Adding attribute reflection to C++. 
Contents

1 Intro ................................................. 1

2 Motivation .......................................... 2

3 Impact on Standard ............................... 3
  3.1 What to forbid .................................. 3
  3.2 Avoid break old code ......................... 3
  3.3 What we get today .............................. 3
  3.4 What to allow .................................. 4

4 Design decisions ................................. 6
  4.1 Typed attributes ............................... 6
  4.2 Attribute Instances ............................ 7
  4.3 Type traits ..................................... 7
  4.4 Attribute Inheritance ........................... 8
  4.5 Which attributes attachment can be reflected 8
  4.6 More complete examples ..................... 8

5 Technical Specification ......................... 10
  5.1 Attempted Wording ............................ 10

6 Acknowledgments ................................. 12
  6.1 Few acks ...................................... 12

Bibliography ....................................... 13
1 Intro

C++ reflection mechanism could benefit from a more complete set of functionality, if we define how to reflect attributes as well as types. This proposal main idea is to make attributes typed, and use the standard attribute syntax `[[ ]]` to attachment but, instead of attach a typed attribute to a target, we propose to attach a constexpr instance of an attribute to a target. After that, we also propose to reflect them using the syntax `typedef<[[T]]>` and `typedef<[[T]] requires C>...` based on [SA14, N3951] to get attributes of type T, restricted by constexpr `bool` function C. However, by current standard, the compiler implementers and vendors are free to use any tokens they need, including keywords. Make attributes typed will lead to some problems, and we show some ways to address them. But before that, let’s introduce an easy Get and Set idiom example:

```cpp
struct serializable {}
{
};

struct getter {
  string name;
};

struct setter {
  string name;
};

struct [[serializable]] X {
  private:
    int index_;
  public:
    [[getter("index")]] int index() const { return index_; }
    [[setter("index")]] void set_index(int i) { index_ = i; }
};

auto mytuple1 = make_tuple(typedef<[[SomeType]]>...);
// auto mytuple1 = make_tuple(serializable());

auto mytuple2 = make_tuple(typedef<[[&SomeType::index]]>...);
// auto mytuple2 = make_tuple(getter("index"));

auto mytuple3 = make_tuple(typedef<[[&SomeType::set_index]]>...);
// auto mytuple3 = make_tuple(setter("index"));
```
2 Motivation

After defining reflection in C++, it’s easy to imagine an augmented version of it, with more information attached to types. And a common way to do this is using attributes, they help to create new ways to pass information from the source code to the frameworks, to choose which members you will use and which you will discard in some process, or improve meta programming techniques like type selection. All of this is expected from a reflection mechanism, and also we can enumerate some more.

— Serialization
— Remote Procedure Calls
— Event Driven Development
— Test Driven Development
— GUI Property Editors
— Database-Object mapping interface
— Documentation
— Describe Axioms, Constraints and Rules over Types
3  Impact on Standard

3.1  What to forbid

In order to take a little more controlled environment we pretend to forbid few things.

— Forbid attributes to be keywords: The definition from [dcl.attr.grammar] says that If a keyword (2.12) or an alternative token (2.6) that satisfies the syntactic requirements of an identifier (2.11) is contained in an attribute-token, it is considered an identifier. No name lookup (3.4) is performed on any of the identifiers contained in an attribute-token. Today, none of the standard attributes are keywords alignas, noreturn, carries_dependency and deprecated, so it still in time. Also, they need to particpate of the name look-up, just as a common types and instantiated as constexpr.

— Attribute balanced braces, they can be [] or () but we propose to forbid [] because it’s confusing since it’s not a standard C++ constructor idiom, and allow only as standard initialization and () as common constructor call.

3.2  Avoid break old code

C++ attributes allows compiler implementers to define their custom features that does not need to be standardized, so we must keep this use. To allow this and avoid break old code and allow attributes to be typed we propose the following:

— First, make all attributes typed, for standard defined attributes like [[deprecated]] we propose to define their type as a common class or structure;

— According to [dcl.attr.grammar] (7.6.3), the implementers can put attributes in a non-standard attribute-namespace, the standard does not say that attribute-namespace is a common namespace, but since attributes does not participate of a name look-up we conclude that namespace != attribute-namespace. But since now the attributes will become common types, may be it’s a good idea to put standard attributes in std namespace and allow implementers to create a new namespace std::imp that stands for implementation defined. Where implementers can put their non-standardizable features that needs a namespace. And also, create freely their own namespaces inside of this, where they can put their own attributes. [Example: GNU may create std::imp::gcc to put gnu_inline attribute class. — end example]

— Also, implementers will be free to add an implicit using std::imp::xxx to any compilation unit to make their attributes work as typed and not break old code.

This will help the migration from non-typed do typed attributes.

3.3  What we get today

We will not propose forbid any place that attributes can be attached today, but we need to specify few new ones, and define also new ways to understand what is allowed by the standard. Today, attributes can be attached to the following items.

— enums, lambdas, classes, base classes, arrays

— member objects, member functions, member operators, member types, constructors, destructors, friend declaration, bit fields

— functions, function parameters, operators

— variables
— template functions, template classes
— statements, for parameters, ranged for parameters, conditions
— labels
— try blocks, exception declaration
— expressions
— using alias, using namespace directive
— trailing return types

3.4 What to allow [impact.allow]

Now we show the few important new places that you should be able to attach attributes.
— Surprisingly in enumerator-definitions [dcl.enum] it cannot be attached today, the standard says:

```
enumerator-list:
  enumerator-definition
  enumerator-list , enumerator-definition
enumerator-definition:
  enumerator
  enumerator = constant-expression
enumerator:
  identifier
```

Enumerator should change to:

```
enumerator:
  attribute-specifier-seq opt identifier
```

— Attributes attached to class inside the class definition, by the current standard, class attributes must lay between class name and the start of the class definition.

```
class-head:
  class-key attribute-specifier-seq opt class-head-name class-virt-specifier opt base-clause opt
  class-key attribute-specifier-seq opt base-clause opt
```

Looking at the standard in item [class.mem] of [ISO14, N3797] we got this:

```
member-declaration:
  attribute-specifier-seq opt decl-specifier-seq opt member-declarator-list opt ;
  function-definition
  using-declaration
  static_assert-declaration
  template-declaration
  alias-declaration
  empty-declaration
```

In the first line, it says that attribute-specifier-seq can appear before a member-declarator-list that can be optional, allows it at the end of the class, just before "}

```
struct X {
  int a;
  int b;
  [[ Att{1, 'hello'} ]] ;
};
```
The standard address this case directly with a restriction in 9.2.7 that says a `attribute-seqopt` cannot appear without a, `member-declarator-list`, but this restriction should be removed. This way it’s not bounded to any particular member, it becomes a class attribute. It’s a good place to put some like `Property` attributes. Also, this can help human view of attributes.

— Another issue that can help human view, comes from great idea in [Tom14, N3955] that leads rapidly to adding group attributes on it to attach attributes to many member at once;

```cpp
class AnotherType {
public [[qt::slot]]:
    void fun();
    void gun();
};
```

Using this syntax should means that each member have their own instance of `qt::slot` attribute, but this is pointless, since all instances will get the exact same constexpr values, we could instantiate only one object and share it to members `fun()` and `gun()`.

§ 3.4
4 Design decisions

4.1 Typed attributes

Among few other problems, attributes are not typed nor part of type system, characteristic they share with macros and throws specification. These two features of C++ become deprecated over the years for different reasons, but not participate in type system is their common problem. So the propose is about three things:

— Turn attributes into typed attributes;

— Allow any type to be attached to target. The way of attachment is the usual general attribute [++] syntax as defined in [dcl.attr.grammar].

— Retrieve them via reflection. This proposal is based on [SA14, N3951] syntax, the first idea is to use typedef<[[T]]>... and typedef<[[T]]> requires C... syntax to get attributes. Also propose new type traits to help reflecting them. If you think that this syntax is ugly, you probably right, but it’s ugliness is derived from ugly [++] attribute C++ syntax. Soon after publishing the proposal [SA14, N3951], this syntax received a big criticism on the forum, in favor to drop changes in keywords and use a as-if-lib approach like reflect<T>::attribute<C>::types..., but when we tried to define the interface of it we find something like this:

```cpp
template <typename T>
struct reflect {
    template <typename C>
    struct attribute {
        ?? types...; // <-what we put here ?
    }
};
```

What kind of function or typedef or directive have a return type of variadic template ? Well, today, we have tr2::bases from [Spe09, N2965]:

```cpp
template<typename _Tp>
struct bases {
    typedef __reflection_typelist<__bases(_Tp)...> type;
};
```

What is the interface of __bases(_Tp) ? In this case, it’s need a language support. If you look at a [Spe12, N3416] you can find some ideas about typelists, that can turn __bases(_Tp) interface into a plain standard. May be variadic template expansion is just a struct version of lambda algorithms, structs with unnamed members, a more pure form of lambda types, a type part of a tuple. But, avoid being too theoretical, we propose an approach, does not depend of implement the whole typelists issue and does not harm the interface forcing not-standard features and brings us the benefit ot template expansion. We think that as-if-lib should be used since it’s possible to do it without violating any C++ rule. So, as soon as we got some advances in this issue, we would be glad to update this proposal and add typelists to it, but until there, we still go for typedef<[[T]]>..., anyway if you prefer, you can imagine the samples using the syntax reflect<T>::attribute<C>::types... without loosing anything of the rest of the proposal.
4.2 Attribute Instances

Since attributes will be typed, instead of attach the type of the attribute to target type, we can attach an instance of an attribute type to a target type, instantiate it at compile time, and then use object information to improve their use. So \([A]\) is not a typed attribute “A” anymore, but it becomes a constexpr instance of type A, and constructing A using the syntax of standard initialization, we can keep attribute attachment short like this:

```
Example:
  class A {
    int x;
    int y;
  }
  class [[A{10,20}]] T {
  }
— end example
```

4.3 Type traits

4.3.1 Applying type traits

The proposed method to applying type traits in `typedef<[[T]] requires C>...` is analogue to [SA14, N3951], `typedef<T requires C>...`, in both cases C is expected to be a `template callable function bool constexpr` that returns true for all typed that should be returned by `typedef<[[T]] requires C>...`. It’s important to show at least one example of this usage; [Example:

```
struct TestCase {
  string nm;
};
struct Serializable {};

template<typename T>
constexpr bool is_test() {
  return std::is_same<T, TestCase>::value;
}

struct X [[Serializable, TestCase{"test1"}]] {
  int mem1();
  int mem2;
};

auto my_tuple = make_tuple(typedef<[[X]] requires is_test>... >);
// auto my_tuple = make_tuple(TestCase{"test1"}); <- ignores Serializable
— end example
```

4.3.2 Has Attribute type trait

As expected we should create a type trait to discover if a type has an attribute attached to it, so we propose `std::has_attribute<X,A>` that returns true if type X has attached a attribute A or a attribute derived from A; [Example:

```
struct att {};
struct deprecated : att { std::string why; };    
struct X [[deprecated{"this is problematic"}]] {};

void foo() {
  std::cout << std::boolalpha << std::has_attribute<X, att>::value << std::endl;
}
```

§ 4.3.2
4.4 Attribute Inheritance

Does the class and member attributes should be inherited? Derived classes and members are inherited, and also you can set their access, but you cannot define a access specifier to an attribute. This means that you cannot decide what is “passed away” to derived classes and what should be hidden from them. Also, in C++ if a base class has virtual methods, they can be re-declared in derived classes, in this case, according to standard, you can re-declare new attributes as well, do this new attributes should override the previous or should they be added to previous function attributes? Instead of defining if the class and member attributes must be inherited or not, since exists uses for both cases, we propose the creation of a `std::is_derived<T>` type trait that could solve this problem. Applying this trait as a constraint when you reflect the attributes will bring what you want. So we propose some new traits.

— `is_derived<M>` return true if T is a inherited member type, data or function, and false otherwise

```
Example:

struct A {
    int x;
};
struct B : A {
};

— end example
```

std::is_derived<B::x>::value is true, but std::is_derived<A::x>::value false, since x is defined in A;

— `is_derived_from<T,X>` The use for this trait is to check if a member T was derived from class X;

4.5 Which attributes attachment can be reflected

Not all attribute can be reflected, since not all attachments occurs in classes or typed items, we must define which cases is possible to reflect attributes.

— classes
— class members ( classes, enums, object members and function members, including constructors and destructors )
— functions
— lambda types
— parameters of functions

4.6 More complete examples

4.6.1 Property idiom

By the end let’s show a sample implementation of a Property idiom:

```
struct serializable {};
struct desc { std::string description; };
struct getter { std::string name; };
struct setter { std::string name; };

template< typename _Reader, typename _Writer, typename _Notifier >
struct prop {
    std::string name;
};
```
_Reader reader;
_Writer writer;
_Notifier notifier;
constexpr property(_Reader r, _Writer w, _Notifier n)
  : reader(r), writer(w), notifier(n) {}
};

class [[make_tuple<serializable>]] X {
private:
  [[getter("index")] int index_; // it is a member attribute
  [[setter("index")] void set_index(int i) { index_ = index; }}

  enum Status {
    Half_empty [[desc("It's about a half")]],
    Half_full [[desc("Optimistic approach to half")]]
  };

  int val_;
  virtual void notifier() = 0;

public:
  void set_val(int v) { val_ = val; }
  [[ prop("val", val_, set_val, notifier) ]] // it is a class attribute
  }

  auto mytuple = make_tuple(typedef<[[SomeType]]>...);
  /* auto mytuple =
     make_tuple(serializable(),
                property("val", &X::val_, &X::set_val, &X::notifier));
  */

4.6.2 Attribute Idioms [design.complete.idioms]

Just to show how things can get complicated fast, by the definition of standard, the placement of an attribute
attachment can occur after the template definition of a class declaration, and now being part of the name
look-up, it’s possible to create an analogue of CRTP a Curious Recurrent Attribute Pattern¹:

  template<typename T> struct A {};
  class C [[A<C>]] {}

Just for fun let’s show another few more idioms:

  template<typename ...T> class Mixim [[T()...]] {}

  struct A { template<typename ...T> U(const T&...t) {} };
  struct B {};
  class VariadicCtor [[A(10, 20, 30, 1.0, B(), "what was that?")] {};

  struct X {};
  struct Y [[X]] {};
  struct Z [[Y]] {}; // Y is an attribute that has attributes

¹) Avoid the anachronism

§ 4.6.2
5 Technical Specification

5.1 Attempted Wording

It’s not all wording needed about attribute reflection, it’s just small new changes in standard where attributes should be placed.

— [dcl.enum],

\[ \text{enumerator:} \]
\[ \text{attribute-specifier-seq} \text{ opt} \text{ identifier} \text{ attribute-specifier-seq} \]

— [class],

\[ \text{class-head:} \]
\[ \text{class-key} \text{ attribute-specifier-seq} \text{ opt} \text{ class-head-name} \text{ attribute-specifier-seq} \text{ opt} \text{ class-virt-specifier} \text{ opt} \text{ base-clause} \text{ opt} \]
\[ \text{class-key} \text{ attribute-specifier-seq} \text{ opt} \text{ base-clause} \text{ opt} \]

— [class.mem],

7 The optional attribute-specifier-seq in a member-declaration appertains to each of the entities declared by the member-declators; it shall not appear if the optional member-declarator-list is omitted. If it appear when the optional member-declarator-list is omitted, it shall be a class attribute.

— [dcl.attr.grammar]

1 Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

\[ \text{attribute-specifier-seq:} \]
\[ \text{attribute-specifier-seq opt attribute-specifier} \]
\[ \text{attribute-specifier:} \]
\[ [ \text{ attribute-list } ] \]
\[ \text{alignment-specifier} \]
\[ \text{alignment-specifier:} \]
\[ \text{alignas ( type-id ... opt) } \]
\[ \text{alignas ( constant-expression ... opt) } \]
\[ \text{attribute-list:} \]
\[ \text{attribute opt attribute-list , attribute opt attribute } \]
\[ \text{attribute-list , attribute ... } \]
\[ \text{attribute:} \]
\[ \text{type-specifier initializer opt} \]
\[ \text{attribute token attribute argument-clause } \]
\[ \text{attribute token:} \]
\[ \text{identifier} \]
\[ \text{attribute-scoped token} \]
\[ \text{attribute-scoped token:} \]
\[ \text{attribute namespace :: identifier} \]
\[ \text{attribute namespace:} \]
\[ \text{identifier} \]
\[ \text{attribute argument-clause:} \]
\[ \{} \text{balanced-token-seq } \} \]
balanced-token-seq:
  balanced-token-seq
  balanced-token

balanced-token:
  ( balanced-token-seq )
  { balanced-token-seq }~
  [ balanced-token-seq ]

any token other than a parenthesis, a bracket, or a brace

[Note: For each individual attribute, the form of the balanced-token-seq will be specified. — end note]

In an attribute-list, an ellipsis may appear only if that attribute’s specification permits it. An attribute followed by an ellipsis is a pack expansion (11.5.3). An attribute specifier that contains no attributes has no effect. The order in which the attribute tokens appear in an attribute list is not significant. If a keyword (2.12) or an alternative token (2.6) that satisfies the syntactic requirements of an identifier (2.11) is contained in an attribute token, it is considered an identifier. No name lookup (3.4) is performed on any of the identifiers contained in an attribute token. The attribute token determines additional requirements on the attribute argument clause (if any). The use of an attribute-scoped token is conditionally-supported, with implementation-defined behavior. [Note: Each implementation should choose a distinctive name for the attribute namespace in an attribute-scoped token. — end note]

Each attribute-specifier-seq is said to appertain to some entity or statement, identified by the syntactic context where it appears (Clause 6, Clause 7, Clause 8). If an attribute-specifier-seq that appertains to some entity or statement contains an attribute that is not allowed to apply to that entity or statement, the program is ill-formed. If an attribute-specifier-seq appertains to a friend declaration (11.3), that declaration shall be a definition. No attribute-specifier-seq shall appertain to an explicit instantiation (14.7.2).

For an attribute token not specified in this International Standard, the behavior is implementation defined.

Two consecutive left square bracket tokens shall appear only when introducing an attribute specifier. [Note: If two consecutive left square brackets appear where an attribute specifier is not allowed, the program is ill-formed even if the brackets match an alternative grammar production. — end note]

In the attribute definition, before type-specifier the compiler must add a implicit constexpr.

The attribute information must be carried from the definition to the point of type trait calls.

[Example:

```cpp
struct Att {
};

struct X {
  int v1;
  [[Att]] int v2;
};

template< typename T >
bool foo(T t) {
  return std::has_attribute<T,Att>::value;
}
```

The call to foo must distinguish `foo(&X::v1)` from `foo(&X::v2)` — end example]
6 Acknowledgments

6.1 Few acks

Thanks to forum people: Sean Middleditch, Tiago Macieira, Andrew Tomazos, Ville Voutilainen for reading and feedback and Oliver Gottart for that good [Gof14, MOC] form wich we pick some ideas.
Bibliography


