemented as attributes in the C++ standard, rather than separate specifiers. In this draft, I have left the function and alignment specifiers alone to reduce drafting churn. It may make sense to alter these productions in a follow-up paper exploring the changes, though it should not result in a difference to existing code.

Modify 6.7.6p1:

Drafting notes:

It might make sense to allow an optional *attribute-specifier-seq* to precede the *type-qualifier-list* in an array [abstract] declarator with the same semantics as in a *type-qualifier-list*: the attributes would appertain to the pointer type formed by array-to-pointer decay. However, this would diverge from C++ by allowing `int a[ [[foo]] 5];`, which is invalid in C++ due to containing a `[[` that does not denote an attribute. For this reason, the syntax is not being proposed at this time.

declarator:

pointer$_{opt}$ *direct-declarator*

direct-declarator:

identifier *attribute-specifier-sequence*$_{opt}$

(*declarator* **)**

array-declarator attribute-specifier-sequence$_{opt}$

function-declarator attribute-specifier-sequence$_{opt}$

direct-declarator **(** *identifier-list*$_{opt}$ **)**

array-declarator:

direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**

direct-declarator **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**

direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**

direct-declarator **[** *type-qualifier-list*$_{opt}$ ***** **]**

function-declarator:

direct-declarator **(** *parameter-type-list* **)**

pointer:

***** *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$

***** *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:

type-qualifier

type-qualifier-list type-qualifier

parameter-type-list:

parameter-list

parameter-list **, ...**

parameter-list:

parameter-declaration

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

parameter-declaration:

attribute-specifier-sequence$_{opt}$ *declaration-specifiers declarator*

attribute-specifier-sequence$_{opt}$ *declaration-specifiers abstract-declarator*$_{opt}$

identifier-list:

identifier

identifier-list **,** *identifier*

Modify 6.7.6p5:

If, in the declaration "**T D1**", **D1** has the form

identifier *attribute-specifier-sequence*$_{opt}$

then the type specified for *ident* is **T** and the optional attribute specifier sequence appertains to **D1**.

Modify 6.7.6.1p1:

If, in the declaration "**T D1**", **D1** has the form

> ***** *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$ **D**

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

Modify 6.7.6.2p3:

If, in the declaration "**T D1**", **D1** has one of the forms:

> **D [** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]** *attribute-specifier-sequence*$_{opt}$
> **D [static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]** *attribute-specifier-sequence*$_{opt}$
> **D [** *type-qualifier-list* **static** *assignment-expression* **]** *attribute-specifier-sequence*$_{opt}$
> **D [** *type-qualifier-list*$_{opt}$ ***]** *attribute-specifier-sequence*$_{opt}$

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[142)] The optional attribute specifier sequence appertains to the array.
(See 6.7.6.3 for the meaning of the optional type qualifiers and the keyword **static**.)

Modify 6.7.6.3p5:

If, in the declaration "**T D1**", **D1** has the form

> **D(** *parameter-type-list* **)** *attribute-specifier-sequence*$_{opt}$

or

> **D(** *identifier-list*$_{opt}$ **)**

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list* T", then the type specified for *ident* is "*derived-declarator-type-list* function returning T". The optional attribute specifier sequence appertains to the function type.

Modify 6.7.6.3p13:

The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition. The optional attribute specifier sequence in a parameter declaration appertains to the parameter.

Modify 6.7.7p1:

type-name:

> *specifier-qualifier-list abstract-declarator*$_{opt}$

abstract-declarator:

pointer

pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:

(*abstract-declarator* **)**

array-abstract-declarator attribute-specifier-sequence$_{opt}$

function-abstract-declarator attribute-specifier-sequence$_{opt}$

array-abstract-declarator:

direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**

direct-abstract-declarator$_{opt}$ **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**

direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list* **static** *assignment-expression* **]**

direct-abstract-declarator$_{opt}$ **[*]**

function-abstract-declarator:

direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

Modify 6.7.7p2:

In several contexts, it is necessary to specify a type. This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.[151)] The optional attribute specifier sequence in a direct abstract declarator appertains to the preceding array or function type. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

Add new Subclause after 6.7.10.

Drafting notes:

This new subclause specifies the syntax and semantics of attributes in general, and is followed by sub-subclauses for each of the standardized attributes. Since this proposal is concerned only with the attribute syntax and semantics rather than specific attributes, no concrete attributes are included in this draft.

The primary concerns are that attributes are introduced as a list contained within double-square brackets (as individual tokens, rather than a single token). Attributes come in two forms, one is a single identifier which should only be used for standardized attributes and the other is a “scoped” form, which is a pair of identifiers delimited by double colons (as a single token) and should be used by implementations for implementation-defined attributes. Each implementation is recommended to select a unique identifier for their attribute namespace. Any attribute not specified by the standard is implementation-defined, and implementations are required to ignore unknown attributes. Each attribute specifies its own requirements on whether it accepts arguments or not, but the parsing constraints on arguments are left purposefully loose so that implementations have flexibility (for instance, an attribute argument could be arbitrary source code).

6.7.11 Attributes

1 Attributes specify additional information for various source constructs such as types, variables, identifiers, or blocks. They are identified by an *attribute token*, which can either be an *attribute prefixed token* (for implementation-specified attributes) or an identifier (for attributes specified in this document).

2 Support for any of the individual attributes specified in this document is implementation-defined and optional. For an attribute token (including an attribute prefixed token) not specified in this document, the behavior is implementation-defined. Any attribute token that is not supported by the implementation is ignored.

3 Attributes are said to appertain to some source construct, identified by the syntactic context where they appear (6.7, 6.8), and for each individual attribute, the corresponding clause constrains the contexts in which this appertainance is valid. The attribute specifier sequence appertaining to some source construct shall contain only attributes that are allowed to apply to that source construct.

4 In all aspects of the language, an attribute token specified by this document as an identifier `attr` and an identifier of the form `__attr__` shall behave the same when used as an attribute token, except for the spelling. [X)]

Recommended Practice

5 Implementations should support all attributes defined in this document.

6.7.11.1 General

Syntax

1 *attribute-specifier-sequence:*

attribute-specifier-sequence$_{opt}$ *attribute-specifier*

attribute-specifier:

[[*attribute-list* **]]**

attribute-list:

attribute$_{opt}$

attribute-list **,** *attribute*$_{opt}$

attribute:

attribute-token attribute-argument-clause$_{opt}$

attribute-token:

identifier

attribute-prefixed-token

attribute-prefixed-token:

attribute-prefix **::** *identifier*

attribute-prefix:

identifier

attribute-argument-clause:

(*balanced-token-sequence*$_{opt}$ **)**

balanced-token-sequence:

balanced-token

balanced-token-sequence balanced-token

balanced-token:

(*balanced-token-sequence*$_{opt}$ **)**

[*balanced-token-sequence*$_{opt}$ **]**

{ *balanced-token-sequence*$_{opt}$ **}**

any *token* other than a parenthesis, a bracket, or a brace

Semantics

2 An attribute specifier that contains no attributes has no effect. The order in which attribute tokens appear in an attribute list is not significant. If a keyword (6.4.1) that satisfies the syntactic requirements of an identifier (6.4.2) is contained in an attribute token, it is considered an identifier. A strictly conforming program using an attribute token other than an attribute prefixed token remains strictly conforming in the absence of that attribute token.[Y)]

3 Note For each individual attribute, the form of balanced token sequence will be specified.

Recommended Practice

4 Each implementation should choose a distinctive name for the attribute prefix in an attribute prefixed token. Implementations should not define attributes without an attribute prefixed token unless the attribute is specified in this document.

Add new footnotes to 6.7.11:

[X)] Thus, the attributes `[[nodiscard]]` and `[[__nodiscard__]]` can be freely interchanged. Implementations are encouraged to behave similarly for attribute tokens (including attribute prefixed tokens) they provide.

[Y)] Attributes specified by this document can be parsed but ignored by an implementation without changing the semantics of a correct program; the same is not true for attributes not specified by this document.

6.8 Statements and blocks

Drafting notes:

Introduce optional attribute specifier sequences that precede the statement. The attribute will appertain to the statement itself. In the case of labels, ensure that the attribute appertains to the declaration of the label name rather than the subsequent statement being labeled.

Modify 6.8p1:

statement:

labeled-statement

~~*compound-statement*~~

expression-statement
attribute-specifier-sequence$_{opt}$ *compound-statement*
attribute-specifier-sequence$_{opt}$ *selection-statement*
attribute-specifier-sequence$_{opt}$ *iteration-statement*
attribute-specifier-sequence$_{opt}$ *jump-statement*

Modify 6.8p2:

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

Modify 6.8.1p1:

labeled-statement:
attribute-specifier-sequence$_{opt}$ *identifier* **:** *statement*
attribute-specifier-sequence$_{opt}$ *case constant-expression* **:** *statement*
attribute-specifier-sequence$_{opt}$ *default* **:** *statement*

Modify 6.8.1p4:

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

Modify 6.8.3p1:

Drafting notes:

This is required to prevent ambiguous parses with the *attribute-declaration* production through *declaration*, the result is:

```
[[something]]; // Parses as an attribute-declaration.

void func(void) {
  [[something]]; // Parses as an attribute-declaration.
  [[something]]1; // Parses as an expression-statement.
}
```

expression-statement:
expression$_{opt}$ ***;***
attribute-specifier-sequence expression ***;***

Modify 6.8.3p2:

The attribute specifier sequence appertains to the expression. The expression in an expression statement is evaluated as a void expression for its side effects.

6.9 External Definitions

Modify 6.9.1p1:

function-definition:

attribute-specifier-sequence$_{opt}$ *declaration-specifiers declarator*
declaration-list$_{opt}$ *compound-statement*

declaration-list:

no-leading-attribute-declaration
declaration-list no-leading-attribute-declaration

Add 6.9.1p7 to the Semantics section:

7 The optional attribute specifier sequence in a function definition appertains to the function.

7.1.3 Reserved Identifiers

Drafting notes:

C++ disallows a standard *attribute-token* from being defined or undefined as a macro by the user. The *attribute-token*s are not reserved (it is not UB to have an identifier with the same name as an *attribute-token*), but disallowing the interactions with macros eases the burden on standard library implementations.

Modify 7.1.3p2:

No other identifiers are reserved. If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 7.1.4), or defines a reserved identifier or attribute token described in 6.7.11 as a macro name, the behavior is undefined.

Modify 7.1.3p3:

If the program removes (with `#undef`) any macro definition of an identifier in the first group listed above or attribute token described in 6.7.11, the behavior is undefined.

Acknowledgements

I would like to recognize the following people for their help in this work: Clark Nelson, David Keaton, David Svoboda, Jens Gustedt, Jens Maurer, Joseph Myers, Martin Sebor, Michael Wong, and Richard Smith.

References

[N1229]
Potential Extensions For Inclusion In a Revision of ISO/IEC 9899. <unknown>. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1229.pdf

[N1264]
Potential Extensions For Inclusion In a Revision of ISO/IEC 9899. <unknown>. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1264.pdf

[N1403]
Towards support for attributes in C. David Svoboda. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1403.pdf

[N2021]
C - Preliminary C2x Charter. David Keaton. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2021.htm

[WG21 N2761]
Towards support for attributes in C++. Jens Maurer, Michael Wong. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf

[SC22WG14.14310]
attributes. David Keaton. http://www.open-std.org/jtc1/sc22/wg14/14310