Type Property Queries (rev 4)

Document Number: **N4428**
Revises: N4113
Date: 2015-04-08
Project: Programming Language C++ - SG7 Reflection
Reply-To: Andrew Tomazos <zos@google.com>,
Christian Kaeser <christiankaeser87@gmail.com>

Summary

We propose the addition of two class templates to the C++ Metaprogramming and Type Traits Standard Library - `std::class_traits` and `std::enum_traits` - that will enable basic reflection of classes, unions and enumerations - without core language changes.

Further, to show how these two new class templates integrate into a long-term vision for C++ reflection - we layout a tentative roadmap for a future new core language operator **reflectid** that will build upon these two traits. We show that this roadmap addresses previous concerns regarding both access control and namespace reflection.

This proposal revises N4113 in accordance with SG7 feedback from Urbana. (N4113 revises N4027 with feedback from Rapperswil, N4027 revises N3815 with feedback from Issequah).

To represent compile-time text we use std::string_literal from N4121. N4121 has passed SG7 (Rapperswil) and EWG (Urbana), and is due to be looked at by LEWG (Lenexa).

Synopsis

We propose the addition of the following two class templates to <type_traits>:

```
namespace std
{
    template<typename E>
    struct enum_traits {
        struct enumerators {
            static constexpr size_t size;
            template<size_t I>
            struct get {
                static constexpr string_literal identifier;
                static constexpr E value;
            };
        };
    };

    template<class C>
```

```
    struct class_traits {
        struct base_classes {
            static constexpr size_t size;
            template<size_t I>
            struct get {
                typedef /* */ type;
                static constexpr bool is_virtual;
            };
        };

        struct class_members {
            static constexpr size_t size;
            template<size_t I>
            struct get {
                static constexpr string_literal name;
                static constexpr /* */ pointer;
            };
        };

        struct nested_types {
            static constexpr size_t size;
            template<size_t I>
            struct get {
                static constexpr string_literal identifier;
                typedef /* */ type;
            };
        };
    };
}
```

Specification

std::enum_traits<E>

Requires: std::is_enum<E>

Provides information about the enumeration type E.

std::enum_traits<E>::enumerators

Provides information about the enumerator list of E.

std::enum_traits<E>::enumerators::size

The number of enumerators in the enumerator list of E.

std::enum_traits<E>::enumerators::get<I>

Requires: I >= 0 && I < size

Provides information about the I'th (zero-indexed) enumerator in the enumerator list of E, in declared order.

std::enum_traits<E>::enumerators::get<I>::identifier

A std::string_literal (N4121) holding the identifier of the enumerator. The identifier is encoded in UTF-8 format, with any UCNs decoded.

std::enum_traits<E>::enumerators::get<I>::value

A value of type E, that is the value of the enumerator.

std::class_traits<C>

Requires: std::is_class<C> || std::is_union<C>
Requires: C does not contain a member of reference type or a bit field.
Requires: C is not a lambda type.

Provides information about the class or union type C.

std::class_traits<C>::base_classes

Provides information about the direct public base classes of C, as they appear in the base-clause after pack expansion.

std::class_traits<C>::base_classes::size

The number of public base classes in the base-clause of C, or 0 if C does not have a base-clause.

std::class_traits<C>::base_classes::get<I>

Requires: I >= 0 && I < size

Provides information about the I'th (zero-based) public base class of C, in declared order.

std::class_traits<C>::base_classes::get<I>::type

The type of the base class.

std::class_traits<C>::base_classes::get<I>::is_virtual

True iff the base class is virtual.

std::class_traits<C>::class_members

Provides information about some public class members of C. The included members are functions or objects that have a direct simple declaration in the definition of C, and are not member templates or instantiations thereof. (Note: Members that are implicitly generated are not shown and members imported with a using declaration, or inherited are not shown. Constructors and destructors are not shown.)

std::class_traits<C>::class_members::size

The number of public members of C that satisfy the criteria given above, or 0 if none.

std::class_traits<C>::class_members::get<I>

Requires: I >= 0 && I < size

Provides information about the I'th (zero-indexed) public member of C that satisfies the above criteria, in declared order.

std::class_traits<C>::class_members::get<I>::name

A std::string_literal that holds the name of the member. If the name is an identifier, then it shall hold the UTF-8 encoded text of that identifier, with any UCNs decoded. If the name is a operator-function-id, then the text "operator" appended with a space character and then the canonical non-terminal of the operator ("operator +", "operator new[]", "operator <<=", etc). If the member is unnamed, the empty string. Otherwise, the text is implementation-defined.

std::class_traits<C>::class_members::get<I>::pointer

The result of the expression &C::m applied to the member. That is, a pointer-to-member for a non-static member, or a pointer for a static member. Anonymous unions are reflected as a single subobject of union type. (Note: The subobjects of the union are not visible in the class_members of the enclosing class type, they are visible by recursion on the type of the union subobject.)

std::class_traits<C>::nested_types

Provides information about some direct public nested types of C. Each nested type shall be one that is declared with a name in the class definition of C. The reflected nested types can be introduced by an alias declaration, a typedef member declaration, or from a member declaration with a class specifier, enum specifier or an elaborated-type-specifier. (Note: Unnamed classes and unions

can be reached through class_members instead, by deducing the type of the data members of the enclosing class type.)

std::class_traits<C>::nested_types::size

The number of public nested types of C meeting the above criteria.

std::class_traits<C>::nested_types::get<I>

Provides information about the I'th (zero-indexed) public nested type of C that satisfies the above criteria, in declared order.

std::class_traits<C>::nested_types::get<I>::identifier

A std::string_literal holding the identifier of the nested type. The identifier is encoded in UTF-8 format, with any UCNs decoded.

std::class_traits<C>::nested_types::get<I>::type

The type of the nested type.

Future Roadmap

As a first point of extension, we note that additional members can be added by a future proposal to either std::class_traits or std::enum_traits, without breaking changes. For example, we are considering proposing additional list members such as std::class_traits<C>::constructors and std::class_traits<C>::member_templates in a future proposal. Also, we could add other kinds of members other than these compile-time list style members.

As a second point of extension, in any of these list-style members we could, in a future proposal, add new members. For example, we are considering a std::class_traits<C>::class_members::get<I>::specifiers member to reflect the members specifiers.

The third and major point of extension we mention here is a core language operator called **reflectid**. We offer a rough possible specification here and show how it unifies and encloses std::class_traits<C> and std::enum_traits<E> - and also how it addresses previous discussions about both Access Control and Namespace Reflection.

The result of the construct reflectid(X) is a type, in a similar fashion to decltype(e). It can be used wherever a type may be used. The argument to reflectid identifies an entity, and the type that results is the "reflection" of that entity.

For an enumeration type E, reflectid(E) is equivalent to std::enum_traits<E>.

In cases where C is a class type, and reflectid(C) appears in an unrelated context to C, then reflectid(C) is equivalent to std::class_traits<C>. That is, it only reflects public members. If reflectid(C) appears in a context that has protected access, it reflects public and protected members. If it appears in a context that has private access, it reflects public, protected and private members.

There could additionally be ways to "break in" to a class, but we think there will be insufficient consensus on such a feature to get passed. In any event it is clear that reflectid will at least support public reflection of classes and enumerations, and under such use it will be implemented by mapping to std::class_traits<C> and std::enum_traits<E>.

In cases where N is a namespace, reflectid(N) will generate a reflection of the members of the namespace declared before the appearance of reflectid(N). This shows us why we need a core language operator and not a std::reflect template. First, namespace template parameters do not exist, and, even if they did, the reflection of a namespace is dependant on the position in the translation unit, so two occurrences of reflectid(N) will not result in the same type.

In cases where Tmpl is a template, reflectid(Tmpl) would provide a reflection of the template. Again, we need a core language operator, because a template and a class (for example), differ in template parameter kind.

Likewise for other entities.

Example Use Cases

We show a couple of basic things you can do with these traits here:

```
template<typename E> string EnumToString(E e); // generic enum-to-string
template<typename E> E StringToEnum(string s); // generic string-to-enum
template<typename E> std::set<E> AllEnums; // generic get set of all enums in
enumerator
template<class S> DumpStruct<S>(); // outputs members of any struct by
iterating over members and printing member name, ": " and then cout the member
value.

enum Color { red, blue, green };

color c1 = red;

string s1 = EnumToString(c1); // s == "red"

string s2 = "red";
```

```
color c2 = StringToEnum(s2); // c2 == red

for (color c : AllEnums<Color>)
   cout << EnumToString(c) << ", "; // prints: red, blue green

struct Rec { string name; Color favorite_color; Date birthday; }

Person person = { "tom", blue, Date(12,8,1982) };

DumpStruct<Person>();
// prints:
//   name: tom
//   favorite_color: blue
//   birthday: 12/8/1982
```

In general, by enumerating the names (as strings), types and pointer-to-members of the members of classes and enumerations - one can walk the subobject lattice and implement a large number of use cases. This can be done at both run-time, and, with standard C++14 constexpr programming, at compile-time.

FAQ

Is this implemented?

Yes. The cosmetic changes in this proposal have not been applied but the bulk of the implementation is at https://github.com/ChristianKaeser/clang-reflection. The implementation involves a handful of intrinsics that inspect the AST on instantiation of the traits, much the same as the existing type traits library works.

Does this have wording?

Yes. A detailed wording was given in a previous proposal, it has not been updated for the cosmetic changes here. Our intention is to update the wording based on the Specification section if SG7 passes std::class_traits and std::enum_traits at Lenexa.

Why the size/index interface?

This was discussed in N4113 under "Size / Index Interface". An alternative was considered involving passing template template parameters, but it turned out to be impractical and inconsistent with the standard library, that makes no use of template template parameters as yet. Like std::tuple, std::get and similar components, the proposed "compile-time lists" are rolled up with pack expansion of std::index_sequences.

What about access control?

This proposal includes reflection of public members, on which there is consensus. We also reveal a possible roadmap for future proposals under "Future Roadmap" that may include access-checked private member reflection. This proposal is forward compatible with any of the options discussed previously regarding access control.

What about use cases?

The previous revisions of this paper go through a number of use cases. There is also a paper called "has_member_function" by Andrew Tomazos at:

https://googlegroups.com/a/isocpp.org/group/reflection/attach/78a90edd12996432/has_member_functionDemoForN4027.pdf?part=0.1

That shows a worked example of how to build, step-by-step, upon the proposed traits to implement high-level reflection library features.

The general use cases for reflection are outlined in N3814.

Some use cases enabled by this proposal include converting enums to and from strings, serializing/deserializing/dumping aggregate types, exposing public interfaces of classes through RMI or to scripting languages/configs, supporting memberwise operations, and many more.