Attributes introspection

Document #:
Date: 2024-06-25
Project: Programming Language C++
Audience: sg7
Reply-to: Aurelien Cassagnes
<aurelien.cassagnes@gmail.com>

Contents

1 Introduction **1**
1.1 Earlier work . . . 1

2 Motivation **1**

3 Optionality rule **2**

4 Proposal **2**
4.1 Scope . . . 2
4.2 std::meta::info . . . 2
4.3 Reflection operator . . . 3
4.4 Splicers . . . 3
4.5 Metafunctions . . . 3
4.6 Queries . . . 4
4.7 Applications . . . 4

5 Discussion **4**

6 References **4**

1 Introduction

The current draft aims at getting the conversation and work started on supporting introspection of attributes. There is ongoing work to refine that draft, especially when it comes to examples, applications and proposed wording.

1.1 Earlier work

While collecting feedback on this draft, we were redirected to [P1887R1] as a pre existing proposal. In this paper the author discusses two topics 'user defined attributes' (also in [P2565R0]) and reflection over said attributes. We believe the two topics need not be conflated together, both have intrinsic values on their own. We aim here to focus the discussion entirely on **standard** attributes reflection.

2 Motivation

Attributes are used to great extent and there likely will be attributes added as the language evolve. What is missing now is a way for generic code to look into the attributes appertaining to an entity. A motivating toy example is the following

```cpp
[[nodiscard]] bool foo(int i) { return i % 2 ; }

template <class F, class... Args>
constexpr std::invoke_result_t<F, Args...> logInvoke(F&& f, Args&&... args) {
    // Do some extra work, log the call...
    // Forward call
    return std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
}

int main() {
    foo(0); // Warning on discarded return
    logInvoke(foo, 0); // No warning on discarded return
}
```

Ideally we would want a mechanism to recover the `[[nodiscard]]` attribute that originally appertained to `foo` declaration. Other examples of wrapping around callables can be found, whether by closure or explicity registering callbacks for dispatch, etc. The current example deals with code as it is in a c++23 world, but other applications can easily be thought of in the context of code injection [P2237R0] where one may want to skip over `[[deprecated]]` members for example.

3 Optionality rule

There is a long standing and confusing discussion around how ignorable are attributes allowed to be. We'll refer the reader to [P2552R3] for an at-length discussion of this problem, and especially on what 'ignorability' really mean. This proposal agrees with the discussion carried there and in [CWG2538]. We also feel that whether an implementation decide to semantically ignore a standard attribute should not matter.

What matters more is self-consistency, when introspecting an entity:

— We should be able to discover appertaining attributes
— Declaring an entity with attributes discovered via introspection yield the same result as what the implementation would offer if directly declaring that entity with those attributes.

4 Proposal

We put ourselves in the context of [P2996R3] for the current proposal to be more illustrative in terms of what is being proposed.

4.1 Scope

Attributes are split into standard and non standard. This proposal wishes to limit itself to standard attributes ([dcl.attr]). We feel that since it is up to implementation to define how they handle non standard attributes, it would lead to obscure situations that we don't claim to tackle here.

A fairly (admittedly artificial) example can be built as such: Given an implementation supporting a non standard `[[no_introspect]]` attributes that suppress all reflection information appertaining to an entity, we would have a hard time coming up with a self-consistent system of rules to start with.

4.2 std::meta::info

We propose that attributes be a supported *reflectable* property of the expression that are reflected upon. That means value of type `std::meta::info` should be able to represent an attribute in addition to the currently supported set.

4.3 Reflection operator

The current proposition for reflection operator grammar does not cover attributes, i.e. the expression`^[[deprecated]]` is ill-formed.
The current proposal advocates for supporting this new grammatical construct under *unary-expression*

unary-expression:

…
`^` `[[`*attribute-list*`]]`

The resulting value is a reflection value embedding relevant info for the standard *attribute* entity. If the *attribute* is not a standard attribute, the expression is ill-formed.

4.4 Splicers

We propose that the form

```
[[ [: r :] ]]
```

be supported in contexts where attributes are allowed.

— `[[ [: r :] ]]` produces a potentially empty sequence of attributes corresponding to the attributes reflected via `r`.

A toy example is as follows

```
[[nodiscard]]
enum class ErrorCode {
  e_Warn,
  e_Fatal,
};

[[ [: ^ErrorCode :] ]]
enum class ClosedErrorCode {
  template for (constexpr auto e : std::meta::enumerators_of(^ErrorCode)) {
    return [:e:],
  }
  e_Final,
};
```

If the attributes produced through introspection violate the rules of what attributes can appertain to what entity, as usual the program is ill-formed.

4.5 Metafunctions

We propose to add two metafunctions to what is discussed already in [P2996R3]

4.5.1 attributes_of

```
consteval auto attributes_of(info entity) -> vector<info>;
```

This being applied to a reflection `entity` will yield a sequence of `std::meta::info` representing the attributes appertaining to `entity`.

4.5.2 is_standard_attribute

```
consteval auto is_standard_attribute(info entity) -> bool;
```

This would return true if r designates a standard `[[attribute-list]]`, otherwise it would return false.

4.6 Queries

We do not think it is necessary to introduce additional query or queries at this point. Especially we would not recommend to introduce a dedicated query per attribute (eg `is_deprecated`, `is_nouniqueaddress`, etc.). Having said that, we feel those should be acheivable via concept, something akin to

```
auto deprecatedAttributes = std::meta::attributes_of(^[[deprecated]]);

template<class T>
concept IsDeprecated = std::ranges::any_of(
  attributes_of(^T),
  [deprecatedAttributes] (auto meta) { meta == deprecatedAttributes[0]; }
);
```

4.7 Applications

coming next

5 Discussion

Originally the idea of introducing a `declattr(Expression)` keyword seemed the most straightforward to tackle on this problem, but from feedback the concern of introspecting on expression attributes was a concern that belongs with the reflection SG. The current proposal shifted away from the original `declattr` idea to align better with the reflection toolbox. Note also that as we advocate here for `[[ [: r :] ]]` to be supported, we recover the ease of use that we first envisioned `declattr` to have.

6 References

[CWG2538] Jens Maurer. 2021-12-02. Can standard attributes be syntactically ignored?
https://wg21.link/cwg2538

[P1887R1] Corentin Jabot. 2020-01-13. Reflection on attributes.
https://wg21.link/p1887r1

[P2237R0] Andrew Sutton. 2020-10-15. Metaprogramming.
https://wg21.link/p2237r0

[P2552R3] Timur Doumler. 2023-06-14. On the ignorability of standard attributes.
https://wg21.link/p2552r3

[P2565R0] Bret Brown. 2022-03-16. Supporting User-Defined Attributes.
https://wg21.link/p2565r0

[P2996R3] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2024-05-22. Reflection for C++26.
https://wg21.link/p2996r3