Strongly-Typed Reflection on Attributes

Document #:	P1887R0
Date:	2019-10-06
Project:	Programming Language C++
Audience:	SG-7

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

The notion of static types and compile time type checking is central to the effective use of C++
- Bjarne Stroustrup

Target

C++23 (Or whenever reflection ships). At this stage, this paper aims at giving an overview of the feature and does not constitute a complete proposal.

Abstract

Current reflection proposals do not propose to reflect on attributes. However we think it is a powerful feature to have as attributes combined with reflection can enable important use cases, as filtering and augmenting entities with additional information.

Unfortunately, attributes are only identified by their names, and if they accept parameters, the parameters form a *balanced token sequence*, which makes it impossible for individual parameters to be extracted from within a program.

We introduce the notion of user-defined attributes and a typesafe mechanism to reflect on these attributes and their parameters at compile time.

Example

```
namespace Catch {
    struct [[decorator]] test_case {
        constexpr test_case(const char* name)
            : name(name) { }
        const char* name;
    };
}

namespace test {
    void g() {}

    [[Catch::test_case("Printing something")]]
    void f() {
        g();
        std::cout << "Hello World\n";
    }
}

int main () {
    static constexpr auto r = meta::range(reflexpr(test));
    for...(constexpr auto member :  r) {
        static constexpr bool b = meta::has_attribute<Catch::test_case>(member);
        if constexpr(meta::is_function(member) && b) {
            constexpr auto data = meta::attribute<Catch::test_case>(member);
            std::cout << "Running test " << data.name << "\n";
            auto ptr = &n::unqualid(meta::name_of(member));
            ptr();
        }
    }
}
```

Proposal

We propose that any `regular` type that can be constructed at compile time can appertain to any C++ entity that can be reflected upon through the same syntax as attributes.

We further propose a couple of apis to enable reflecting on these user-defined attributes

User-Defined Attribute Declaration

A user-defined attribute is any class which meets the requirements of `regular`, and which has `constexpr` constructors, copy-constructor and is equality comparable in constant expressions.

The `[[decorator]]` attribute is proposed to mark types intended to be used as user defined attributes. This attribute can be helpful for documentation purposes as well as to allow an implementation to offer better diagnostics but is not necessary nor does it affect the behavior of a program.

Using User-Defined Attributes

User-Defined Attributes can be used at the same places other attributes can be used, and use the same syntax as a qualified-call to a constructor.

```
namespace n {
    struct [[decorator]] a {
        constexpr a(int);
    };
    [[n::a(42)]] void f(); // constructor
}
```

It is also always possible to invoke the copy constructor

```
namespace n {
    struct [[decorator]] a {
        constexpr a(int);
    };
    a get_attribute();

    [[n::a(get_attribute())]] void f(); // copy constructor
    [[get_attribute()]] void g(); // attach a "get_attribute" attribute to g
}
```

Reflecting on User-Defined Attributes

The primary use of user defined attributes is to be reflected upon. (User defined attributes can still get picked up by tooling, including documentation generators, etc).

To that end, we propose a couple of methods to query the presence and the value of a user defined attribute.

```
namespace std::meta {
    template <typename Attr>
    consteval bool has_attribute(info x);

    template <typename Attr>
    consteval Attr attribute(info x);
}
```

[*Note:* These interfaces are inspired by other reflection proposals and may have to change to reflect any design evolution of these proposals. — *end note*]

Design consideration

Multiple definitions of attributes

If the same user-defined attribute appears multiple times on the same declaration, the program is ill-formed.

Attributes must appertain to the declaration that is being reflected upon. If different declarations do not have the same set of user-defined attributes or have the same attributes with different values, the program is ill-formed, no diagnostic required.

It is ill-formed to redeclare user-defined attributes on definitions if there exists a previous declaration in the same translation units.

Ignorable attributes

Attributes are designed to be ignorable. Therefore, an implementation should not have to parse attributes as user-defined attributes until they are reflected upon.

Notably, the only difference between user-defined attributes and other attributes is that there exists a type of the same name as the attribute.

To that end, user-defined-attributes are only parsed as such upon a call to `has_attribute` or `meta::attribute`.

This also means that the header/module in which the type defining a user-defined attribute only has to be imported before a call to any of these functions.

For any attribute `A`, a call to `has_attribute<A>` or `meta::attribute<A>` on the reflection of an object `obj` is ill-formed if:

- A is not defined
- `obj` has an attribute named "A" but that attribute cannot be evaluated as a valid C++ constant expression constructing an object of type `A`.
- `A` does not model `regular` or does not have a `constexpr` copy constructor and equality comparison operator.

Furthermore, a call to `meta::attribute<A>` on the reflection of an object `obj` is ill-formed if no attribute of that type appertains to `obj`.

Non ignorable attributes

The syntax `[[::foo::bar()]]` un-ambiguously designates a user-defined attribute and in this case, we could enforce that the attribute exists and the constructor expression at the point where the attribute is used.

Use cases

Filtering

One major use case for this feature is to tag entity that must be filtered in or out while reflecting upon a member or a class. Consider the following meta-class:

```
class(qobject) my_class {
public:
    [[qt::slot()]] void show();
    [[qt::signal()]] void stateChanged();
};
```

Not all methods are signals, and not all methods are slots, we need a way to filter them. In the [P0707], Herb Sutter suggests using the return type of functions to distinguish signal from slots.

```
class(qobject) my_class {
    qt::signal mySignal();
    qt::slot mySlot();
};
```

This would have to look like that because slots can have a non-void return type, which means the qobject meta class implementation would have to extract the different pieces of the function signature to recreate it.

```
class(qobject) my_clas {
    qt::signal<void> mySignal();
    qt::slot<void> mySlot();
};
```

It also makes the intent less clear to the reader as they can't know that the 'qt::signal' type is merely a tag that will not be a part of the generated class. More critically it makes it almost impossible to combine several tags.

```
class(qobject) my_class {
    [[qt::signal()]]
    [[qt::notify_property("state")]];
    void stateChanged();
};
```

As showed on pages 2, this could be used by test and bench-marking frameworks, and reduces the scenarios in which the use of macros is still necessary.

Mapping and other information

A primary use case for reflection is serialization and command line parser generators. However, it is often beneficial that the name of the serialized field differs from the C++ variable name, as well as to provide extra information to the generator.

```
struct my_object {
    [[json_serializer::field_name("case")]]
    [[json_serializer::default_value(42)]]
    int caseNumber;
    std::mutex mutex;
};

struct my_options {
    [[command_line_parser::short_name("L")]]
    [[command_line_parser::help("Specify the compression level")]]
    [[command_line_parser::default_value(5)]]
    int compression_level;
};
```

Future work

Along with a way to reflect on user defined attribute, it would be beneficial to copy/inject standard attributes (including `maybe_unused`, `noreturn`, `nodiscard`, `deprecated`), when working with code injection and meta classes. In the absence of code injection, none of the existing standard attributes seems worth reflecting upon.

Other approaches considered

One of the early ideas was to have a special syntax to declare attributes, and have the different parameters exposed as a tuple. This approach added too much complexity and did not provide any benefit compared to the current approach or relying on declared types.

Implementation

I have implemented a basic prototype in a fork of clang supporting reflection, proving the viability of the idea, But the implementation is not yet complete.

Acknowledgments

Many thanks to Peter Bindels for reviewing this work and providing valuable feedback.

References

[N4830] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/n4830

[P0707] Herb Sutter *Metaclasses*
https://wg21.link/p0707