Abstract  In N3996 [1] we presented some ideas on the design and possible implementation of a compile-time reflection facility for standard C++. N3996 also contained extensive discussion about usefulness of reflection, description of several use-cases and motivational examples from the Mirror reflection utilities [2]. The actual proposal was however found to be confusing in some points. This paper tries to make the proposal more concise and detailed.

Contents

1. Introduction 4
  1.1. Differences between n3996 and n4111 .......................... 4

2. Metaobject concepts 5
  2.1. Categorization and Traits ........................................... 6
    2.1.1. Metaobject category tags .................................... 6
    2.1.2.Specifier category tags ...................................... 8
  2.2. StringConstant .................................................. 10
  2.3. Metaobject Sequence .............................................. 11
    2.3.1. size ......................................................... 12
    2.3.2. at ........................................................... 12
  2.4. Metaobject ....................................................... 13
    2.4.1. is_metaobject ............................................... 14
    2.4.2. metaobject_category ...................................... 14
    2.4.3. Traits ...................................................... 14
  2.5. MetaSpecifier .................................................... 15
    2.5.1. specifier_category ........................................ 16
    2.5.2. keyword ................................................... 16
  2.6. MetaNamed ........................................................ 17
    2.6.1. base_name .................................................. 17
    2.6.2. full_name .................................................. 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.3. named typedef</td>
<td>19</td>
</tr>
<tr>
<td>2.6.4. named_mem_var</td>
<td>20</td>
</tr>
<tr>
<td>2.7. MetaScoped</td>
<td>22</td>
</tr>
<tr>
<td>2.7.1. scope</td>
<td>22</td>
</tr>
<tr>
<td>2.8. MetaNamedScoped</td>
<td>23</td>
</tr>
<tr>
<td>2.9. MetaScope</td>
<td>23</td>
</tr>
<tr>
<td>2.9.1. members</td>
<td>23</td>
</tr>
<tr>
<td>2.10. MetaClassMember</td>
<td>24</td>
</tr>
<tr>
<td>2.10.1. access_specifier</td>
<td>24</td>
</tr>
<tr>
<td>2.11. MetaGlobalScope</td>
<td>25</td>
</tr>
<tr>
<td>2.12. MetaNamespace</td>
<td>25</td>
</tr>
<tr>
<td>2.13. MetaType</td>
<td>26</td>
</tr>
<tr>
<td>2.13.1. original_type</td>
<td>26</td>
</tr>
<tr>
<td>2.14. MetaTypedef</td>
<td>27</td>
</tr>
<tr>
<td>2.14.1. type</td>
<td>27</td>
</tr>
<tr>
<td>2.15. MetaClass</td>
<td>28</td>
</tr>
<tr>
<td>2.15.1. elaborated_typeSpecifier</td>
<td>28</td>
</tr>
<tr>
<td>2.15.2. base_classes</td>
<td>29</td>
</tr>
<tr>
<td>2.16. MetaEnum</td>
<td>29</td>
</tr>
<tr>
<td>2.16.1. base_type</td>
<td>29</td>
</tr>
<tr>
<td>2.17. MetaEnumClass</td>
<td>30</td>
</tr>
<tr>
<td>2.17.1. base_type</td>
<td>30</td>
</tr>
<tr>
<td>2.18. MetaOverloadedFunction</td>
<td>31</td>
</tr>
<tr>
<td>2.18.1. overloads</td>
<td>31</td>
</tr>
<tr>
<td>2.19. MetaFunction</td>
<td>32</td>
</tr>
<tr>
<td>2.19.1. linkageSpecifier</td>
<td>32</td>
</tr>
<tr>
<td>2.19.2. constexprSpecifier</td>
<td>32</td>
</tr>
<tr>
<td>2.19.3. result_type</td>
<td>33</td>
</tr>
<tr>
<td>2.19.4. parameters</td>
<td>33</td>
</tr>
<tr>
<td>2.19.5. noexceptSpecifier</td>
<td>33</td>
</tr>
<tr>
<td>2.19.6. exceptions</td>
<td>34</td>
</tr>
<tr>
<td>2.19.7. constSpecifier</td>
<td>34</td>
</tr>
<tr>
<td>2.19.8. is_pure</td>
<td>34</td>
</tr>
<tr>
<td>2.20. MetaInitializer</td>
<td>35</td>
</tr>
<tr>
<td>2.21. MetaConstructor</td>
<td>36</td>
</tr>
<tr>
<td>2.22. MetaOperator</td>
<td>36</td>
</tr>
<tr>
<td>2.23. MetaTemplate</td>
<td>37</td>
</tr>
<tr>
<td>2.23.1. template_parameters</td>
<td>37</td>
</tr>
<tr>
<td>2.23.2. instantiation</td>
<td>38</td>
</tr>
<tr>
<td>2.24. MetaTemplateParameter</td>
<td>38</td>
</tr>
<tr>
<td>2.24.1. position</td>
<td>39</td>
</tr>
<tr>
<td>2.24.2. is_pack</td>
<td>39</td>
</tr>
<tr>
<td>2.25. MetaInstantiation</td>
<td>40</td>
</tr>
<tr>
<td>2.25.1. template_arguments</td>
<td>40</td>
</tr>
</tbody>
</table>
2.25.2. template_ .......................... 40
2.26. MetaInheritance .......................... 41
  2.26.1. access_specifier ......................... 41
  2.26.2. inheritanceSpecifier ............ 42
  2.26.3. base_class ...................... 42
2.27. MetaVariable .............................. 42
  2.27.1. storageSpecifier .................. 43
  2.27.2. type ................................ 43
2.28. MetaParameter ............................. 43
  2.28.1. type ................................ 44
  2.28.2. position ......................... 44
  2.28.3. is_pack ......................... 45
2.29. MetaConstant ............................... 45
  2.29.1. type ................................ 45
  2.29.2. value ......................... 46

3. Reflection operator 46

4. Additions to the library 48
  4.1. Metaobject expressions ............ 48
      4.1.1. evaluate ....................... 48
  4.2. Default implementation of metafunctions ............. 49

5. Rationale 50
  5.1. Why metaobjects, why not reflect directly? .......... 50
  5.2. Why are the metaobjects anonymous? .............. 52
  5.3. Why this rendering of metaobjects? ............ 53

6. Known issues 54

7. Acknowledgements 54

8 References 54

Appendix A. Examples 54
  A.1. Basic traits .......................... 55
  A.2. Global scope reflection ............ 55
  A.3. Namespace reflection ............... 56
  A.4. Type reflection .................... 57
  A.5. Typedef reflection ................. 58
  A.6. Class reflection .................... 60
1. Introduction

Reflection and reflective programming can be used for a wide range of tasks, such as the implementation of serialization-like operations, object-relational mapping, remote procedure calls, scripting, automated GUI-generation, implementation of several software design patterns, etc. C++ as one of the most prevalent programming languages lacks a standardized reflection facility.

In this paper we propose to add native support for compile-time reflection to C++ by the means of compiler generated types providing basic metadata describing various program constructs. These metaobjects, together with some additions to the standard library can later be used to implement other third-party libraries providing both compile-time and run-time high-level reflection utilities.

The basic static metadata provided by compile-time reflection should be as complete as possible to be applicable in a wide range of scenarios and allow to implement custom higher-level static and dynamic reflection libraries and reflection-based utilities.

1.1. Differences between n3996 and n4111

This proposal (n4111) is a conceptual successor of n3996 [1], but it has been rewritten nearly from scratch (some parts of the text were reused). The main differences are listed here:

- n3996 included a brief introduction to reflection, a discussion of its usefulness and presented several use-cases.
- n3996 included multiple examples from the Mirror reflection utilities [2], for example the factory generator.
- In n3996, the metaobject concepts were described just conceptually and a possible renderings of the concepts into valid C++ were discussed. In n4111 the definition of the concepts is more detailed, specific and includes concrete definitions of what should be added to the standard and hints how certain concepts could be implemented.
- The definition of several concepts was changed. For example
  - the requirements for the MetaNamedScoped concept were moved to the MetaNamed concept,
  - the string which is the result of the base_name template class has been changed,
  - the definition of the named_typedef and the named_mem_var template classed is more detailed,
some of the metaobject attributes (rendered as template classes) were renamed.

- This proposal includes additions to the standard library.
- The appendices in n4111 include concrete examples of usage of the proposed metaobjects and the added template classes.

2. Metaobject concepts

We propose that the basic metadata describing a program written in C++ should be made available through a set of *anonymous* types and related functions and templates defined by the compiler. These types should describe various program constructs like, namespaces, types, typedefs, classes, their member variables (member data), member functions, inheritance, templates, template parameters, enumerated values, etc.

The compiler should generate metadata for the program constructs defined in the currently processed translation unit. Indexed sets (ranges) of metaobjects, like scope members, parameters of a function, etc. should be listed in the order of appearance in the processed source code.

Since we want the metadata to be available at compile-time, different base-level constructs should be reflected by "*statically*" different metaobjects and thus by different types. For example a metaobject reflecting the global scope namespace should be a different *type* than a metaobject reflecting the *std* namespace, a metaobject reflecting the *int* type should have a different type then a metaobject reflecting the *double* type, a metaobject reflecting *::foo(int)* function should have a different type than a metaobject reflecting *::foo(double)*, function, etc.

In a manner of speaking these special types (metaobjects) should become "instances" of the meta-level concepts (static interfaces which should not exist as concrete types, but rather only at the " specification-level" similar for example to the iterator concepts). This section describes a set of metaobject concepts, their interfaces, tag types for metaobject classification and functions (or operators) providing access to the metaobjects.

This section conceptually describes the requirements that various metaobjects need to satisfy in order to be considered models of the individual concepts.

Unless stated otherwise all additions proposed and described below should go into the *std* namespace. Alternatively, if any of the definitions proposed here clash with existing members (or new members proposed elsewhere) of the *std* namespace then they can be nested in a namespace like *std::meta* or *std::mirror*.

Also note, that in the sections below, the examples use names for concrete metaobjects, like *__meta_std_string*, etc.. This convention is *NOT* part of this proposal. The actual
naming of the metaobjects should be left to the compiler implementations and for all purposes from the users point of view the metaobjects should be anonymous types.

2.1. Categorization and Traits

In order to provide means for distinguishing between regular types and metaobjects generated by the compiler, the `is_metaobject` trait should be added to the `std` namespace (even if a nested namespace like `std::meta` is used for everything else) and should inherit from `true_type` for metaobjects (types generated by the compiler providing metadata) and from `false_type` for non-metaobjects (native or user defined types):

```cpp
template <typename T>
struct is_metaobject
 : false_type
 { };
```

2.1.1. Metaobject category tags

To distinguish between various metaobject kinds (satisfying different concepts as described below) a set of tag structs (indicating the kind of the metaobject) should be added:

```cpp
struct specifier_tag
{
    typedef specifier_tag type;
};

struct namespace_tag
{
    typedef namespace_tag type;
};

struct global_scope_tag
{
    typedef global_scope_tag type;
};

struct type_tag
{
    typedef type_tag type;
};

struct typedef_tag
{
```
typedef typedef_tag type;
};

struct class_tag
{
    typedef class_tag type;
};

struct function_tag
{
    typedef function_tag type;
};

struct constructor_tag
{
    typedef constructor_tag type;
};

struct operator_tag
{
    typedef operator_tag type;
};

struct overloaded_function_tag
{
    typedef overloaded_function_tag type;
};

struct enum_tag
{
    typedef enum_tag type;
};

struct enum_class_tag
{
    typedef enum_class_tag type;
};

struct inheritance_tag
{
    typedef inheritance_tag type;
};

struct constant_tag
N4111- Static reflection (rev. 2)

```cpp
{
    typedef constant_tag type;
};

struct variable_tag {
    typedef variable_tag type;
};

struct parameter_tag {
    typedef parameter_tag type;
};

These tags are referred-to as MetaobjectCategory below:

2.1.2. Specifier category tags

Similar to the metaobject tag types, a set of tag types for individual C++ specifier keywords should be defined:

```cpp
// indicates no specifier
struct none_tag {
    typedef none_tag type;
};

struct extern_tag {
    typedef extern_tag type;
};

struct static_tag {
    typedef static_tag type;
};

struct mutable_tag {
    typedef mutable_tag type;
};

struct register_tag {

typedef register_tag type;
};

struct thread_local_tag
{
    typedef thread_local_tag type;
};

struct const_tag
{
    typedef const_tag type;
};

struct virtual_tag
{
    typedef virtual_tag type;
};

struct private_tag
{
    typedef private_tag type;
};

struct protected_tag
{
    typedef protected_tag type;
};

struct public_tag
{
    typedef public_tag type;
};

struct class_tag
{
    typedef class_tag type;
};

struct struct_tag
{
    typedef struct_tag type;
};

struct union_tag
{    typedef union_tag type;
};

struct enum_tag
{
    typedef enum_tag type;
};

struct enum_class_tag
{
    typedef enum_class_tag type;
};

struct constexpr_tag
{
    typedef constexpr_tag type;
};

These tags are collectively referred-to as SpecifierCategory below.

2.2. StringConstant

A StringConstant is a class conforming to the following:

struct StringConstant
{
    typedef StringConstant type;

    // null terminated char array with size (string length+1)
    // known to sizeof at compile-time
    static constexpr const char value[Length+1] = {..., '\0'};

    // implicit compile-time conversion to null terminated
    // c-string
    constexpr operator const char* (void) const
    {
        return value;
    }
};

constexpr const char StringConstant::value[Length+1];
Concrete models of `StringConstant` are used to return compile-time string values. For example the `_str_void` type defined below, conforms to the `StringConstant` concept:

```cpp
template <char ... C>
struct string_constant
{
    typedef string_constant type;

    static constexpr const char value[sizeof...(C)+1] = {C..., '\0'};

    constexpr operator const char* (void) const
    {
        return value;
    }
};
```

```cpp
template <char ... C>
constexpr const char string_constant::value[sizeof...(C)+1];
```

//...

typedef string_constant<'v','o','i','d'> _str_void;

cout << _str_void::value << std::endl;
cout << _str_void() << std::endl;
static_assert(sizeof(_str_void::value) == 4+1, "");

The strings stored in the `value` array should be UTF-8 encoded.

Also note, that the `string_constant` class as defined above is just one of the possible implementations of `StringConstant`, we do not however imply that it must be implemented this way.

### 2.3. Metaobject Sequence

As the name implies, `MetaobjectSequence`s are used to store sequences or tuples of metaobjects. A model of `MetaobjectSequence` is a class conforming to the following:

It is a nullary metafunction returning itself:

```cpp
template <typename Metaobject>
struct MetaobjectSequence
{
    typedef MetaobjectSequence type;
};
```
Note: The definition above is only a pseudo-code and the template parameter Metaobject indicates here the minimal metaobject concept which all elements in the sequence must satisfy. For example a MetaobjectSequence<MetaConstructor> denotes a sequence of metaobjects that all satisfy the MetaConstructor concept, etc.

2.3.1. size

A template class size is defined as follows:

```cpp
template <typename T>
struct size;

template <>
struct size<MetaobjectSequence<Metaobject>>
  : integral_type<size_t, N>
{);
```

Where \( N \) is the number of elements in the sequence.

2.3.2. at

A template class at, providing random access to metaobjects in a sequence is defined (for values of \( I \) between 0 and \( N-1 \) where \( N \) is the number of elements returned by size) as follows:

```cpp
template <typename T, size_t I>
struct at;

template <size_t I>
struct at<MetaobjectSequence<Metaobject>, I>
  : Metaobject
{);
```

For example if \_\_meta_seq_ABC\_\_ is a metaobject sequence containing three metaobjects: \_\_meta_A\_\_, \_\_meta_B\_\_ and \_\_meta_C\_\_ (in that order), then:

```cpp
template <>
struct size<\_\_meta_seq_ABC>
  : integral_constant<size_t, 3>
{];
```

and

```cpp
template <>
struct at<\_\_meta_seq_ABC, 0>
  : \_\_meta_A
```
Note: The order of the metaobjects in a sequence is determined by the order of appearance in the processed translation unit.

2.4. Metaobject

A Metaobject is a stateless anonymous type generated by the compiler which provides metadata reflecting a specific program feature. Each metaobject should satisfy the following:

Every metaobject should be a nullary metafunction returning itself:

```cpp
struct Metaobject
{
  typedef Metaobject type;
};
```

One possible way how to achieve this is to define basic metaobjects as plain types (without any internal structure) and define a class template like:

```cpp
template <typename BasicMetaobject>
struct metaobject
{
  typedef metaobject type;
};
```

and then, implement the actual Metaobjects as instantiations of this template. For example if `__base_meta_int` is a basic metaobject reflecting the int type then the actual metaobject `__meta_int` conforming to this concept could be defined as:

```cpp
typedef metaobject<__base_meta_int> __meta_int;
```

Although, this is just a suggestion not a requirement of this proposal.
2.4.1. is_metaobject

The is_metaobject template should return true_type for all Metaobjects.

```cpp
template <>
struct is_metaobject<Metaobject>
: true_type
{};
```

2.4.2. metaobject_category

A template class metaobject_category should be defined in the std namespace (even if everything else is defined inside of a nested namespace like std::meta) and should inherit from one of the metaobject category tags, depending on the actual kind of the metaobject.

```cpp
template <typename T>
struct metaobject_category;
```

```cpp
template <>
struct metaobject_category<Metaobject>
: MetaobjectCategory
{}
```

For example if the __meta_std metaobject reflects the std namespace, then the specialization of metaobject_category should be:

```cpp
template <>
struct metaobject_category::__meta_std
: namespace_tag
{}
```

2.4.3. Traits

The following template classes indicating various properties of a Metaobject should be defined and should by default inherit from false_type unless stated otherwise below:

has_name – indicates that a Metaobject is a MetaNamed:

```cpp
template <typename T>
struct has_name
: false_type
{}
```

has_scope – indicates that a Metaobject is a MetaScoped:
template <typename T>
struct has_scope
    : false_type
{ };  
is_scope – indicates that a Metaobject is a MetaScope:

template <typename T>
struct is_scope
    : false_type
{ };  
is_class_member – indicates that a Metaobject is a MetaClassMember:

template <typename T>
struct is_class_member
    : false_type
{ };  
has_template – indicates that a Metaobject is a MetaInstantiation:

template <typename T>
struct has_template
    : false_type
{ };  
is_template – indicates that a Metaobject is a MetaTemplate or MetaTemplateParameter:

template <typename T>
struct is_template
    : false_type
{ };  

2.5. MetaSpecifier

MetaSpecifier is a Metaobject reflecting a C++ specifier. There also should be a ”none” MetaSpecifier reflecting a missing specifier. For example the const specifier on member functions can be either specified or not. In the latter case this ”none” MetaSpecifier should be ”returned”.

In addition to the requirements inherited from Metaobject, types conforming to this concept must satisfy the following:

The metaobject_category template should return specifier_tag for all MetaSpecifiers.
template <>
struct metaobject_category<MetaSpecifier>
  : specifier_tag
{ };

2.5.1. specifier_category

A template struct specifier_category should be defined and should inherit from one of the specifier category tags, depending on the actual reflected specifier.

template <typename T>
struct specifier_category;

template <>
struct specifier_category<MetaSpecifier>
  : SpecifierCategory
{ };  

For example if the __meta_static metaobject reflects the static C++ specifier, then the specialization of specifier_category should be:

template <>
struct specifier_category<__meta_static>
  : static_tag
{ };  

If __meta_nospec is the MetaSpecifier reflecting the ”none” (missing) specifier, then the specialization of specifier_category should be:

template <>
struct specifier_category<__meta_nospec>
  : none_tag
{ };  

2.5.2. keyword

A template struct keyword should be defined and should return the keyword matching the reflected specifier as a StringConstant.

template <typename T>
struct keyword;

template <>
struct keyword_category<MetaSpecifier>
For example if the `__meta_thread_local` metaobject reflects the `thread local` specifier, then the matching specialization of `keyword` could be following:

```cpp
template <>
struct keyword<__meta_thread_local>
: string_constant<'t','h','r','e','a','d',' ','l','o','c','a','l'>
{ }
```

If `__meta_nospec` is the `MetaSpecifier` reflecting the “none” (missing) specifier, then the specialization of `keyword` should return an empty string constant. For example:

```cpp
template <>
struct keyword<__meta_nospec>
: string_constant<>
{ }
```

The `string_constant<'t','h','r','e','a','d',' ','l','o','c','a','l'>` and the `string_constant<>` classes should be models of `StringConstant` as described above.

### 2.6. MetaNamed

`MetaNamed` is a `Metaobject` reflecting program constructs, which have a name (are identified by an identifier) like namespaces, types, functions, variables, parameters, etc.

In addition to the requirements inherited from `Metaobject`, the following requirements must be satisfied:

The `has_name` template class specialization for a `MetaNamed` should inherit from `true_type`:

```cpp
template <>
struct has_name<MetaNamed>
: true_type
{ }
```

#### 2.6.1. base_name

A template class `base_name` should be defined and should return the base name of the reflected construct, without the nested name specifier nor any qualifications or other decorations, as a `StringConstant`:

```cpp
template <typename T>
struct base_name;
```
template <>
struct base_name<MetaNamed>
  : StringConstant
{ }

For example, if __meta_std_size_t reflects the std::size_t type, then the matching specialization of base_name could be implemented in the following way:

template <>
struct base_name<__meta_std_size_t>
  : string_constant<'s','i','z','e','_','t'>
{ }

where the string_constant<'s','i','z','e','_','t'> class is a model of String-Constant as described above.

For namespace std the value should be "std", for namespace foo::bar::baz it should be "baz", for the global scope the value should be an empty string.

For unsigned long int * const * it should be "unsigned long int".

For std::vector<int>::iterator it should be "iterator". For derived, qualified types like volatile std::vector<const foo::bar::fubar*> * const * it should be "vector", etc.

### 2.6.2. full_name

A template class full_name should be defined and should return the fully qualified name of the reflected construct, including the nested name specifier and all qualifiers.

For namespace std the value should be "std", for namespace foo::bar::baz the value should be "foo::bar::baz", for the global scope the value should be an empty String-Constant. For std::vector<int>::iterator it should be "std::vector<int>::iterator".

For derived qualified types like volatile std::vector<const foo::bar::fubar*> * const * it should be defined as "volatile std::vector<const foo::bar::fubar*> * const *", etc.

```cpp
template <typename T>
struct full_name;
```

```cpp
template <>
struct full_name<MetaNamedScoped>
  : StringConstant
{ }
```
2.6.3. named typedef

A template class `named typedef` should be defined:

```cpp
template <typename X, typename T>
struct named typedef;

template <typename X>
struct named typedef<X, MetaNamedScoped>
{
    typedef X <NAME>;
};
```

The `<NAME>` expression above should be replaced in the actual specialization generated by the compiler by the name of the reflected named object. If the generated identifier would clash with a C++ reserved keyword, then a single trailing underscore should be appended to the identifier. If the generated identifier consists of multiple whitespace separated words then the whitespaces should be replaced by a single underscore.

For example if a type `__meta_std_thread` reflects the `std::thread` class, then the specialization of `named typedef` for this metaobject should be following:

```cpp
template <typename X>
struct named typedef<X, __meta_std_thread>
{
    typedef X thread;
};
```

if a type `__meta_std` reflects the `std` namespace, then the specialization of `named typedef` should be:

```cpp
template <typename X>
struct named typedef<X, __meta_std>
{
    typedef X std;
};
```

if a type `__meta_` reflects the global scope (or another anonymous base-level object), then the specialization of `named typedef` should be:

```cpp
template <typename X>
struct named typedef<X, __meta_>
{
    typedef X _;
};
```

If the types `__meta_int` and `__meta_unsigned_long_long_int` reflect the `int` and the `unsigned long long int` type respectively, then the matching instantiations of
named typedef should be:

```cpp
template <typename X>
struct named typedef<X, __meta_int>
{
    // note the trailing underscore
    typedef X int_;}
```

```cpp
template <typename X>
struct named typedef<X, __meta_long_long_unsigned_int>
{
    // note underscores replacing the spaces
    typedef X long_long_unsigned_int;
}
```

If the types __meta_char_const, __meta_long_const_ref, __meta_int_volatile_ptr and __meta_double_array_5 reflect char const, long const&, int volatile* and double[5] respectively, then the specializations of named typedef should be:

```cpp
template <typename X>
struct named typedef<X, __meta_char_const>
{
    typedef X char_;}
```

```cpp
template <typename X>
struct named typedef<X, __meta_long_const_ref>
{
    typedef X long_int;
}
```

```cpp
template <typename X>
struct named typedef<X, __meta_int_volatile_ptr>
{
    typedef X int_;}
```

```cpp
template <typename X>
struct named typedef<X, __meta_double_array_5>
{
    typedef X double_;}
```

2.6.4. named mem var

A template class named_mem_var should be defined as follows:
template <typename X, typename T>
struct named_mem_var;

template <typename X>
struct named_mem_var<X, MetaNamedScoped>
{
    X <NAME>;

template <typename ... P>
named_mem_var(P&& p)
    : <NAME>(std::forward<P>(p)...) ...
    {}
};

The <NAME> expression above should be replaced in the actual specialization generated by the compiler by the name of the reflected named object. If the generated identifier would clash with a C++ reserved keyword, then a single trailing underscore should be appended to the identifier. If the generated identifier consists of multiple whitespace separated words then the whitespaces should be replaced by a single underscore.

For example if a type __meta_std_string reflects the std::string typedef, then the specialization of named_mem_var for this metaobject should be following:

template <typename X>
struct named_mem_var<X, __meta_std_string>
{
    X string;

template <typename ... P>
named_mem_var(P&& ... p)
    : string(std::forward<P>(p)...) ...
    {}
};

If types __meta_void and __meta_long_double reflect the void and long double types respectively, then the matching instantiations of named_mem_var should be:

template <typename X>
struct named_mem_var<X, __meta_void>
{
    // note the trailing underscore
    X void_;

template <typename ... P>
named_mem_var(P&& ... p)
    : void_(std::forward<P>(p)...)
template <typename X>
struct named_mem_var<X, __meta_long_double>
{
    // note underscores replacing the spaces
    typedef X long_double;

template <typename ... P>
named_mem_var(P&& ... p)
    : long_double_(std::forward<P>(p)...) { }
};

For decorated and qualified types the same rules apply as for named_typedef. If __meta_std_string_const_ref reflects std::string const&, then:

```cpp
template <typename X>
struct named_typedef<X, __meta_std_string_const_ref>
{
    typedef X string;
};
```

### 2.7. MetaScoped

MetaScoped is a Metaobject reflecting program constructs defined inside of a named scope (like the global scope, a namespace, a class, etc.)

In addition to the requirements inherited from Metaobject, the following requirements must be satisfied:

The has_scope template class specialization for a MetaScoped should inherit from true_type:

```cpp
template <>
struct has_scope<MetaScoped>
    : true_type { }
```

#### 2.7.1. scope

A template class scope should be defined and should inherit from the MetaScope which reflects the parent scope of the program construct reflected by this MetaScoped.
template <typename T>
struct scope;

template <>
struct scope<MetaScoped>
 : MetaScope
 {}

2.8. MetaNamedScoped

Metaobject

MetaNamed

MetaNamedScoped

MetaScoped

Models of MetaNamedScoped combine the requirements of MetaNamed and MetaScoped.

2.9. MetaScope

Metaobject

MetaNamed

MetaNamedScoped

MetaScoped

MetaScope

MetaScoped is a MetaNamedScoped reflecting program constructs defined inside of a named scope (like the global scope, a namespace, a class, etc.)

In addition to the requirements inherited from MetaNamedScoped, the following is required:

The is_scope template class specialization for a MetaScope should inherit from true_type:

template <>
struct is_scope<MetaScope>
 : true_type
 {}

2.9.1. members

A template class members should be defined and should inherit from a MetaobjectSequence containing MetaNamedScoped metaobjects reflecting all members of the base-level
scope reflected by this `MetaScope`.

```cpp
template <typename T>
struct members;

template <>
struct members<MetaScope>
  : MetaobjectSequence<MetaNamedScoped>
{ }
```

### 2.10. MetaClassMember

`MetaClass` is a `MetaType` and a `MetaScope` if reflecting a regular class or possibly also a `MetaTemplate` if it reflects a class template.

In addition to the requirements inherited from `MetaNamedScoped`, the following is required for `MetaClassMembers`:

The `is_class_member` template class specialization for a `MetaClassMember` should inherit from `true_type`:

```cpp
template <>
struct is_class_member<MetaClassMember>
  : true_type
{ }
```

### 2.10.1. accessSpecifier

A template class called `accessSpecifier` should be defined and should inherit from a `MetaSpecifier` reflecting the `private`, `protected` or `public` access specifier:

```cpp
template <typename T>
struct accessSpecifier;

template <>
struct accessSpecifier<MetaClassMember>
  : MetaSpecifier
{ }
```
2.11. MetaGlobalScope

MetaGlobalScope is a MetaScope reflecting the global scope.

In addition to the requirements inherited from MetaScope, the following must be satisfied:

The metaobject_category template class specialization for a MetaGlobalScope should inherit from global_scope_tag:

```cpp
template <>
struct metaobject_category<MetaNamespace>
 : global_scope_tag
{ }
```

The scope template class specialization (required by MetaScoped) for MetaGlobalScope should inherit from the MetaGlobalScope itself:

```cpp
template <>
struct scope<MetaGlobalScope>
 : MetaGlobalScope
{ }
```

2.12. MetaNamespace

MetaNamespace is a MetaScope reflecting a namespace.

In addition to the requirements inherited from MetaScope, the following must be satisfied:

The metaobject_category template class specialization for a MetaNamespace should inherit from namespace_tag:

```cpp
template <>
struct metaobject_category<MetaNamespace>
 : namespace_tag
{ }
```
2.13. MetaType

MetaType is a MetaNamedScoped reflecting types.

In addition to the requirements inherited from MetaNamedScoped, the following is required:

The metaobject_category template class specialization for a MetaType should inherit from type_tag:

```cpp
template <>
struct metaobject_category<MetaType>
: type_tag
{
};
```

2.13.1. original_type

A template class original_type should be defined and should "return" the original type reflected by this MetaType:

```cpp
template <typename T>
struct original_type;
```

```cpp
template <>
struct original_type<MetaType>
{
    static_assert(not(is_template<MetaType>::value), "");
    typedef original_type type;
};
```

For example, if __meta_int is a metaobject reflecting the int type, then the specialization of original_type should be following:

```cpp
template <>
struct original_type<__meta_int>
{
    typedef int type;
};
```
Note: If a concept derived from MetaType, for example a MetaClass, is also a MetaTemplate (i.e. is reflecting a template not a concrete type), then the original_type template should be left undefined.

2.14. MetaTypedef

MetaTypedef is a MetaType reflecting typedefs.
In addition to the requirements inherited from MetaType, the following is required:
The metaobject_category template class specialization for a MetaTypedef should inherit from typedef_tag:

```cpp
template <>
struct metaobject_category<MetaTypedef>
  : typedef_tag
{ };  
```

2.14.1. type

A template class called type should be defined and should inherit from the MetaType reflecting the "source" type of the typedef:

```cpp
template <typename T>
struct type;

template <>
struct type<MetaTypedef>
  : MetaType
{ };  
```
For example if __meta_std_string is a MetaTypedef reflecting the std::string typedef and __meta_std_basic_string_char is the MetaType that reflects the std::basic_string<char> type, and std::string is defined as:

```cpp
namespace std {
 typedef basic_string<char> string;
}
```
then the specialization of type for __meta_std_string should be following:
template <>
struct type<__meta_std_string>
 : __meta_std_basic_string_char
 { };

2.15. MetaClass

MetaClass is a MetaType and a MetaScope if reflecting a regular class or possibly also a MetaTemplate if it reflects a class template.

In addition to the requirements inherited from MetaType, MetaScope and optionally from MetaTemplate, models of MetaClass are subject to the following:

The metaobject_category template class specialization for a MetaClass should inherit from class_tag:

```cpp
template <>
struct metaobject_category<MetaClass>
 : class_tag
 { };
```

If a MetaClass reflects a class template, then the is_template trait should inherit from true_type

2.15.1. elaborated_type_specifier

A template class called elaborated_type_specifier should be defined and should inherit from a MetaSpecifier reflecting the class, struct or union specifiers:

```cpp
template <typename T>
struct elaborated_type_specifier;
```

```cpp
template <>
struct elaborated_type_specifier<MetaClass>
 : MetaSpecifier
 { };```
2.15.2. base_classes

A template class `base_classes` should be defined and should inherit from a `Metaobject-Sequence` of `MetaInheritances`, each one of which reflects the inheritance of a single base class of the class reflected by the `MetaClass`:

```cpp
template <typename T>
struct base_classes;

template <>
struct base_classes<MetaClass>
: MetaobjectSequence<MetaInheritance>
{
};
```

2.16. MetaEnum

`MetaEnum` is a `MetaType` reflecting an enumeration type.

In addition to the requirements inherited from `MetaType`, `MetaEnum` requires also the following:

The `metaobject_category` template class specialization for a `MetaEnum` should inherit from `enum_tag`:

```cpp
template <>
struct metaobject_category<MetaEnum>
: enum_tag
{ }
```

The members of `MetaEnumClass` are only `MetaNamed MetaConstants`.

2.16.1. base_type

A template class `base_type` should be defined and should inherit from a `MetaType` reflecting the underlying type of the enumeration:

```cpp
template <typename T>
struct base_type;
```
2.17. MetaEnumClass

*MetaEnumClass* is a *MetaType* and a *MetaScope* reflecting a strongly type enumeration. In addition to the requirements inherited from *MetaType* and *MetaScope*, the following must be satisfied:

The `metaobject_category` template class specialization for a *MetaEnumClass* should inherit from `enum_class_tag`:

```cpp
template <>
struct metaobject_category<MetaEnumClass>
  : enum_class_tag
{ };
```

The members of *MetaEnumClass* are only *MetaNamedScoped* *MetaConstants*.

2.17.1. base_type

A template class `base_type` should be defined and should inherit from a *MetaType* reflecting the base type of the enumeration:

```cpp
template <typename T>
struct base_type;
```

```cpp
template <>
struct base_type<MetaEnumClass>
  : MetaType
{ };
```
2.18. MetaOverloadedFunction

Models of `MetaOverloadedFunction` reflect overloaded functions. `MetaFunctions` (and `MetaOperators`, `MetaInitializers`, `MetaConstructors`, etc.) are not direct members of scopes (they are not listed in the `MetaobjectSequence` "returned" by the `members<MetaScope>` template class). Instead, all functions, operators and constructors with the same name, (and even those that are not overloaded in a specific scope) are grouped into a `MetaOverloadedFunction`. Individual overloaded `MetaFunctions` in the group can be obtained through the interface of `MetaOverloadedFunction` (specifically through the `overloads` template described below). The same also applies to `MetaConstructors` and `MetaOperators`.

The rationale for this is that direct scope members, i.e. metaobjects accessible through the `MetaScope`'s `members` template class should have unique names, which would not be the case if `MetaFunctions` were direct scope members.

The `scope` of an `MetaOverloadedFunction` is the same as the `scope` of all `MetaFunctions` grouped by that `MetaOverloadedFunction`.

In addition to the requirements inherited from `MetaNamedScoped`, models of `MetaOverloadedFunction` are subject to the following:

The `metaobject_category` template class specialization for a `MetaOverloadedFunction` should inherit from `overloaded_function_tag`:

```cpp
template <>
struct metaobject_category<MetaOverloadedFunction>
  : overloaded_function_tag
{};
```

2.18.1. overloads

A template class called `overloads` should be defined and should return a `MetaobjectSequence` of `MetaFunctions`, reflecting the individual overloads:

```cpp
template <>
struct overloads<MetaOverloadedFunction>
  : MetaobjectSequence<MetaFunction>
{};
```
2.19. **MetaFunction**

*MetaFunction* is a *MetaNamedScoped* which reflects a function or a function template. In addition to the requirements inherited from *MetaNamedScoped*, models of *MetaFunction* are subject to the following:

The `metaobject_category` template class specialization for a *MetaFunction* should inherit from `function_tag`:

```cpp
template <>
struct metaobject_category<MetaFunction>
  : function_tag
{);
```

If a *MetaFunction* reflects a function template, then the `is_template` trait should inherit from `true_type`

### 2.19.1. linkageSpecifier

A template class `linkageSpecifier` should be defined and should inherit from a `MetaSpecifier` reflecting the linkage specifier of the function reflected by the *MetaFunction*:

```cpp
template <typename T>
struct linkageSpecifier;
```

```cpp
template <>
struct linkageSpecifier<MetaFunction>
  : MetaSpecifier
{);
```

### 2.19.2. constexprSpecifier

A template class `constexprSpecifier` should be defined and should inherit from a `MetaSpecifier` reflecting the constexpr specifier of the reflected function:

```cpp
template <typename T>
struct constexprSpecifier;
```
 template <>
 struct constexpr_specifier<MetaFunction>
  : MetaSpecifier
 { };  

In case the reflected function does not have the constexpr specifier, then the result should be a MetaSpecifier reflecting the "none" specifier.

2.19.3. result_type

A template class result_type should be defined and should inherit from a MetaType reflecting the return value type of the reflected function:

 template <typename T>
 struct result_type;

 template <>
 struct result_type<MetaFunction>
  : MetaType
 { };  

2.19.4. parameters

A template class parameters should be defined and should inherit from a Metaobject-Sequence or MetaParameters reflecting the individual parameters of the function:

 template <typename T>
 struct parameters;

 template <>
 struct parameters<MetaFunction>
  : MetaobjectSequence<MetaParameter>
 { };  

2.19.5. noexceptSpecifier

A template class noexceptSpecifier should be defined and should inherit from a MetaSpecifier reflecting the noexcept specifier of the reflected function:

 template <typename T>
 struct noexceptSpecifier;

 template <>
struct noexcept_specifier<MetaFunction>
  : MetaSpecifier
{ }

In case the reflected function does not have the noexcept specifier, then the result should be a MetaSpecifier reflecting the "none" specifier.

2.19.6. exceptions

A template class exceptions should be defined and should inherit from a MetaobjectSequence of MetaTypes reflecting the individual exception types that the reflected function is allowed to throw:

```cpp
template <typename T>
struct exceptions;
```

```cpp
template <>
struct exceptions<MetaFunction>
  : MetaobjectSequence<MetaType>
{ }
```

2.19.7. const_specifier

In case a MetaFunction is also a MetaClassMember, a template class const_specifier should be defined and should inherit from a MetaSpecifier reflecting the const specifier of the reflected member function:

```cpp
template <typename T>
struct const_specifier;
```

```cpp
template <>
struct const_specifier<MetaFunction>
  : MetaSpecifier
{ 
  static_assert(is_class_member<MetaFunction>::value, "");
};
```

In case the reflected member function does not have the const specifier, then the result should be a MetaSpecifier reflecting the "none" specifier.

2.19.8. is_pure

In case a MetaFunction is also a MetaClassMember, a template class is_pure should be defined and should inherit from true_type if the reflected function is pure virtual or
from `false_type` otherwise:

```cpp
template <typename T>
struct is_pure;

struct is_pure<MetaFunction>
  : std::integral_constant<bool, B>
{
    static_assert(is_class_member<MetaFunction>::value, "");
};
```

### 2.20. MetaInitializer

`MetaInitializer` reflects an initializer (constructor) of a native type.

In addition to the requirements inherited from `MetaFunction`, models of `MetaInitializer` must conform to the following:

The `metaobject_category` template class specialization for a `MetaInitializer` should inherit from `constructor_tag`:

```cpp
template <>
struct metaobject_category<MetaInitializer>
  : constructor_tag
{
};
```

The specialization of the `result_type` template class for a `MetaInitializer` should inherit from a `MetaType` reflecting the initialized type:

```cpp
template <>
struct result_type<MetaInitializer>
  : MetaType
{ };```
2.21. MetaConstructor

MetaConstructor reflects a constructor of an elaborated type.

In addition to the requirements inherited from MetaFunction and MetaClassMember, the following is required for MetaConstructors:

The metaobject_category template class specialization for a MetaConstructor should inherit from constructor_tag:

```cpp
template <>
struct metaobject_category<MetaConstructor>
  : constructor_tag
{ }
```

The specialization of the result_type template class for a MetaConstructor should inherit from a MetaClass reflecting the constructed type.

```cpp
template <>
struct result_type<MetaConstructor>
  : MetaClass
{ }
```

The specialization of the scope template class for a MetaConstructor should inherit from a MetaClass reflecting the constructed type.

```cpp
template <>
struct scope<MetaConstructor>
  : MetaClass
{ }
```

2.22. MetaOperator
MetaOperator is a MetaFunction which reflects an operator.

In addition to the requirements inherited from MetaFunction, models of MetaOperator must conform to the following:

The metaobject_category template class specialization for a MetaOperator should inherit from operator_tag:

```cpp
template <>
struct metaobject_category<MetaOperator>
  : operator_tag
{ }
;
```

2.23. MetaTemplate

MetaTemplate is a MetaNamedScoped and either a MetaClass or a MetaFunction.

Note: The MetaTemplate concept slightly modifies the requirements of the MetaClass and MetaFunction concepts.

In addition to the requirements inherited from MetaNamedScoped, models of MetaTemplate must conform to the following:

The is_template template class specialization for a MetaTemplate should inherit from true_type:

```cpp
template <>
struct is_template<MetaTemplate>
  : true_type
{ }
;
```

2.23.1. template_parameters

A template class called template_parameters should be defined and should inherit from a MetaobjectSequence of MetaTemplateParameters, reflecting the individual type or constant template parameters:

```cpp
template <typename T>
struct template_parameters;
```
template <>
struct template_parameters<MetaTemplate>
  : MetaobjectSequence<MetaTemplateParameter>
{ };  

2.23.2. instantiation

A template class instantiation should be defined and should inherit from a MetaInstantiation reflecting a concrete instantiation of the template reflected by this MetaTemplate:

template <typename T, typename ... P>
struct instantiation;

template <typename ... P>
struct instantiation<MetaTemplate, P...>
  : MetaInstantiation
{ };  

For example if __meta_std_pair is a MetaTemplate and a MetaClass reflecting the std::pair template and __meta_std_pair_int_double is a MetaInstantiation and a MetaClass reflecting the std::pair<int, double> class then:

static_assert(
  is_base_of<
    __meta_std_pair_int_double,
    instantiation<__meta_std_pair, int, double>
  >(), ""
);  

2.24. MetaTemplateParameter

MetaTemplateParameter is a MetaNamedScoped and either a MetaTypedef or a MetaConstant.

In addition to the requirements inherited from MetaNamedScoped, models of MetaTemplateParameter must conform to the following:
The `is_template` template class specialization for a `MetaTemplateParameter` should inherit from `true_type`:

```cpp
template <>
struct is_template<MetaTemplateParameter>
  : true_type
{ };  
```

The `full_name` inherited from `MetaNamed` should return the same `StringConstant` as `base_name` for models of `MetaTemplateParameter`, i.e. the plain template parameter name without any qualifications.

### 2.24.1. position

A template class `position` should be defined and should inherit from `integral_constant<size_t, I>` where `I` is a zero-based position (index) of the parameter.

```cpp
template <typename T>
struct position;
```

```cpp
template <>
struct position<MetaTemplateParameter>
  : integral_constant<size_t, I>
{ };  
```

### 2.24.2. is_pack

A template class called `is_pack` should be defined and should inherit from `true_type` if the template parameter is a pack parameter or from `false_type` otherwise.

```cpp
template <typename T>
struct is_pack;
```

```cpp
template <>
struct is_pack<MetaTemplateParameter>
  : integral_constant<bool, B>
{ };  
```
2.25. MetaInstantiation

MetaInstantiation is a MetaNamedScoped and either a MetaClass or a MetaFunction.

In addition to the requirements inherited from MetaNamedScoped, models of MetaInstantiation must conform to the following:

The has_template template class specialization for a MetaInstantiation should inherit from true_type:

```cpp
template <>
struct has_template<MetaInstantiation>
 : true_type
{ };```

2.25.1. template_arguments

A template class called template_arguments should be defined and should inherit from a MetaobjectSequence of MetaNamedScoped metaobjects each of which is either a MetaType or a MetaConstant and reflects the i-th template argument:

```cpp
template <typename T>
struct template_arguments;

template <>
struct template_arguments<MetaInstantiation>
 : MetaobjectSequence<MetaNamed Scoped>
{ };```

2.25.2. template_

A template class called template_ should be defined and should inherit from a MetaTemplate reflecting the instantiation’s template:

```cpp
template <typename T>
struct template_;

template <>
struct template_<MetaInstantiation>
For example if \texttt{\_\_meta\_std\_pair} is a \textit{MetaTemplate} and a \textit{MetaClass} reflecting the \texttt{std::pair} template and \texttt{\_\_meta\_std\_pair\_int\_double} is a \textit{MetaInstantiation} and a \textit{MetaClass} reflecting the \texttt{std::pair<int, double>} class then:

\begin{verbatim}
static_assert(
    is_base_of<
        \_\_meta\_std\_pair,
        template_<\_\_meta\_std\_pair\_int\_double>
    >(), ""
);
\end{verbatim}

\section*{2.26. \textit{MetaInheritance}}

\begin{tikzpicture}
    \node[draw] (M) {\textit{MetaInheritance}};
    \node[draw, left of=M, xshift=-2cm] (MS) {\textit{MetaScoped}};
    \node[draw, left of=MS, xshift=-2cm] (MO) {\textit{MetaObject}};
    \node[draw, right of=M, xshift=2cm] (MI) {\textit{MetaInheritance}};
    \draw[->] (MO) -- (MS); \draw[->] (MS) -- (M); \draw[->] (M) -- (MI);
\end{tikzpicture}

\textit{MetaInheritance} is a \textit{MetaScoped} reflecting class inheritance.

In addition to the requirements inherited from \textit{MetaScoped}, types conforming to this concept must satisfy the following:

The \texttt{metaobject\_category} template should return \texttt{inheritance\_tag} for models of \textit{MetaInheritance}:

\begin{verbatim}
template <>
struct metaobject\_category<MetaInheritance>
  : inheritance\_tag
{ }
\end{verbatim}

The \texttt{scope} member function should inherit from a \textit{MetaClass} reflecting the derived class in the inheritance:

\begin{verbatim}
template <>
struct scope<MetaInheritance>
  : MetaClass
{ }
\end{verbatim}

\subsection*{2.26.1. access\_specifier}

A template struct \texttt{access\_specifier} should be defined and should inherit from a \textit{MetaSpecifier} reflecting one of the \texttt{private}, \texttt{protected} and \texttt{public} access specifiers.

\begin{verbatim}
template <typename T>
struct access\_specifier;
\end{verbatim}
template <>
struct access_specifier<MetaInheritance>
    : MetaSpecifier
{ }

2.26.2. inheritanceSpecifier

A template struct inheritanceSpecifier should be defined and should inherit from a MetaSpecifier reflecting one of the virtual and "none" access specifiers.

template <typename T>
struct inheritanceSpecifier;

template <>
struct inheritanceSpecifier<MetaInheritance>
    : MetaSpecifier
{ }

2.26.3. baseClass

A template struct baseClass should be defined and should inherit from a MetaClass reflecting the base class in the inheritance:

template <typename T>
struct baseClass;

template <>
struct baseClass<MetaInheritance>
    : MetaClass
{ }

2.27. MetaVariable

MetaVariable is a MetaNamedScoped reflecting a variable.
In addition to the requirements inherited from `MetaNamedScoped`, the following must be satisfied:

The `metaobject_category` template class specialization for a `MetaVariable` should inherit from `variable_tag`:

```cpp
template <>
struct metaobject_category<MetaVariable>
  : variable_tag
{ };
```

### 2.27.1. storage_specifier

A template class `storage_specifier` should be added and should inherit from a `MetaSpecifier` reflecting a storage class specifier:

```cpp
template <typename T>
struct storage_specifier;

template <>
struct storage_specifier<MetaVariable>
  : MetaSpecifier
{ };
```

### 2.27.2. type

A template class `type` should be added and should inherit from a `MetaType` reflecting the type of the variable:

```cpp
template <typename T>
struct type;

template <>
struct type<MetaVariable>
  : MetaType
{ };
```

### 2.28. MetaParameter

[Diagram of MetaObject, MetaNamed, MetaScoped, MetaParameter relationships]


**MetaParameter** is a **MetaNamed** and **MetaScoped**, reflecting a function parameter or a parameter pack.

In addition to the requirements inherited from **MetaNamed** and **MetaScoped**, the following must be satisfied:

The `metaobject_category` template class specialization for a **MetaParameter** should inherit from `parameter_tag`:

```cpp
template <>
struct metaobject_category<MetaParameter>
    : parameter_tag
{ }
```

The `scope` of a **MetaParameter** should be a **MetaFunction** reflecting the function to which the parameter belongs:

```cpp
template <>
struct scope<MetaParameter>
    : MetaFunction
{ }
```

The `full_name` inherited from **MetaNamed** should return the same **StringConstant** as `base_name` for models of **MetaParameter**, i.e. the plain parameter name without any qualifications.

### 2.28.1. type

A template class `type` should be added and should inherit from a **MetaType** reflecting the type of the parameter:

```cpp
template <typename T>
struct type;
```

```cpp
template <>
struct type<MetaParameter>
    : MetaType
{ }
```

### 2.28.2. position

A template class `position` should be defined and should inherit from `integral_constant<size_t, I>` type where `I` is a zero-based position (index) of the parameter:

```cpp
template <typename T>
struct position;
```
```cpp
template <>
struct position<MetaParameter>
  : integral_constant<size_t, I>
{ };
```

2.28.3. is_pack

A template class called is_pack should be defined and should inherit from true_type if the parameter is a pack parameter or from false_type otherwise.

```cpp
template <typename T>
struct is_pack;
```

```cpp
template <>
struct is_pack<MetaParameter>
  : integral_constant<bool, B>
{ };
```

2.29. MetaConstant

MetaConstant is a Metaobject reflecting a compile-time constant value.

In addition to the requirements inherited from Metaobject, the following must be satisfied:

The metaobject_category template class specialization for a MetaConstant should inherit from constant_tag:

```cpp
template <>
struct metaobject_category<MetaConstant>
  : constant_tag
{ };
```

2.29.1. type

A template class type should be added and should inherit from a MetaType reflecting the type of the constant value.

```cpp
template <typename T>
struct type;
```
template <>
struct type<MetaVariable>
 : MetaType
{ };

2.29.2. value

A template class value should be added and should inherit from an integral_constant<T, V>:

```cpp
template <typename T>
struct value;
```

```cpp
template <>
struct value<MetaConstant>
 : integral_constant<T, V>
{ };
```

3. Reflection operator

The metaobjects reflecting some program feature \( X \) as described above should be made available to the user by the means of a new operator or expression. More precisely, the reflection operator returns a type conforming to a particular metaobject concept, depending on the reflected expression.

Since adding a new keyword has the potential to break existing code, we do not insist on any particular expression, here follows a list of suggestions in order of preference (from the most to the least preferrable):

- \( \text{mirrored}(X) \)
- \( \text{reflected}(X) \)
- \( \text{reflexpr}(X) \)
- \( |X| \)
- \( [[X]] \)
- \( <<X>> \)

The reflected expression \( X \) in the items listed above can be any of the following:

- \( :: \): – The global scope, the returned metaobject is a \( \text{MetaGlobalScope} \).
- \( Namespace \ text{name} \): (std) the returned metaobject is a \( \text{MetaNamespace} \).
- \( Type \ text{name} \): (long double) the returned metaobject is a \( \text{MetaType} \).
• typedef name – (std::size_t or std::string) the returned metaobject is a MetaTypedef.

• Template name – (std::tuple or std::map) the returned metaobject is a MetaTemplate.

• Class name – (std::thread or std::map<int, double>) the returned metaobject is a MetaClass.

• Function name – (std::sin or std::string::size) the returned metaobject is a MetaOverloadedFunction.

• Function signature – (std::sin(double) or std::string::front(void) const) the returned metaobject is a MetaFunction. The signature may be specified without the return value type.

• Constructor signature – (std::pair<char, double>::pair(char, double) or std::string::string(void)) the returned metaobject is a MetaConstructor.

• Variable name – (std::errno) the returned metaobject is a MetaVariable.

The reflection operator or expression should have access to private and protected members of classes. The following should be valid:

```cpp
struct A
{
    int a;
};

class B
{
  protected:
    int b;
};

class C
  : protected A
  , public B
{
  private:
    int c;
};
typedef mirrored(A::a) meta_A_a;
typedef mirrored(B::b) meta_B_b;
typedef mirrored(C::a) meta_C_a;
typedef mirrored(C::b) meta_C_b;
typedef mirrored(C::c) meta_C_c;
```
4. Additions to the library

In order to simplify composition of the metaobjects and metafunctions defined above, several further additions to the standard library should be made.

4.1. Metaobject expressions

A *metaobject expression* is a class which can be *evaluated* into a *Metaobject*. By default any class, that has a member typedef which is a model of *Metaobject* is a metaobject expression.

```cpp
typedef Metaobject type;
```

And thus, any *Metaobject* is also a *metaobject expression*.

Generally, however, any type for which the *evaluate* metafunction (described below), "returns" a *Metaobject* is a *metaobject expression*.

4.1.1. evaluate

A class template called *evaluate* should be defined and should "return" a *Metaobject* resulting from a *metaobject expression*:

```cpp
template <class MetaobjectExpression>
struct evaluate
    : Metaobject
{ }
```

that could be implemented for example as follows:

```cpp
template <class X, bool IsMetaobject>
struct do_evaluate;

template <class X>
struct do_evaluate<X, true>
    : X
{ }

template <class X>
struct do_evaluate<X, false>
    : do_evaluate<
        typename X::type,
```
```cpp
is_metaobject<typename X::type>::value
> { };

template <class X>
struct evaluate
 : do_evaluate<X, is_metaobject<X>::value>
{ }

The users should be allowed to add specializations of evaluate for other types if necessary.

4.2. Default implementation of metafunctions

The default implementation of the metafunction template classes defined above, should follow this pattern:

```cpp
template <typename T>
struct Template;

template <typename T>
struct Template
 : Template<typename evaluate<T>::type>
{ }
```

Where Template is each of the following:

- `metaobject_category`
- `specifier_category`
- `keyword`
- `base_name`
- `full_name`
- `named_typedef`
- `named_mem_var`
- `scope`
- `members`
- `overloads`
- `type`
- `base_classes`
- `base_class`
- `base_type`
- `result_type`
- `parameters`
- `template_parameters`
- `template_arguments`
- `template_`
- `exceptions`
- `instantiation`
- `position`
- `value`
- `elaborated_typeSpecifier`
- `accessSpecifier`
- `constexprSpecifier`
- `noexceptSpecifier`
- `constSpecifier`
- `inheritanceSpecifier`
- `linkageSpecifier`
- `storageSpecifier`
- `is_pure`
- `is_pack`
- `original_type`

For example:
template <typename T>
struct metaobject_category
  : metaobject_category<typename evaluate<T>::type>
{ };  

This will allow to compose metaobject expressions into algorithms. For example:

// print the number of members of the scope where mycls is defined
cout << size<members<scope<mirrored(mycls)>>>() << endl;

// print the name of the first base class of mycls
cout <<
    base_name<base_class<at<base_classes<mirrored(mycls)>, 0>>>()  
    << endl;

// print the access specifier keyword of the second base of mycls
cout <<
    keyword<access_specifier<at<base_classes<mirrored(mycls)>, 1>>>()  
    << endl;

// print the fully qualified name of the scope of
// the source type of the third member of mycls
cout <<
    full_name<scope<type<at<members<mirrored(mycls)>, 2>>>>()  
    << endl;

5. Rationale

This section explains some of the design decisions behind this proposal and answers several frequently asked questions.

5.1. Why metaobjects, why not reflect directly?

Q: Why should we define a set of metaobject concepts, let the compiler generate models of these concepts and use those to obtain the metadata? Why not just extend the existing type traits?

A: The most important reason is the completeness and the scope of reflection. Type traits (as they are defined now) work just with types. A reflection facility should however provide much more metadata. It should be able to reflect namespaces, functions, constructors, inheritance, variables, etc.

For example:
pair<long, string> my_var;

// OK, we can print the name of the type of a variable:
cout << type_name<decltype(my_var)>() << endl;
// But we really, really want to print the name of the variable
// (without the use of the preprocessor)
cout << type_name<my_var>() << endl; // Error
// similar with namespaces:
cout << type_name<std::chrono>() << endl; // Error
// etc.

Doing reflection with type traits limits the scope, because of the rules defining what can
be a template parameter. This rules could be updated to allow for example an expression
representing a particular class constructor to be passed as a template argument. Also
currently there is no expression for specifying (not invoking) a constructor or a particular
function overload, so additional rules would have to be added.

This would (in our opinion) be a much more drastic change to the standard, than
the adoption of this proposal. If expressions denoting a namespace or a particular
constructor or a function overload were added just for the purpose of reflecting them
(with the mirrored keyword), then all the changes could be localized in the reflection
subsystem and remain invalid in the core language:

mirrored(std::current_thread); // OK - MetaNamespace
std::current_thread; // error - not a primary expression

mirrored(std::sin); // OK - MetaOverloadedFunction
std::sin; // error - cannot resolve overload

mirrored(std::sin(double)); // OK - MetaFunction
std::sin(double); // error - invalid expression
// etc.

Second reason is access to private and protected members. There are many use-cases
where access to non-public class members through reflection is desired. If reflection was
done through type traits directly on the class members, it would be either impossible
to reflect non-public members or the access rules would have to be changed to somehow
allow access in reflection expressions:

class C
{
private:
    typedef int T;
public:
};
assert(some_trait<C::T>::value); //OK, we are reflecting so we have access
but not outside:

C::T x = 0; // Error, C::T is private

With the reflection operator like mirrored(X), the access rules would have to be updated
only to allow the reflection operator to have access to everything. At the first glance, the
following two expressions;

some_trait<C::T>::value

and

mirrored(C::T)

look similar and so the changes to the access rules could seem similar too, but that is
not the case. The (single) mirrored operator would have special status, on the other
hand type traits are regular templates (with some magic inside) and all (several dozens of
them) would need to be distinguished from all the other templates in the std namespace,
which should not have private access.

Having said that, we do not object to extending the type traits where it does make sense.

One other reason for having a new reflection operator is, that there already is an ex-
isting (very limited) reflection operator, namely typeid which “returns” a compiler-
generated ”metaobject” – std::type_info. We are aware that there are differences
between typeid and mirrored, but the basic idea is similar.

5.2. Why are the metaobjects anonymous?

Q: Why should the metaobjects be anonymous types as opposed to types with well defined
and standardized names or concrete template classes, (possibly with some special kind of
parameter accepting different arguments than types and constants)?

A: We wanted to avoid defining a specific naming convention, because it would be
difficult to do so and very probably not user friendly (see C++ name mangling). There
already is a precedent for anonymous types – for example C++ lambdas.

Another option would be to define a concrete set of template classes like:

```cpp
namespace std {

template <typename T>
class meta_type /* Model of MetaType */ {
};
}
```
which could work with types, classes, etc., but would not work with namespaces, constructors, etc. (see also the Q/A above):

```cpp
namespace std {

    template <something X> //<- Problem
class meta_constructor /* Model of MetaConstructor */
    { };

    template <something X> //<- Problem
class meta_namespace /* Model of MetaNamespace */
    { };
}

typedef std::meta_namespace<std> meta_std; //<- Problem
```

Instead of this, the metaobjects are anonymous and their (internal) identification is left to the compiler. From the users POV, the metaobject can be distinguished by the means of the metaobject traits and tags as described above.

### 5.3. Why this rendering of metaobjects?

**Q:** Why were the metaobject concepts from n3996 transformed into a set of types with external template classes operating on them?

**A:** N3996 defined a set of abstract metaobject concepts including their instances, traits, attributes and functions. Then two possible concrete transformations into valid C++ were shown in the appendices.

1. Structures where the attributes were constexpr static member variables and the functions were constexpr static member functions.
2. Types where the traits, attributes and functions are implemented as specializations of class templates defined on namespace-level.

In this proposal the second option was picked for several reasons:

- It is closer to standard type traits and to popular metaprogramming libraries like *Boost.MPL*, etc.
- It should require less resources and time to compile, since the implementation of various metafuncions (class templates) like `base_name`, `scope` or `position` and especially those returning *MetaobjectSequences*, like `members`, `overloads`, `base_classes` etc. for concrete metaobjects can be instantiated individually and need to be instantiated only if they are actually used in the code. It would probably be harder to do so if they were implemented as (non-template) class members.
6. Known issues

- Some better name for the template metafunction template class.

- *Something similar to source contexts from D3972*: Should *Metaobject* contain information about the source file, line and possibly line column or function name where the base-level construct reflected by the *Metaobject* was defined?

- *Normalization of names returned by MetaNamed base_name() and MetaNamedScoped full_name()*: The strings returned by the *base_name* and *full_name* metafunctions should be implementation-independent and the same on every platform/compiler.

- *The reflection of C++11/14 features not covered by this proposal.*

- *Explicit specification of what should be reflected.* It might be useful to have the ability to explicitly specify either what to reflect or what to hide from reflection. For example the ”whitelisting” (explicitly specifying of what should be reflected) of namespace or class members could simplify reflective meta-algorithms so that they would not have to implement complicated filters when traversing scope members, to hide implementation details and to improve compilation times. It is important that this functionality is decoupled from the scope member declarations, since it would allow applications to cherry-pick what should be reflected even in third-party libraries.

7. Acknowledgements

Thanks to Fabio Fracassi for presenting the n3996 proposal at the Rapperswil meeting.

8 References


A. Examples

This section contains multiple examples of usage of the additions proposed above. The examples assume that the *mirrored* operator (described above) is used to obtain the metaobjects and the types, templates, etc. are defined in the *std::meta* namespace.
For the sake of brevity

```cpp
using namespace std;
```
is assumed.

### A.1. Basic traits

Usage of the `is_metaobject` trait on non-metaobjects:

```cpp
static_assert(not(is_metaobject<int>()), "");
static_assert(not(is_metaobject<std::string>()), "");
static_assert(not(is_metaobject<my_class>()), "");
static_assert(not(meta::is_class_member<meta_gs>()), "");
```

### A.2. Global scope reflection

```cpp
// reflected global scope
typedef mirrored(::) meta_gs;
```

```cpp
static_assert(is_metaobject<meta_gs>(), "");
```

```cpp
// Is a MetaNamed
static_assert(meta::has_name<meta_gs>(), "");
// Is a MetaScoped
static_assert(meta::has_scope<meta_gs>(), "");
// Is a MetaScope
static_assert(meta::is_scope<meta_gs>(), "");
// Is not a MetaTemplate
static_assert(not(meta::is_template<meta_gs>()) , "");
// Is not a MetaInstantiation
static_assert(not(meta::has_template<meta_gs>() ), "");
// Is not a MetaClassMember
static_assert(not(meta::is_class_member<meta_gs>()) , "");

// Is a MetaGlobalScope
static_assert(
    is_base_of<
        meta::global_scope_tag,
        metaobject_category<meta_gs>
    >(), "");
```

```cpp
// Global scope is its own scope
```
static_assert(
    is_base_of<
        meta_gs,
        meta::scope<meta_gs>
    >(), ""
);

// Empty base and full name
assert(strlen(meta::base_name<meta_gs>()) == 0);
assert(strcmp(meta::base_name<meta_gs>(), "") == 0);

assert(strlen(meta::full_name<meta_gs>()) == 0);
assert(strcmp(meta::full_name<meta_gs>(), "") == 0);

// the sequence of members
typedef meta::members<meta_gs>::type meta_gs_members;

static_assert(
    meta::size<meta_gs_members>() == 20, // YMMV
    ""
);

A.3. Namespace reflection

// reflected namespace std
typedef mirrored(std) meta_std;

static_assert(is_metaobject<meta_std>(), "");

// Is a MetaNamed
static_assert(meta::has_name<meta_std>(), "");
// Is a MetaScoped
static_assert(meta::has_scope<meta_std>(), "");
// Is a MetaScope
static_assert(meta::is_scope<meta_std>(), "");
// Is not a MetaTemplate
static_assert(not(meta::is_template<meta_std>()), "");
// Is not a MetaInstantiation
static_assert(not(meta::has_template<meta_std>()), "");
// Is not a MetaClassMember
static_assert(not(meta::is_class_member<meta_std>()), "");
// Is a MetaNamespace
static_assert(
    is_base_of<
        meta::namespace_tag,
        metaobject_category<meta_std>
    >(), ""
);

// The scope of namespace std is the global scope
static_assert(
    is_base_of<
        meta_gs,
        meta::scope<meta_std>
    >(), ""
);

// The base and full name
assert(strlen(meta::base_name<meta_std>()) == 3);
assert(strcmp(meta::base_name<meta_std>(), "std") == 0);
assert(strlen(meta::full_name<meta_std>()) == 3);
assert(strcmp(meta::full_name<meta_std>(), "std") == 0);

A.4. Type reflection

// reflected type unsigned int
typedef mirrored(unsigned int) meta_uint;

static_assert(is_metaobject<meta_uint>(), "");

// Is a MetaNamed
static_assert(meta::has_name<meta_uint>(), "");
// Is a MetaScoped
static_assert(meta::has_scope<meta_uint>(), "");
// Is not a MetaScope
static_assert(not(meta::is_scope<meta_uint>()), "");
// Is not a MetaTemplate
static_assert(not(meta::is_template<meta_uint>()), "");
// Is not a MetaInstantiation
static_assert(not(meta::has_template<meta_uint>()), "");
// Is not a MetaClassMember
static_assert(not(meta::is_class_member<meta_uint>()), "");

// Is a MetaType
static_assert(
is_base_of<
  meta::type_tag,
  metaobject_category<meta_uint>
>() , ""
);

// The scope of unsigned int is the global scope
static_assert(
  is_base_of<
    meta_gs,
    meta::scope<meta_uint>
 >() , ""
);

// The original type
static_assert(
  is_same<
    unsigned int,
    meta::original_type<meta_uint>::type
 >() , ""
);

assert(strlen(meta::base_name<meta_uint>()) == 12);
assert(strcmp(meta::base_name<meta_uint>(), "unsigned int") == 0);
assert(strlen(meta::full_name<meta_uint>()) == 12);
assert(strcmp(meta::full_name<meta_uint>(), "unsigned int") == 0);

A.5. Typedef reflection

// reflected typedef std::size_t
typedef mirrored(std::size_t) meta_size_t;

static_assert(is_metaobject<meta_size_t>() , ""
);
static_assert(meta::has_name<meta_size_t>() , ""
);
static_assert(meta::has_scope<meta_size_t>() , ""
);
static_assert(not(meta::is_scope<meta_size_t>()) , ""
);
static_assert(not(meta::is_template<meta_size_t>()) , ""
);
static_assert(not(meta::has_template<meta_size_t>()) , ""
);
static_assert(not(meta::is_class_member<meta_size_t>()) , ""
);

// Is a MetaTypedef
static_assert(
  ...
is_base_of<
    meta::typedef_tag,
    metaobject_category<meta_size_t>
  >(), ""
);

// The scope of std::size_t is the namespace std
static_assert(
  is_base_of<
    meta_std,
    meta::scope<meta_size_t>
  >(), ""
);

// The original type
static_assert(
  is_same<
    std::size_t,
    meta::original_type<meta_size_t>::type
  >(), ""
);

// the "source" type of the typedef
typedef meta::type<meta_size_t>::type meta_size_t_type;
static_assert(
  is_base_of<
    meta::type_tag,
    metaobject_category<meta_size_t_type>
  >(), ""
);

// The original type
static_assert(
  is_same<
    std::size_t,
    meta::original_type<meta_size_t_type>::type
  >(), ""
);

assert(strlen(meta::base_name<meta_size_t>()) == 6);
assert(strcmp(meta::base_name<meta_size_t>(), "size_t") == 0);
assert(strlen(meta::full_name<meta_size_t>()) == 11);
assert(strcmp(meta::full_name<meta_size_t>(), "std::size_t") == 0);
// YMMV
assert(strlen(meta::base_name<meta_size_t_type>()) == 12);
assert(strcmp(meta::base_name<meta_size_t_type>(), "unsigned int") == 0);

### A.6. Class reflection

```cpp
struct A
{
    int a;
};

class B
{
private:
    bool b;
public:
    typedef int T;
};
class C
: public A
, virtual protected B
{
public:
    static constexpr char c = 'C';
    
    struct D : A
    {
        static double d;
    } d;
};
union U
{
    long u;
    float v;
};
typedef mirrored(A) meta_A;
typedef mirrored(B) meta_B;
typedef mirrored(C) meta_C;
typedef mirrored(C::D) meta_D;
typedef mirrored(B::T) meta_T;
typedef mirrored(U) meta_U;
```
// classes are scopes
static_assert(meta::is_scope<meta_A>(), ""s);
static_assert(meta::is_scope<meta_B>(), ""s);
static_assert(meta::is_scope<meta_C>(), ""s);
static_assert(meta::is_scope<meta_D>(), ""s);
static_assert(meta::is_scope<meta_U>(), ""s);

// A, B, C, C::D and U are all elaborated types
assert(is_base_of<meta::class_tag, metaobject_category<meta_A>>());
assert(is_base_of<meta::class_tag, metaobject_category<meta_B>>());
assert(is_base_of<meta::class_tag, metaobject_category<meta_C>>());
assert(is_base_of<meta::class_tag, metaobject_category<meta_D>>());
assert(is_base_of<meta::class_tag, metaobject_category<meta_U>>());

static_assert(!meta::is_class_member<meta_A>(), ""s);
static_assert(!meta::is_class_member<meta_B>(), ""s);
static_assert(!meta::is_class_member<meta_C>(), ""s);
static_assert(meta::is_class_member<meta_D>(), ""s);
static_assert(meta::is_class_member<meta_T>(), ""s);
static_assert(!meta::is_class_member<meta_U>(), ""s);

// typenames
assert(strcmp(meta::base_name<meta_A>(), "A") == 0);
assert(strcmp(meta::base_name<meta_B>(), "B") == 0);
assert(strcmp(meta::full_name<meta_D>(), "C::D") == 0);

// reflected laborated type specifiers for A, B and U
typedef meta::elaborated_type_specifier<meta_A>::type meta_A_ets;
typedef meta::elaborated_type_specifier<meta_B>::type meta_B_ets;
typedef meta::elaborated_type_specifier<meta_U>::type meta_U_ets;

// specifier keywords
assert(strcmp(meta::keyword<meta_A_ets>(), "struct") == 0);
assert(strcmp(meta::keyword<meta_B_ets>(), "class") == 0);
assert(strcmp(meta::keyword<meta_U_ets>(), "union") == 0);

// specifier tags
assert(is_base_of<meta::struct_tag, meta::specifier_category<meta_A_ets>>());
assert(is_base_of<meta::class_tag, meta::specifier_category<meta_B_ets>>());
assert(is_base_of<meta::union_tag, meta::specifier_category<meta_U_ets>>());

// reflected sequences of members of the A, B and C classes
typedef meta::members<meta_A>::type meta_A_members;
typedef meta::members<meta_B>::type meta_B_members;
typedef meta::members<meta_C>::type meta_C_members;

static_assert(meta::size<meta_A_members>() == 1, ""); // A::a
static_assert(meta::size<meta_B_members>() == 2, ""); // B::b, B::C
static_assert(meta::size<meta_C_members>() == 3, ""); // C::c, C::D, C::d

// reflected members of B and C
typedef meta::at<meta_B_members, 0>::type meta_B_b;
typedef meta::at<meta_B_members, 1>::type meta_B_T;
typedef meta::at<meta_C_members, 0>::type meta_C_c;
typedef meta::at<meta_C_members, 1>::type meta_C_D;
typedef meta::at<meta_C_members, 2>::type meta_C_d;

assert(is_base_of<meta::variable_tag, metaobject_category<meta_B_b>>());
assert(is_base_of<meta::typedef_tag, metaobject_category<meta_B_T>>());
assert(is_base_of<meta::class_tag, metaobject_category<meta_C_D>>());

// MetaClassMembers
static_assert( meta::is_class_member<meta_B_b>(), "" );
static_assert( meta::is_class_member<meta_B_T>(), "" );
static_assert( meta::is_class_member<meta_C_D>(), "" );
static_assert( meta::is_class_member<meta_C_d>(), "" );

// access specifiers
typedef meta::access_specifier<meta_B_b>::type meta_B_b_access;
typedef meta::access_specifier<meta_C_D>::type meta_C_D_access;

// specifier keywords
assert(strcmp(meta::keyword<meta_B_b_access>(), "private") == 0);
assert(strcmp(meta::keyword<meta_C_D_access>(), "public") == 0);

// sequence of base classes of C
typedef meta::base_classes<meta_C>::type meta_C_bases;

static_assert(meta::size<meta_C_bases>() == 2, ""); // A, B

// MetaInheritances of C->A and C->B
typedef meta::at<meta_C_bases, 0>::type meta_C_base_A;
typedef meta::at<meta_C_bases, 1>::type meta_C_base_B;

// inheritance specifiers
typedef meta::inheritance_specifier<meta_C_base_A>::type meta_C_base_A_it;
typedef meta::inheritance_specifier<meta_C_base_B>::type meta_C_base_B_it;
// access specifiers
typedef meta::access_specifier<
    meta_C_base_A>::type meta_C_base_A_acc;
typedef meta::access_specifier<
    meta_C_base_B>::type meta_C_base_B_acc;

// specifier keywords
assert(strcmp(meta::keyword<
    meta_C_base_A_it>(), "") == 0);
assert(strcmp(meta::keyword<
    meta_C_base_B_it>(), "virtual") == 0);
assert(strcmp(meta::keyword<
    meta_C_base_A_acc>(), "public") == 0);
assert(strcmp(meta::keyword<
    meta_C_base_B_acc>(), "protected") == 0);

// specifier tags
static_assert(
    is_base_of<
        meta::none_tag,
        meta::specifier_category<
            meta_C_base_A_it>
    >(), ""
);
static_assert(
    is_base_of<
        meta::virtual_tag,
        meta::specifier_category<
            meta_C_base_B_it>
    >(), ""
);
static_assert(
    is_base_of<
        meta::public_tag,
        meta::specifier_category<
            meta_C_base_A_acc>
    >(), ""
);

// base classes
static_assert(
    is_base_of<
        meta_A,
        meta::base_class<
            meta_C_base_A>
    >(), ""
);
static_assert(
    is_base_of<
        meta_B,
        meta::base_class<
            meta_C_base_B>
    >(), ""
);