On the ignorability of standard attributes

Timur Doumler (papers@timur.audio)

Document #: P2552R0
Date: 2022-02-15
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 this means exactly. This paper is proposing a definition that could be useful as a design guide for future language feature proposals.

1 Motivation

Whenever proposals for new C++ attributes or attribute-like language features are considered, there is usually a discussion around the idea that attributes should be "ignorable", and that new attributes should be compatible with this principle. However, there does not seem to be a common understanding of what this means exactly. The principle of ignorability of standard attributes in C++ is currently some kind of "gentlemen's agreement" that has its origin in the standardisation of attribute syntax for C++11 [N2761]. However, this "agreement" is not codified anywhere and is therefore often misunderstood.

Does it mean that an implementation can choose to ignore any standard attribute and still be conforming? That doesn't sound right: a malformed *attribute-specifier*, for example, `[[fallthrough(42)]]`, should result in a compilation error (and all major C++ compilers issue one). Rather than just saying that attributes are "ignorable", a better rule seems to be: if the program is well-defined *with* the attribute, then ignoring the attribute should not change the observable behaviour of that program. But that does not quite work either: if an implementation implements the functionality of the `[[no_unique_address]]` attribute, its effects are clearly observable, as it can change the layout of a type. So what is the rule then?

The purpose of this proposal is to clearly define what we mean by "ignorability" of standard attributes. The goal is to avoid such discussions in the future and to guide future C++ language proposals by helping to decide whether or not a particular new language feature could or should be specified as an attribute.

Further, attributes are a design space that C++ shares with C. The C standard actually *does* define what the common semantics of all its standard attributes are (see 3.5). We should seek to be as compatible with this as reasonably possible.

2 Proposed rule

We propose to define the semantics of standard attributes as follows:

> Given a well-formed program with well-defined behaviour, omitting all occurrences of a particular standard attribute shall result in a well-formed program whose observable behaviour is consistent with a correct execution of the original program.

It is not entirely clear whether the C++ standard itself is the ideal place to codify such a definition. On the one hand, the C standard does something similar (see 3.5). On the other hand, the C++ standard does not typically try to constrain future evolution of the language, only to define what is and isn't conforming with the current standard. So, alternatively, such a definition could instead be published in a new standing document containing design principles for new core language features. We would like to seek guidance from the C++ committee on this point.

Our own position is that this definition should go into the C++ standard itself (in [dcl.attr]), so it would formally only apply to the attributes that are already currently in the standard, and not add any new information per se. However, the existence of such an explicit rule in the standard would force any new attribute proposal that doesn't want to follow the above principle to carve out an explicit exception for itself. We believe this would be a strong enough motivation for future proposals to stick to the rule in order to avoid introducing inconsistencies to the standard.

Another question is whether such a rule should apply only to true standard attributes (i.e. those following the *attribute-specifier* grammar), or also to hypothetical new language features that are "attribute-like" (i.e. using double square brackets but a novel syntax), such as attribute-like syntax for contract annotations [P2461R1]. Again, we believe that it is enough to spell out the rule for existing attributes in the standard, and then "attribute-like" proposals will be likely to follow suit in order to be consistent.

3 Discussion

3.1 Well-formedness

In order for the above "ignorability" rule to apply, the program has to be well-formed *with* the attribute, i.e. the implementation is not allowed to ignore ill-formed standard attributes. The standard says:

> The *attribute-token* determines additional requirements on the *attribute-argument-clause* (if any). Each *attribute-specifier-seq* is said to appertain to some entity or statement, identified by the syntactic context where it appears ([stmt.stmt], [dcl.dcl], [dcl.decl]). 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.

In other words, an attribute can only be "ignorable" if it is well-formed, i.e. if the attribute appertains to an entity that it is allowed to appertain to, and if the *attribute-argument-clause* has the correct shape. Otherwise, the program is ill-formed, and the compiler is required to issue a diagnostic. All major C++ compilers follow this rule and emit an error if the user tries to use the wrong type of argument (`[[deprecated(42)]]`), use an argument where none is allowed (`[[noreturn("x")]]`), use an attribute in the wrong place (`[[fallthrough]] int x;`), etc.

Currently, all standard attributes either have no arguments, or a single string literal as an argument, so verifying the well-formedness of an *attribute-argument-clause* is trivial. However, new proposals such as `[[assume(`*`expr`*`)]]` [P1774R6] and `[[trivially_relocatable(`*`expr`*`)]]` [P1144R5] propose attributes with an expression as an argument. This has the consequence that even if an implementation chooses not to implement the functionality of such an attribute, it will have to parse the expression inside the *attribute-argument-clause* and issue a diagnostic if the expression

is ill-formed. Further, parsing the expression might trigger template instantiations, and again the implementation will have to issue a diagnostic in case this fails. This is fully compatible with the above "ignorability" rule.

According to Jens Maurer, one of the original authors of standard attributes [N2761], it was always intended that a conforming compiler needs to syntax-check all attributes specified in the standard. After having done so, the compiler may choose to ignore the attribute, i.e. ignore the "recommended practice" sections for the attribute. This means that the sentence "The *attribute-token* determines additional requirements on the *attribute-argument-clause* (if any)" is always in force for attributes specified in the standard. The standard already implies this today, but in order to make it unambiguous, we should change the wording

> 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.

to

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

Jens Maurer suggested to open a Core issue for this; however, we might as well just roll this wording tweak into the present paper.

As should be clear from the above, the requirement of well-formedness should apply only to standard attributes. We do not propose to introduce any restrictions for non-standard attributes such as `[[gnu::unused]]` or `[[omp::directive (parallel for, schedule(static))]]`. They are truly ignorable in the sense that if implementation does not recognise them, it can completely skip them. The existing rule that the *attribute-argument-clause* can be any *balanced-token-seq* was created specifically to allow implementations to cleanly ignore non-standard attributes whose details (structure of arguments) are unknown to the implementation.

3.2 Well-definedness

In order for the above "ignorability" rule to apply, the program must not have undefined behaviour *with* the attribute. This means that the addition of an attribute is allowed to introduce undefined behaviour, however the removal of an attribute is *not* allowed to do that.

This is compatible with the existing standard attribute `[[noreturn]]` as well as the proposed new attribute `[[assume(`*`expr`*`)]]` [P1774R6], both of which can introduce undefined behaviour into an otherwise valid program. On the other hand, this means that new language features that turn undefined behaviour into well-defined behaviour cannot be attributes.

3.3 Omission of all occurrences

The above "ignorability" rule works by considering omission of all occurrences of a particular standard attribute token, such as `noreturn`, regardless of whether it occurs in its own double square brackets or as part of an *attribute-list* such as `[[noreturn, deprecated]]`. We are not interested in considering the consequences of omitting some but not all occurrences of `noreturn` in a given program.

3.4 Consistency with original program

Attributes are allowed to have effects that cause a change in the observable behaviour of a program; notably, `[[no_unique_address]]` does so by altering the memory layout of a class. However, any such effects need to be optional. This is what we mean by "ignorability": not that ignoring the

attribute will not change the behaviour of the program, but that the program you get without the attribute is a valid and conforming version of the original program with the attribute. Note that the effects of `[[no_unique_address]]` are optional. The non-static data member marked with `[[no_unique_address]]` can share the address of another non-static data member or that of a base class, but it doesn't have to; therefore, an implementation could choose to not implement the functionality of `[[no_unique_address]]` and still be conforming. If `[[no_unique_address]]` would cause a mandatory change in class layout rather than an optional one, it could not be an attribute.

By the same token, the existing feature `alignas` cannot be an attribute, because its effects on the alignment of an object are mandatory, not optional. Indeed, `alignas` was initially proposed as an attribute, but this was later reverted before C++11 was finalised [N3190].

Any core language feature that does not influence the observable behaviour of a program at all, but affects only "non-behavioural" things like warnings or providing information to the back-end is a good candidate for a standard attribute. Most existing standard attributes fall into this category: `[[carries_dependency]]`, `[[deprecated]]`, `[[fallthrough]]`, `[[likely]]`, `[[unlikely]]`, `[[maybe_unused]]`, and `[[nodiscard]]`.

3.5 Compatibility with C

The C standard specifies the following rule for standard attributes:

> 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.

The existing wording for C++, together with the additional wording we propose here, would create a rule that is compatible with the above, but arguably more precise.

4 Wording

Modify [dcl.attr.grammar] as follows:

> Given a well-formed program with well-defined behaviour, omitting all occurrences of an *attribute-token* specified in this document shall result in a well-formed program whose observable behaviour is consistent with a correct execution of the original program.
>
> 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.

References

[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.

[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.

[P1774R6] Timur Doumler. Portable assumptions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r6.pdf, 2022-02-15.

[P2461R1] Andrzej Krzemieński. Attribute-like syntax for contract annotations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2487r0.html, 2021-11-12.