Static reflection

How to read this document  The first two sections are devoted to the introduction to reflection and reflective programming, they contain some motivational examples and some experiences with usage of a library-based reflection utility. These can be skipped if you are knowledgeable about reflection. Section 3 contains the rationale for the design decisions. The most important part is the technical specification in section 4, the impact on the standard is discussed in section 5, the issues that need to be resolved are listed in section 7, and section 6 mentions some implementation hints.

Contents

1. Introduction 4

2. Motivation and Scope 6
   2.1. Usefullness of reflection .............................................. 6
   2.2. Motivational examples ............................................... 7
       2.2.1. Factory generator ................................................ 7

3. Design Decisions 11
   3.1. Desired features .................................................... 11
   3.2. Layered approach and extensibility ............................. 11
       3.2.1. Basic metaobjects ............................................. 12
       3.2.2. Mirror .......................................................... 12
       3.2.3. Puddle ........................................................... 12
       3.2.4. Rubber .......................................................... 13
       3.2.5. Lagoon ........................................................... 13
   3.3. Class generators .................................................... 14
   3.4. Compile-time vs. Run-time reflection .......................... 16

4. Technical Specifications 16
   4.1. Metaobject Concepts ............................................... 17
       4.1.1. Categorization and Traits .................................... 17
       4.1.2. String .......................................................... 17
       4.1.3. Range {Element} .............................................. 18
### 4.1. Static reflection

6. Implementation hints 45
   6.1. Generation of metaobjects 45

7. Unresolved Issues 45

8. Acknowledgements 46

9 References 46

A. Transforming the concepts to C++ 46
   A.1. Concept models – variant 1 (preferred) 46
      A.1.1. String 46
      A.1.2. Range 46
      A.1.3. MetaobjectCategory 47
      A.1.4. SpecifierCategory 47
      A.1.5. Metaobject 48
      A.1.6. Specifier 49
      A.1.7. Named 49
      A.1.8. Scoped 49
      A.1.9. NamedScoped 50
      A.1.10. Scope 50
      A.1.11. Namespace 51
      A.1.12. GlobalScope 52
      A.1.13. Type 53
      A.1.14. Typedef 53
      A.1.15. Class 54
      A.1.16. Function 55
      A.1.17. ClassMember 56
      A.1.18. Initializer 57
      A.1.19. Constructor 59
      A.1.20. Operator 60
      A.1.21. OverloadedFunction 62
      A.1.22. Template 63
      A.1.23. TemplateParameter 65
      A.1.24. Instantiation 66
      A.1.25. Enum 68
      A.1.26. EnumClass 68
      A.1.27. Inheritance 69
      A.1.28. Variable 70
      A.1.29. Parameter 71
      A.1.30. Constant 72
   A.2. Concept models – variant 2 (alternative) 72
      A.2.1. String 72
      A.2.2. Range 73
1. Introduction

Reflection and reflective programming can be used for a wide range of tasks such as implementation of serialization-like operations, 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 structures providing basic metadata describing various program constructs. These metaobjects can later be used to implement a library or a set of 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.

The term reflection refers to the ability of a computer program to observe and possibly alter its own structure and/or its behavior. This includes building new or altering the existing data structures, doing changes to algorithms or changing the way the program code is interpreted. Reflective programming is a particular kind of metaprogramming.

Reflection should follow the principle of Ontological correspondence, i.e. should reflect the base-level program constructs as closely as possible to a reasonable level. Reflection should not omit existing language features not invent new ones that do not exist at the base-level.

What reflection "looks like" is therefore very language-specific. Reflection for C++ is necessary different from reflection in Smalltalk since these are two quite different languages.

The "reasonability” applies to the level-of-detail of the metadata provided by reflection. It is a tradeoff between the complexity of the reflection system and its usefulness. The "metadata" provided by the currently standard typeid operator are rather simple (which may be good), but their usefulness is very limited (which is bad). On the other hand a fictional reflection facility that would allow to inspect the individual instructions of a function could be useful for some specific applications, but this system would also be very complex to implement and use. The proposed reflection system tries to walk a "middle ground" and be usable in many situations without unmanageable complexity.

The advantage of using reflection is in the fact that everything is implemented in a single programming language, and the human-written code can be closely tied with the customizable reflection-based code which is automatically generated by compiler metaprograms, based on the metadata provided by reflection.

The solution proposed in this paper is based on the experience with Mirror reflection utilities [1] and with reflection-based metaprogramming. In the following chapters and appendices several examples of usage of various components of the Mirror library are provided. These are included just to show what can be done with a sufficiently advanced reflection facility. The Mirror library, however is not part of the current proposal.
2. Motivation and Scope

2.1. Usefulness of reflection

There is a wide range of computer programming tasks that involve the execution of the same algorithm on a set of types defined by an application or on instances of these types, accessing member variables, calling free or member functions in an uniform manner, converting data between the language's intrinsic representation and external formats, etc., for the purpose of implementing the following:

- serialization or storing of persistent data in a custom binary format or in XML, JSON, XDR, etc.,
- (re-)construction of class instances from external data representations (like those listed above), from the data stored in a relational database, from data entered by a user through a user interface or queried through a web service API,
- automatic generation of a relational schema from the application object model and object-relational mapping (ORM),
- support for scripting
- support remote procedure calls (RPC) / remote method invocation (RMI),
- inspection and manipulation of existing objects via a (graphic) user interface or a web service,
- visualization of objects or data and the relations between objects or relations in the data,
- automatic or semi-automatic implementation of certain software design patterns,
- etc.

There are several aproaches to the implementation of such functionality. The most straightforward and also usually the most error-prone is manual implementation. Many of the tasks listed above are inherently repetitive and basically require to process programming language constructs (types, structures, containers, functions, constructors, class member variables, enumerated values, etc.) in a very uniform way that could be easily transformed into a meta-algorithm.

While it is acceptable (even if not very advantageous) for example, for a design pattern implementation to be made by a human, writing RPC/RMI-related code is a task much better suited for a computer.

This leads to the second, heavily used approach: preprocessing and parsing of the program source text by a (usually very specific) external program (documentation generation tool, interface definition language compiler for RPC/RMI, web service interface generator, a rapid application development environment with a form designer, etc.) resulting
in additional program source code, which is then compiled into the final application binary.

This approach has several problems. First, it requires the external tools which may not fit well into the build system or may not be portable between platforms or be free; second, such tools are task-specific and many of them allow only a limited, if any, customization of the output.

Another way to automate these tasks is to use reflection, reflective programming, metaprogramming and generic programming as explained below.

2.2. Motivational examples

This section describes some of the many possible uses of reflection and reflective programming on concrete real-world examples.

2.2.1. Factory generator

As already said above, it is possible (at least partially) to automate the implementation of several established software design patterns. This example shows how to implement a variant of the Factory pattern.

By factory we mean here a class, which can create instances of a Product type, but does not require that the caller chooses the manner of the construction (in the programming language) nor supplies the required arguments directly in the C++ intrinsic data representation.

So instead of direct construction of a Product type,

```cpp
// get the values of arguments from the user
int arg1 = get_from_user<int>("Product arg1");
double arg2 = get_from_user<double>("Product arg2");
std::string arg3 = get_from_user<std::string>("Product arg3");
```

```cpp
// call a constructor with these arguments
Product* pp = new Product(arg1, arg2, arg3);
```

```cpp
// default construct a Product
Product p;
```

```cpp
// copy construct a Product
Product cp = p;
```

which involves selection of a specific constructor, getting the values of the required arguments and possibly converting them from an external representation and calling the selected constructor with the arguments, factories pick or let the application user pick
the Product’s most appropriate constructor, they gather the necessary parameters in a
generic way and use the selected constructor to create an instance of the Product:

```
// get data necessary for construction in xml
XMLNode xml_node_1 = get_xml_node(...);
XMLNode xml_node_2 = get_xml_node(...);

// make a factory for the product type
Factory<Product, XMLWalker> xml_factory;

// use the factory to create instances of Product
// from the external representation
Product p = xml_factory(xml_node_1);
Product* pp = xml_factory.new_(xml_node_2);
```

One of the interesting features of these factories is, that they separate the caller (who
just needs to get an instance of the specified type) from the actual method of creation.

By using a factory, the constructor to be called can be automatically picked depending
on the data available only at run-time and not be chosen by the programmer (at least
not directly as in the code above). Factory can match the constructor to best fit the
data available in the external representation (XML or JSON fragment, dataset resulting
from a RDBS query, etc.)

Even more interesting is, that such factories can be implemented semi-automatically
with the help of reflection.

Every factory is a composition of two distinct (and nearly orthogonal) parts:

- **Product-type-dependent**: includes the enumeration of Product’s constructors, enu-
  meration of their parameters, information about the context in which a constructor
  is called, etc. This part is based on reflection and independent on the representa-
  tion of the input data.

- **Data representation-dependent**: includes the scanning of the available input data,
  conversion into C++ intrinsic data representation, and the selection of the best
  constructor. This part is user-defined and specifies how the input data is gathered
  and converted into the C++ representation.

These two parts are then tied together into the factory class. Based on the input-data
related components, the factory can include a script parser or XML document tree
walker or code dynamically generating a GUI for the input of the necessary values and
the selection of the preferred constructor. Figure 1 shows such a GUI created by factory
automatically generated by the Mirror’s *factory generator* utility for a tetrahedron class
with the following definition:

```
struct vector
{
```
Figure 1: Example of a GUI created by a factory generated by the Mirror’s factory generator.

```c
double x, y, z;

vector(double _x, double _y, double _z)
   : x(_x), y(_y), z(_z)
{
}

vector(double _w)
   : x(_w), y(_w), z(_w)
{
}

vector(void)
   : x(0.0), y(0.0), z(0.0)
{
}

/* other members */
```

};

struct triangle
{
    vector a, b, c;

    triangle(
        const vector& _a,
        const vector& _b,
        const vector& _c
    ): a(_a), b(_b), c(_c)
    { }

    triangle(void){ }

    /* other members */
};

struct tetrahedron
{
    triangle base;
    vector apex;

    tetrahedron(const triangle& _base, const vector& _apex)
        : base(_base), apex(_apex)
    { }

    tetrahedron(
        const vector& a,
        const vector& b,
        const vector& c,
        const vector& d
    ): base(a, b, c), apex(d)
    { }

    /* other members */
};
3. Design Decisions

3.1. Desired features

The proposed reflection facility is designed with the following goals in mind:

- **Reusability**: The provided metadata should be reusable in many situations and for many different purposes, not only the obvious ones. This is closely related to completeness (below).

- **Flexibility**: The basic reflection and the libraries built on top of it should be designed in a way that they are eventually usable during both compile-time and run-time and under various paradigms (object-oriented, functional, etc.), depending on the application needs.

- **Encapsulation**: The metadata should be accessible through conceptually well-defined interfaces.

- **Stratification**: Reflection should be non-intrusive, and the meta-level should be separated from the base-level language constructs it reflects. Also, reflection should not be implemented in a all-or-nothing manner. Things that are not needed, should not generally be compiled-into the final application.

- **Ontological correspondence**: The meta-level facilities should correspond to the ontology of the base-level C++ language constructs which they reflect. This basically means that all existing language features should be reflected and new ones should not be invented. This rule may have some important exceptions, like the reflection of containers.

- **Completeness**: The proposed reflection facility should provide as much useful metadata as possible, including various specifiers, (like constness, storage-class, access, etc.), namespace members, enumerated types, iteration of namespace members and much more.

- **Ease of use**: Although reflection-based metaprogramming allows to implement very complicated things, simple things should be kept simple.

- **Cooperation with other libraries**: Reflection should be usable with the existing introspection facilities (like `type_traits`) already provided by the standard library and with other libraries.

3.2. Layered approach and extensibility

The purpose of this section is to show that a static → dynamic and basic → complex approach in designing reflection can accommodate a wide variety of programming styles and is arguably the ”best” one. We do not propose to add all layers described below
into the standard library. They are mentioned here only to show that a well designed compile-time reflection is a good foundation for many (if not all) other reflection facilities.

The Mirror reflection utilities [1] on which this proposal is based, implements several distinct components which are stacked on top of each other. From the low-level metadata, through a functional-style compile-time interface to a completely dynamic object-oriented run-time layer (all described in greater detail below).

3.2.1. Basic metaobjects

The very basic metadata, which are in Mirror provided (registered) by the user (or an automated command-line tool) via a set of preprocessor macros. This approach is both inconvenient and error-prone in many situations, but also has its advantages.

We propose that a standard compiler should make these metadata available to the programmer through the static basic metaobject interfaces described below. These should serve as the basis for other (standard and non-standard) higher-level reflection libraries and utilities.

In the Mirror utilities the basic metadata is not used directly by applications.

3.2.2. Mirror

Mirror is a compile-time functional-style reflective programming library, which is based directly on the basic metadata and is suitable for generic programming, similar to the standard type_traits library.

Mirror is the original library from which the Mirror reflection utilities started.

It provides a more user-friendly and rich interface than the basic-metaobjects, and a set of metaprogramming utilities which allow to write compile-time meta-programs, which can generate efficient and optimized program code using only those metadata that are required.

The appendix B contains several (rather simple) examples of usage and the functional style of the algorithms based on metadata provided by Mirror.

3.2.3. Puddle

Puddle is an OOP-style (mostly) compile-time interface built on top of Mirror. It copies the metaobject concept hierarchy of Mirror, but provides a more ”object-ish” interface as shown below:

Instead of Mirror’s:
N3996- Static reflection

```cpp
static_assert(
    is_public<
        access_type<
            at_c<
                member_variables<
                    reflected<person>
                , reflected<person>
            >, 0
        >
    >::value,
    "Shoot, persons first mem. variable is not public!"
)
```

Puddle allows to do the following:

```cpp
assert(
    reflected_type<person>()
    member_variables().
    at_c<0>().
    access_type().
    is_public()
);
```

For a more complex use-case see appendix C.

### 3.2.4. Rubber

Rubber is a OOP-style run-time type erasure utility built on top of Mirror and Puddle. It again follows the metaobject concept hierarchy of Mirror and Puddle. Rubber allows to access and store metaobjects of the same category in a single type, so in contrast to Mirror and Puddle where a meta-type reflecting the `int` type and a meta-type reflecting the `double` type have different types in Rubber they can both be stored in a variable of the same type. Rubber does not use virtual functions but rather pointers to existing functions implemented by Mirror to achieve run-time polymorphism.

For examples of usage see appendix D.

### 3.2.5. Lagoon

Lagoon defines run-time polymorphic interfaces and classes implementing these interfaces and wrapping the compile-time metaobjects from Mirror and Puddle. While Rubber is more suitable for simple decoupling of reflection-based algorithms from the real types of the metaobjects that the algorithms operate on, Lagoon is full-blown run-time
reflection utility that can be even decoupled from the application using it and loaded dynamically on-demand.

See appendix E for examples of usage.

3.3. Class generators

There are situations where the following transformation of scopes (classes, enumerations, etc.) and their members would be very useful. Consider a simple user-defined structs address and person,

```cpp
struct address
{
    std::string street;
    std::string number;
    std::string postal_code;
    std::string city;
    std::string country;
};

struct person
{
    std::string name;
    std::string surname;

    address residence;

    std::tm birth_date;
};
```

and an object-relational mapping (ORM) library, that would allow automatic generation of SQL queries from strongly typed expressions in a DSEL in C++. It would be advantageous to have some counterparts for all "ORM-aware" classes having members with the same names as the original class, but with different types, like:

```cpp
template <class T>
struct orm_table;

template <>
struct orm_table<address>
    : public base_table
{
    orm_column<std::string> street;
    orm_column<std::string> number;
    orm_column<std::string> postal_code;
};
```
N3996- Static reflection

```cpp
orm_column<std::string> city;
orm_column<std::string> country;

orm_table(orm_param& param)
  : base_table(param)
  , street(this, param)
  , number(this, param)
  , postal_code(this, param)
  , city(this, param)
  , country(this, param)
{ }
};

template <>
struct orm_table<person>
  : public base_table
{
  orm_column<std::string> name;
  orm_column<std::string> surname;
  orm_column<address> residence;
  orm_column<std::tm> birth_date;

  orm_table(orm_param& param)
    : base_table(param)
    , name(this, param)
    , surname(this, param)
    , residence(this, param)
    , birth_date(this, param)
  { }
};

Generating such or similar classes can also be achieved with reflection. The Mirror library implements the `by_name` metafunction template and the `class_generator` utility for this purpose.

The Puddle layer, described above, uses this functionality and allows access to metadata reflecting member variables of a class or free variables of a namespace through the overloaded `operator ->` of a meta-class or meta-namespace:

```cpp
auto meta_person = puddle::reflected_type<person>();
// access the metavariable reflecting
// the birth_date member of person
assert(meta_person->birth_date().access_type().is_public());
// access the metadata for person::name
// and person::surname by their names
```
assert(
    meta_person->name() == 
    meta_person.member_variables().at_c<0>()
);  
assert(
    meta_person->surname() != 
    meta_person.member_variables().at_c<0>()
);  

This functionality could be extended to any scope member and the mechanism is described below.

### 3.4. Compile-time vs. Run-time reflection

Run-time, dynamic reflection facilities may seem more readily usable, but with the increasing popularity of compile-time metaprogramming, the need for compile-time introspection (already taken care of by `type_traits`) and reflection also increases.

Also, if compile-time reflection is well supported it is relatively easy to implement run-time or even dynamically loadable reflection on top of it. The opposite is not true: One cannot use run-time metaobjects or the value returned by their member functions as template parameters or compile-time constants.

From the performance point of view, algorithms based on static meta-data offer much more possibilities for the compiler to do optimizations.

Thus, taking shortcuts directly to run-time reflection, without compile-time support has obvious drawbacks.

### 4. Technical Specifications

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.

4.1. Metaobject Concepts

This section conceptually describes the requirements that various metaobjects need to satisfy in order to be considered models of the individual concepts. There are several ways how the conceptual model can be transformed into the final C++ form. These are discussed in the 4.2 subsection below. For examples of concrete renderings please see Appendix A.

4.1.1. Categorization and Traits

In order to provide means for distinguishing between regular types and metaobjects the is_metaobject trait should be added and should "return" true for metaobjects (types defined by the compiler providing metadata) and false for non-metaobjects (native or user defined types). See the definition of the Metaobject concept for further metaobject traits and tags.

4.1.2. String

String is an "object" which can be examined at compile-time that represents constant C-character string storing for example a name of a type, function, namespace, etc. or the keyword of a specifier. It allows compile-time metaprograms to examine and make decisions based on the value of such strings. If necessary, the stored string can be returned as a regular C-string.

One of the use-cases for these strings is the filtering of scope members based on their names if a good naming policy is consistently applied. For example: filter out all scope members whose name starts with an underscore or process only classes with names starting with DB, Persistent, etc.

Concrete metaobjects modelling String must satisfy the following:
• **attribute**: `size_t size` – Specifies the length (in chars) of the *String*, not counting any terminating characters.

• **attribute**: `const char* c_str` – The encapsulated constant character string value.

4.1.3. **Range** {Element}

*Range* is a static constant parametrized container, containing and providing random-access to elements satisfying the *Element* concept.

Concrete metaobjects modelling *Range* must satisfy the following:

• **attribute**: `size_t size` – Specifies the number of elements in the *Range*.

• **element**: `Element at(position)` – Returns the element at the specified position in the *Range*.

  – **position**: The position of the element to be returned. Valid values for this argument are from the range `{0...size − 1}`. For other arguments the result is undefined.

4.1.4. **MetaobjectCategory**

*MetaobjectCategory* is a compile-time tag specifying the category of a metaobject.

Instances of the *MetaobjectCategory* concept are listed below.

• **instance**: `namespace` – Indicates that the tagged metaobject satisfies the *Namespace* concept.

• **instance**: `global_scope` – Indicates that the tagged metaobject satisfies the *GlobalScope* concept.

• **instance**: `type` – Indicates that the tagged metaobject satisfies the *Type* concept.

• **instance**: `typedef` – Indicates that the tagged metaobject satisfies the *Typedef* concept.

• **instance**: `class` – Indicates that the tagged metaobject satisfies the *Class* concept.

• **instance**: `function` – Indicates that the tagged metaobject satisfies the *Function* concept.

• **instance**: `constructor` – Indicates that the tagged metaobject satisfies the *Constructor* concept.
• instance: **operator** – Indicates that the tagged metaobject satisfies the *Operator* concept.

• instance: **overloaded_function** – Indicates that the tagged metaobject satisfies the *OverloadedFunction* concept.

• instance: **enum** – Indicates that the tagged metaobject satisfies the *Enum* concept.

• instance: **enum_class** – Indicates that the tagged metaobject satisfies the *Enum-Class* concept.

• instance: **inheritance** – Indicates that the tagged metaobject satisfies the *Inheritance* concept.

• instance: **constant** – Indicates that the tagged metaobject satisfies the *Constant* concept.

• instance: **variable** – Indicates that the tagged metaobject satisfies the *Variable* concept.

• instance: **parameter** – Indicates that the tagged metaobject satisfies the *Parameter* concept.

### 4.1.5. SpecifierCategory

*SpecifierCategory* is a compile-time tag specifying an exact C++ specifier.

Instances of the *SpecifierCategory* concept are listed below.

• instance: **none** – Indicates missing specifiers; for example a reflected non-const member function would have a *none* constness specifier tag or a variable with automatic storage class would have a *none* storage class specifier, etc. The *keyword* attribute in *Specifiers* with this category tag is an empty *String*.

• instance: **extern** – Indicates that the tagged metaobject satisfies the *Specifier* concept and is reflecting the *extern storage* or *linkage* specifier.

• instance: **static** – Indicates that the tagged metaobject satisfies the *Specifier* concept and is reflecting the *static storage* or *linkage* specifier.

• instance: **mutable** – Indicates that the tagged metaobject satisfies the *Specifier* concept and is reflecting the *mutable storage* or *linkage* specifier.

• instance: **register** – Indicates that the tagged metaobject satisfies the *Specifier* concept and is reflecting the *register storage* or *linkage* specifier.

• instance: **thread_local** – Indicates that the tagged metaobject satisfies the *Specifier* concept and is reflecting the *thread_local storage* or *linkage* specifier.
• instance: const – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the const constness specifier.

• instance: virtual – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the virtual inheritance type or linkage specifier.

• instance: private – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the private access type specifier.

• instance: protected – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the protected access type specifier.

• instance: public – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the public access type specifier.

• instance: class – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the class elaborated type specifier.

• instance: struct – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the struct elaborated type specifier.

• instance: union – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the union elaborated type specifier.

• instance: enum – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the enum elaborated type specifier.

• instance: enum_class – Indicates that the tagged metaobject satisfies the Specifier concept and is reflecting the enum_class elaborated type specifier.

• instance: constexpr – Indicates that the tagged metaobject satisfies the Specifier constness specifier.

4.1.6. Metaobject

Metaobject is a stateless or (monostate) anonymous type which provides metadata reflecting certain program features.

Concrete metaobjects modelling Metaobject must satisfy the following:

• The is_metaobject trait inherited from Metaobject is true.

• trait: MetaobjectCategory category – A tag specifying the category of the concrete metaobject, which allows metaprograms to do tag dispatching and indicates which concepts the concrete metaobject models.

• trait: bool is_metaobject – Indicates that the examined metaobject satisfies the Metaobject concept.
• trait: bool has_name – Indicates that the examined metaobject satisfies the Named concept.

• trait: bool has_scope – Indicates that the examined metaobject satisfies the Scoped concept.

• trait: bool is_scope – Indicates that the examined metaobject satisfies the Scope concept.

• trait: bool is_class_member – Indicates that the examined metaobject satisfies the ClassMember concept.

• trait: bool has_template – Indicates that the examined metaobject satisfies the Instantiation concept.

• trait: bool is_template – Indicates that the examined metaobject satisfies the Template concept.

4.1.7. Specifier

Specifier is a Metaobject which reflects specifiers (like const, static, virtual, etc.) used in the definition the base-level program constructs.

Metaobject ← Specifier

Specifier has these requirements:

• trait: SpecifierCategory category – Refines the category. The resulting tag can be used to identify the concrete specifier reflected by this Specifier.

• attribute: String keyword – The C++ keyword of the reflected specifier.

4.1.8. Named

Named is a Metaobject that reflects program constructs, which have a name, like namespaces, types, functions, variables, parameters, etc.

Metaobject ← Named

Concrete metaobjects modelling Named must satisfy the following:

• The has_name trait inherited from Metaobject is true.

• attribute: String base_name – The base name of the reflected construct, without the nested name specifier. 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 `std::vector<int>::iterator` it should be "iterator". For derived, qualified types like `volatile std::vector<const foo::bar::fubar*> * const *` it should be "volatile vector<const fubar*> * const *", etc.

4.1.9. Scoped

Scoped is a Metaobject reflecting program constructs, which are defined inside of a named Scope (global scope, namespace, class, etc.).

Concrete metaobjects modelling Scoped must satisfy the following:

- The has_scope trait inherited from Metaobject is true.
- attribute: Scope scope – A Scope metaobject reflecting the scope of the scoped object reflected by this Scoped metaobject. In concrete metaobjects the result can be a Namespace, GlobalScope, Class, etc.

4.1.10. NamedScoped

Many of the concepts are specializations of both the Scoped and Named concepts.

Concrete metaobjects modelling NamedScoped must satisfy the following:

- attribute: String full_name – The full name of the reflected construct, with the nested name specifier. 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 c-string literal. 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. For some metaobjects this value may be the same as the base_name attribute.
- template: unspecified named_typedef(X) – A template, instantiation of which should result in a type equivalent to the struct in the following pseudo-code:

```c++
struct unspecified
{
```
typedef X <NAME>;
};

The <NAME> expression above should be replaced by the name of the reflected named scoped object. This structure could be used to generate new classes with member typedefs having the same names as the members of the scope of the named object reflected by this NamedScoped metaobject. One way to combine the <NAME> typedefs from various reflected scope members into a single class would be to let the class inherit from multiple types generated by the named_TypeDef template from the metaobjects obtained by reflection.

– X: The parameter passed to the named_TypeDef template by the end-user. It is used as the "source" type of the typedef (with the same name as the reflected language construct) in the resulting type.

• template: unspecified named_mem_var(X) – A template, instantiation of which should result in a type equivalent to the struct in the following pseudo-code:

```cpp
struct unspecified
{
  X <NAME>;

  unspecified(void) = default;

  template <typename Param>
  unspecified(Param&& param)
  : <NAME>(std::forward<Param>(param))
  { }
};
```

The <NAME> expression above should be replaced by the name of the reflected named scoped object. This structure could be used to generate new classes with member variables having the same names as the members of the scope of the named object reflected by this NamedScoped metaobject. One way to combine the <NAME> member variables from various reflected scope members into a single class would be to let the class inherit from multiple types generated by the named_mem_var template from the metaobjects obtained by reflection.

– X: The parameter passed to the named_mem_var template by the end-user. It is used as the type of the member variable (with the same name as the reflected language construct) in the resulting type.

4.1.11. Scope

Scope is a NamedScoped which reflects scopes like namespaces, classes, enum classes, etc.
Scope has these requirements:

- The `is_scope` trait inherited from `Metaobject` is true.
- `range: Range {Scoped} members` – A range of `Scoped` metaobjects reflecting the individual members like types, namespaces, functions, variables, etc. defined inside the scope reflected by this `Scope`.

4.1.12. Namespace

Namespace is a Scope which reflects a namespace.

Namespace has these requirements:

- The value of the `MetaobjectCategory` category attribute inherited from `Metaobject` is `namespace`.

4.1.13. GlobalScope

GlobalScope is a Namespace which reflects the global scope.

GlobalScope has these requirements:

- The value of the `MetaobjectCategory` category attribute inherited from `Metaobject` is `global_scope`. 
4.1.14. Type

Type is a NamedScoped which reflects types

Type has these requirements:

- The value of the MetaobjectCategory category attribute inherited from Metaobject is type.
- attribute: original-type original_type – The original base-level type that this Type is reflecting. Note, that if a concept derived from Type, for example Class, is also a Template (i.e. is reflecting a template not a concrete type), then this attribute is not inherited.

4.1.15. Typedef

Typedef is a Type which reflects typedefs, i.e. types that were defined as alternate names for other types using the C++ typedef expression.

Typedef has these requirements:

- The value of the MetaobjectCategory category attribute inherited from Metaobject is typedef.
- attribute: Type typedef_type – A Type metaobject reflecting the "source" type of the typedef.

4.1.16. Class

Class is a Type and Scope that reflects elaborated types (class, struct, union) or class templates. Note, that if a Class is also a Template, i.e. is reflecting a class template not a concrete class, then the original_type attribute is not inherited from Type.
Concrete metaobjects modelling *Class* must satisfy the following:

- The value of the *MetaobjectCategory* attribute inherited from *Metaobject* is *class*.

- *attribute: Specifier elaborated_type* – Specifier reflecting the elaborated type specifier used to define the class (*class, struct, union*).

- *range: Range \{ Inheritance \} base_classes* – A *Range of Inheritance* metaobjects reflecting the base classes that the class reflected by this *Class* inherits from.

### 4.1.17. Function

*Function* is a *Scope* reflecting a function or a function template. *Function* metaobjects are not direct members of scopes. Instead, all functions with the same name, even those that are not overloaded in a specific scope are grouped into an *OverloadedFunction*. Individual overloaded *Functions* in the group can be obtained through the interface of *OverloadedFunction*. The same should also apply to *Constructors* and *Operators*.

The rationale for this is that direct scope members, i.e. metaobjects accessible through the *Scope's members* attribute should have unique names, which would not be the case if *Functions* were direct scope members.

The *scope* attribute of an *OverloadedFunction* is the same as the *scope* attribute of all *Functions* grouped by that *OverloadedFunction*.

Concrete metaobjects modelling *Function* must satisfy the following:

- The value of the *MetaobjectCategory* attribute inherited from *Metaobject* is *function*.

- *attribute: Specifier linkage* – Specifier reflecting the linkage specifier of the function reflected by this *Function*. 
• **attribute**: `bool constexpr` – Indicates if the reflected function is defined as `constexpr`.

• **attribute**: `Type result_type` – `Type` reflecting the result type of the function.

• **range**: `Range {Parameter} parameters` – A `Range` of `Parameter` metaobject reflecting the parameters of the function reflected by this `Function`.

• **attribute**: `bool noexcept` – Indicates if the reflected function is defined as `noexcept`.

• **range**: `Range {Type} exceptions` – A `Range` of `Type` metaobject reflecting the exception types that the function reflected by this `Function` is allowed to throw.

• **attribute** (conditional): `Specifier constness` – `Specifier` reflecting the the constness specifier of the function reflected by this `Function`. This attribute is available only if the `is_class_member` trait is true for this `Function`, i.e. when it also satisfies the `ClassMember` concept.

• **attribute** (conditional): `bool pure` – Indicates if the function is a pure virtual function. This attribute is available only if the `is_class_member` trait is true for this `Function`, i.e. when it also satisfies the `ClassMember` concept.

• **function**: `unspecified call(...)` – Function with the same return value type and the same number and type of parameters as the original function reflected by this `Function`. Calls to this function should be equivalent to the call of the reflected function with the arguments passed to `call`. Additionally if the reflected function is a member function, then the first of the parameters of `call` should be a reference to the class where the member function is defined and should be used as the `this` argument when calling the member function. If the member function is declared as `const` then the reference to the class should also be `const`.

### 4.1.18. ClassMember

`ClassMember` is a `NamedScoped` that reflects a member of a class.

```
Metaobject <- NamedScoped <- ClassMember
```

Concrete metaobjects modelling `ClassMember` must satisfy the following:

• The `is_class_member` trait inherited from `Metaobject` is true.

• **attribute**: `Specifier access_type` – `Specifier` reflecting the access type specifier of the class member reflected by this `ClassMember`.
4.1.19. **Initializer**

*Initializer* is a *Function* which reflects an initializer (constructor) of a native type.

Concrete metaobjects modelling *Constructor* must satisfy the following:

- The value of the *MetaobjectCategory* category attribute inherited from *Metaobject* is *constructor*.

4.1.20. **Constructor**

*Constructor* is a *ClassMember* and *Function* that reflects a constructor.

4.1.21. **Operator**

*Operator* is a *Function* and possibly a *ClassMember* reflecting an operator.

Concrete metaobjects modelling *Operator* must satisfy the following:
• The value of the MetaobjectCategory category attribute inherited from Metaobject is operator.

4.1.22. OverloadedFunction

OverloadedFunction is a NamedScoped and possibly a ClassMember reflecting an overloaded function. Function metaobjects are not direct members of scopes. Instead, all functions with the same name, even those that are not overloaded in a specific scope are grouped into an OverloadedFunction. Individual overloaded Functions in the group can be obtained through OverloadedFunction. The same should also apply to Constructors and Operators.

The rationale for this is that direct scope members, i.e. metaobjects accessible through the Scope’s members attribute should have unique names, which would not be the case if Functions were direct scope members.

The scope attribute of an OverloadedFunction is the same as the scope attribute of all Functions grouped by that OverloadedFunction.

Concrete metaobjects modelling OverloadedFunction must satisfy the following:

• The value of the MetaobjectCategory category attribute inherited from Metaobject is overloaded_function.

• range: Range {Function} overloads – A range of Function metaobjects reflecting the individual overloaded functions.

4.1.23. Template

Template is a NamedScoped and possibly a Class, ClassMember or Function reflecting a class or (member) function template. Note, that the Template concept slightly modifies the requirements of the Class and Function concepts.
Concrete metaobjects modelling `Template` must satisfy the following:

- The `is_template` trait inherited from `Metaobject` is true.
- `range`: `Range {TemplateParameter} template_parameters` – A range of `TemplateParameter` metaobjects reflecting the individual template parameters.
- `template`: `Instantiation instantiation(...)` – A template, instantiation of which should result in the instantiation of the template, reflected by this `Template`, with the specified parameters. The template parameters for the `instantiation` template should be the same as the parameters of the original template.

### 4.1.24. TemplateParameter

`TemplateParameter` is a either `Typedef` or `Constant` which reflects a type or non-type template parameter or parameter pack. The `category` tag should be used to distinguish between type and non-type (integral constant) template parameters. The `is_template` trait should be used to distinguish reflected template parameters from typedefs and constants.

`TemplateParameter` has these requirements:

- `trait`: `bool is_template` – This trait, inherited from `Metaobject`, should return true for template parameters.
- `attribute`: `size_t position` – The position of the template parameter.
- `attribute`: `bool pack` – Indicates whether the reflected parameter is a parameter pack of a variadic template.

### 4.1.25. Instantiation

`Instantiation` is a either `Class`, `ClassMember` or `Function` that reflects an instantiation of a class or (member) function template.
Concrete metaobjects modelling *Instantiation* must satisfy the following:

- The *has_template* trait inherited from *Metaobject* is true.
- *attribute: Template template* – A *Template* metaobject reflecting the template that the class or function, reflected by this *Instantiation*, is an instantiation of.

### 4.1.26. *Enum*

*Enum* is a *Type* reflecting an enumeration type. The members of an *Enum* are only *Constant* metaobjects.

Concrete metaobjects modelling *Enum* must satisfy the following:

- The value of the *MetaobjectCategory* category attribute inherited from *Metaobject* is `enum`.
- *attribute: Type base_type* – A *Type* reflecting the underlying type of the enumeration type.

### 4.1.27. *EnumClass*

*EnumClass* is a *Type* and *Scope* which reflects a strongly typed enumeration. The members of an *EnumClass* are only *Constant* metaobjects.
EnumClass has these requirements:

- The value of the MetaobjectCategory category attribute inherited from Metaobject is enum_class.
- attribute: Type base_type – A Type reflecting the underlying type of the enumeration type.

4.1.28. Inheritance

Inheritance is a Metaobject which reflects a class inheritance.

Inheritance has these requirements:

- The value of the MetaobjectCategory category attribute inherited from Metaobject is inheritance.
- attribute: Specifier access_type – Specifier that reflects the inheritance access type specifier (private, protected, public).
- attribute: Specifier inheritance_type – Specifier that reflects the inheritance type specifier (virtual, non-virtual).
- attribute: Class base_class – A Class reflecting the base-class in the inheritance.
- attribute: Class derived_class – A Class reflecting the derived class in the inheritance.

4.1.29. Variable

Variable is a NamedScoped and possibly a ClassMember which reflects a variable defined in a namespace, class, function, etc.
Variable has these requirements:

- The value of the \textit{MetaobjectCategory category} attribute inherited from \textit{Metaobject} is \textit{variable}.
- \textit{attribute: Specifier storage\_class} – \textit{Specifier} that reflects the storage class specifier of the variable.
- \textit{attribute: Type type} – A \textit{Type} reflecting the type of the variable.

4.1.30. Parameter

\textit{Parameter} is a \textit{Variable} which reflects a function parameter or a parameter pack.

Parameter has these requirements:

- The value of the \textit{MetaobjectCategory category} attribute inherited from \textit{Metaobject} is \textit{parameter}.
- \textit{attribute: size\_t position} – The position of the parameter.
- \textit{attribute: bool pack} – Indicates whether the reflected parameter is a parameter pack.
- \textit{attribute: Function scope} – This attribute, inherited from \textit{Scoped}, returns the function that the reflected parameter belongs to.

4.1.31. Constant

\textit{Constant} is a \textit{Named} and possibly a \textit{NamedScoped} or \textit{ClassMember} which reflects a named compile-time constant values, like non-type template parameters and enumeration values.
Constant has these requirements:

- The value of the MetaobjectCategory category attribute inherited from Metaobject is constant.
- attribute: unspecified-constant-value value – The value of the reflected constant.

4.2. Concept transformation

The concepts described above need to be transformed into concrete C++ code in order to be usable to the programmer and there are several possible sets of transformation rules, with various advantages and disadvantages.

Please see the appendix A for concrete examples of several possible transformations.

Consider a Concept with the following requirements:

- instance: some_instance – Instance of a tag concept.
- attribute: Constant constant_attrib – Attribute specifying a compile-time constant value.
- attribute: Type type_attrib – Attribute specifying a base-level type.
- attribute: Concept concept_attrib – Attributes specifying a metaobject conforming to a concept.
- trait: bool some_trait – Boolean trait.
- element: Element element(position) – Unary random access getter for indexed attributes.
- template: some_template(X) – Class template which when instantiated results in a concrete type.
- function: result_type some_function(...) – N-ary function.

This logical concept could be transformed into usable C++ as:
• an anonymous structure with member typedefs, functions, constant values, etc.:

    // definition
    struct Concept {
        static constexpr size_t constant_attrib = /*...*/;
        typedef /*...*/ type_attrib;
        static constexpr AnotherConcept concept_attrib(void);
        static constexpr bool some_trait = /*...*/;
        static Element element(integral_constant<size_t, /*...*/>);
        template <typename X> struct some_template { /*...*/ };
        static result_type some_function(/*...*/);
    };

    // usage
    auto metaobject = /* use reflection to get an instance of Concept */
    size_t x = metaobject.constant_attrib;
    auto y = metaobject.concept_attrib();
    // etc.

• an anonymous type (with its own identity so that it could be used in function
  overloads or template specializations) without any internal structure, where the
  related attributes, etc. would be accessed by free functions or by the means of
  templates, for example:

    // by function
    // declared as
    constexpr size_t constant_attrib(Concept);
    constexpr AnotherConcept concept_attrib(Concept);
    //
    // used as
    auto metaobject = /* use reflection to get an instance of Concept */
    size_t x = constant_attrib(metaobject);
    AnotherConcept y = concept_attrib(metaobject);
    result_type some_function(Concept, /*...*/);

    // or by template
// defined as
template <typename Concept>
struct constant_attr
{
    static constexpr size_t value = /*...*/;
};
template <typename Concept>
struct concept_attrib
{
    typedef AnotherConcept type;
};
template <typename Concept, typename X>
struct some_template
{
    /*...*/
};
template <typename Concept>
result_type some_function(/*...*/);

// used as
typedef /* get metaobject */ metaobject;
size_t x = constant_attrib<metaobject>::value;
typedef concept_attrib<metaobject>::type y;

The advantage of the first approach is that it does not "pollute" the std (nor any other namespace) with the getter functions or templates. Everything necessary to get the metadata from the metaobject is contained in the metaobject itself.

The advantage of the second approach is that the getters for a particular metaobject do not need to be defined all-in-one, which may simplify resolution of circular-dependencies.

4.2.1. Resolving conflicts with C++ keywords

In the concepts defined above, several requirements like instances, attributes, etc. have names which if transformed directly to C++ would conflict with C++ keywords.

There are several ways to resolve these conflicts:

- Changing the letter case or naming convention for example to ALL_CAPS, CamelCase, etc. – this approach is however inconsistent with the naming convention adopted by the standard library.

- Applying a prefix and/or postfix – the prefixes/suffixes can be meaningful or just underscores, for which there is a precedent in the standard - the placeholders.

- Changing the name to a synonym – this can however lead to confusion.
4.3. Suggested transformation

We suggest the following set of rules to transform the individual requirement kinds to C++ (see appendix A.1 for details).

Concrete metaobjects should be anonymous structures with the attributes, instances, elements, templates, etc. defined as members. If the name is in conflict with a C++ keyword it should be postfixed with an underscore, unless stated otherwise.

4.3.1. Instances of tag concepts

Tag concepts define a set of ”instances” tagging other metaobjects. The instances are intended to be used for metaobject classification and tag dispatching of function overloads and template specializations – i.e. they must have their own identity.

Tags can be either types or compile-time constant values, for example the values of a strongly typed enumeration.

We suggest that the instances of the tag concepts are transformed into tag types and in order to avoid conflicts with keywords they should be prefixed with meta_ (in case of MetaobjectCategory) and spec_ (in case of SpecifierCategory) and suffixed with _tag.

For example the namespace instance of MetaobjectCategory should be transformed into:

```cpp
namespace std {
  struct meta_namespace_tag { }; // namespace std
} // namespace std
```

and the static instance of SpecifierCategory should be transformed into:

```cpp
namespace std {
  struct spec_static_tag { }; // namespace std
} // namespace std
```

4.3.2. Traits

Concept traits should be transformed to templates similar to type_traits. For example the is_template trait of Metaobject should be transformed:

```cpp
struct _unspecified_instance_of_Metaobject { }; // _unspecified_instance_of_Metaobject
```

```cpp
template <typename Metaobject>
struct is_template
  : integral_constant<bool, true-or-false>
{ };
```
4.3.3. Constant value attributes

Attributes specifying a constant value should be transformed to static constexpr data members. For example the size attribute of String should be transformed into:

```cpp
struct _unspecified_instance_of_String
{
    static constexpr size_t size = /*...*/;
    /*...*/
};
```

4.3.4. Type attributes

Attributes specifying a (base-level not meta-level) type should be transformed to member typedefs. For example the original_type attribute of Type should be transformed to:

```cpp
struct _unspecified_instance_of_Type
{
    typedef unspecified-type original_type;
    /*...*/
};
```

4.3.5. Concept attributes

Attributes specifying a (meta-level) type conforming to a Concept should be transformed to static constexpr member functions. For example the scope attribute of Scoped should be transformed to:

```cpp
struct _unspecified_instance_of_Scoped
{
    static constexpr Scope scope(void);
    /*...*/
};
```

The template attribute of Instantiation that is in conflict with a reserved identifier should be transformed to:

```cpp
struct _unspecified_instance_of_Instantiation
{
    static constexpr Template template_(void);
    /*...*/
};
```

There are circular dependencies in the metaobjects and using functions can help to avoid problems when defining the metaobjects.
4.3.6. Range elements

Indexed attributes specifying the i-th element in a `Range` should be transformed to a set of static constexpr functions overloaded for every element in the range. The `integral_constant` template is used at the parameter distinguishing the overloads. For example the `at` element of a `Range` should be transformed to:

```cpp
struct _unspecified_instance_of_Range<Element>
{
    // overloaded for i in (0, 1, ..., N-1) where
    // N is the value of Range::size
    static constexpr Element at(integral_constant<size_t, i>);
    /*...*/
};
```

4.3.7. Templates

Class templates should be transformed to nested member templates. For example the `named_typedef` template in `NamedScoped` should be transformed to:

```cpp
struct _unspecified_instance_of_NamedScoped
{
    template <typename X>
    struct named_typedef
    {
        /*...*/
    };  /*...*/
};
```

4.3.8. Functions

Functions should be transformed to static member functions. For example the `call` function in `Function` should be transformed into:

```cpp
struct _unspecified_instance_of_Function
{
    static result_type call(/*same as the reflected function*/);
    /*...*/
};
```
4.4. Reflection

The metaobjects can be provided either via a set of overloaded functions or template classes similar to `type_traits` defined in the `std` namespace or by a new operator. Both of these approaches have advantages and disadvantages and both also depend on how the logical concepts are transformed to the C++ types generated by the compiler.

Another important aspect is, whether the metaobjects are returned as types or as objects. This is closely tied to how the logical concepts are transformed to concrete C++ which determines how the metaobjects are used.

4.4.1. Reflection functions

In this approach at least two functions should be defined in the `std` namespace:

- **unspecified-type** `reflected_global_scope(void);` (or alternatively `mirrored_global_scope()`)  
  This function should return a type conforming to the `GlobalScope` concept, reflecting the global scope. The real type of the result is not defined by the standard, i.e. it is an implementation detail. If the caller needs to store the result of this function the `auto` type specifier should always be used.

- **template <typename Type>**
  
  **unspecified-type** `reflected(void);` (or alternatively `mirrored<Type>()`)  
  This function should return a type conforming to the `Type` concept, reflecting the `Type` passed as template argument to this function. The real type of the result is not defined by the standard, i.e. it is an implementation detail. If the caller needs to store the result of this function the `auto` type specifier should always be used.

Several other similar functions could be added to the list above for reflection of templates, enumerated values, etc. without defining new rules for what regular function and template parameters can be. The advantages of using reflection functions are following:

- No need to add a new keyword to the language.
- Reduced chance of breaking existing code. The `reflected_global_scope()` and `reflected<Type>()` (or `mirrored_global_scope()` and `mirrored<Type>()`) functions are currently not defined in the `std` namespace and therefore should not clash with existing user code.

This approach has the following disadvantages:

- Less direct reflection. Using this approach it is not possible (at least without adding new rules for possible values of template and function parameters) to reflect constructors, overloaded functions and some other things.
4.4.2. Reflection operator

In this approach a new operator (we suggest the name \texttt{mirrored}(\texttt{param}) (or alternatively \texttt{reflected}(\texttt{param}) for additional alternatives see below) should be added. Depending on \texttt{param} – which could be a type name, namespace name, template name, overloaded function name, enumerated value name, etc. – the operator should return a Named metaobject reflecting the specified feature. If the parameter is omitted a type conforming to the \texttt{GlobalScope} metaobject concept should be returned. The exact types returned by the operators should be implementation details. For example:

```cpp
//
typedef integral_constant<size_t, 0> _0;
typedef integral_constant<size_t, 1> _1;
typedef integral_constant<size_t, 2> _2;
typedef integral_constant<size_t, 3> _3;
//
// reflect the global scope
// meta_gs conforms to the GlobalScope concept
mirrored() meta_gs;

static_assert(
    meta_gs.member_count() > 0,
    "The global scope has no members!"
);

static_assert(
    meta_gs.base_name().size() == 0,
    "Name of the global scope is not an empty string!"
);

//
// reflect the std namespace
// meta_std conforms to the Namespace concept
mirrored(std) meta_std;

static_assert(
    is_same<
        meta_gs,
        decltype(meta_std.scope())
    >::value,
    "Namespace std is not in the global scope!"
);
```
static_assert(
    meta_std.base_name().size() == 3,
    "Name of the std namespace does not have 3 characters!"
);

static_assert(
    meta_std.base_name().at(0) == 's',
    "Name of the std namespace does not start with 's'!"
);

assert(strcmp(meta_std.base_name().c_str(), "std") == 0);

//
// reflect the errno variable
// meta_errno conforms to the Variable concept
mirrored(errno) meta_errno;

//
// reflect the int type
// meta_int conforms to the Type concept
mirrored(int) meta_int;

//
// reflect the std::string typedef
// meta_std_string conforms to the Typedef concept
mirrored(std::string) meta_std_string;

//
// reflect the std::map template
// meta_std_map conforms to the Template
// and Class concepts
mirrored(std::map) meta_std_map;

//
// reflect the std::map<int, std::string> type
// meta_std_map_int_std_string conforms to Class
// and Instantiation concepts
mirrored(std::map<int, std::string>)
    meta_std_map_int_std_string;

//
// reflect the std::string’s (overloaded) constructors
// meta_std_string_string conforms to
// the OverloadedFunction concept and the individual // overloads that it allows to traverse conform // to the Constructor concept
mirrored(std::string::string) meta_std_string_string;

// reflect the std::string's copy constructor
// meta_std_string_string_copy conforms to // the Constructor concept
mirrored(std::string::string(const std::string&))
    meta_std_string_string_copy;

// reflect the std::swap overloaded free function
// meta_std_swap conforms to OverloadedFunction
mirrored(std::swap) meta_std_swap;

// reflect the (local) variable i
// meta_i conforms to Variable
int i = 42;
mirrored(i) meta_i;

Alternatively the mirrored keyword could return a value (instead of a type) and would be used as in the following examples:

// overloaded constructors of std::string
auto meta_std_string_string = mirrored(std::string::string);
// the first constructor
auto meta_std_string_string_0 = meta_std_string_string.overloads().at(0);

Using a new operator has the following advantages:

- More direct reflection. Even features that could not be reflected by using a (tem- plated) function could be reflected with an operator.
- More consistent reflection. Everything is reflected with a single operator.

and these disadvantages:

- Requires a new keyword or the usage of an existing keyword in a new context or the usage of a character sequence that is currently invalid.
- Increased risk of breaking existing code. Could be resolved by using an existing operator like %, |, etc., or the use of a currently invalid character or character sequence like @, $ or the usage of a new set of quotations like ' (backtick character).
For example:
5. Impact On the Standard

The impact on the standard and the existing applications depends mainly on the method of reflection (functions vs. operators). Reflection functions pose a very small risk of breaking existing standard-conforming code. The `%` operator on the other hand has a considerable potential of breaking existing applications. This can be alleviated by using existing keywords like `%` as suggested above.

Since compilers already have all the metadata required to generate the proposed metaob-
jects, making them available to programmers should not pose a big problem to the compiler vendors.

TODO: to be revised/completed

6. Implementation hints

6.1. Generation of metaobjects

The metaobjects should be generated / instantiated by the compiler only when explicitly requested. This also applies to members of the metaobjects. For example when a Namespace reflecting the std namespace is generated the individual member(...) functions (and the resulting metaobjects) should not be generated automatically unless the Scope::member(...) function is called or its type queried (by decltype or otherwise).

This should probably improve the compilation times and avoid reflection-related overhead when reflection is not used.

7. Unresolved Issues

- **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 Named::base_name() and NamedScoped::full_name():** The strings returned by the base_name and full_name functions 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.
8. Acknowledgements

9 References


A. Transforming the concepts to C++

A.1. Concept models – variant 1 (preferred)

A.1.1. String

```cpp
struct String
{
    static constexpr size_t size;

    static constexpr const char* c_str;
};
```

A.1.2. Range

```cpp
template <typename Element>
struct Range
{
    static constexpr size_t size;

    static constexpr Element at(
```
integral_constant<size_t, unspecified> position

A.1.3. MetaobjectCategory

This concept has the following instances:

struct meta_namespace_tag { }
struct meta_global_scope_tag { }
struct meta_type_tag { }
struct meta_typedef_tag { }
struct meta_class_tag { }
struct meta_function_tag { }
struct meta_constructor_tag { }
struct meta_operator_tag { }
struct meta_overloaded_function_tag { }
struct meta_enum_tag { }
struct meta_enum_class_tag { }
struct meta_inheritance_tag { }
struct meta_constant_tag { }
struct meta_variable_tag { }
struct meta_parameter_tag { }

A.1.4. SpecifierCategory

This concept has the following instances:

struct spec_none_tag { }
struct specExtern_tag { }
struct spec_static_tag { }
struct spec_mutable_tag { }
struct spec_register_tag { }
struct spec_thread_local_tag { }
struct spec_const_tag { }
struct spec_virtual_tag { }
struct spec_private_tag { }
struct spec_protected_tag { }
struct spec_public_tag { }
struct spec_class_tag { }
struct spec_struct_tag { }
struct spec_union_tag { }
struct spec_enum_tag { }
struct spec_enum_class_tag {};  
struct specconstexpr_tag {};  

A.1.5. Metaobject  

struct Metaobject  
{  
};  

template <>  
struct category<Metaobject>  
{  
    typedef MetaobjectCategory type;  
};  

template <>  
struct is_metaobject<Metaobject>  
: integral_constant<bool, true>  
{ };  

template <>  
struct has_name<Metaobject>  
: integral_constant<bool, false>  
{ };  

template <>  
struct has_scope<Metaobject>  
: integral_constant<bool, false>  
{ };  

template <>  
struct is_scope<Metaobject>  
: integral_constant<bool, false>  
{ };  

template <>  
struct is_class_member<Metaobject>  
: integral_constant<bool, false>  
{ };  

template <>  
struct has_template<Metaobject>  
: integral_constant<bool, false>
struct Specifier
{
    static constexpr String keyword(void);
};

template <>
struct category<Specifier>
{
    typedef SpecifierCategory type;
};

A.1.7. Named

struct Named
{
    static constexpr String base_name(void);
};

template <>
struct has_name<Named>
: integral_constant<bool, true>
{  
};

A.1.8. Scoped

struct Scoped
{
    static constexpr Scope scope(void);
};

template <>
struct has_scope<Scoped>
A.1.9. NamedScoped

```cpp
struct NamedScoped
{
    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    static constexpr String full_name(void);

    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };
};
```

A.1.10. Scope

```cpp
struct Scope
{
    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
```
template <typename X>
struct named_typedef
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var
{
    typedef unspecified type;
};

static constexpr Range<Scoped> members(void);

template <>
struct is_scope<Scope> : integral_constant<bool, true> { };

A.1.11. Namespace

struct Namespace
{
    typedef meta_namespace_tag category;
    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };
// inherited from NamedScoped
template <typename X>
struct named_mem_var
{
    typedef unspecified type;
};

// inherited from Scope
static constexpr Range<Scoped> members(void);

A.1.12. GlobalScope

struct GlobalScope
{
    typedef meta_global_scope_tag category;
    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // inherited from Scope
    static constexpr Range<Scoped>
A.1.13. Type

struct Type
{
    typedef meta_type_tag category;

    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    }

    typedef original-type original_type;
};

A.1.14. Typedef

struct Typedef
{
    typedef meta_typedef_tag category;

// inherited from Named
static constexpr String base_name(void);

// inherited from Scoped
static constexpr Scope scope(void);

// inherited from NamedScoped
static constexpr String full_name(void);

// inherited from NamedScoped
template <typename X>
struct named_typedef
{
  typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var
{
  typedef unspecified type;
};

// inherited from Type
typedef original-type original_type;

static constexpr Type typedef_type(void);
};

A.1.15. Class

struct Class
{
  typedef meta_class_tag category;

  // inherited from Named
  static constexpr String base_name(void);

  // inherited from Scoped
  static constexpr Scope scope(void);

  // inherited from NamedScoped
  static constexpr String full_name(void);
// inherited from NamedScoped
template <typename X>
struct named_typedef
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var
{
    typedef unspecified type;
};

// inherited from Type
typedef original-type original_type;

// inherited from Scope
static constexpr Range<Scoped> members(void);

static constexpr Specifier elaborated_type(void);

static constexpr Range<Inheritance> base_classes(void);

A.1.16. Function

struct Function
{
    typedef meta_function_tag category;

    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);
// inherited from NamedScoped
template <typename X>
struct named_typedef
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var
{
    typedef unspecified type;
};

// inherited from Scope
static constexpr Range<Scoped> members(void);

static constexpr Specifier linkage(void);

static constexpr bool constexpr_;

static constexpr Type result_type(void);

static constexpr Range<Parameter> parameters(void);

static constexpr bool noexcept_;

static constexpr Range<Type> exceptions(void);

static constexpr Specifier constness(void);

static constexpr bool pure;

static unspecified call(...);}

A.1.17. ClassMember

struct ClassMember
{
    // inherited from Named

static constexpr String base_name(void);

// inherited from Scoped
static constexpr Scope scope(void);

// inherited from NamedScoped
static constexpr String full_name(void);

// inherited from NamedScoped
template <typename X>
struct named_typedef
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var
{
    typedef unspecified type;
};

static constexpr Specifier access_type(void);

A.1.18. Initializer

struct Initializer
{
    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);
};
// inherited from NamedScoped
template <typename X>
struct named_typedef
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var
{
    typedef unspecified var;
};

// inherited from Scope
static constexpr Range<Scoped> members(void);

// inherited from Function
static constexpr Specifier linkage(void);

// inherited from Function
static constexpr bool constexpr_;

// inherited from Function
static constexpr Type result_type(void);

// inherited from Function
static constexpr Range<Parameter> parameters(void);

// inherited from Function
static constexpr bool noexcept_;

// inherited from Function
static constexpr Range<Type> exceptions(void);

// inherited from Function
static constexpr Specifier constness(void);

// inherited from Function
static constexpr bool pure;
A.1.19. Constructor

struct Constructor
{
    typedef meta_constructor_tag category;
    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // inherited from ClassMember
    static constexpr Specifier access_type(void);

    // inherited from Scope
    static constexpr Range<Scoped> members(void);

    // inherited from Function
    static constexpr Specifier linkage(void);

    // inherited from Function
static constexpr bool constexpr_;  
// inherited from Function
static constexpr Type result_type(void);  
// inherited from Function
static constexpr Range<Parameter> parameters(void);  
// inherited from Function
static constexpr bool noexcept_;  
// inherited from Function
static constexpr Range<Type> exceptions(void);  
// inherited from Function
static constexpr Specifier constness(void);  
// inherited from Function
static constexpr bool pure;  
// inherited from Function
static unspecified call(...);
};

A.1.20. Operator

struct Operator
{
    typedef meta_operator_tag category;  
    // inherited from Named
    static constexpr String base_name(void);
    // inherited from Scoped
    static constexpr Scope scope(void);
    // inherited from NamedScoped
    static constexpr String full_name(void);
    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
{  typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var
{  
    typedef unspecified type;
};

// inherited from Scope
static constexpr Range<Scoped> members(void);

// inherited from Function
static constexpr Specifier linkage(void);

// inherited from Function
static constexpr bool constexpr_;

// inherited from Function
static constexpr Type result_type(void);

// inherited from Function
static constexpr Range<Parameter> parameters(void);

// inherited from Function
static constexpr bool noexcept_

// inherited from Function
static constexpr Range<Type> exceptions(void);

// inherited from Function
static constexpr Specifier constness(void);

// inherited from Function
static constexpr bool pure;

// inherited from Function
static unspecified call(...);

// possibly inherited from ClassMember
A.1.21. **OverloadedFunction**

```cpp
struct OverloadedFunction
{
    typedef meta_overloaded_function_tag category;

    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // possibly inherited from ClassMember
    static constexpr Specifier access_type(void);

    static constexpr Range<Function> 
    overloads(void);
};
```
A.1.22. Template

```cpp
template struct Template
{
    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // possibly inherited from Type
    typedef original-type original_type;

    // possibly inherited from Scope
    static constexpr Range<Scoped> members(void);

    // possibly inherited from Class
    static constexpr Specifier elaborated_type(void);

    // possibly inherited from Class
    static constexpr Range<Inheritance> base_classes(void);

    // possibly inherited from ClassMember
    static constexpr Specifier access_type(void);
};
```
// possibly inherited from Function
static constexpr Specifier linkage(void);

// possibly inherited from Function
static constexpr bool constexpr_

// possibly inherited from Function
static constexpr Type result_type(void);

// possibly inherited from Function
static constexpr Range<Parameter> parameters(void);

// possibly inherited from Function
static constexpr bool noexcept_

// possibly inherited from Function
static constexpr Range<Type> exceptions(void);

// possibly inherited from Function
static constexpr Specifier constness(void);

// possibly inherited from Function
static constexpr bool pure;

// possibly inherited from Function
static unspecified call(...);
static constexpr Range<TemplateParameter> template_parameters(void);

template <...>
struct instantiation
{
    typedef Instantiation type;
};

template <>
struct is_template<Template>
: integral_constant<bool, true>
{
};
A.1.23. TemplateParameter

```cpp
struct TemplateParameter
{
    // possibly inherited from Named
    static constexpr String base_name(void);

    // possibly inherited from Scoped
    static constexpr Scope scope(void);

    // possibly inherited from NamedScoped
    static constexpr String full_name(void);

    // possibly inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // possibly inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // possibly inherited from Type
    typedef original-type original_type;

    // possibly inherited from Typedef
    static constexpr Type typedef_type(void);

    // possibly inherited from ClassMember
    static constexpr Specifier access_type(void);

    // possibly inherited from Constant
    static constexpr unspecified-constant-value value;

    static constexpr size_t position;

    static constexpr bool pack;
};
```
template <>
struct is_template<TemplateParameter>
    : integral_constant<bool, true>
{ }; 

A.1.24. Instantiation 

struct Instantiation
{
    // possibly inherited from Named
    static constexpr String base_name(void);

    // possibly inherited from Scoped
    static constexpr Scope scope(void);

    // possibly inherited from NamedScoped
    static constexpr String full_name(void);

    // possibly inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // possibly inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // possibly inherited from Type
    typedef original-type original_type;

    // possibly inherited from Scope
    static constexpr Range<Scoped> members(void);

    // possibly inherited from Class
    static constexpr Specifier elaborated_type(void);
/possibly inherited from Class
static constexpr Range<Inheritance> base_classes(void);

// possibly inherited from ClassMember
static constexpr Specifier access_type(void);

// possibly inherited from Function
static constexpr Specifier linkage(void);

// possibly inherited from Function
static constexpr bool constexpr_

// possibly inherited from Function
static constexpr Type result_type(void);

// possibly inherited from Function
static constexpr Range<Parameter> parameters(void);

// possibly inherited from Function
static constexpr bool noexcept_

// possibly inherited from Function
static constexpr Range<Type> exceptions(void);

// possibly inherited from Function
static constexpr Specifier constness(void);

// possibly inherited from Function
static constexpr bool pure;

// possibly inherited from Function
static unspecified call(...);
static constexpr Template template_(void);

};

#include<iostream>
#include<assert.h>

int main()
{ return 0; }


A.1.25. Enum

```cpp
struct Enum
{
    typedef meta_enum_tag category;

    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // inherited from Type
    typedef original_type original_type;

    static constexpr Type base_type(void);
};
```

A.1.26. EnumClass

```cpp
struct EnumClass
{
    typedef meta_enum_class_tag category;

    // inherited from Named
    static constexpr String base_name(void);
};
```
// inherited from Scoped
class static constexpr Scope scope(void);

// inherited from NamedScoped
class static constexpr String full_name(void);

// inherited from NamedScoped
template <typename X>
class struct named_typedef
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
class struct named_mem_var
{
    typedef unspecified type;
};

// inherited from Type
typedef original_type original_type;

// inherited from Scope
class static constexpr Range<Scoped>
members(void);

    static constexpr Type base_type(void);
};

A.1.27. Inheritance

class struct Inheritance
{
    typedef meta_inheritance_tag category;

    static constexpr Specifier access_type(void);

    static constexpr Specifier inheritance_type(void);

    static constexpr Class base_class(void);
}
static constexpr Class derived_class(void);

A.1.28. Variable

struct Variable
{
    typedef meta_variable_tag category;

    // inherited from Named
    static constexpr String base_name(void);

    // inherited from Scoped
    static constexpr Scope scope(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template<typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template<typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // possibly inherited from ClassMember
    static constexpr Specifier access_type(void);

    static constexpr Specifier storage_class(void);

    static constexpr Type type(void);
};
A.1.29. Parameter

```cpp
struct Parameter
{
    typedef meta_parameter_tag category;

    // inherited from Named
    static constexpr String base_name(void);

    // inherited from NamedScoped
    static constexpr String full_name(void);

    // inherited from NamedScoped
    template <typename X>
    struct named_typedef
    {
        typedef unspecified type;
    };

    // inherited from NamedScoped
    template <typename X>
    struct named_mem_var
    {
        typedef unspecified type;
    };

    // possibly inherited from ClassMember
    static constexpr Specifier access_type(void);

    // inherited from Variable
    static constexpr Specifier storage_class(void);

    // inherited from Variable
    static constexpr Type type(void);

    static constexpr size_t position;

    static constexpr bool pack;

    static constexpr Function scope(void);
};
```
A.1.30. Constant

```cpp
struct Constant {
    typedef meta_constant_tag category;

    // inherited from Named
    static constexpr String base_name(void);

    // possibly inherited from Scoped
    static constexpr Scope scope(void);

    // possibly inherited from NamedScoped
    static constexpr String full_name(void);

    // possibly inherited from NamedScoped
    template <typename X> static constexpr String full_name(void);

    // possibly inherited from NamedScoped
    template <typename X> static constexpr String full_name(void);

    // possibly inherited from NamedScoped
    template <typename X> static constexpr String full_name(void);

    // possibly inherited from ClassMember
    static constexpr Specifier access_type(void);

    static constexpr unspecified-constant-value value;
};
```

A.2. Concept models – variant 2 (alternative)

A.2.1. String

```cpp
struct String { };

template <>
struct size<String>
: integral_constant<size_t, unspecified> 
{ }; 

template <> 
struct c_str<String> 
{ 
    static constexpr const char* value; 
}; 

A.2.2. Range 

template <typename Element> 
struct Range { }; 

template <typename Element> 
struct size<Range<Element>> 
: integral_constant<size_t, unspecified> 
{ }; 

template <typename Element, size_t Index> 
struct at<Range<Element>, integral_constant<size_t, Index>> 
{ 
    typedef Element type; 
}; 

A.2.3. MetaobjectCategory 

This concept has the following instances: 

struct meta_namespace_tag { }; 
struct meta_global_scope_tag { }; 
struct meta_type_tag { }; 
struct meta_typedef_tag { }; 
struct meta_class_tag { }; 
struct meta_function_tag { }; 
struct meta_constructor_tag { }; 
struct meta_operator_tag { }; 
struct meta_overloaded_function_tag { }; 
struct meta_enum_tag { }; 
struct meta_enum_class_tag { }; 
struct meta_inheritance_tag { }; 
struct meta_constant_tag { };
struct meta_variable_tag { };  
struct meta_parameter_tag { };  

A.2.4. SpecifierCategory  

This concept has the following instances:  
struct spec_none_tag { };  
struct specExtern_tag { };  
struct specStatic_tag { };  
struct specMutable_tag { };  
struct specRegister_tag { };  
struct specThread_local_tag { };  
struct specConst_tag { };  
struct specVirtual_tag { };  
struct specPrivate_tag { };  
struct specProtected_tag { };  
struct specPublic_tag { };  
struct specClass_tag { };  
struct specStruct_tag { };  
struct specUnion_tag { };  
struct specEnum_tag { };  
struct specEnum_class_tag { };  
struct specConstexpr_tag { };  

A.2.5. Metaobject  

struct Metaobject { };  

template <>  
struct category<Metaobject>  
{  
typedef MetaobjectCategory type;  
};  

template <>  
struct is_metaobject<Metaobject>  
: integral_constant<bool, true>  
{ };  

template <>  
struct has_name<Metaobject>
A.2.6. Specifier

struct Specifier { };
A.2.7. Named

```cpp
struct Named {};

template <>
struct base_name<Named>
{
    typedef String type;
};

template <>
struct has_name<Named>
    : integral_constant<bool, true>
{};
```

A.2.8. Scoped

```cpp
struct Scoped {};

template <>
struct scope<Scoped>
{
    typedef Scope type;
};

template <>
struct has_scope<Scoped>
    : integral_constant<bool, true>
{};
```

A.2.9. NamedScoped

```cpp
struct NamedScoped {};

    // inherited from Named
    template <>
    struct base_name<NamedScoped>
    {
        typedef String type;
    };

    // inherited from Scoped
```
template <>
struct scope<NamedScoped>
{
    typedef Scope type;
};

template <>
struct full_name<NamedScoped>
{
    typedef String type;
};

template <typename X>
struct named_typedef<NamedScoped, X>
{
    typedef unspecified type;
};

template <typename X>
struct named_mem_var<NamedScoped, X>
{
    typedef unspecified type;
};

A.2.10. Scope

struct Scope { };

// inherited from Named
template <>
struct base_name<Scope>
{
    typedef String type;
};

// inherited from Scoped
template <>
struct scope<Scope>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Scope>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Scope, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Scope, X>
{
    typedef unspecified type;
};

template <>
struct members<Scope>
{
    typedef Range<Scoped> type;
};

template <>
struct is_scope<Scope>
    : integral_constant<bool, true>
{
};

A.2.11. Namespace

struct Namespace { };
{  
typedef String type;
};

// inherited from Scoped
template <>
struct scope<Namespace>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Namespace>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Namespace, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Namespace, X>
{
    typedef unspecified type;
};

// inherited from Scope
template <>
struct members<Namespace>
{
    typedef Range<Scoped> type;
};

A.2.12. GlobalScope

struct GlobalScope { };
template <typename X>
struct category<GlobalScope>
{
  typedef meta_global_scope_tag type;
};
// inherited from Named

template <>
struct base_name<GlobalScope>
{
  typedef String type;
};
// inherited from Scoped

template <>
struct scope<GlobalScope>
{
  typedef Scope type;
};
// inherited from NamedScoped

template <typename X>
struct full_name<GlobalScope, X>
{
  typedef String type;
};
// inherited from NamedScoped

template <typename X>
struct named_typedef<GlobalScope, X>
{
  typedef unspecified type;
};
// inherited from NamedScoped

template <typename X>
struct named_mem_var<GlobalScope, X>
{
  typedef unspecified type;
};
// inherited from Scope

template <>
struct members<GlobalScope>
{


```cpp
typedef Range<Scoped> type;
};

A.2.13. Type

struct Type { }
;

template typename <>
struct category<Type>
{
    typedef meta_type_tag type;
};

    // inherited from Named
template <>
struct base_name<Type>
{
    typedef String type;
};

    // inherited from Scoped
template <>
struct scope<Type>
{
    typedef Scope type;
};

    // inherited from NamedScoped
template <>
struct full_name<Type>
{
    typedef String type;
};

    // inherited from NamedScoped
template <typename X>
struct named_typedef<Type, X>
{
    typedef unspecified type;
};

    // inherited from NamedScoped
template <typename X>
```
struct named_mem_var<Type, X>
{
    typedef unspecified type;
};

template <>
struct original_type<Type>
{
    typedef original-type type;
};

A.2.14. Typedef

struct Typedef { };  

template typename <>
struct category<Typedef>
{
    typedef meta_typedef_tag type;
};

    // inherited from Named
template <>
struct base_name<Typedef>
{
    typedef String type;
};

    // inherited from Scoped
template <>
struct scope<Typedef>
{
    typedef Scope type;
};

    // inherited from NamedScoped
template <>
struct full_name<Typedef>
{
    typedef String type;
};

    // inherited from NamedScoped
template <typename X>
struct named_typedef<Typedef, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Typedef, X>
{
    typedef unspecified type;
};

// inherited from Type
template <>
struct original_type<Typedef>
{
    typedef original-type type;
};

template <>
struct typedef_type<Typedef>
{
    typedef Type type;
};

A.2.15. Class

struct Class { };

template typename <>
struct category<Class>
{
    typedef meta_class_tag type;
};

    // inherited from Named
    template <>
    struct base_name<Class>
    {
        typedef String type;
    };

83
// inherited from Scoped
template <>
struct scope<Class>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Class>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Class, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Class, X>
{
    typedef unspecified type;
};

// inherited from Type
template <>
struct original_type<Class>
{
    typedef original-type type;
};

// inherited from Scope
template <>
struct members<Class>
{
    typedef Range<Scoped> type;
};

template <>
struct elaborated_type<Class>
{  
typedef Specifier type;
};

template <>
struct base_classes<Class>
{
  typedef Range<Inheritance> type;
};

A.2.16. Function

struct Function { };

template typename <>
struct category<Function>
{
  typedef meta_function_tag type;
};

    // inherited from Named
    template <>
    struct base_name<Function>
    {
      typedef String type;
    }

    // inherited from Scoped
    template <>
    struct scope<Function>
    {
      typedef Scope type;
    }

    // inherited from NamedScoped
    template <>
    struct full_name<Function>
    {
      typedef String type;
    }

    // inherited from NamedScoped
    template <typename X>
struct named_typedef<Function, X>
{
  typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Function, X>
{
  typedef unspecified type;
};

// inherited from Scope
template <>
struct members<Function>
{
  typedef Range<Scoped> type;
};

template <>
struct linkage<Function>
{
  typedef Specifier type;
};

template <>
struct constexpr_<Function>
: integral_constant<bool, >
{
};

template <>
struct result_type<Function>
{
  typedef Type type;
};

template <>
struct parameters<Function>
{
  typedef Range<Parameter> type;
};

template <>
struct noexcept_<Function>
: integral_constant<bool, >
{
};

template <>
struct exceptions<Function>
{
    typedef Range<Type> type;
};

template <>
struct constness<Function>
{
    typedef Specifier type;
};

template <>
struct pure<Function>
: integral_constant<bool, >
{
};

template <>
struct call<Function>
{
    static unspecified apply(...);
};

A.2.17. ClassMember

struct ClassMember { };

    // inherited from Named
template <>
struct base_name<ClassMember>
{
    typedef String type;
};

    // inherited from Scoped
template <>
struct scope<ClassMember>
{
    typedef Scope type;
};
// inherited from NamedScoped
template <>
struct full_name<ClassMember>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<ClassMember, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<ClassMember, X>
{
    typedef unspecified type;
};

template <>
struct access_type<ClassMember>
{
    typedef Specifier type;
};

template <>
struct is_class_member<ClassMember>
    : integral_constant<bool, true>
{};

A.2.18. Initializer

struct Initializer { };
template <>
struct scope<Initializer>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Initializer>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Initializer, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Initializer, X>
{
    typedef unspecified type;
};

// inherited from Scope
template <>
struct members<Initializer>
{
    typedef Range<Scoped> type;
};

// inherited from Function
template <>
struct linkage<Initializer>
{
    typedef Specifier type;
};
```cpp
#include <iostream>

// Inherited from Function
template <>
struct constexpr_<Initializer>
: integral_constant<bool, true>
{
};

// Inherited from Function
template <>
struct result_type<Initializer>
{
    typedef Type type;
};

// Inherited from Function
template <>
struct parameters<Initializer>
{
    typedef Range<Parameter> type;
};

// Inherited from Function
template <>
struct noexcept_<Initializer>
: integral_constant<bool, true>
{
};

// Inherited from Function
template <>
struct exceptions<Initializer>
{
    typedef Range<Type> type;
};

// Inherited from Function
template <>
struct constness<Initializer>
{
    typedef Specifier type;
};

// Inherited from Function
template <>
struct pure<Initializer>
: integral_constant<bool, true>
{
};
```
// inherited from Function
template <>
struct call<Initializer>
{
    static unspecified apply(...);
};

A.2.19. Constructor

struct Constructor { };

template typename <>
struct category<Constructor>
{
    typedef meta_constructor_tag type;
};

// inherited from Named
template <>
struct base_name<Constructor>
{
    typedef String type;
};

// inherited from Scoped
template <>
struct scope<Constructor>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Constructor>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Constructor, X>
{
    typedef unspecified type;
};

    // inherited from NamedScoped
template <typename X>
struct named_mem_var<Constructor, X>
{
    typedef unspecified type;
};

    // inherited from ClassMember
template <>
struct access_type<Constructor>
{
    typedef Specifier type;
};

    // inherited from Scope
template <>
struct members<Constructor>
{
    typedef Range<Scoped> type;
};

    // inherited from Function
template <>
struct linkage<Constructor>
{
    typedef Specifier type;
};

    // inherited from Function
template <>
struct constexpr_<Constructor>
: integral_constant<bool, >
{
};

    // inherited from Function
template <>
struct result_type<Constructor>
{
    typedef Type type;
};

    // inherited from Function
A.2.20. Operator

struct Operator {};
template typename <>
struct category<Operator>
{
    typedef meta_operator_tag type;
};
// inherited from Named
template <>
struct base_name<Operator>
{
    typedef String type;
};

// inherited from Scoped
template <>
struct scope<Operator>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Operator>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Operator, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Operator, X>
{
    typedef unspecified type;
};

// inherited from Scope
template <>
struct members<Operator>
{
typedef Range<Scoped> type;

// inherited from Function
template <>
struct linkage<Operator>
{
    typedef Specifier type;
};

// inherited from Function
template <>
struct constexpr_<Operator>
: integral_constant<bool, >
{ }; // inherited from Function

template <>
struct result_type<Operator>
{
    typedef Type type;
};

// inherited from Function
template <>
struct parameters<Operator>
{
    typedef Range<Parameter> type;
};

// inherited from Function
template <>
struct noexcept_<Operator>
: integral_constant<bool, >
{ }; // inherited from Function

// inherited from Function
template <>
struct exceptions<Operator>
{ // inherited from Function
    typedef Range<Type> type;
};

// inherited from Function

```cpp
template <>
struct constness<Operator>
{
    typedef Specifier type;
};

    // inherited from Function
template <>
struct pure<Operator>
    : integral_constant<bool, >
{ };
// inherited from Scoped
template <>
struct scope<OverloadedFunction>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<OverloadedFunction>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<OverloadedFunction, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<OverloadedFunction, X>
{
    typedef unspecified type;
};

// possibly inherited from ClassMember
template <>
struct access_type<OverloadedFunction>
{
    typedef Specifier type;
};

template <>
struct overloads<OverloadedFunction>
{
    typedef Range<Function> type;
};
A.2.22. Template

```cpp
struct Template {};

// inherited from Named
template <>
struct base_name<Template>
{
    typedef String type;
};

// inherited from Scoped
template <>
struct scope<Template>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Template>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Template, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Template, X>
{
    typedef unspecified type;
};

// possibly inherited from Type
template <>
struct original_type<Template>
{
```
typedef original-type type;
};

// possibly inherited from Scope
template <>
struct members<Template>
{
    typedef Range<Scoped> type;
};

// possibly inherited from Class
template <>
struct elaborated_type<Template>
{
    typedef Specifier type;
};

// possibly inherited from Class
template <>
struct base_classes<Template>
{
    typedef Range<Inheritance> type;
};

// possibly inherited from ClassMember
template <>
struct access_type<Template>
{
    typedef Specifier type;
};

// possibly inherited from Function
template <>
struct linkage<Template>
{
    typedef Specifier type;
};

// possibly inherited from Function
template <>
struct constexpr_<Template>
: integral_constant<bool, >
{
};
// possibly inherited from Function
template <>
struct result_type<Template>
{
    typedef Type type;
};

// possibly inherited from Function
template <>
struct parameters<Template>
{
    typedef Range<Parameter> type;
};

// possibly inherited from Function
template <>
struct noexcept_<Template>
    : integral_constant<bool, >
{
};

// possibly inherited from Function
template <>
struct exceptions<Template>
{
    typedef Range<Type> type;
};

// possibly inherited from Function
template <>
struct constness<Template>
{
    typedef Specifier type;
};

// possibly inherited from Function
template <>
struct pure<Template>
    : integral_constant<bool, >
{
};

// possibly inherited from Function
template <>
struct call<Template>
{
N3996- Static reflection

    static unspecified apply(...);
};

template <>
struct template_parameters<Template>
{
    typedef Range<TemplateParameter> type;
};

template <...>
struct instantiation<Template, ...>
{
    typedef Instantiation type;
};

template <>
struct is_template<Template>
    : integral_constant<bool, true>
{ };

A.2.23. TemplateParameter

struct TemplateParameter { };  // possibly inherited from Named

template <>
struct base_name<TemplateParameter>
{
    typedef String type;
};  // possibly inherited from Scoped

template <>
struct scope<TemplateParameter>
{
    typedef Scope type;
};  // possibly inherited from NamedScoped

    typedef String type;
};

// possibly inherited from NamedScoped
template <typename X>
struct named_typedef<TemplateParameter, X>
{
    typedef unspecified type;
};

// possibly inherited from NamedScoped
template <typename X>
struct named_mem_var<TemplateParameter, X>
{
    typedef unspecified type;
};

// possibly inherited from Type
template <>
struct original_type<TemplateParameter>
{
    typedef original-type type;
};

// possibly inherited from Typedef
template <>
struct typedef_type<TemplateParameter>
{
    typedef Type type;
};

// possibly inherited from ClassMember
template <>
struct access_type<TemplateParameter>
{
    typedef Specifier type;
};

// possibly inherited from Constant
template <>
struct value<TemplateParameter>
{
    static constexpr unspecified-constant-value value;
};
template <>
struct position<TemplateParameter>
  : integral_constant<size_t, >
{ }; 

template <>
struct pack<TemplateParameter>
  : integral_constant<bool, >
{ }; 

template <>
struct is_template<TemplateParameter>
  : integral_constant<bool, true>
{ }; 

A.2.24. Instantiation

struct Instantiation { }; 

// possibly inherited from Named
template <>
struct base_name<Instantiation>
{ 
  typedef String type;
};

// possibly inherited from Scoped
template <>
struct scope<Instantiation>
{ 
  typedef Scope type;
};

// possibly inherited from NamedScoped
template <>
struct full_name<Instantiation>
{ 
  typedef String type;
};

// possibly inherited from NamedScoped
template <typename X>
struct named_typedef<Instantiation, X>
{  
    typedef unspecified type;
};

// possibly inherited from NamedScoped
template <typename X>
struct named_mem_var<Instantiation, X>
{
    typedef unspecified type;
};

// possibly inherited from Type
template <>
struct original_type<Instantiation>
{
    typedef original-type type;
};

// possibly inherited from Scope
template <>
struct members<Instantiation>
{
    typedef Range<Scoped> type;
};

// possibly inherited from Class
template <>
struct elaborated_type<Instantiation>
{
    typedef Specifier type;
};

// possibly inherited from Class
template <>
struct base_classes<Instantiation>
{
    typedef Range<Inheritance> type;
};

// possibly inherited from ClassMember
template <>
struct access_type<Instantiation>
{
    typedef Specifier type;


// possibly inherited from Function
template <>
struct linkage<Instantiation>
{
  typedef Specifier type;
};

// possibly inherited from Function
template <>
struct constexpr_<Instantiation>
  : integral_constant<bool, >
{
};

// possibly inherited from Function
template <>
struct result_type<Instantiation>
{
  typedef Type type;
};

// possibly inherited from Function
template <>
struct parameters<Instantiation>
{
  typedef Range<Parameter> type;
};

// possibly inherited from Function
template <>
struct noexcept_<Instantiation>
  : integral_constant<bool, >
{
};

// possibly inherited from Function
template <>
struct exceptions<Instantiation>
{
  typedef Range<Type> type;
};
struct constness<Instantiation>
{
    typedef Specifier type;
};

    // possibly inherited from Function
template <>
struct pure<Instantiation>
    : integral_constant<bool, >
{
};

    // possibly inherited from Function
template <>
struct call<Instantiation>
{
    static unspecified apply(...);
};

template <>
struct template_<Instantiation>
{
    typedef Template type;
};

template <>
struct has_template<Instantiation>
    : integral_constant<bool, true>
{
};

A.2.25. Enum

struct Enum { 
};

template typename <>
struct category<Enum>
{
    typedef meta_enum_tag type;
};

    // inherited from Named
template <>
struct base_name<Enum>
{

typedef String type;
;
// inherited from Scoped
template <>
struct scope<Enum>
{
  typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<Enum>
{
  typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Enum, X>
{
  typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Enum, X>
{
  typedef unspecified type;
};

// inherited from Type
template <>
struct original_type<Enum>
{
  typedef original-type type;
};

template <>
struct base_type<Enum>
{
  typedef Type type;
};
A.2.26. EnumClass

struct EnumClass { };

template typename <>
struct category<EnumClass>
{
    typedef meta_enum_class_tag type;
};

// inherited from Named
template <>
struct base_name<EnumClass>
{
    typedef String type;
};

// inherited from Scoped
template <>
struct scope<EnumClass>
{
    typedef Scope type;
};

// inherited from NamedScoped
template <>
struct full_name<EnumClass>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<EnumClass, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<EnumClass, X>
{
    typedef unspecified type;
}
// inherited from Type
template <>
struct original_type<EnumClass>
{
    typedef original_type type;
};

// inherited from Scope
template <>
struct members<EnumClass>
{
    typedef Range<Scoped> type;
};
template <>
struct base_type<EnumClass>
{
    typedef Type type;
};

A.2.27. Inheritance

struct Inheritance { };
template <>
struct base_class<Inheritance>
{
    typedef Class type;
};

template <>
struct derived_class<Inheritance>
{
    typedef Class type;
};

A.2.28. Variable

struct Variable { };  // inherited from Named

template typename <>
struct category<Variable>
{
    typedef meta_variable_tag type;
};

    // inherited from Scoped
template <>
struct base_name<Variable>
{
    typedef String type;
};

    // inherited from NamedScoped
template <>
struct full_name<Variable>
{
    typedef String type;
};
// inherited from NamedScoped
template <typename X>
struct named_typedef<Variable, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Variable, X>
{
    typedef unspecified type;
};

// possibly inherited from ClassMember
template <>
struct access_type<Variable>
{
    typedef Specifier type;
};

template <>
struct storage_class<Variable>
{
    typedef Specifier type;
};

template <>
struct type<Variable>
{
    typedef Type type;
};

A.2.29. Parameter

struct Parameter {
};

template typename <>
struct category<Parameter>
{
    typedef meta_parameter_tag type;
};
// inherited from Named
template <>
struct base_name<Parameter>
{
    typedef String type;
};

// inherited from NamedScoped
template <>
struct full_name<Parameter>
{
    typedef String type;
};

// inherited from NamedScoped
template <typename X>
struct named_typedef<Parameter, X>
{
    typedef unspecified type;
};

// inherited from NamedScoped
template <typename X>
struct named_mem_var<Parameter, X>
{
    typedef unspecified type;
};

// possibly inherited from ClassMember
template <>
struct access_type<Parameter>
{
    typedef Specifier type;
};

// inherited from Variable
template <>
struct storage_class<Parameter>
{
    typedef Specifier type;
};
template <>
struct type<Parameter>
{
    typedef Type type;
};

template <>
struct position<Parameter>
    : integral_constant<size_t, >
{ }; 

template <>
struct pack<Parameter>
{
    static constexpr bool value;
};

template <>
struct scope<Parameter>
{
    typedef Function type;
};

A.2.30. Constant

struct Constant { }; 

template typename <>
struct category<Constant>
{
    typedef meta_constant_tag type;
};

    // inherited from Named

template <>
struct base_name<Constant>
{
    typedef String type;
};

    // possibly inherited from Scoped 

template <>
struct scope<Constant>
{  
typedef Scope type;
};

// possibly inherited from NamedScoped
template <>
struct full_name<Constant>
{
  typedef String type;
};

// possibly inherited from NamedScoped
template <typename X>
struct named_typedef<Constant, X>
{
  typedef unspecified type;
};

// possibly inherited from NamedScoped
template <typename X>
struct named_mem_var<Constant, X>
{
  typedef unspecified type;
};

// possibly inherited from ClassMember
template <>
struct access_type<Constant>
{
  typedef Specifier type;
};

template <>
struct value<Constant>
{
  static constexpr unspecified-constant-value value;
};

B. Mirror examples

The first example prints some information about the members of selected namespaces to std::cout.
struct info_printer
{
    template <typename MetaObject>
    void operator()(MetaObject mo) const
    {
        MIRRORED_META_OBJECT(MetaObject) mmo;
        std::cout
              << mmo.construct_name()
              << ": "
              << mo.full_name()
              << std::endl;
    }
};

int main(void)
{
    using namespace mirror;

    // print the info about each of the members
    // of the global scope
    mirror::mp::for_each<
        members<

            // this should be in standard C++
            // be replaced by a special standard library
            // function or operator
            MIRRORED_GLOBAL_SCOPE()
        >
    >(info_printer());

    // print the info about each of the members
    // of the std namespace
    mp::for_each<
        members<

            // this should be in standard C++
            // be replaced by a special standard
            // library function or operator
            MIRRORED_NAMESPACE(std)
        >
    >(info_printer());

    //
    return 0;
}
This program produces the following output:

```cpp
namespace: std
namespace: boost
type: void
type: bool
type: char
type: unsigned char
type: wchar_t
type: short int
type: int
type: long int
type: unsigned short int
type: unsigned int
type: unsigned long int
type: float
type: double
type: long double
class: std::string
class: std::wstring
class: std::tm
template: std::pair
template: std::tuple
template: std::allocator
template: std::equal_to
template: std::not_equal_to
template: std::less
template: std::greater
template: std::less_equal
template: std::greater_equal
template: std::vector
template: std::list
template: std::deque
template: std::map
template: std::set
```

The next example gets all types in the global scope, applies some `type_traits` modifiers like `std::add_pointer` `std::add_const` and for each of such modified types calls a functor that prints the names of the individual types to the standard output:

```cpp
struct name_printer
{
    template <typename MetaNamedObject>
    void operator()(MetaNamedObject mo) const
    {
```
```cpp
std::cout << mo.base_name() << std::endl;
}
};

int main(void)
{
    using namespace mirror;

    // this function calls the name_printer functor passed
    // as the function argument on each element in the
    // range that is passed as the template argument
    mp::for_each<

    // this template transforms the elements in the range
    // passed as the first argument by the unary template
    // passed as the second argument
    mp::transform<

    // this template filters out only those metaobjects
    // that satisfy the predicate passed as the second
    // argument from the range of metaobjects passed
    // as the first argument
    mp::only_if<

    // this template "returns" a range of metaobjects
    // reflecting the members of the namespace
    // (or other scope) that is passed as argument
    members<

    // this macro expands into a class
    // conforming to the Mirror’s MetaNamespace
    // concept and provides metadata describing
    // the global scope namespace.
    // in the proposed solution for standard C++
    // this should be relaced by a special stdlib
    // function or by an operator.
    MIRRORED_GLOBAL_SCOPE()
>,

    // this is a lambda function testing if its first
    // argument falls to the MetaType category
    mp::is_a<
        mp::arg<1>,
        meta_type_tag
```
// this is a unary lambda function that modifies
// the type passed as its argument by
// the add_pointer and add_const type traits
apply_modifier<
  mp::arg<1>,
  mp::protect<
    std::add_pointer<
      std::add_const<
        mp::arg<1>
    >
  >
  >
>
>(name_printer());
std::cout << std::endl;
return 0;
}

This short program produces the following output:

void const *
bool const *
char const *
unsigned char const *
wchar_t const *
short int const *
int const *
long int const *
unsigned short int const *
unsigned int const *
unsigned long int const *
float const *
double const *
long double const *

For other examples of usage see [2].
C. Puddle examples

In this example a reflection-based algorithm traverses the global scope namespace and its nested scopes and prints information about their members:

```cpp
struct object_printer
{
    std::ostream& out;
    int indent_level;

    std::ostream& indented_output(void)
    {
        for(int i=0;i!=indent_level;++i)
            out << "  ";
        return out;
    }

    template <class MetaObject>
    void print_details(MetaObject obj, mirror::meta_object_tag)
    {
    }

    template <class MetaObject>
    void print_details(MetaObject obj, mirror::meta_scope_tag)
    {
        out << ":  ";
        if(obj.members().empty())
        {
            out << "{}";
        }
        else
        {
            out << "{" << std::endl;
            object_printer print_members = {out, indent_level+1};
            obj.members().for_each(print_members);
            indented_output() << "}";
        }
    }

    template <class MetaObject>
    void print(MetaObject obj, bool last)
    {
        indented_output()
            << obj.self().construct_name()
```
<< " "
<< obj.base_name();
print_details(obj, obj.category());
if(!last) out << ",";
out << std::endl;
}
template <class MetaObject>
void operator()(MetaObject obj, bool first, bool last)
{
    print(obj, last);
}

template <class MetaObject>
void operator()(MetaObject obj)
{
    print(obj, true);
}

int main(void)
{
    object_printer print = {std::cout, 0};
    print(puddle::adapt<MIRRORED_GLOBAL_SCOPE>()());
    return 0;
}

which prints the following on the standard output:

global scope : {
namespace std: {
    class string: { },
    class wstring: { },
    template pair,
    template tuple,
    template initializer_list,
    template allocator,
    template equal_to,
    template not_equal_to,
    template less,
    template greater,
    template less_equal,
    template greater_equal,
    template deque,
    class tm: {
        member variable tm_sec,
member variable tm_min,
member variable tm_hour,
member variable tm_mday,
member variable tm_mon,
member variable tm_year,
member variable tm_wday,
member variable tm_yday,
member variable tm_isdst
},
template vector,
template list,
template set,
template map
},
namespace boost: {
    template optional
},
namespace mirror: { },
type void,
type bool,
type char,
type unsigned char,
type wchar_t,
type short int,
type int,
type long int,
type unsigned short int,
type unsigned int,
type unsigned long int,
type float,
type double,
type long double
}

For more examples of usage see [3].

D. Rubber examples

The first example shows the usage of type-erased metaobjects with a C++11 lambda function which could not be used with Mirror’s or Puddle’s metaobjects (because lambdas are not templated):
```cpp
#include <mirror/mirror.hpp>
#include <rubber/rubber.hpp>
#include <iostream>

int main(void)
{
    // use the Mirror's for_each function, but erase
    // the types of the iterated compile-time metaobjects
    // before passing them as arguments to the lambda function.
    mirror::mp::for_each<
        mirror::members<
            MIRRORED_GLOBAL_SCOPE()>
    >(
        // the rubber::meta_named_scoped_object type is
        // constructible from a Mirror MetaNamedScopedObject
        [] (const rubber::meta_named_scoped_object & member)
        {
            std::cout <<
                member.self().construct_name() <<
                " " <<
                member.base_name() <<
                std::endl;
        }
    );
    return 0;
}
```

This simple application prints the following on the standard output:

```plaintext
namespace std
namespace boost
namespace mirror

type void
type bool
type char
type unsigned char
type wchar_t
type short int
type int
type long int
type unsigned short int
type unsigned int
type unsigned long int
type float
```
type double
type long double

The next example prints different information for different categories of metaobjects:

```cpp
#include <mirror/mirror.hpp>
#include <rubber/rubber.hpp>
#include <iostream>
#include <vector>

int main(void)
{
    using namespace rubber;
    mirror::mp::for_each<
        mirror::members<
            MIRRORED_GLOBAL_SCOPE()
        >
    >
    eraser<
        meta_scope, meta_type, meta_named_object>
    [](const meta_scope& scope)
    {
        std::cout <<
            scope.self().construct_name() <<
            " ": " 
            scope.base_name() <<
            ", number of members = " <<
            scope.members().size() <<
            std::endl;
    },
    [](const meta_type& type)
    {
        std::cout <<
            type.self().construct_name() <<
            " ": " 
            type.base_name() <<
            ", size in bytes = " <<
            type.sizeof_() <<
            std::endl;
    },
    [](const meta_named_object& named)
    {
        std::cout <<
            named.self().construct_name() <<
            " ": " 
            named.base_name() <<
```
It has the following output:

namespace 'std', number of members = 20
namespace 'boost', number of members = 0
namespace 'mirror', number of members = 0

type 'void', size in bytes = 0
type 'bool', size in bytes = 1
type 'char', size in bytes = 1
type 'unsigned char', size in bytes = 1
type 'wchar_t', size in bytes = 4
type 'short int', size in bytes = 2
type 'int', size in bytes = 4
type 'long int', size in bytes = 8
type 'unsigned short int', size in bytes = 2
type 'unsigned int', size in bytes = 4
type 'unsigned long int', size in bytes = 8
type 'float', size in bytes = 4
type 'double', size in bytes = 8
type 'long double', size in bytes = 16

For more examples of usage see [4].

E. Lagoon examples

This example queries the meta-types reflecting types in the global scope, orders them by the value of sizeof and prints their names:

```cpp
#include <mirror/mirror.hpp>
#include <lagoon/lagoon.hpp>
#include <lagoon/range/extract.hpp>
#include <lagoon/range/sort.hpp>
#include <iostream>

int main(void) {
```
using namespace lagoon;
typedef shared<meta_named_scoped_object> shared_mnso;
typedef shared<meta_type> shared_mt;
//
// traverses the range of meta-objects passed as
// the first argument and on each of them executes
// the functor passed as the second argument
for_each(

    // sorts the range passed as the first argument
    // using the functor passed as the second argument
    // for comparison
    sort(

        // extracts only those having the meta_type
        // interface
        extract<meta_type>(

            // gets all members of the global scope
            reflected_global_scope()->members()
        ),

        // compares two meta-types on the value
        // of sizeof(reflected-type)
        [](const shared_mt& a, const shared_mt& b)
        {
            return a->size_of() < b->size_of();
        }
    ),

    // prints the full name of a type
    [](const shared_mt& member)
    {
        std::cout << member->full_name() << std::endl;
    }
)

return 0;

This application prints the following on the standard output:

void
bool
char
unsigned char
short int
unsigned short int
wchar_t
int
long int
unsigned int
unsigned long int
float
double
long double

The following example is more complex and shows the usage of Lagoon’s object factories, in this case a factory using a text-script similar to C++ uniform initializers to provide input data from which a set of instances is constructed:

```cpp
#include <mirror/mirror_base.hpp>
#include <mirror/pre_registered/basic.hpp>
#include <mirror/pre_registered/class/std/vector.hpp>
#include <mirror/pre_registered/class/std/map.hpp>
#include <mirror/utils/quick_reg.hpp>
#include <lagoon/lagoon.hpp>
#include <lagoon/utils/script_factory.hpp>
#include <iostream>

namespace morse {

    // declares an enumerated class for morse code symbols
    enum class signal : char { dash = '-', dot = '.'};

    // declares a type for a sequence of morse code symbols
    typedef std::vector<signal> sequence;

    // declares a type for storing morse code entries
    typedef std::map<char, sequence> code;

} // namespace morse

MIRROR_REG_BEGIN

    // registers the morse namespace
    MIRROR_QREG_GLOBAL_SCOPE_NAMESPACE(morse)
    // registers the signal enumeration
    MIRROR_QREG_ENUM(morse, signal, (dash)(dot))

MIRROR_REG_END
```
```c
int main(void)
{
    try
    {
        using namespace lagoon;

        // a factory builder class provided by Lagoon
        // that can be used together with a meta-type
        // to build a factory
        c_str_script_factory_builder builder;

        // a class storing the input data for the factory
        // built by the builder
        c_str_script_factory_input in;

        // the input data for the factory
        auto data = in.data();

        // polymorphic meta-type reflecting the morse::code type
        auto meta_morse_code = reflected_class<morse::code>();

        // a polymorphic factory that can be used to construct
        // instances of the morse::code type, that is built by
        // the builder and the meta-type reflecting morse::code.
        auto morse_code_factory = meta_morse_code->make_factory(
            builder,
            raw_ptr(&data)
        );

        // the input string for this factory
        const char input[] = "{" \
            "'A', {dot, dash}}," \
            "'B', {dash, dot, dot, dot}}," \
            "'C', {dash, dot, dash, dot}}," \
            "'D', {dash, dot, dot}}," \
            "'E', {dot}}," \
            "'F', {dot, dot, dash, dot}}," \
            "'G', {dash, dash, dot}}," \
            "'H', {dot, dot, dot, dot}}," \
            "'I', {dot, dot}}," \
            "'J', {dot, dash, dash, dash}}," \
            "'K', {dash, dot, dash}}," \
            "'L', {dot, dash, dot, dot}}," \
```

// passes the input data to the factory
in.set(input, input+sizeof(input));

// use the factory built above to create
// a new instance of the morse::code type
raw_ptr pmc = morse_code_factory->new_();

// cast of the raw pointer returned by the factory
// to the concrete type (morse::code)
morse::code& mc = *raw_cast<morse::code*>(pmc);

// the morse::code type is just a map of char to
// a vector of morse signals, this prints them
// to cout in a standard way
for(auto i = mc.begin(), e = mc.end(); i != e; ++i)
{
  std::cout << "Morse code for " << i->first << ": ";
  auto j = i->second.begin(), f = i->second.end();
while (j != f) {
    std::cout << char(*j);
    ++j;
}
std::cout << std::endl;

// uses the meta-type reflecting morse::code to delete
// the instance constructed by the factory
meta_morse_code->delete_(pmc);
}
catch (std::exception const & error) {
    std::cerr << "Error: " << error.what() << std::endl;
}
//
return 0;

This application has the following output:

Morse code for '0': -----  
Morse code for '1': .----  
Morse code for '2': ..---  
Morse code for '3': ...--  
Morse code for '4': ....-  
Morse code for '5': .....  
Morse code for '6': -....  
Morse code for '7': --...  
Morse code for '8': ---..  
Morse code for '9': ----.  
Morse code for 'A': .-    
Morse code for 'B': -...  
Morse code for 'C': -.-.  
Morse code for 'D': -..   
Morse code for 'E': .     
Morse code for 'F': ..-.  
Morse code for 'G': --.   
Morse code for 'H': .....  
Morse code for 'I': ..    
Morse code for 'J': .---  
Morse code for 'K': -.-.  
Morse code for 'L': .-..  
Morse code for 'M': --   

129
Morse code for 'N': -.
Morse code for 'O': ---
Morse code for 'P': .--.
Morse code for 'Q': --.-
Morse code for 'R': .-.
Morse code for 'S': ...
Morse code for 'T': -
Morse code for 'U': ..-
Morse code for 'V': ...-
Morse code for 'W': .--
Morse code for 'X': -..-
Morse code for 'Y': -.--
Morse code for 'Z': --..

For more examples of usage see [5].