Reflection Type Traits For Classes, Unions and Enumerations (rev 3)

Document Number: N4113
Revises: N4027
Date: 2014-07-02
Project: Programming Language C++ - SG7 Reflection
Reply-To: Andrew Tomazos <andrewtomazos@gmail.com>,
Christian Kaeser <christiankaeser87@gmail.com>

Summary

We propose the addition of two new kinds of type trait, and four new type traits to the Metaprogramming and Type Traits Standard Library [meta] for the purposes of low-level reflection of classes, unions and enumerations. These primitives are thin wrappers for compiler instrinsics, require zero core language changes, and are designed to be consistent with the existing [meta] library. Pure library authors can then compose the primitives to provide higher-level reflection libraries and facilities.

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

Synoposis

This proposal adds the following entities to `<type_traits>`:

```
// (I) IntegralTrait, (T) TypeTrait, (S) TextTrait, (L) ListTrait

namespace std
{
    enum access_levels
    { public_access, protected_access, private_access };

    namespace enumerator                                  // L
    {
        template<typename E> struct list_size;            // I
        template<typename E, size_t I> struct identifier; // S
        template<typename E, size_t I> struct value;      // I
    }

    namespace base_class                                  // L
    {
        template<class C> struct list_size;               // I
        template<class C, size_t I> struct type;          // T
        template<class C, size_t I> struct is_virtual;    // I
        template<class C, size_t I> struct access_level;  // I
    }

    namespace class_member                                // L
    {
        template<class C> struct list_size;               // I
        template<class C, size_t I> struct name;          // S
        template<class C, size_t I> struct pointer;       // I
        template<class C, size_t I> struct access_level;  // I
    }

    namespace nested_type                                 // L
    {
        template<class C> struct list_size;               // I
        template<class C, size_t I> struct identifier;    // S
        template<class C, size_t I> struct type;          // T
        template<class C, size_t I> struct access_level;  // I
    }
}
```

Background

The Metaprogramming and Type Traits Standard Library contains a set of templates, we shall call *Traits,* for reflecting types.

Conceptually there are currently two kinds of Traits, we shall call *IntegralTrait* and *TypeTrait*.

IntegralTrait

An IntegralTrait is a class template that produces a value of a type suitable for a non-type template parameter on instantiation (which is formally actually a large superset of integral types). It is derived from `std::integral_constant` and (from N3854) has a variable template with _v appended. For example `std::extent` is an IntegralTrait and might be implemented as follows:

```
template<typename A, unsigned I = 0> struct extent
    : integral_constant<unsigned, __extent(A,I)> {};

template <typename A, unsigned I = 0>
constexpr size_t extent_v = extent<T, I>::value;
```

and so can be used as follows:

```
static_assert(std::extent_v<int[42]> == 42); // ok
```

TypeTrait

A TypeTrait is a class template that produces a type on instantiation. It contains a member typedef named `type`.   For example `std::remove_pointer` is a TypeTrait, and might be implemented as follows:

```
template<typename T> struct remove_pointer     { typedef T type; }
template<typename T> struct remove_pointer<T*> { typedef T type; }
...

template<typename T> using remove_pointer_t
    = remove_pointer<T>::type;
```

and so can be used as follows:

```
static_assert(std::is_same_v<float, std::remove_pointer_t<float*>>);
```

New Kinds of Trait

We propose the addition of two new kinds of trait, *TextTrait* and *ListTrait*.

TextTrait

A TextTrait is a class template that produces a compile-time string on instantiation. It contains a static data member named `value` of a compile-time string type in UTF-8 encoding. There is a variable template of reference type referring to the value member _v. For example, a theoretical `std::foo` TextTrait might be implemented as follows:

```
template<typename T> struct foo
{
    static constexpr std::string_literal<...> value = __foo(T);
};

template<typename T> constexpr std::string_literal<...>& foo_v
    = foo<T>::value;
```

And so could be used as follows:

```
static_assert(foo_v<C> == “bar”); // ok
```

ListTrait

A *ListTrait* is a namespace containing a set of related Traits that produce information about a list of entities. It contains an IntegralTrait named `list_size` that produces the number of elements of the list. The remaining Traits are called *ListAccessors*. Each ListAccessor has the same template parameters as list_size with an index I of type size_t appended that identifies which element of the list is being inspected. For example a theoretical std::foo ListTrait with 3 ListAccessors, (1) `std::foo::bar` an IntegralTrait; (2) `std::foo::baz` a TextTrait; and (3) `std::foo::qux` a TypeTrait; might be implemented as follows:

```
namespace foo // (ListTrait)
{
    // list_size
    template<typename T> struct list_size
        : integral_constant<size_t, __foo_list_size(A,I)> {};

    template<typename T>
    constexpr size_t list_size_v = list_size<T>::value;
```

```
        // foo::bar (IntegralTrait)
        template<typename T, size_t I>
        struct bar : integral_constant<size_t, __foo_bar(T,I)> {};

        template <typename T, size_t I>
        constexpr size_t bar_v = bar<T, I>::value;

        // foo::baz (TextTrait)
        template<typename T, size_t I>
        struct baz
        {
            static constexpr std::string_literal<...> value
                = __foo_baz(T);
        };

        template<typename T, size_t I>
        constexpr std::string_literal<...>& baz_v
            = baz<T,I>::value;

        // foo::qux (TypeTrait)
        template<typename T, size_t I>
        struct qux
        {
            using type = __foo_qux(T,I);
        };

        template<typename T, size_t I>
        using qux_t = qux<T,I>::type;
    };
```

And could be used as follows:

```
    static_assert(std::foo::list_size_v<C> == 42);

    // We then inspect the 14th member...
    static_assert(std::foo::bar_v<C,13> == 109);
    static_assert(std::foo::baz_v<C,13> == "blah");
    static_assert(std::is_same_v<U, std::foo::qux_t<C,13>>);
```

New Traits

We propose the addition of four new ListTraits. They are:

`std::enumerator` : The enumerators of an enumeration type.
`std::base_class` : The base classes of a class type
`std::class_member` : Some members of some class or union types
`std::nested_type` : Some member types of a class type.

We also will mention two other possible future ListTraits:

`std::constructor` : The constructors of a class type
`std::member_template` : The member templates of a class type

Similarly, other ListTraits not mentioned here could be added in the future.

std::enumerator

std::enumerator is a ListTrait which provides information about the enumerators of an enumeration in declared order. It contains two ListAccessors:

`value`, an IntegralTrait, the enumerator value
`identifier`, a TextTrait, the enumerators identifier

So it could be used as follows:

```
enum foo { bar = 13, baz = 42 };

static_assert(std::enumerator::list_size<foo> == 2);

static_assert(std::enumerator::value_v<foo,0> == bar);
static_assert(std::enumerator::value_v<foo,0> == foo(13));
static_assert(std::enumerator::identifier_v<foo,0> == "bar");

static_assert(std::enumerator::value_v<foo,1> == baz);
static_assert(std::enumerator::value_v<foo,1> == foo(42));
static_assert(std::enumerator::identifier_v<foo,1> == "baz");
```

This provides complete reflection of an enum-specifier as can be shown from the grammar, so no further ListAccessors of std::enumerator will be needed.

std::base_class

`std::base_class` is a ListTrait which provides information about the base classes of a class type, in the declared order they appear in the base-clause (after pack expansion).

`type`, a TypeTrait, the type of the base class
`is_virtual`, an IntegralTrait, true iff the base class is virtual
`access_level`, an IntegralTrait, the access control level (public, private, protected) of the base class.

So it could be used as follows:

```
class A {}; class B {}; class C {};

class D : public A, protected virtual B, private C {};

static_assert(std::base_class::list_size_v<D> == 3);

static_assert(std::is_same_v<A, std::base_class::type_t<D,0>>);
static_assert(std::is_same_v<B, std::base_class::type_t<D,1>>);
static_assert(std::is_same_v<C, std::base_class::type_t<D,2>>);

static_assert(std::base_class::is_virtual_v<D,0>> == false);
static_assert(std::base_class::is_virtual_v<D,1>> == true);
static_assert(std::base_class::is_virtual_v<D,2>> == false);

static_assert(std::base_class::access_level_v<D,0>>
              == std::public_access));
static_assert(std::base_class::access_level_v<D,1>>
              == std::protected_access));
static_assert(std::base_class::access_level_v<D,2>>
              == std::private_access));
```

std::class_member

`std::class_member` is a ListTrait which provides information about certain class members of some class and union types, shown in declared order.

The subject type is required to not contain data members of reference type or bit fields. The subject type shall not be a lambda type. A violation of either of these requirements is ill-formed, and should result in a compile-time error.

The reason for this design decision is that we may propose separately later the addition of a pointer-to-member of reference type and perhaps a pointer-to-member of a bitfield to the core language. It is not known yet whether such a proposal will be made and accepted. If it is not, then we can create a seperate ListTrait for reference members or bitfields. Until such time as it is known which of the two directions we will go in, we should leave reflection of such types ill-formed so as not to break unchanged code that depends on it after a compiler upgrade. That is, we do not know yet whether members of reference type or bitfields will be included in std::class_member.

Likewise for lambda types, they are compiler-generated, so do not have a standardized class definition on which std::class_member can be easily defined. We make them ill-formed for use with the first iteration of this proposal, so we keep the option to add them later.

The members of the class/union type shown in std::class_member are functions or objects that have a direct simple declaration in the definition of the type, and are not member templates or instantiations thereof. Notably 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_member` has 3 ListAccessors, they are:

`name`, a TextTrait, if the member name is an identifier, the text of that identifier, if the member name is an operator-function-id, then the text “operator“ appended with a space character and then the canonical non-terminal of operator (“operator +”, “operator new[]”, “operator <<=”, etc), if the member is unnamed the empty string, otherwise implemented-defined
`pointer`, an IntegralTrait, the result of the expression &C::m. That is a pointer-to-member for a non-static member and a pointer for a static member. Anonymous unions are reflected as a single sub-object of union type. The sub-objects of the union are not visible in the std::class_member list of the enclosing class type (they are visible by recursion on the type of the union sub-object).
`access_level`, an IntegralTrait, the access level of the member (public, protected or private)

So it could be used as follows:

```
class C
{
public: int x;
protected: void f();
private: static char32_t* p;
};

static_assert(std::class_member::list_size<C> == 3);

static_assert(std::class_member::name<C,0> == “x”);
static_assert(std::class_member::name<C,1> == “f”);
static_assert(std::class_member::name<C,2> == “p”);

// within a member function of C so that RHS is well-formed
```

```
static_assert(std::class_member::pointer<C,0> == &C::x);
static_assert(std::class_member::pointer<C,1> == &C::f);
static_assert(std::class_member::pointer<C,2> == &C::p);

static_assert(std::class_member::access_level<C,0>
    == std::public_access);
static_assert(std::class_member::access_level<C,1>
    == std::protected_access);
static_assert(std::class_member::access_level<C,2>
    == std::private_access);
```

std::nested_type

`std::nested_type` is a ListTrait which provides information about the nested types of a class type. Each nested type shall be one that is declared with a name in a class definition, and the list is in declared order. (Unnamed classes and unions can be reached through std::class_member by deducing the type of the data members of the enclosing class type). 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. In particular member template classes and member alias declarations (and instantiations thereof) are not reflected.

So all of T1 through T7 are reflected in std::nested_type of S:

```
    struct S
    {
        using T1 = int;
        typedef int T2;
        class T3;
        class T4 {};
        union T5;
        union T6 {};
        enum T7 {};

        union { int x; } // not shown in std::nested_type,
                         // shown in std::class_member
    };
```

`std::nested_type` has three ListAccessors, they are:

`identifier`, a TextTrait, the identifier of the nested type.
`type`, a TypeTrait, the type of the nested type
access_level, an IntegralTrait, the access level of the nested type (public, private, protected)

So it could be used as follows:

```
struct foo
{
public: typedef char32_t bar;
protected: using baz = float;
private: struct qux {};
};

static_assert(std::nested_type::list_size<foo> == 3);

static_assert(std::nested_type::identifier_v<foo,0> == “bar”);
static_assert(std::nested_type::identifier_v<foo,1> == “baz”);
static_assert(std::nested_type::identifier_v<foo,2> == “qux”);

static_assert(std::is_same_v<foo::bar,
              std::nested_type::type_t<foo,0>>);
static_assert(std::is_same_v<foo::baz,
              std::nested_type::type_t<foo,1>>);
static_assert(std::is_same_v<foo::qux,
              std::nested_type::type_t<foo,2>>);

static_assert(std::nested_type::access_level<foo, 0>
    == std::public);
static_assert(std::nested_type::access_level<foo, 1>
    == std::protected);
static_assert(std::nested_type::access_level<foo, 2>
    == std::private);
```

Access Control

The most controversial aspect of this proposal has been what to do about access control.

First, we should note that in every other language that has both access control and a reflection facility (notably Java and C#), it is possible to access and modify private members using reflection.

Second, you should notice that what we propose are low-level primitives for reflection library builders. It is extremely difficult to use them directly to access a specific private member by name of a specific class. It would require quite some determination to do it intentionally, and would be impossible to happen by accident.

We will now enumerate the options that were considered, and why we have settled on the one we have.

Option #1: Only reflect public members.
Pros: No risk of breaking encapsulation
Cons: Key reflection use cases become impossible

Option #2: Reflect all members, but make their access level visible.
Pros: Simple. Makes the reflection use cases possible.
Cons: Risk of intentional abuse to access and modify private members

Option #3: Require some opt-in line in a class definition like “friend reflection”
Pros: Requires an opt-in from class designers
Cons: Because class designers can opt-out, if they do not, they become responsible for misuse of private members through reflection. This is not appropriate, the blame should be on the party that misused reflection to access private members in non-aggregate fashion.

Option #4: Reflect public members by default, but have an override switch.
Pros: Same as Option #2 with a reduction of the Simple part.
Cons: Increases complexity, risk of abuse still exists.

Option #5: Extend the primitives to give enough information to allow a reflection library to check friendship within pure library code. Pass in a context class.
Pros: Allows reflection libraries to do fine-grained access check
Cons: Extremely complex. Risk of intentional abuse still exists by using primitives directly.

Option #6: Check access control where primitives are used with core language extension
Pros: Similar to how access control is checked with name lookup
Cons: Crazy complex. Requires core language changes. Not clearly specified / infeasible. Doesn’t really solve the problem, ends up being equivalent to Option #3.

After extensive discussions we ended up settling on Option #2. Using Option #2 in the low-level primitives enables reflection library authors to choose between Option #1, #2, #3 and #4 for their external interface.

To implement Option #2 we introduce an enumeration std::access_levels

```
enum access_levels
{
    public_access,
    protected_access,
    private_access
};
```

And then expose a ListAccessor `access_level` of that type for each ListTrait that is subject to access control. High-level reflection libraries can then use this information to implement one of the first four options (or possibly others not considered).

Size / Index Interface

There appears to be clear and strong consensus on the schema, but there has been some discussion of the interface of ListTrait, and they have undergone a change since last revision to reduce the number of names introduces into namespace std.

We considered a few different options on the transformation. Our design decisions were as follows:

We didn't want to have member templates of templates. There are uncomfortable restrictions on what can be done with such member templates as opposed to "free" namespace-scope templates. Also there are usages issues on dependant names. It was also felt that keeping them as namespace-scope templates was more consistent with the existing traits, so we could specify a ListTrait as a namespace containing a set of Traits with the "usual" interface.

We also considered having the ListAccessors as data members of a single complete object (per element of a ListTrait). The concern here was regarding the zero overhead principle. Under certain usages the enclosing complete object would be odr-used, and so the user would pay for ListAccessors they did not use. For example using pointer from std::class_member would entail odr-using name.

With regard to the compiler-side performance of the size-index interface, the implementation of the instrinsics backing a ListTrait can lazily construct an internal random-access index in O(list_size) time and space (if the data structure it uses doesn't already have one). In the typical case each element will be accessed (in constant time), and so the total time cost per access remains amortized constant (keep in mind every ListTrait is immutable) which is optimal, and the O(n) size of the index is the same as the parse tree from the syntax so makes no asymptotic contribution to size.

There were ideas suggested in Rapperswil SG7 about what we understood to be a possible compile-time iterator-style interface to ListTrait rather than a size-index interface, that still did not require core language changes but could use a linked structure without constructing an internal array index. We don't have a clear specification of this idea to evaluate the pros/cons, and have reached out to the commenter for more information, but at time-of-writing have not heard back.

Wording and Reference Implementation

There is a detailed wording of the previous proposal revision N4027, it has not been updated with the minor changes and additions stated in this proposal.

There is a reference implementation linked from N3815, it contains a set of size-index intrinsics similar in style to those specified here for accessing class members, base classes, enumerators and so on, but likewise has not been updated against every minor detail / change / addition.

There are two sister papers of the previous proposal (N4027) of interest that show how to reflect the fundamental types and built-in compound types (we already have this in C++), google “angloname tomazos”. There is also a demo use case for class_member, google “has_member_function tomazos”. These were presented to SG7 at Rapperswil.