Document Number: N4447 - Revision of N3951/N3984/N3987
Date: 2015-04-10
Project: SG7 - Reflection
Reply to: Cleiton Santoia Silva
<cleitonsantoia@gmail.com>
Daniel Auresco
<auresco@gmail.com>

From a type T, gather members name and type information, via variadic template expansion.

Contents

1 Revision history [revision]

— Change the layout of the document to more C++ standard

— Changed the meaning of `typename<T, C>` to type-list

— Introduce `typeid<T, C>` in replacement of `typename<T, C>`

— Changed `requires` by `,`

— Address private access problem

— Explain more why is important to reflect namespaces and modules

— Resume some parts (but still a big document)

— Changes in impact

— Merged with attribute proposal N3984 (typed attributes)

— Merged with new type traits proposal N3987 (type traits)

2 Intro [intro]

1. Overview of the new three instructions:

   — `typedef<T, C>` that expands the definition of type T into a parameter back instantiation.

   — `typename<T, C>` that expands types of the definition of type T into a parameter back.

   — `typeid<T, C>` that expands members names of type T into a parameter back instantiation.

2. Simple Sample Code

```
using namespace std;
namespace ns {
 struct X {
  int x, y;
 };
}

std::vector<std::string> names{ typeid<ns::X, is_member_object_pointer>... };
std::tuple<typename<ns::X, is_member_object_pointer>...>
     mytuple = std::make_tuple(typedef<ns::X, is_member_object_pointer>...);
```

3. Compiler will turn the last two lines into this:

```
vector<string> names { "x","y" };
std::tuple<ns::X::int, ns::X::int>
       mytuple = std::make_tuple(
                &ns::some_struct::x,
                &ns::some_struct::y);
```

4. What it's not:

   — we don't need no new keywords

   — we don't need no AST control

   — no magic namespaces or runtime structs

   — not an as-if-lib approach

   — all we need by now just another syntax keywords

5. That's it.

3 Motivation [motivation]

— Serialization

— Better meta-programming

— Type transformations

— Event Driven Development

— Test Driven Development

— GUI Property Editors

— Database-Object mapping interface

— Auto Documentation

— Automatic verification of concepts (assert on templates)

— Reflecting constructors, allow GUI Forms with widgets as sub objects (without pointers)

4 Impact on Standard [impact]

4.1 Impact of reflection [impact.reflect]

1. This is a proposal of compile time reflection, strongly typed, using the variadic template expansion;

2. `typename<T, C>...`, `typedef<T, C>...` and `typeid<T, C>...` will be new ways to use existing keywords, that are not used today, only `typename type`, `typedef type id` and `typeid(typeid)` are used in C++14;

3. The expansion of `typedef<T, C>...` and `typeid<T, C>...` may appear in the source code as a constant expression-list, and a single const-expression, if the result is only one element.

4. The expansion of `typename<T, C>...` may appear in the source code as a type parameter pack expansion.

5. Allow typed attributes.

6. To allow a terse use of type traits `typedef<T, is_enum>...`, they must implement some function call operator as follows.

```
template<>
struct is_class<void> {
 template<typename T>
 constexpr bool operator() const noexcept {
   return is_class<T>();
 }
};
```

4.2 Impact of attributes [impact.attributes]

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

1. Forbid attributes to be keywords: today, none of the standard attributes are keywords *alignas, noreturn, carries_dependency and deprecated*, so it still in time. Also, they need to participate of the name look-up, just as a common types and instantiated as constexpr.

2. 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;

   — The implementers can put typed attributes in a non-standard attribute-namespace. [*Example:* GNU may create `gcc::attribute` namespace to put `gnu_inline` attribute class. — *end example*]

   — Implementers will be free to add an implicit `using gcc::attribute` to any attribute lookup to make their attributes work as typed and not break old code.

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

5 Design [design]

5.1 Guidelines [guidelines]

1. Allow all types as parameters a.k.a. `typedef<int>` or `typedef<std::printf>` should not break anything.
2. Allow not typed parameters a.k.a. `typedef<namespace>` and `typedef<module>`.
3. Allow to choose what to get a.k.a. `typedef<T, is_member>`.
4. Allow automatic "search" for classes via `typedef<namespace>` a.k.a. `class.forName()`.
5. Allow intended access to private parts, but not accidentally.

To fulfill this guidelines we considered the instructions `typename<T, C>...`, `typedef<T, C>...` and `typeid<T, C>...>`, where C is a concept, a typename or a template parameter, that provides a boolean constexpr function call operator() template, that will be applied to each element of T, to choose if it will be retrieved or not.

5.2 Overview of `typedef<T, C>...` [typedef]

1. We propose to gather from a type T, base classes, members functions, members objects, member enums, member typedefs, member classes, member constants, instantiated member templates and attributes restricted by a predicate.
2. Here some examples of type parameter.

Table 1 — Get some items from a class T

member functions of T	`typedef<T, is_member_function_pointer>`
member enums of T	`typedef<T, is_enum>`
member classes of T	`typedef<T, is_member_class>`
member arithmetic consts of T	`typedef<T, is_const && is_arithmetic>`
parent classes of T	`typedef<T, is_derived<T> >`
only private members of T	`typedef<T, is_private && is_member_pointer>`

3. Some examples of non-type parameter.

Table 2 — Get some items from non-type T

functions of namespace N	`typedef<N, is_function>`
classes of namespace N	`typedef<N, is_class>`
classes of module M	`typedef<M, is_class>`
classes of namespace N that are inherited from Factory class and which name starts with TFac and in singleton and have the attribute [[pimpl]] attached	`typedef<N, your_foo>`

4. Some reflection frameworks that allows `class.for_name(std::string)` idiom, need you to register each class that will be available into the reflection library. Gather information of a namespace allows automatic registration, avoiding boring, error prone, and static fiasco problems. You can just attach an attribute to your class definition and your favored reflection framework will search for you in the correct moment.

5. Another use for it is a concept checker, if you need to ensure a concept over a class, but you must wait until all member (classes and functions) template instantiations occurs.

6. `typedef<T,C>` may appear in same context as an instantiation of a pack expansion, as 14.5.3.7 [ISO14, N4296]

5.3 Overview of `typename<T, C>...` [typename]

1. Must be the same order than `typedef<T, C>...`

2. Brings the types of the elements of `typedef<T, C>...`, except for classes, unions and enums that, contrary to `typedef<T, C>...` (which brings pointers), `typename<T, C>...` brings "directly" types;

3. `typename<T,C>` may appear in same context as a pack expansion, as 14.5.3.4 [ISO14, N4296]

5.4 Overview of `typeid<T, C>...` [typeid]

1. Must be the same order than `typedef<T, C>...`

2. `typeid<T,C>` may appear in same context as an instantiation of a pack expansion, as 14.5.3.7 [ISO14, N4296]

3. Encoding should be utf-8.

4. Type names must be fully qualified, expanding all namespaces to avoid ambiguity if you will create a map of classes by name.

5. Member names may not be qualified, we don't see the need for full namespaces into each member, that you already knows what class it belongs to, when you call `typeid<T>...`;

6. When passed an argument to `typeid<T>...`, it will transitively search until it find the definition of it in the code, to allow the following :

```
struct X {
 int i;
 int j;
}

template<class T>
const char* foo(const T& t) {
 return typeid<T>...;
}

void foo {
 std::cout << foo(&X::i) << std::endl;
 std::cout << foo(&X::j) << std::endl;
}

output:
i
j
```

5.5 Minimalist and orthogonal [minimalist]

1. This syntax received a big criticism on the forum, in favor to drop changes in keywords and use a as-if-lib approach, but it can't address "not typed parameters", unless we open some "exception" and break some rules and accepts namespaces in the place of the type parameter. However, forum comments must be carefully considered, and we decided to put some effort to address this issue. Consider the following code:

   ```
   template<typename T>
   struct reflect {
     ... private_member_function_types;
     ... private_member_functions;
     ... private_member_object_types;
     ... private_member_objects;
     ... public_member_function_types;
     ... public_member_functions;
     ... public_member_object_types;
     ... public_member_objects;
     ... base_classes;
     ... private_base_classes;
     ... public_base_classes;
     ... virtual_base_classes;
   };
   ```

   It's a big interface, and still did not take virtual functions, abstract functions, constants, volatile, mutable, member classes, member enums, member typedefs, and so on... Instead of growing this class definition for each trait on the `<type_traits>` header, we pretend to use them directly. So even if as-lib proposal will be approved, it's better put some predicate on it.

2. Other use for a parameter pack expansion is the possibility to maintain the order of members, allowing section idioms:

   ```
   struct X {
    int a;
    int b;
   [[start_serialize]]
    int i;
    int j;
    int k;
    int get_ip();
   [[end_serialize]]
   };
   ```

   The above sample shows how define a serialize "session" in the class.

5.6 Constraints [constraints]

1. Template functions, template classes, constexpr function names and constexpr function objects and concepts may be passed as constraints.

2. The way `typedef<T, C>...` expands C is as follows:

   ```
   struct X {
    int i;
    int f();
   };
   ```

The instruction `typedef<X, C>...` will check for a parametrized function call `C(&X::i)` and `C(&X::f)` if they don't exist, try `C<decltype(&X::i)>()` and `C<decltype(&X::f)>()` it they also don't exist, the instruction is ill formed. If it is well formed and the member evaluate to true, the member is retrieved.

5.7 Type traits [traits]

1. Many new type traits will be useful, like `is_virtual, is_friend, is_override, is_final, has_-throws_specification, is_private, is_protected, is_public, etc`, some are defined in the session 7.

5.8 Private access [access]

1. We propose make the access accordingly to the normal rules of the language, if the reflector class is same or friend of the reflected class, `typedef<>...` have private access, if reflector class is descendent, it gets protected access, if other, just public access. Since one can choose via traits `is_private, is_protected, is_public`, it's possible to ask what you want. If the compiler finds a `is_private` or a `is_protected` among the constraints, then it overrides the default access.

2. However, it's much more important to some serialization framework, the fact the member is tagged with some "serialize" attribute than if it's private or public.

```
class X {
 [[serialize]] int a;
 int b;
public:
 [[serialize]] int i;
 int get_b();
};
```

5.9 Compatibility [compatibility]

1. When new "kinds" of declarations, suppose "member concepts", become part of the language, they must be included in the domain of `typedef<>...`, to maintain backward compatibility, the use of a predicate C, in `typedef<T, C>...`, force the return to be exactly what you want. If you ask for `typedef<T, std:is_member_pointer>...` and some new feature is added to class definition, the return will change only if the semantics of `std::is_member_pointer` changes.

2. Also, new features that are added in the type_traits, could be used in the reflection mechanism without any changes.

3. It's also a intuition problem if the rules for `reflect<T>::member_function` are different from `std::is_-member_function_pointer` a good way of ensure this is just making `std::is_member_function_-pointer` part of the instruction.

5.10 T in `typedef<T,C>`, `typename<T,C>` and `typeid<T,C>` [typedef.parameter]

1. T must be a template-argument of type or non-type, but cannot be a template.

2. T is constexpr `typedef<T>...` must calculate it and return it's type and value;

3. T is a member of enumeration it must return it's type and value:

```
enum E {
  e1 = 0;
  e2 = 2;
};
```

In this sample, `typedef<E::e2>...` should expand to a constant of type E, and value 2;

4. T is a constant or a member constant `typedef<(int)4>...` should expand to a constant of type int and value 4;

5. T is constant data member it return type must be a member-pointer type of member and return value must be a pointer to that member-typed constant;

6. `typedef<const/volatile/& T>...` and their combinations, must have the same result as `typedef<T>...`

7. T can be a lambda type `typedef<decltype([](int i)->int { return i+1; } )>....`

   — The standard says that the lambda must have a function call operator, but not where.

   — The lambdas may contain more member functions and member objects, and also the order of implementation is not defined in the standard.

   — The solution is via type_traits, one can define a new type trait `std::is_function_call_-operator` constraint that brings you the expected operator.

   — Otherwise the behavior of `typedef<lambda>` is undefined.

```
namespace std {
 template<typename T>
 constexpr is_function_call_operator() {...}
}

void foo() {
 auto lamb = []() -> int { return 7; };
 auto tuple = make_tuple( typedef<decltype(lamb), is_function_call_operator>... );
 auto ptr = std::get<0>(tuple);
 lamb.*ptr(1);
}
```

8. T can also be a namespace;

5.11 T in `typename<T,C>` and `typeid<T,C>` [typename.parameter]

. Here we explore arguments that are not be accepted in `typedef<T,C>....`

1. T can be a function pointer or a member function pointer, possibly a returned element of `typedef<T,C>`.

2. T can be primary types, in this case `typename<T,C>` get cast operators, operators and constructors.

5.12 Returns of `typedef<T>...` [typedef.return]

1. For each class T, the types of `typedef<T>...` should be:
   a) first the pointer type to class itself;
2. The following items in order of appearance in the class
   a) the pointer types of its direct base classes;
   b) types of the member objects pointers;
   c) types of the member functions pointers;
   d) types of the static member objects pointers;
   e) types of the static member functions pointers;
   f) the pointer types to typedefs;
   g) the pointer types to inner classes;
   h) lambda types for constructors;
   i) lambda types destructor;
   j) the pointer types enumerations;
   k) attribute types.
3. For each class T, the values of `typedef<T>...` should be:
   a) first the nullptr pointer to class type;

   a) nullptr for each base classes;
   b) value of the member objects pointers;
   c) value of the member functions pointers;
   d) value of the static member objects pointers;
   e) value of the static member functions pointers;
   f) nullptr for typedefs;
   g) nullptr for inner classes;
   h) lambda constructors;
   i) lambda destructor;
   j) nullptr for enumerations;
   k) instances of the attributes.
4. Since C++ standard says that we cannot get constructor address, cleverly (bad cleverly), they can be reflected by lambda proxies; A lambda function with a `operator()` overload for each constructor of T.
   a) For each namespace T, the types of `typedef<T>...` should be:
      i. the pointer type to classes/enums/unions/typedef of the namespace and it's inner namespaces;
      ii. the pointer type to each namespace-level function;
      iii. the pointer type to each namespace-level variable;
      iv. the pointer type to each namespace-level constant;

   a) For each namespace T, the values of `typedef<T>...` should be:
      i. nullptr to classes/enums/unions/typedef of the namespace and it's inner namespaces;
      ii. the pointer to each namespace-level function;
      iii. the pointer to each namespace-level variable;
      iv. the pointer to each namespace-level constant;

5.13 Returns of `typename<T>...` [typename.return]

1. For each class T, the types of `typename<T>...` should be:

   a) type to class itself;

   b) types of its direct base classes;

   c) types of the member objects pointers;

   d) types of the member functions pointers;

   e) types of the static member objects pointers;

   f) types of the static member functions pointers;

   g) types to typedefs;

   h) types to inner classes;

   i) lambda types for constructors;

   j) lambda types destructor;

   k) the pointer types enumerations;

   l) attribute types.

5.14 Returns of `typeid<T>...` [typeid.return]

1. For each class T, the types of `typeid<T>...` should be:

   a) name of the class itself;

   b) name of the direct base classes;

   c) name of types of the member objects pointers;

   d) name of types of the member functions pointers;

   e) name of types of the static member objects pointers;

   f) name of types of the static member functions pointers;

   g) name of types to typedefs;

   h) name of types to inner classes;

   i) name of constructors;

   j) name of destructor;

   k) name of the enumerations;

   l) name of attribute types.

2. The general rules for returned values are that types, classes, enums and unions it returns the fully qualified names (if defined in the global namespace the namespace name is"::"), but for members, only it's names;

3. For member operators values, `typeid<T>...` returns `operator@`, for `X operator+(X x) const` the result should be "operator+" without spaces;

4. For member cast operators values, `typeid<T>...` returns `operator @`, for `operator Y(X x) const` the result should be "operator Y" with a space;

5. For non member operators values, `typeid<T>...` returns `operator@`, for `::operator+(X x, X x)` the result should be "operator+" without spaces;

5.15 Returns of `typename<T>`,`typeid<T>` for functions [function.return]

1. Global function arguments will be reflected is this order:
   a) `typeid<T>...` returns first the name of function prefixed with "::", than the parameter names;
   b) `typename<T>...` returns the type of return of function than the types of parameters;
2. Non member function arguments will be reflected is this order:
   a) `typeid<T>...` returns first the name of function prefixed with namespace name, than the parameter names;
   b) `typename<T>...` returns the type of return of function than the types of parameters;
3. Member function arguments will be reflect is this order:
   a) `typeid<T>...` returns first the name of function unqualified, than the class name where the function is defined, than the parameter names;
   b) `typename<T>...` returns type of return of function, than the class that the function is enclosed together with the constness or ref. signature of the function, than the types of parameters.
4. Static member function arguments will be reflect is this order:
   a) `typeid<T>...` returns first the name of function unqualified inside the class, than the parameter names;
   b) `typename<T>...` gets the type of return of function than, than the types of parameters;
5. Constructors will be reflect is this order:
   a) `typeid<T>...` returns first the name of class unqualified, than the parameter names;
   b) `typename<T>...` gets the type of of the class, the types of parameters;
6. For non member functions argument T `typeid<T>...` returns the name of the function prefixed with it's namespace or "::" if it's a global function, without parentheses and without any noexcept/exception declaration or inline or other modifiers, only the qualified name of the function, then the names of each parameter. for a functin foo() in the namespace ns, `typeid<ns::foo>...` will return "ns::foo";
7. For member function arguments T `typeid<T>...` returns the qualified name of the function, without parentheses and without any noexcept/exception declaration or inline or other modifiers like virtual or public or protected constness or volatile, etc..., only the unqualified name of the function, then the names of each parameter;

5.16 Returns of `typename<T>`,`typeid<T>` for primary types [primary.return]

1. For primary types, `typename<T>...` will get cast operators, common operators, constructors and destructors, casts to convertible types must appear as well, if you call `typename<int>...` must receive cast operators to double/float/long... etc.
   — `typename<void>...` this should not return any data members neither function members, only the constructor and also void should be only "incomplete type" allowed;
   — `typename<nullptr_t>...` this should not return any data members neither function members, may be not even constructors here;
   — `typename<T*>...` this should return assignment operators, dereference operator*, ++,– and conversions from it to it's base classes, but not return any members of T;

— `typename<const T*>...` this should return same as `typename<T*>...` ++, – and dereference operator* changed to const, conversions from it to it's base const classes;

2. For primary type argument `typeid<T>...` returns the name of the type unqualified, and operators as "operator+" without spaces;

6 Attributes [attributes]

6.1 Intro to attributes [attributes.intro]

C++ reflection mechanism could benefit from a more complete set o 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 a type (as an attribute) to a target. After that, we also propose to reflect them using the syntax `typedef<T, C>` and `typename<T,C>...` 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:

```
struct serializable {
};

struct getter {
 const char* name;
};

struct setter {
 const char* name;
};

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

serializable ser =typedef<X, has_attribute<serializable>>...; // get class attribute
getter g = typedef<&X::index, has_attribute<getter>)...;      // member attribute
setter s = typedef<&X::set_index, has_attribute<setter>)...;  // member attribute
```

6.2 Typed attributes [attributes.typed]

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 tree 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 `typedef<T, has_atribute<A»...` syntax.

6.3 Attribute Instances [design.instances]

Since attributes will be typed, we will attach an instance of an attribute type to a target type, instantiate it at compile time. 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*]

6.4 Which attributes attachment can be reflected [design.attachment]

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

6.5 More complete examples [design.complete]

6.5.1 Property idiom [design.complete.property]

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

```
struct serializable {};
struct desc { const char* description; };
struct getter { const char* name; };
struct setter { const char* name; };

template< typename _Reader, typename _Writer, typename _Notifier >
struct prop {
 const char* name;
 _Reader reader;
 _Writer writer;
 _Notifier notifier;
 constexpr prop(const char* nm, _Reader r, _Writer w, _Notifier n)
  : name(nm), reader(r), writer(w), notifier(n) {}
};

class [[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<X, has_attribute_prop>...);
 /* auto mytuple =
      make_tuple(serializable(),
                 prop("val", &X::val_, &X::set_val, &X::notifier));
 */
```

6.5.2 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 occurs 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 Pattern[1]:

```
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> A(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

7 Type traits [type.traits]

1. First some unary type traits:

   a) `std::is_constexpr<D>` returns true if declaration D is marked as constexpr.
   b) `std::is_extern<D>` returns true if declaration D is marked as extern.
   c) `std::is_explicit<F>` returns true if function F is marked as explicit.
   d) `std::is_export<D>` return true if declaration D is marked as export.
   e) `std::is_final<D>` return true if declaration D is marked as final.
   f) `std::is_inline<F>` return true if function F is marked as inline.
   g) `std::is_lambda<T>` return true if the class T is a lambda.
   h) `std::is_local<T>` return true if the class is a local class (defined inside a function).
   i) `std::is_mutable<M>` return true if member M is marked as mutable.
   j) `std::is_override<F>` return true if function F is marked as override.
   k) `std::is_pure_virtual<F>` return true if the function F is a pure virtual function.
   l) `std::is_private<M>` return true if member M is inside a private scope.
   m) `std::is_protected<M>` return true if member M is inside a protected scope.
   n) `std::is_public<M>` return true if member M is inside a public scope.
   o) `std::is_thread_local<V>` return true if variable V is has a thread local defined.
   p) `std::is_overload<F>` return true if function F has some overload.
   q) `std::is_virtual<F>` return true if function F is virtual.

   For all unary type traits, one can use them directly in `typedef<T, C>...`

2. Then some binary template predicates:

   a) `std::has_attribute<D, A>` returns a
   b) `std::is_direct_base_of<Base,Derived>` return true if class “Base” is a direct base class of “Derived”.
   c) `std::is_friend<T,U>` return true if type T is declared as friend of U.
   d) `std::is_private_base_of<Base,Derived>` return true if class “Base” is a private base class of “Derived”.
   e) `std::is_protected_base_of<Base,Derived>` return true if class “Base” is a protected base class of “Derived”.
   f) `std::is_public_base_of<Base,Derived>` return true if class “Base” is a public base class of “Derived”.
   g) `std::is_virtual_base_of<Base,Derived>` return true if class “Base” is a virtual base class of “Derived”.

   For all binary type traits it’s expected to have a specialization when the user give one parameter and a definition of a template `template<typename T> constexpr bool operator()(T& t)` that checks the predicate to the elements of `typedef<T, C>...` in the way that is possible to ask for `typedef<T, has_attribute<atrib_name>....`

8 Technical Specification [spec]

1. Attempted wording for `typedef<T>`

   reflective-element:

    type-expression:

    constant-expression:

    namespace-name:

   reflective-constraint:

    function-template:

    class-template:

    constant-expression:

   reflective-typedef:

    `typedef<` *reflective-element:* `,` *reflective-constraint*$_{opt}$`>...`

   initializer-list:

    reflective-typedef:

   postfix-expression:

    reflective-typedef:

   [Note: postfix-expression: is allowed when reflect-pack: has only one element — end note]

2. Grammar for `typedef<T>`

9 Acknowledgments [ack]

9.1 Few acks [ack.ack]

Thanks to forum people: Sean Middleditch, Tiago Macieira, Andrew Tomazos, Ville Voutilainen for reading and feedback and Oliver Gottart.

Bibliography

[ISO14] C++ ISO. N4296 programming languages c++. Technical report, Programming Language C++, 2014.