On the ignorability of standard attributes

Timur Doumler (papers@timur.audio)

Document #:	P2552R1
Date:	2022-11-15
Project:	Programming Language C++
Audience:	Evolution Working Group, Core Working Group

Abstract

There is a general notion in C++ that standard attributes should be *ignorable*. However, currently there does not seem to be a common understanding of what "ignorable" means, and the C++ standard itself is ambiguous on this matter. In this paper, we consider three aspects of ignorability: syntactic ignorability, semantic ignorability, and the behaviour of `__has_cpp_attribute`. We discuss where and how the C++ standard is underspecified and why that is problematic, survey existing implementation practice, and propose different options to resolve existing ambiguities.

1 Motivation

The C++ standard says about the ignorability of attributes ([dcl.attr.grammar]/6):

> For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined. Any *attribute-token* that is not recognized by the implementation is ignored.

This wording is ambiguous. It is not clear at all whether the intent is to allow the implementation to ignore any *attribute-token not specified in this document* (i.e. only non-standard attributes), or any *attribute-token, including those specified in this document* (i.e. including standard attributes). This ambiguity is a known defect: there is a Core issue [CWG2538] and a recent NB comment (GB 9.12.1p6).

Standard attributes are a feature shared between C and C++. The C standard says:

> A strictly conforming program using a standard attribute remains strictly conforming in the absence of that attribute. [...] Standard 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.

It is clear from the C wording that the intent is for standard attributes to be ignorable, however it is not entirely clear what that means: "parse but ignore" implies that the compiler needs to at least parse them (i.e. it cannot treat a standard attribute as token soup). So the C standard might be talking about a different kind of ignorability than the C++ standard does (ignoring the *attribute-token*).

Before we can try to fix the defect in the C++ wording, we need to answer two questions:

— Should an implementation be allowed to ignore a standard attribute?

— What does it mean to ignore a standard attribute?

We will start by defining what *ignore* means. There are different properties of standard attributes that could or could not be declared *ignorable*, with different consequences for the standard. In particular, we can draw a distinction between *syntactic ignorability* (i.e., ignoring the form of the argument clause, the attribute's appertainment, and so forth) and *semantic ignorability* (i.e., ignoring the effect that the attribute would have on the program).

Whether standard attributes are *syntactically* ignorable is a matter of contention. At the heart of the issue is the question whether a compiler is required to properly parse a standard attribute (which includes syntax-checking the argument clause, appertainment, and so forth) even if it then does not implement any semantics for that attribute.

On the other hand, it is uncontroversial that attributes are meant to be *semantically* ignorable. However, we have a problem here as well: it is not quite clear what semantic ignorability means exactly. The principle of semantic ignorability is currently some kind of gentlemens' agreement that originated in the standardisation of attribute syntax for C++11 [N2761]. This agreement, however, is not codified anywhere. The rule exists only implicitly and can be interpreted in different ways. Such manifest ambiguity is anathema to sound and consistent language design.

Finally, the behaviour of `__has_cpp_attribute` is ambiguous as well. In particular, it is unclear whether `__has_cpp_attribute` should return a positive value for a standard attribute if the compiler is aware of the attribute and can parse it correctly, but does not implement any useful semantics for it. There is current implementation divergence on this point, so the standard should specify the correct behaviour.

2 The status quo

2.1 Syntactic ignorability

2.1.1 Argument clause

The C++ grammar defines the *attribute-argument-clause* of an attribute to have the form:

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

where *balanced-token-seq* is any token sequence with balanced parentheses, square brackets, and curly braces. This base grammar allows for a wide variety of possible arguments for standard attributes. It is up to the specification of each individual attribute to constrain the grammar for its arguments further ([dcl.attr.grammar]/4):

> [...] The *attribute-token* determines additional requirements on the *attribute-argument-clause* (if any).

Every standard attribute specifies explicitly whether it can have an argument clause, whether this argument clause is optional or mandatory, and what form the argument clause shall have, for example in [dcl.attr.noreturn/1]:

> The *attribute-token* `noreturn` specifies that a function does not return. No *attribute-argument-clause* shall be present.

or in [dcl.attr.deprecated]/1:

> The *attribute-token* `deprecated` can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. An *attribute-argument-clause* may be present and, if present, it shall have the form:

(*string-literal*$_{opt}$)

On the one hand, this wording is all normative; it therefore seems that a program violating these requirements should be ill-formed, and a conforming compiler must emit a diagnostic. On the other hand, due to the ambiguity in [dcl.attr.grammar]/6, it is unclear whether [dcl.attr.grammar]/6 overrides these requirements and allows an implementation to completely ignore the argument clause:

```
[[noreturn("cannot have a reason")]] int f();                  // Ill-formed or ignorable?
[[deprecated(not_a_string)]] int g();                          // Ill-formed or ignorable?
[[nodiscard(this?is!a:balanced%{token[sequence]})]] int h();   // Ill-formed or ignorable?
```

Existing practice

Clang, GCC, ICC, and MSVC are all very good at diagnosing syntax errors in the argument clause. We tried many different ill-formed constructions like the above and got a diagnostic on all four compilers in all cases. The only questionable (but still conforming) case we found was `[[carries_dependency(some_argument)]]` on GCC, where the emitted diagnostic said that the `carries_dependency` attribute is not supported, but did not specifically call out the syntax error in the argument clause.

2.1.2 Appertainment

On the appertainment of a standard attribute, [dcl.attr.grammar]/5 says:

> Each *attribute-specifier-seq* is said to *appertain* to some entity or statement, identified by the syntactic context where it appears. If an *attribute-specifier-seq* that appertains to some entity or statement contains an *attribute* or *alignment-specifier* that is not allowed to apply to that entity or statement, the program is ill-formed.

Every standard attribute has normative requirements on appertainment. For example, `noreturn` "may be applied to a function or a lambda call operator" ([dcl.attr.noreturn]/1); `no_unique_address` "may appertain to a non-static data member other than a bit-field" ([dcl.attr.nouniqueaddr]/1); `fallthrough` "may be applied to a null statement" ([dcl.attr.fallthrough]/1); and so forth.

Similarly to syntax errors in the argument clause, whether [dcl.attr.grammar]/6 allows the compiler to ignore these appertainment rules is currently ambiguous:

```
int main() {
  [[fallthrough]] int i;  // Ill-formed or ignorable?
}
```

Existing practice

We found that generally, Clang, GCC, ICC, and MSVC are very good at diagnosing appertainment errors as well. But, unlike with argument clause errors, with appertainment errors we did find some false negatives on all four compilers. For example, no compiler diagnoses `[[deprecated]]` or `[[maybe_unused]]` on static data members, and GCC allows any standard attribute to appertain to an empty declaration at class scope without warning:

```
struct X { [[nodiscard]]; };  // no diagnostic on GCC
```

The code triggering those false negatives, however, is typically quite obscure. Moreover, we could not find any cases on any compiler where the failure to diagnose appertainment rules introduced a bug or changed the behaviour of a program.

2.1.3 Additional syntactic requirements

Some standard attributes have additional normative syntactic requirements on top of syntactic rules for the argument clause and appertainment. In particular, [dcl.attr.likelihood]/1 constraints which *attribute-token*s can appear in an *attribute-specifier-seq*:

> The *attribute-token* `likely` shall not appear in an *attribute-specifier-seq* that contains the *attribute-token* `unlikely`.

and ([dcl.attr.fallthrough]/1) specifies:

> A fallthrough statement may only appear within an enclosing `switch` statement. The next statement that would be executed after a fallthrough statement shall be a labeled statement whose label is a case label or default label for the same `switch` statement and, if the fallthrough statement is contained in an iteration statement, the next statement shall be part of the same execution of the substatement of the innermost enclosing iteration statement. The program is ill-formed if there is no such statement.

Just as with the other requirements, the question here is whether a program violating these syntactic requirements is ill-formed, or whether [dcl.attr.grammar]/6 allows ignoring such violations.

Existing practice

The case of `likely` and `unlikely` appearing in the same *attribute-specifier-seq* is reliably diagnosed as ill-formed by all of Clang, GCC, ICC, and MSVC. On the other hand, the additional rules for `fallthrough` are not consistently diagnosed. In [dcl.attr.fallthrough]/3, the C++ standard gives a code example that contains four syntax errors explicitly marked as such:

```
void f(int n) {
  void g(), h(), i();
  switch (n) {
  case 1:
  case 2:
    g();
    [[fallthrough]];
  case 3:                  // warning on fallthrough discouraged
    do {
      [[fallthrough]];     // error: next statement is not part of the same substatement execution
    } while (false);
  case 6:
    do {
      [[fallthrough]];     // error: next statement is not part of the same substatement execution
    } while (n--);
  case 7:
    while (false) {
      [[fallthrough]];     // error: next statement is not part of the same substatement execution
    }
  case 5:
    h();
  case 4:                  // implementation may warn on fallthrough
    i();
    [[fallthrough]];       // error
  }
}
```

Only ICC and Clang diagnose all four syntax errors. GCC only diagnoses the second and the fourth, and MSVC diagnoses only the fourth.

2.1.4 Expression parsing and ODR-use

With attribute `assume` [P1774R8], we added an attribute to C++23 that contains an expression in its argument clause. Having an attribute that includes an expression brings with it several interesting consequences, which are likewise affected by the current ambiguity in [dcl.attr.grammar]/6.

First, to detect syntax errors inside the expression such as

```
void f(int i) {
  [[assume(i >=)]];  // Ill-formed or ignorable?
}
```

the compiler has to parse expression grammar inside the attribute's argument clause; merely treating the argument clause as a *balanced-token-sequence* is not enough (it would be enough for the other standard attributes having an argument clause, `deprecated` and `nodiscard`, since their argument is merely a *string-literal*). One compiler vendor, MSVC, has told us that this is technically challenging for them to implement (see also NB comment FR 9.12.3). On the other hand, two other vendors, GCC and Clang, have told us that their compilers have no problem parsing expressions inside an attribute's argument clause, and in fact this is already existing practice for their vendor-specific non-standard attributes. MSVC itself also does not seem to have a problem with parsing expressions inside other constructs such as `__declspec(...)`.

Apart from syntax errors in the expression grammar, expression parsing also involves ODR-use of the entities in the expression, which can trigger template instantiations. If such an instantiation in turn triggers a failing `static_assert`, the program would be rendered ill-formed as well:

```
template <typename T>
struct X {
  static_assert(sizeof(T) > 1);
  bool f() { return true; }
};

int main() {
  [[assume(X<char>().f())]];  // Ill-formed or ignorable?
}
```

In addition, ODR-use can also trigger lambda capture, which is observable both at compile time and at run time. We can even construct an example where the lambda capture has an effect on the layout of a class:

```
constexpr auto f(int i) {
  return sizeof( [=] { [[assume(i == 0)]]; } );
}

struct X {
  char data[f(0)];
};
```

Here, `sizeof(X)` and therefore the ABI of `struct X` will depend on whether the `assume` is syntactically ignored.

Of course, this code example is a highly contrived usage of assumptions. In real code, it is not useful to use a variable *only* in an assumption, but not anywhere else in the surrounding code. So, in real-world usage, the assumption would never end up triggering the lambda capture (and if anyone were to write such code, they would have much bigger problems than the class layout of `struct X`). Note also that changing the layout of a class through a language construct is nothing new: `[[no_unique_address]]`, `assert`, and other constructs can also trigger changes in class layout. This possibility of affecting class layout does not cause any problems in practice as far as we know. But we still need to define the exact behaviour: is the implementation required to ODR-use the

expression, therefore triggering the template instantiations and lambda captures, or may ODR-use be skipped?

If [dcl.attr.grammar]/6 is interpreted as "only non-standard attributes can be syntactically ignored" (following the suggested resolution of [CWG2538]), then in the template instantiation example, the compiler must instantiate the template, and must trigger the failing `static_assert` (or whatever other effects on the program the template instantiation will cause). In the lambda capture example, the compiler must perform the lambda capture, and therefore change the layout of the class, even if the compiler then decides to ignore the attribute *semantically*, i.e. not implement an assumptions facility.

On the other hand, if [dcl.attr.grammar]/6 is interpreted as "all attributes, including standard attributes, can be syntactically ignored", the compiler is free to not ODR-use the entities in the expression, not perform the template instantiations or the lambda capture, and in fact not parse the expression at all, but to treat the entire thing as token soup, and just skip over it.

Note that the question of whether an expression must be ODR-used is in no way related to the *semantics* of the `assume` attribute, which is entirely orthogonal. This question concerns the design space of attributes as a whole; `assume` just happens to be the only standard attribute currently containing an expression. Any attribute containing an expression would run into the same question, such as the proposed `trivially_relocatable` attribute [P1144R5], or a hypothetical attribute-like syntax for Contracts.

Existing practice

`assume` was added recently for C++23. GCC already implements it with the standard attribute syntax. The other three major compilers implement the same functionality as a built-in: `__assume` on MSVC and ICC, and `__builtin_assume` on Clang.

On all four compilers, regardless of whether the attribute syntax or the built-in syntax is used, the expression in the argument clause is always ODR-used, and the side effects of this ODR-use (such as lambda captures changing class layout) are always triggered. Interestingly, the ODR-use also happens when the actual assumption is then semantically ignored by the compiler, as is the case on Clang for expressions that have side effects. This behaviour is consistent with the interpretation that standard attributes can be ignored only semantically, but not syntactically.

2.2 Semantic ignorability

2.2.1 Design of existing standard attributes

The original paper that introduced attributes to C++ [N2761] says — somewhat vaguely — that a standard attribute should be "something that helps but can be ignorable with little serious side-effects", but that paper does not mention a strict rule of semantic ignorability. The paper also contains a list of possible future features, indicating which ones in the opinion of the authors at the time would be good or bad candidates for a standard attribute.

The list in [N2761] contains `alignas` as a good candidate: `alignas` was initially proposed as an attribute, but later changed to a keyword before C++11 was finalised [N3190]. The agreement had evolved: attributes should now be features that are semantically ignorable in the strict sense, i.e., their effect on the program is optional, and `alignas` does not fit the bill: its effects on the alignment of an object are mandatory, not optional.

The original paper also says that attributes should appertain to declarations only, not statements, which also changed later: `likely`, `unlikely`, `fallthrough`, and `assume` can all apply to statements (the latter two only to null statements). What should or should not be an attribute has clearly evolved over the years, so we should base our rules on the attributes that have been standardised so far.

2.2.2 Semantic categories of existing standard attributes

All standard attributes are currently normatively specified in such a way that they are syntactically ignorable. However, how this ignorability is achieved varies from attribute to another. We can distinguish four different categories:

— Attributes that produce or suppress diagnostics and otherwise have no effect: `deprecated`, `fallthrough`, `maybe_unused`, and `nodiscard`. These attributes are normatively defined to do nothing. The desired effect is described in a section called "recommended practice".

— Attributes that serve as optimisation hints to the compiler and otherwise have no effect: `likely`, `unlikely`, and `carries_dependency`. These attributes are also defined to do nothing and have a "recommended practice" section.

— Attributes that can turn defined behaviour into undefined behaviour: `noreturn` and `assume`. These attributes are semantically ignorable because undefined behaviour means the implementation can do literally anything, including ignoring the effects of the attribute and compiling and executing the program as if they were not there.

— Attributes that change the semantics of the program in an observable way. We currently have only one such attribute: `no_unique_address`. This attribute is semantically ignorable because its effect is carefully specified to be so: it introduces a *potentially-overlapping subobject*, i.e. a subobject that either *is* or is *not* overlapping, depending on whether the compiler chooses to implement or semantically ignore the attribute.

2.2.3 Attempts to define a semantic ignorability rule

Unfortunately, there is currently no explicit definition of what constitutes an attribute and what it means for it to be semantically ignorable. As a result, different people have a different mental model.

Some say that semantic ignorability means that a program has *the same* behaviour (or *identical* semantics) with or without the attribute. This characterisation is clearly wrong: it applies to only the first two of the four categories listed above. Constructing a counterexample is easy:

```
[[noreturn]] int f() { return 0; }
int main() { return f(); }
```

This program returns `0` without the attribute, but has undefined behaviour with the attribute, which means that adding the attribute can change the behaviour, and will often do so in practice.

Others say that — and this is the version we hear most often — given a well-formed program, omitting an attribute does not change the observable behaviour (or the semantics) of the program. However, this characterisation too is wrong, and we can again construct a counterexample:

```
struct X {};
struct Y {
  [[no_unique_address]] X x;
  int i;
};

int main() { return (sizeof(Y) == sizeof(int)); }
```

This program might return `1` *with* the attribute, but it will *always* return `0` *without* the attribute. However, because the compiler is not required to implement the semantic effects of `no_unique_address`, the program may also return `0` with the attribute. If we add a

```
static_assert (sizeof(Y) == sizeof(int));
```

we get an even more obvious violation of the rule above: this program might or might not be ill-formed with the attribute, depending on whether the compiler implements it, but is definitely ill-formed without the attribute. Therefore, the omission of the attribute can render a program ill-formed.

If we had codified a rule earlier, we might have ended up with the rule above (given a well-formed program, omitting an attribute does not change the behaviour/semantics of the program). As a result, we would perhaps never have standardised `no_unique_address`, which violates this rule, as an attribute; however, that ship sailed with C++20. In order to find a rule for semantic ignorability that matches existing practice, we need to take a closer look at what it actually means when we say that two programs have "the same semantics" or "the same behaviour".

2.2.4 Regions of program behaviour

The standard distinguishes undefined behaviour, unspecified behaviour, and implementation-defined behaviour. Let us call the behaviour that falls into none of these regions, i.e. the behaviour that is fully specified by the standard itself, the *mandated behaviour* of a C++ program. An example for mandated behaviour is `sizeof(char)`, which must evaluate to `1`.

We can then distinguish the following regions of program behaviour:

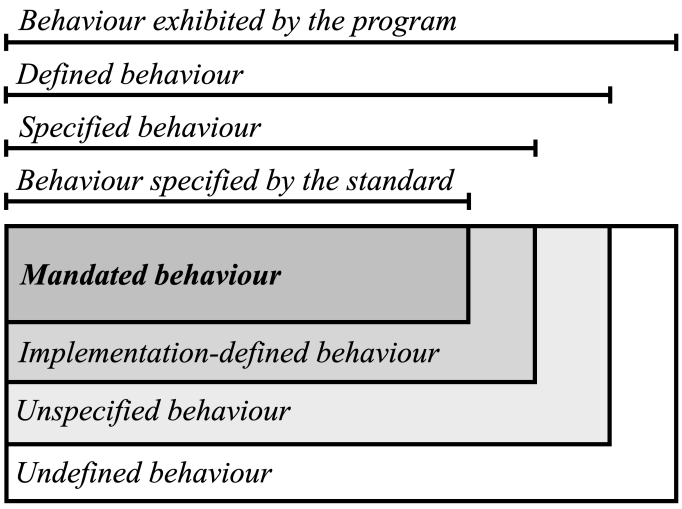

Figure 1: Regions of behaviour of a C++ program

The *specified behaviour* is the union of mandated behaviour (specified by the standard) and implementation-defined behaviour (specified by the implementation). The *defined behaviour*, i.e. behaviour that is not undefined, is the union of specified behaviour and unspecified behaviour. Finally, the full set of behaviour exhibited by a C++ program is the union of defined behaviour and undefined behaviour.

2.2.5 Proposed rule for semantic ignorability

With this framework in place, we can now formulate a rule for semantic ignorability of standard attributes:

> Given a well-formed program, omitting all occurrences of a particular standard attribute shall result in a well-formed program that exhibits the same *mandated behaviour* as the original program.

It is easy to see that the above rule works for all categories of standard attributes we identified in section 2.2.2: semantically ignoring any of them does not change the *mandated behaviour* of a C++ program in any way. For `no_unique_address` in particular, [intro.object] says:

> A *potentially-overlapping subobject* is either:
>
> — a base class subobject, or
>
> — a non-static data member declared with the `no_unique_address` attribute.
>
> An object has nonzero size if it
>
> — is not a potentially-overlapping subobject, or
>
> — is not of class type, or
>
> — is of a class type with virtual member functions or virtual base classes, or
>
> — has subobjects of nonzero size or unnamed bit-fields of nonzero length.
>
> Otherwise, if the object is a base class subobject of a standard-layout class type with no non-static data members, it has zero size. Otherwise, the circumstances under which the object has zero size are implementation-defined.

Therefore, in the code example above, whether `sizeof(Y) == sizeof(int)` holds is implementation-defined, and therefore neither option constitutes mandated behaviour of the program.

Conversely, for `alignas`, which sits in the same space in the grammar as standard attributes, but is not an attribute, [dcl.align]/4 says:

> The alignment requirement of an entity is the strictest nonzero alignment specified by its *alignment-specifier*s, if any; otherwise, the *alignment-specifier*s have no effect.

The effect that the alignment requirement of an entity has is fully defined by the standard: it constitutes mandated behaviour of the program. Therefore, `alignas` cannot be an attribute.

Codifying the proposed rule would enable consistent language design going forward, which would greatly simplify future discussions about what should and should not be an attribute. Having such a rule would also help guide language design for other language features such as Contracts [P2521R2]. One of the possible syntaxes for contract annotations is an attribute-like syntax [P2487R0]. If we choose this syntax, we should be consistent with attribute ignorability semantics, too. We believe that the framework we developed here for standard attributes can be used for specifying contracts checks as well.

Finally, we believe that the rule proposed here is compatible with, but more precise than the rule in the C language (see section 1).

Existing practice

Which attributes are being semantically ignored in practice by today's major compilers? Again, we looked at the latest available versions of Clang, GCC, ICC, and MSVC. It seems that none of them implement any semantics for `carries_dependency` (so perhaps we should not have standardised `carries_dependency`, but that is another story).

In addition, MSVC does not implement any semantics for `no_unique_address`, which has the consequence that class layout is inconsistent across different compilers on the same platform.

All other standard attributes seem to have semantically functional implementations on all major compilers.

2.3 `__has_cpp_attribute`

The behavior of `__has_cpp_attribute` as specified in the standard today is also ambiguous and should be fixed. On the one hand, the standard currently requires implementations to report a nonzero value even for recognised, but semantically ignored attributes ([cpp.cond]/6):

> For an attribute specified in this document, the value of the *has-attribute-expression* is given by Table 22. For other attributes recognized by the implementation, the value is implementation-defined.

On the other hand, the standard simultaneously does *not* require implementation to do that when they do not support the attribute ([cpp.cond]/5):

> Each *has-attribute-expression* is replaced by a non-zero *pp-number* matching the form of an *integer-literal* if the implementation supports an attribute with the name specified by interpreting the *pp-token*s, after macro expansion, as an *attribute-token*, and by `0` otherwise.

The wording regarding `__has_cpp_attribute` is therefore contradictory. As a result, it is unclear whether `__has_cpp_attribute` should return a positive value if a compiler recognise and syntactically checks a standard attribute but then semantically ignores it.

Existing practice

The `__has_cpp_attribute` feature is a victim of implementation divergence. Clang and ICC both report a positive value for `__has_cpp_attribute(carries_dependency)`, even though they semantically ignore it; however, GCC reports `0` (and emits a diagnostic that it is being ignored). MSVC is inconsistent even with itself: it reports a positive value for `__has_cpp_attribute(carries_dependency)`, but `0` for `__has_cpp_attribute(no_unique_address)`, even though it does not implement semantics for either attribute.

3 What happened so far?

Regarding syntactic ignorability, the authors of [N2761] have told us that the original design intent was to allow standard attributes to be ignored only semantically, not syntactically. The same interpretation is also recommended by CWG. Consider the proposed resolution of [CWG2538] and NB comment GB 9.12.1p6, as approved by CWG:

> For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined~~. Any~~; any such *attribute-token* that is not recognized by the implementation is ignored. [*Note:* A program is ill-formed if it contains an *attribute* specified in [dcl.attr] that violates the rules to which entity or statement the attribute may apply or the syntax rules for the attribute's *attribute-argument-clause*, if any. — *end note*]

EWG conducted a poll on the question in a teleconference earlier this year. The result was consensus in favour of the resolution recommended by CWG:

> It is EWG's intent that [dcl.attr]/6 *only* permits an implementation to ignore a standard attribute's effect, but not appertainment and argument parsing.

SF	F	N	A	SA
2	6	3	0	2

The same question was then put to an electronic poll for confirmation:

> Resolve CWG2538 by clarifying that it is EWG's intent that [dcl.attr]/6 *only* permits an implementation to ignore a standard attribute's effect, but not appertainment and argument parsing.

SF	F	N	A	SA
11	6	4	0	3

Despite the strong majority in favour of the approach recommended by CWG, this poll was deemed by the chair to be No Consensus, because the three strongly against votes came from implementers. Thus, the issue remains unresolved. Some interesting comments with arguments in favour and against can be found in [P1018R17].

Defining a rule for semantic ignorability is something we had already proposed in an earlier version of this paper [P2552R0], but the paper has not been seen by EWG yet.

The problem with `__has_cpp_attribute` has no opened Core issue of which we are aware. This issue is not unique to C++; C has a similar problem with `__has_c_attribute` (C2x 6.10.1p9 and 6.7.12.*). Aaron Ballman told us that an NB comment will be filed proposing to amend each attribute specification to add "if the implementation supports the attribute" to make it clear that `__has_c_attribute` returns `0` for an attribute ignored by the implementation. C++ should follow suit.

4 Where do we go from here?

We should resolve the ambiguities surrounding standard attributes:

1. Clarify whether an implementation is allowed to syntactically ignore a standard attribute,
2. Clarify what we mean when we say that standard attributes are semantically ignorable,
3. Clarify the behaviour of `__has_cpp_attribute` for attributes that the implementation ignores (either syntactically and semantically, or only semantically)

With each question, the committee has different options. Below we describe the options we think are possible, discuss the tradeoffs that come with each option, and propose our recommendation.

4.1 Syntactic ignorability

We believe that the question about syntactic ignorability is the most urgent of the three questions to resolve, and should be addressed within the C++23 timeframe. We have four options:

— **Option 1A.** Specify that standard attributes *cannot* be syntactically ignored (= adopt proposed resolution for [CWG2538] as recommended by CWG).

— **Option 1B.** Specify that standard attributes *can* be syntactically ignored (= parsed as a *balanced-token-sequence* and then skipped entirely; argument clause and appertainment do not need to be syntax checked; entities inside the argument clause do not need to be ODR-used).

— **Option 1C.** Specify that standard attributes are conditionally supported (similar to 1B, but implementations have to explicitly document if they do not support a particular standard attribute, and emit a warning at least the first time such an unsupported standard attribute is used).

— **Option 1D.** Do nothing (leave this ambiguous).

From our perspective, option 1D, while being the default if no consensus can be reached, is by far the worst. It does nothing to resolve [CWG2538] and NB comment GB 9.12.1p6, and it does not even save the committee any time, as this debate surely will come around again. To fix the problem, we must choose 1A, 1B, or 1C.

We propose 1A. This option is the resolution that matches the original design intent of C++ attributes (according to the authors of [N2761]), the resolution approved and recommended by CWG, and the option that had majority support in EWG every time there was a poll on it. We believe that 1A is the choice of sound language design. Standard attributes are not just arbitrary token sequences, even if they are *semantically* ignorable. We should try hard not to add optional language features on the syntax level to the language, and we should avoid treating standard attributes as a bucket for any language feature that we do not care about being implemented. There are a small number of standard attributes. If we bothered to specify something in the standard, implementations should at least bother to syntax-check it. This will ensure safety, predictability, portability, and consistency across implementations. Option 1A is also broadly consistent with existing practice (see section 2.1).

With regards to parsing expressions inside an attribute, if we do not choose 1A, then a compiler is allowed to silently accept an ill-formed expression, and the code will compile and appear to work fine; when we go to port the code to another compiler, the code will break. From a production and maintenance perspective, anything but 1A therefore seems like a very bad idea.

With regards to ODR-using the entities inside such an expression, as discussed in detail in section 2.1.4, we believe that 1A, i.e. requiring the ODR-usage even if the assumption is semantically ignored, is the option that is most consistent and portable. It also matches existing practice with the existing implementations of `assume`. Leaving it up to the implementation whether a particular template instantiation happens, or whether a particular lambda capture gets triggered, would introduce areas of gratuitous non-portability to the language. We must avoid this.

We would like to remind the reader that this issue is completely orthogonal to the semantics of `assume`. It concerns any hypothetical attribute or attribute-like thing that contains an expression, which includes the proposed `trivially_relocatable` attribute [P1144R5] and a hypothetical attribute-like syntax for Contracts [P2487R0].

1B is the option supported by the three compiler implementers that voted against the recommended resolution in the EWG electronic poll. Implementer concerns should of course always be taken seriously. In particular, we heard from Clang that their interpretation of [dcl.attr.grammar]/6 has always been 1B; that switching to 1A would be an unacceptable implementation burden in particular with regards to checking appertainment; that no users are actually asking for this status quo to change; and that existing practice should take priority over the original design. Interestingly, we have found that all major compilers (including Clang) are actually very good at syntax-checking the argument clause and appertainment of all existing standard attributes (see existing practice discussion in sections 2.1.1 and 2.1.2), so we are not sure where the problem actually is.

Another implementer concern is MSVC's comment that parsing expressions inside an attribute-argument clause is technically challenging for them and that they would instead prefer to treat them as token soup that can be skipped entirely (see discussion in section 2.1.4). Yet another argument in favour of option 1B is that the benefits of checking syntactic requirements for something with no semantic effect are negligible, and therefore the standard should not require it.

1C has not been discussed before, but we consider it a superior alternative to 1B. It still allows the implementation to completely ignore a standard attribute and treat it as token soup, but it requires that the implementation explicitly documents that, and emit a warning at least the first time such an unsupported standard attribute is used. We believe that this is more user-friendly than 1B.

4.2 Semantic ignorability

If we pick 1B or 1C (full syntactic ignorability), then the issue of defining semantic ignorability becomes a non-issue: syntactic ignorability implies semantic ignorability. This has the interesting consequence that it would be no longer necessary to hide the semantics of standard attribute in a "recommended practice" section or make the semantics explicitly optional: if attributes are ignorable

on the syntax level, then any effects they have are completely optional by default, overriding any normative wording in [dcl.attr]. We can therefore move wording out of the "Recommended practice" sections and make it regular normative wording; we can simplify the notion of *potentially-overlapping subobject* to just *overlapping subobject*; and so forth. This transformation will simplify the standard, but not change its meaning, and thus does not need EWG's involvement.

If we pick 1A (no syntactic ignorability), then we see the following options:

— **Option 2A.** Specify that standard attributes are meant to be semantically ignorable, and what this means exactly, normatively in the C++ standard.

— **Option 2B.** Specify this in a separate new Standing Document (SD) instead. This SD might also contain other design principles for new language features.

— **Option 2C.** Do nothing (leave this implicit).

We propose 2A. We believe that the rule we outlined in section 2.2.5 accurately captures the current rules of semantic ignorability:

> Given a well-formed program, omitting all occurrences of a particular standard attribute shall result in a well-formed program that exhibits the same *mandated behaviour* as the original program.

We therefore propose that this wording should go through CWG wording review and then be added to [dcl.attr], and the required concept of *mandated behaviour* that we developed in section 2.2.4 added to [intro.defs]. Formally, it would only apply to the attributes that are already in the standard, and thus not add any new information per se, as those attributes are already defined to be semantically ignorable in different ways (see discussion in section 2.2). However, the existence of such an explicit rule in the standard would be very helpful for codifying the intended behaviour of all future standard attributes, too: any new attribute proposal that does not want to follow the rule would have to carve out an explicit exception for itself. We believe that the presence of the normative rule would be a strong enough deterrent for future proposals that they will stick to it, thus leading to consistent language design.

Alternatively, one could hold the position that the C++ standard is not the place to try and constrain future evolution of the language, only to define what is and is not conforming with the current standard. Therefore, this rule could instead be published in a new standing document (option 2B). If this option were to be chosen, it would make most sense to create a new standing document for all such design guidelines for new core language features, not just for attributes.

Doing nothing (2C) means that the standard will continue to say nothing about the semantic ignorability of standard attributes. This means that misunderstandings on this subject will continue, and the discussions around what should or should not be an attribute will keep wasting precious committee time. We believe that choosing 2C only makes sense if we decide to throw away the idea of ignorability of attributes as a design guideline entirely, and consciously allow new features using attributes syntax to modify the *mandated behaviour* of a program. In other words, 2C is the correct choice if we want to open up the design space of attributes to any feature that could be implemented as a keyword, but that we do not want to reserve an actual keyword for, and that cannot (or should not) be specified as a contextual keyword for technical reasons.

4.3 `__has_cpp_attribute`

With regards to the intended behaviour of `__has_cpp_attribute`, we have the following options:

— **Option 3A.** Specify that `__has_cpp_attribute` should return a positive value for a standard attribute only if an an implementation has a useful implementation of its semantics (GCC behaviour).

— **Option 3B.** Specify that `__has_cpp_attribute` should also return a positive value for a standard attribute if an implementation can syntax-parse it, even if it does not implement any useful semantics (Clang and ICC behaviour).

— **Option 3C.** Do nothing (leave this ambiguous).

We propose 3A. `__has_cpp_attribute` is typically used in a preprocessor `#if` to query whether the compiler implements a certain feature. Being able to query, for example, whether the compiler will honour the class layout changes introduced by a `no_unique_address`, is a lot more useful than merely querying if the compiler can recognise the syntax of the attribute. From what we have heard, this option is also in line with what WG14 intends to do for `__has_c_attribute`, and we should not end up in a world where the specifications of `__has_cpp_attribute` and `__has_c_attribute` contradict each other (but this particular point should be confirmed with SG22 before approving any wording on it).

Acknowledgements

We would like to thank John Lakos for his thorough review of the paper and for contributing the idea of regions of program behaviour and the concept of *mandated behaviour*; Corentin Jabot for contributing the code example where removing a `[[no_unique_address]]` attribute can cause a program to stop compiling; and Aaron Ballman, Erich Keane, Jens Maurer, and Peter Brett for their valuable comments on earlier drafts of this paper.

References

[CWG2118] Richard Smith. Core Issue 2118: Stateful metaprogramming via friend injection. https://cplusplus.github.io/CWG/issues/2118.html, 2022-06-27.

[CWG2538] Jens Maurer. Core issue 2538. Can standard attributes be syntactically ignored? https://cplusplus.github.io/CWG/issues/2538.html, 2022-03-26.

[N2761] Jens Maurer and Michael Wong. Towards support for attributes in C++ (Revision 6). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf, 2008-09-18.

[N3190] Lawrence Crowl and Daveed Vandevoorde. C and C++ Alignment Compatibility. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3190.htm, 2008-09-18.

[P1018R17] JF Bastien. C++ Language Evolution status pandemic edition 2022/06–2022/07. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1018r17.html, 2022-07-10.

[P1144R5] Arthur O'Dwyer. Object relocation in terms of move plus destroy. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1144r5.html, 2020-03-01.

[P1774R8] Timur Doumler. Portable assumptions. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r8.pdf, 2022-06-14.

[P2487R0] Andrzej Krzemieński. Attribute-like syntax for contract annotations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2487r0.html, 2021-11-12.

[P2521R2] Gašper Ažan, Joshua Berne, Bronek Kozicki, Andrzej Krzemieński, Ryan McDougall, and Caleb Sunstrum. Contract support – Working paper. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2521r2.html, 2022-03-15.

[P2552R0] Timur Doumler. On the ignorability of standard attributes. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2552r0.pdf, 2022-02-15.