Document Number:	**N4027**
Revises:	N3815
Date:	2014-05-22
Project:	Programming Language C++
	SG7 Reflection Working Group
Reply to:	Andrew Tomazos, Christian Kaeser
	andrewtomazos@gmail.com
	christiankaeser87@googlemail.com

Type Member Property Queries (rev 2)

0.1 Proposal [proposal]

0.1.1 Informal Summary [proposal.summary]

We propose the addition of a set of property queries and helper types to the Metaprogramming and Type Traits Standard Library, for the purposes of compile-time reflection of user-defined types. User-defined types include classes, unions and enumerations. The queries serve as low-level primitives to enable enumeration and inspection of type members, such as base classes, data members, member functions and enumerators. These building blocks can then be composed by pure library authors to provide higher-level reflection facilities. This proposal is a superset of N3815 based on SG7 feedback from Issaquah. The detailed rationale of N3815 will not be repeated here.

0.2 Design [design]

1 The design is based on a *ListTrait* concept. A ListTrait is a related set of class templates. Each ListTrait has a *basis identifier* which is a singular noun that indicates the kind of the subject of the list. Each ListTrait also has a set of *identifying template parameters* associated with it, and *associated requirements* on those template parameters, that identify a legal subject of the list. In this proposal we put forward three ListTraits for standardization. They are `enumerator` (N3815 - the list of enumerators of an enumeration), `base_class` (the list of base classes of a class type) and `class_member` (a subset of class members of a subset of class types). We also mention possible future ListTraits `nested_type` (nested type members of a class scope), `constructor` (the constructors of a class type), `member_template` (member templates of a class type). In all cases so far, the identifying template parameters are simply a single type parameter T that identify the subject type.

2 For each ListTrait `LT` there is a class template named with the basis identifier appended with `_list_size` (ie `LT_list_size`). As per a UnaryTrait `LT1_list_size` shall be derived from a BaseCharacteristic `std::integral_constant` and hold an integer representing the number of elements of the list. It shall take the identifying template parameters and (effectively) static assert the associated requirements on instantiation.

3 Each ListTrait has an associated set of *ListAccessors*. Each ListAccessor has an *accessor identifier* in singular tense describing the characteristic of the list item it describes. Each ListAccessor is realized by a class template with a name formed by concatenating the ListTrait basis identifier, underscore and the accessor identifier. Each ListAccessor has a BaseCharacteristic (eg type, integral, text) that specifies its form. The template parameters of a ListAccessor are the identifying template parameters followed by a non-type template parameter `I` of type `size_t` that is the index of the subject in the list. The associated requirements are statically asserted on the identifying template parameters and, additionaly, `I` is staticly asserted to be in-bounds of the list size. (I is zero-indexed)

4 There are essentially two kinds of existing BaseCharacteristic that ListAccessors can have. The first is a type characteristic like TypeTransform. In these cases there is a nested type named `type`, and a secondary alias template formed to that nested member by appending `_t`. The second is an integral characteristic like UnaryTrait, where the class template derives from `std::integral_constant`. We introduce additional kinds of BaseCharacteristic. The first one is a text characteristic, where the ListAccessor is derived from a `std::text_constant` class which wraps a value that behaves like a string literal (compile-time string). The second is a pointer characteristic which derives from `std::pointer_constant` and holds pointers and pointers-to-members. The third is an access level characteristic which derives from `std::access_constant` and holds an enum representing one of the three access levels.

5 The ListAccessors of the `enumerator` ListTrait are as per N3815: (1) `identifier` of text characteristic, and (2) `value` of integral characteristic.

6 The ListAccessors of the `base_class` ListTrait are (1) `type` of type characteristic, (2) `is_virtual` of integral characteristic and (3) `access_level` of access level characteristic.

7 The ListAccessors of the `class_member` ListTrait are (1) `name` of text characteristic, (2) `pointer` of pointer characteristic and (3) `access_level` of access level characteristic.

8 Future directions would involve (a) creating additional kinds of BaseCharacteristic; (b) adding ListAccessors to existing ListTraits; (c) adding new ListTraits; and/or (d) relaxing associated requirements. None of these directions create breaking changes.

0.3 Technical Specifications [proposal.techspecs]

0.3.1 Header <type_traits> synopsis [meta.type.synop]

```
namespace std {

  // helper class:
  enum access_level { public_access, protected_access, private_access };

  template<access_level A> using access_constant = integral_constant<access_level, A>;
  template<typename Ptr, Ptr p> using pointer_constant = integral_constant<Ptr, p>;

  template <char... cs> struct text_constant;

  // type properties:
  template <class T> struct is_reflectible_class;

  // type property queries:
  template<class E> struct enumerator_list_size;
  template<class E, size_t I> struct enumerator_identifier;
  template<class E, size_t I> struct enumerator_value;

  template<class C> struct base_class_list_size;
  template<class C, size_t I> struct base_class_type;
  template<class C, size_t I> struct base_class_is_virtual;
  template<class C, size_t I> struct base_class_access_level;

  template<class C> struct class_member_list_size;
  template<class C, size_t I> struct class_member_name;
  template<class C, size_t I> struct class_member_pointer;
  template<class C, size_t I> struct class_member_access_level;

} // namespace std
```

0.3.2 Helper classes [meta.help]

```
namespace std {
    enum access_level {
      public_access,
      protected_access,
      private_access
    };

    template<access_level A>
    using access_constant = integral_constant<access_level, A>;

    template<typename Ptr, Ptr p>
    using pointer_constant = integral_constant<Ptr, p>;

    template <char... cs>
    struct text_constant {
      static constexpr char value[] = {cs..., '\0'};
      static constexpr size_t size = sizeof...(cs);
      typedef char value_type[sizeof...(cs)+1];
      typedef text_constant<cs...> type;
    };
  }
```

1 The class template `text_constant` shall hold text in UTF-8 encoding. If holding source text, it shall have any universal-character-names decoded.

0.3.2.1 Type properties [meta.unary.prop]

1 A class or union is a *reflectible class type* if it does not contain a a data member of reference type, or a bit field.

2 A member `M` of a reflectible class type `C` is a *reflectible member* if `M` is a direct class member of `C`, `M` has a declaration other than a using declaration in the class specifier of `C`, it is a function or an object, it is not a constructor or destructor, it is not a member template or the instantiation of a member template.

Table 1 — Type property queries

Template	Value
`template <class T>` `struct type_name;`	An implementation-defined `std::text_constant` such that for a type A and a type B `std::type_name<A>` holds the same text as `std::type_name<B>` if and only if `std::is_same<A,B>`
`template <class T>` `struct is_reflectible_class;`	`true_type` if T is a reflectible class type, `false_type` otherwise. *Requires:*`std::is_class<T>` or `std::is_union<T>`
`template<class E>` `struct enumerator_list_size;`	An integer value representing the number of enumerators in E. *Requires:*`std::is_enum<E>`
`template<class E, size_t I>` `struct enumerator_identifier;`	A `std::text_constant` holding the identifier of the I'th enumerator of E in declared order, where indexing of I is zero-based. *Requires:*`std::is_enum<E>` *Requires:* I shall be nonnegative and less than `std::enumerator_list_size<E>`
`template<class E, size_t I>` `struct enumerator_value;`	A value of type E that is the I'th enumerator of E in declared order, where indexing of I is zero-based. *Requires:*`std::is_enum<E>` *Requires:*I shall be nonnegative and less than `std::enumerator_list_size<E>`
`template <class C>` `struct base_class_list_size;`	An integer, the number of direct base classes of class C. *Requires:*`std::is_class<C>`

Table 1 — Type property queries (continued)

Template	Value
`template <class C, size_t I>` `struct base_class_type;`	The type of the I'th direct base class of C in declared order after pack expansion, where indexing of I is zero-based. *Requires:*`std::is_class<T>` *Requires:*`I` shall be non-negative and less than `std::base_class_list_size<C>`
`template <class C, size_t I>` `struct base_class_is_virtual;`	`true_type` if the I'th direct base class of C is virtual, `false_type` otherwise. Indexing of I is zero-based, and of declared order after pack expansion. *Requires:*`std::is_class<T>` *Requires:*`I` shall be non-negative and less than `std::base_class_list_size<C>`
`template <class C, size_t I>` `struct base_class_access_level;`	An `std::access_constant` representing the access level of the I'th direct base class of C in declared order after pack expansion, where indexing of I is zero-based. *Requires:*`std::is_class<T>` *Requires:*`I` shall be non-negative and less than `std::base_class_list_size<C>`
`template <class C>` `struct class_member_list_size;`	An integer, the number of reflectible members of class C. *Requires:*`std::is_reflectible_class<C>`
`template <class C, size_t I>` `struct class_member_name;`	A `std::text_constant` holding the name of the I'th reflectible member of C, where indexing of I is zero-based and in declared order. If the name is an identifier it shall be that identifier, otherwise the format of the name is implementation-defined. *Requires:*`std::is_reflectible_class<C>` *Requires:*`I` shall be non-negative and less than `std::class_member_list_size<C>`
`template <class C, size_t I>` `struct class_member_pointer;`	A `std::pointer_constant` holding a pointer to the I'th reflectible member of C, where indexing of I is zero-based and in declared order. If the member is non-static it shall be a pointer-to-member, otherwise for static members it shall be a pointer. *Requires:*`std::is_reflectible_class<C>` *Requires:*`I` shall be non-negative and less than `std::class_member_list_size<C>`
`template <class C, size_t I>` `struct class_member_access_level;`	An `std::access_constant` representing the access level of the I'th reflectible member of C, where indexing of I is zero-based and in declared order. *Requires:*`std::is_reflectible_class<C>` *Requires:*`I` shall be non-negative and less than `std::class_member_list_size<C>`

0.4 Appendix A: Example Implementation [impl]

1 We show a stub implementation for the additional properties added since N3815, with the specializations hand-written for the following example classes. (The N3815 reference implementation shows how the property queries are supported via real intrinsics.)

```
struct foo
```

```
{
        int bar;
        char baz();
        static int qux;
};

struct B1 {};
struct B2 {};
struct V {};
struct D1 : private B1, public virtual V {};
struct D2 : protected B2, private virtual V {};
struct E : public D1, protected D2 {};
```

2 The compiler will then automatically generate the properties as the property queries are instantiated as if the following specializations were provided:

```
#include <type_traits>

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

  template<access_level A>
  using access_constant = integral_constant<access_level, A>;

  template<typename Ptr, Ptr p>
  using pointer_constant = integral_constant<Ptr, p>;

  template <char... cs>
  struct text_constant {
    static constexpr char value[] = {cs..., '\0'};
    static constexpr size_t size = sizeof...(cs);
    typedef char value_type[sizeof...(cs)+1];
    typedef text_constant<cs...> type;
  };
  template<class C> struct class_member_list_size;
  template<class C, size_t I> struct class_member_name;
  template<class C, size_t I> struct class_member_pointer;
  template<class C, size_t I> struct class_member_access_level;

  template<class C> struct base_class_list_size;
  template<class C, size_t I> struct base_class_type;
  template<class C, size_t I> struct base_class_is_virtual;
  template<class C, size_t I> struct base_class_access_level;

  template<> struct class_member_list_size<foo> : integral_constant<size_t, 2> {};

  template<> struct class_member_name<foo,0> : text_constant<'b','a','r'> {};
  template<> struct class_member_name<foo,1> : text_constant<'b','a','z'> {};
  template<> struct class_member_name<foo,2> : text_constant<'q','u','x'> {};

  template<> struct class_member_pointer<foo,0> : pointer_constant<int foo::*, &foo::bar> {};
```

```
    template<> struct class_member_pointer<foo,1> : pointer_constant<char (foo::*)(), &foo::baz> {};
    template<> struct class_member_pointer<foo,2> : pointer_constant<int*, &foo::qux> {};

    template<> struct class_member_access_level<foo,0> : access_constant<public_access> {};
    template<> struct class_member_access_level<foo,1> : access_constant<public_access> {};
    template<> struct class_member_access_level<foo,2> : access_constant<public_access> {};

    template<> struct base_class_list_size<B1> : integral_constant<size_t, 0> {};
    template<> struct base_class_list_size<B2> : integral_constant<size_t, 0> {};
    template<> struct base_class_list_size<V>  : integral_constant<size_t, 0> {};
    template<> struct base_class_list_size<D1> : integral_constant<size_t, 2> {};
    template<> struct base_class_list_size<D2> : integral_constant<size_t, 2> {};
    template<> struct base_class_list_size<E>  : integral_constant<size_t, 2> {};

    template<> struct base_class_type<D1,0> { typedef B1 type; };
    template<> struct base_class_type<D1,1> { typedef V  type; };
    template<> struct base_class_type<D2,0> { typedef B2 type; };
    template<> struct base_class_type<D2,1> { typedef V  type; };
    template<> struct base_class_type<E,0>  { typedef D1 type; };
    template<> struct base_class_type<E,1>  { typedef D2  type; };

    template<> struct base_class_access_level<D1,0> : access_constant<private_access> {};
    template<> struct base_class_access_level<D1,1> : access_constant<public_access> {};
    template<> struct base_class_access_level<D2,0> : access_constant<protected_access> {};
    template<> struct base_class_access_level<D2,1> : access_constant<private_access> {};
    template<> struct base_class_access_level<E,0>  : access_constant<public_access> {};
    template<> struct base_class_access_level<E,1>  : access_constant<protected_access> {};
  }
```