Proposal for C2x
WG14 N2481

Title:	Querying attribute support
Author, affiliation:	Aaron Ballman, GrammaTech
Date:	2020-02-10
Proposal category:	New features
Target audience:	General developers, compiler/tooling developers

Abstract: Users with cross-platform code bases need the ability to interrogate a given implementation to determine whether an attribute is supported or not. This provides a preprocessor mechanism to perform that interrogation.

Prior art: C++ standardized this feature in C++2a, spelled __has_cpp_attribute. Clang is shipping this feature under the proposed spelling.

Querying attribute support

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2481
Revises Document No: N2411
Date: 2020-02-10

Summary of Changes

N2481

- Added information about the NB comment resolution from WG21 Belfast.

N2411

- Added the new 6.10.1p4 paragraph to allow `__has_c_attribute` to be treated like a macro for `#ifdef`, `#ifndef`, and `#if defined` support.
- Update the values for the concrete attributes to be based on when they were adopted (if they were adopted) or a placeholder value.
- Updated 6.10.8p2 to disallow `#define` or `#undef` of `__has_c_attribute`.
- Added rationale describing why `__has_c_attribute` should behave like `defined` as opposed to being a function-like macro.
- Updated the original 6.10.1p4 to give semantics to macro expansion of `__has_c_attribute`.

N2333

- Original proposal.

Introduction

To keep code portable, especially for library code which may be consumed by implementations unknown or unavailable to the author of the code, users need the ability to interrogate a compiler to determine whether an attribute [N2269] is supported or not. The proposed `__has_c_attribute` function-like macro provides users with a way to query whether an implementation supports a given attribute.

Rationale

The list of supported attributes changes with each revision of the C Standard and implementations are allowed to provide their own attributes. Users writing portable code may wish to guard against quality implementations diagnosing use of unknown or unsupported attributes while still preserving the functionality provided by the attribute where possible. Additionally, some implementations may provide a non-attribute fallback for the desired functionality, possibly allowing a user's code base to degrade more gracefully in the absence of support for an attribute.

Proposal

C++ uses the `__has_cpp_attribute` function-like macro to interrogate an implementation's support for an attribute [P0941R2], and this document proposes adding the `__has_c_attribute` function-like macro for the same purposes. Separate macros are useful to allow independent queries for both C and

C++ code, as an attribute may be supported in only one of the two languages for a given implementation.

This function-like macro is intended to be used in conjunction with other user-defined macros exposing the attribute, as in the following example.

```
/* Fallback for compilers not yet implementing this feature. */
#ifndef __has_c_attribute
#define __has_c_attribute(x)  0
#endif /* __has_c_attribute */

#if __has_c_attribute(fallthrough)
/* Attribute is available, use it. */
#define FALLTHROUGH [[fallthrough]]
#else
/* Fallback implementation. */
#define FALLTHROUGH
#endif
```

This feature-test macro can be used with either a standards-based attribute or with a vendor-supplied attribute. The result of the macro expansion will be 0 if the attribute is unknown or unsupported, and will return nonzero if the attribute is supported for that build configuration. Standards-based attributes will return the latest date of modification to the standard for that attribute (e.g., `201811L` if an attribute was voted into the standard in Nov 2018) in order to allow more fine-grained feature testing capabilities should an attribute evolve over time. For example, imagining a hypothetical situation in which the deprecated attribute was standardized as not taking a string argument until Nov 2018:

```
#if __has_c_attribute(deprecated) >= 201811L
#define DEPRECATED(MSG) [[deprecated(MSG)]]
#elif __has_c_attribute(deprecated)
#define DEPRECATED(MSG) [[deprecated]]
#else
/* Fallback implementation; perhaps use an alternative impl. */
#define DEPRECATED(MSG)
#endif
```

C++ lists attribute feature test macro values alongside a list of other, non-attribute feature testing macros. Given that C does not have a similar list serving the same purpose, this document proposes adding the feature testing macro value for standards-based attributes as a normative paragraph attached to each attribute.

C++ based the behavior of `__has_cpp_attribute` on the `defined` preprocessor token, where it is only available for use within a conditional inclusion expression. This proposal is consistent with the C++ treatment and does not propose adding this as a predefined macro available outside of the preprocessor. It is valuable for C to follow suit because restricting the usage to only preprocessor conditionals does not interfere with any intended use cases; runtime branching based on the existence of an attribute is not intended to be easily supported.

Why Two Macros?

The question was asked at the London 2019 meeting as to why we need two separate macros for `__has_c_attribute` and `__has_cpp_attribute` given that a translation unit can only be compiled for one language at a time. It was observed that it would be easier for users were there to be a single macro that answers the question "is this attribute supported"? Given that `__has_cpp_attribute` has not been released in a standard, WG21 had an opportunity to rename the macro in C++ to a language-agnostic name to provide a better user experience for users of both languages. Consequently, an NB comment was raised [US129] requesting that WG21 reconsider their choice of names for the macro. The Evolution Working Group rejected the comment saying there was no credible argument that such an approach was possible and that they wished to leave the door open to having incompatible attribute semantics under the same attribute name in both languages. Regardless of the validity of the claims that this is not possible, the reality is that C++20 is expected to be standardized using the name `__has_cpp_attribute` over our objections. While other approaches for querying attribute support are possible, the model provided by `__has_cpp_attribute` is familiar to users and has proven implementation experience. Because of this, the proposal continues to use `__has_cpp_attribute` as the model. Given that an identifier like `__has_cpp_attribute` would be deeply confusing to C users, this proposal continues to use `__has_c_attribute` as the name of the macro, which already has some early implementation experience in Clang [Clang]. Having distinct names for the macros also obviates the need to add additional object-like macros for feature release dates to handle adopting attributes (or their features) into their respective standards at different dates.

Proposed Wording

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

Modify 6.10.1p1, splitting it into two paragraphs:

1 The expression that controls conditional inclusion shall be an integer constant expression except that: identifiers (including those lexically identical to keywords) are interpreted as described below.~~;~~[167] ~~and it~~

2 The conditional inclusion expression may contain unary operator expressions of the form

`defined` *identifier*

or

`defined (` *identifier* `)`

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject identifier), 0 if it is not.

Add a third paragraph to the Constraints section after the new 2nd paragraph from above:

3 The conditional inclusion expression may contain unary operator expressions of the form

`__has_c_attribute (` *pp-tokens* `)`

which are replaced by a nonzero *pp-number* matching the form of an *integer-constant* if the implementation supports an attribute with the name specified by interpreting the *pp-tokens* as an *attribute-token*, and by 0 otherwise. The *pp-tokens* shall match the form of an *attribute-token*.

Add a fourth paragraph as the initial paragraph in the Semantics section:

4 The `#ifdef` and `#ifndef` directives, and the `defined` conditional inclusion operator, shall treat `__has_c_attribute` as if it was the name of a defined macro. The identifier `__has_c_attribute` shall not appear in any context not mentioned in this subclause.

Modify 6.10.1p4 (as originally numbered):
Drafting note: I do not think we need to make it explicitly undefined to generate `__has_c_attribute` through macro replacement because of the "shall not appear in any context" above. The C++ standard has similar words in [cpp.cond]p10 but they do not mention __has_cpp_attribute in that paragraph.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and evaluations of the `defined` and `__has_c_attribute` unary operators have been performed, all remaining identifiers (including those lexically identical to keywords) are replaced with the pp-number 0, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types `intmax_t` and `uintmax_t` defined in the header <stdint.h>.[168)] This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.[169)] Also, whether a single-character character constant may have a negative value is implementation-defined.

Add new paragraphs after 6.10.1p6:

8 EXAMPLE

```
/* Fallback for compilers not yet implementing this feature. */
#ifndef __has_c_attribute
#define __has_c_attribute(x)  0
#endif /* __has_c_attribute */

#if __has_c_attribute(fallthrough)
/* Standard attribute is available, use it. */
#define FALLTHROUGH [[fallthrough]]
#elif __has_c_attribute(vendor::fallthrough)
/* Vendor attribute is available, use it. */
#define FALLTHROUGH [[vendor::fallthrough]]
#else
/* Fallback implementation. */
```

```
#define FALLTHROUGH
#endif
```

Modify 6.10.8p2:

Drafting note: This is intended to continue to allow the common coding pattern:

```
#ifndef __has_c_attribute
#define __has_c_attribute(x) 0
#endif
```

because 6.10.1p4 allows `__has_c_attribute` *to appear in the* `#ifndef`*. The prohibition added here prevents subversion of* `__has_c_attribute` *behavior when the feature is supported by an implementation while still allowing useful idioms in practice when an implementation does not yet support* `__has_c_attribute`*.*

2 None of these macro names, nor the identifiers **defined** or `__has_c_attribute`, shall be the subject of a **#define** or a **#undef** preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

Add a new paragraph after 6.7.11.1p3, to be applied only if the `deprecated` attribute is adopted. The editors are expected to replace the given value with the appropriate date of adoption for the attribute:

4 The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value `20YYMML` when given `deprecated` as the *pp-tokens* operand.

Add a new paragraph after 6.7.11.2p1, to be applied only if the `fallthrough` attribute is adopted. The editors are expected to replace the given value with the appropriate date of adoption for the attribute:

2 The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value `201904L` when given `fallthrough` as the *pp-tokens* operand.

Add a new paragraph after 6.7.11.3p2, to be applied only if the `maybe_unused` attribute is adopted. The editors are expected to replace the given value with the appropriate date of adoption for the attribute:

3 The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value `201904L` when given `maybe_unused` as the *pp-tokens* operand.

Add a new paragraph after 6.7.11.4p1, to be applied only if the `nodiscard` attribute is adopted. The editors are expected to replace the given value with the appropriate date of adoption for the attribute:

2 The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value `201904L` when given `nodiscard` as the *pp-tokens* operand.

Acknowledgements

I would like to acknowledge the following people for their contributions to this work: Jens Gustedt, Joseph Myers, Martin Sebor, and Richard Smith.

References

[P0941R2]
Integrating feature-test macros into the C++WD (rev. 2). Ville Voutilainen, Jonathan Wakely. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0941r2.html

[N2269]
Attributes in C. Aaron Ballman. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2269.pdf

[US129]
US129 15.1 Rename __has_cpp_attribute to __has_attribute. https://github.com/cplusplus/nbballot/issues/128

[Clang]
Language Extensions. https://clang.llvm.org/docs/LanguageExtensions.html#has-c-attribute