Attributes in C

Reply-to: Aaron Ballman (aaron@aaronballman.com; aballman@cert.org)
Document No: N2049
Date: 2016-06-27

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.

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.

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++ [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, name, statement, etc. Concrete examples include:

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

`attr1` appertains to the declarator-ids `s1` and `s2`, `attr2` appertains to the declaration of `struct S`, `attr3` appertains to the type `struct S`, `attr4` appertains to the declarator-id `s1`, and `attr5` appertains to the declarator-id `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 name `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 a name.

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

`attr1` appertains to the variable declaration `i` (in both cases).

```
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.

Proposed Wording

This document does not currently cover proposed wording changes at this time, but if the WG14 committee believes the direction to be an appropriate one, proposed wording changes will follow in a separate revision of this document.

Acknowledgements

I would like to recognize the following people for their help in this work: David Keaton, David Svoboda, Jens Maurer, and Michael Wong. I would also like to thank the US Department of Homeland Security, without whose funding this proposal would not have been made.

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

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