Document Number: D2565R0
Date: 2022-03-15
Reply-to: Bret Brown <bbrown105@bloomberg.com>
Audience: SG-15

Supporting User-Defined Attributes

Contents

Abstract

While the standard supports vendor-provided and otherwise user-defined attributes in C++, actual use of nonstandard attributes while preventing trivial misspellings is a challenge. In particular, diagnostics in compilers used to prevent misspellings of standard and well-known attributes will reject attributes provided for other contexts, including attributes used to support other compilers. This document explores how this problem presents itself in practice and proposes a potential attribute declaration syntax to address the problem.

Standard Attributes

Standard attributes should, and generally do, work as expected on all toolchains that fully support C++14. For example:

```
[[ deprecated("Broken. Use `alpha`.") ]]
double apple(double num);
```

Misspelling Attributes

It's beneficial and common to provide diagnostics in the case that the intention of relevant C++ code is to use a standard attribute but a typing mistake is introduced instead:

```
[[ edprecated("Broken. Use `bravo`.") ]]
double banana(double num);
```

Major compilers all correctly identify `edprecated` as an incorrect attribute with some combination of warnings flags.

Clang 13.0.1 Diagnostic

```
<source>:6:4: warning: unknown attribute 'edprecated' ignored
[-Wunknown-attributes]
[[ edprecated("Broken. Use `bravo`.") ]]
   ^~~~~~~~~~
```

GNU 11.2 Diagnostic

```
<source>:7:25: warning: 'edprecated' attribute directive ignored [-Wattributes]
    7 | double banana(double num);
      |
```

MSVC v19.30 Diagnostic

```
<source>(6): warning C5030: attribute 'edprecated' is not recognized
```

User-Defined Attributes

For the purposes of illustration, let's say we wanted a more featureful deprecation. Perhaps we would like to associate a severity with each deprecation to provide some flexibility for consumers. Let's accomplish that by creating a user-defined attribute that adds a second deprecation field -- a string representing severity. Valid values would include: "off", "warn", and "error".

It could look like so:

```
[[ bespoke::deprecated("Broken. Use `charlie`.", "warn") ]]
double carrot(double num);
```

Custom attributes like these are supported by the standard and should be fully accepted by all tools. This includes issuing no warnings and no errors against the custom attribute. Quoting the standard: “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.” [§9.12.1.6].

Instead, vendors usually consider this to be a diagnosable event. For instance:

Clang 13.0.1 Diagnostic

```
<source>:9:4: warning: unknown attribute 'deprecated' ignored
[-Wunknown-attributes]
[[ bespoke::deprecated("Broken. Use `charlie`.", "warn") ]]
   ^~~~~~~~~~~~~~~~~~>
```

GNU 11.2 Diagnostic

```
<source>:10:25: warning: 'bespoke::deprecated' scoped attribute directive
ignored [-Wattributes]
   10 | double carrot(double num);
      |
```

MSVC v19.30 Diagnostic

```
<source>(9): warning C5030: attribute 'bespoke::deprecated' is not recognized
```

Vendor-Specific Attributes

In the same way, vendors such as compilers are allowed to define their own attributes. As far as the C++ standard is concerned, this use case is the same as the previous one, but the implications in this case are distinct, so it's worth explaining this as an interesting use case compared to “user-defined attributes”.

```
[[ clang::no_sanitize("undefined") ]]
[[ gnu::access(read_only, 1) ]]
[[ msvc::known_semantics ]]
double daikon(double num);
```

Vendor-specific attributes like these are supported by the standard and should be fully accepted by all tools. This includes issuing no warnings and no errors against the vendor-specific attributes.

Again, vendors usually consider these attributes to be diagnosable events except that they recognize their own attributes. Clang gets a notable exception for supporting many attributes from other toolchain providers, but there are attributes from other vendors that are not supported, at least not yet. And, to extrapolate into the future, it's safe to say that current versions of Clang do not support vendor-specific attributes that have not been invented yet, including future Clang-specific attributes.

Clang 13.0.1 Diagnostic

```
<source>:13:4: warning: unknown attribute 'access' ignored
[-Wunknown-attributes]
[[ gnu::access(read_only, 1) ]]
   ^~~~~~~~~~
<source>:14:4: warning: unknown attribute 'known_semantics' ignored
[-Wunknown-attributes]
[[ msvc::known_semantics ]]
   ^~~~~~~~~~~~~~~~~~~~
```

GNU 11.2 Diagnostic

```
<source>:15:21: warning: 'clang::no_sanitize' scoped attribute directive
ignored [-Wattributes]
   15 | int* daikon(int* num);
      |                     ^
<source>:15:21: warning: 'msvc::known_semantics' scoped attribute directive
ignored [-Wattributes]
Compiler returned: 0
```

MSVC v19.30 Diagnostic

```
<source>(12): warning C5030: attribute 'clang::no_sanitize' is not recognized
<source>(13): warning C5030: attribute 'gnu::access' is not recognized
```

Impact

Use of Preprocessor to Work Around Limitations

It is common in cross-platform code to still use preprocessor macros to wrap vendor-specific and user defined attributes, including preprocessor conditionals and feature testing logic. This pattern seems to be commonplace enough to be nearly universally adopted. For instance, both MSVC and Clang support the gsl::suppress attribute, but GCC and other tools do not, so the following code exists in the gsl library:

```
//
// make suppress attributes parse for some compilers
// Hopefully temporary until suppression standardization occurs
//
#if defined(__clang__)
#define GSL_SUPPRESS(x) [[gsl::suppress("x")]]
#else
#if defined(_MSC_VER) && !defined(__INTEL_COMPILER)
#define GSL_SUPPRESS(x) [[gsl::suppress(x)]]
#else
#define GSL_SUPPRESS(x)
#endif // _MSC_VER
#endif // __clang__
```

Source:
https://github.com/microsoft/GSL/blob/b26f6d5ec7b043f9d459c1dfdd6da4d930d4e9b4/include/gsl/assert#L44-L56

Note that, as of that commit, standard attributes like `[[noreturn]]` are in use, so presumably the macros are not for compatibility modes for C++ versions previous to C++11.

Vendor-Specific Attributes Effectively Not Portable

The implication of widespread preprocessor wrapping logic is that vendor-defined attributes are not working as originally intended.

Limitations to Tool Designs

Other "vendors" that would be interested in supporting custom attributes include:

- analysis tools
  - static analysis tools
  - instrumentation tools
- code generation tools
  - serialization frameworks
  - dependency injection frameworks
  - tools that provide bindings between C++ and other languages
- projects researching proposals for new standard attributes

There are probably other use cases as well, but requiring preprocessor macros or cross-toolchain support for specific attributes is undesirable for these use cases, especially for projects without the expertise or resources to coordinate with multiple toolchain providers.

Recommendations

Preserve warnings on unknown attributes

The current behaviors of the compiler have an important advantage in which they detect typos in the use of attributes. Therefore this paper recommends that compilers continue to issue warnings on unknown attributes.

New syntax for declaring attributes

In order to remove the need to wrap the use of attributes in preprocessor macros, this paper proposes a new syntax to declare attributes that may be unknown to the compiler. This is in line with a recent change in GCC, which allows the declaration of a pragma to silence warnings on specific attributes that are intentionally used in the code.

This would, therefore, introduce a standard syntax for indicating that the usage of an unknown attribute is not the result of a typography error. For instance:

```
[[ extern gnu::access(...) ]];
```

This mechanism would support providing these declarations in header files in relevant libraries. It would also be available in the cases where such headers do not exist and instead libraries could use these declarations to suppress diagnostics.

Alternatives

Alternatives are possible and could be investigated, including metadata files to be packaged with relevant projects like gsl to declare the existence of vendor-specific attributes for the benefit of various tools that diagnose C++ code, including compilers.

Future Work

Standard Attribute Declaration Header(s)

It could be interesting to consider a `<stdattrib>` header to provide declarations for standard attributes like `std::deprecated`. Note that, even if such a facility was interesting, it would not be sufficient to resolve the problems described in this paper. In particular, older releases of compilers would not be aware of future standard attributes. In those cases, some way to declare that attribute as valid would be needed or else users will likely turn to nonstandard mechanisms like backward compatibility preprocessor macros.

Validation of Attribute Parameters

Validation of parameters to standard attributes is possible on a per-attribute basis as the standard can clearly define what parameters are valid. This paper does not attempt to define interesting parameter definitions in the interest of supporting parameter validation. It could be an interesting future paper to add this ability. In the meantime, the tools that do support the user-defined attributes should provide validation of the number and types of parameters provided to the user-defined attribute.

References

Examples in Compiler Explorer

https://godbolt.org/z/9451YPeT5

GCC Issue for Supporting Bespoke Attribute Scopes and `-Wattributes`

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101940

Original Paper on C++ Attributes

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf

`clang::no_sanitize` Documentation

https://clang.llvm.org/docs/AttributeReference.html#no-sanitize

`gnu::deprecated` Documentation

https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc/Common-Function-Attributes.html

`gsl::suppress` Documentation

https://docs.microsoft.com/en-us/cpp/cpp/attributes?view=msvc-170#microsoft-specific-attributes