

Document number: P0385R2  
Date: 2017-02-06  
Project: Programming Language C++  
Audience: Reflection (SG7) / EWG  
Matúš Chochlík(chochlik@gmail.com),  
Reply-to: Axel Naumann (axel@cern.ch),  
David Sankel (dsankel@bloomberg.net)

# Static reflection

## Rationale, design and evolution.

*Matúš Chochlík, Axel Naumann, David Sankel*

**Abstract** The aim of this paper is to provide the rationale behind the design of the static reflection facility proposed in P0194, to enumerate and describe its potential use-cases and to keep a written record of its evolution. It also answers questions frequently asked in regard to the proposal.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Terminology . . . . .	4
1.1.1	Base-level, meta-level . . . . .	4
1.1.2	Metadata . . . . .	4
1.1.3	Metaprogramming . . . . .	5
1.1.4	Reflection . . . . .	5
1.1.5	First-class object, second-class object . . . . .	5
1.1.6	Reification . . . . .	7
<b>2</b>	<b>Revision history</b>	<b>7</b>
2.1	P0385R0 . . . . .	7
2.2	P0385R1 . . . . .	8
2.3	P0385R2 – current revision . . . . .	8
<b>3</b>	<b>Motivation</b>	<b>9</b>
<b>4</b>	<b>Design</b>	<b>11</b>
4.1	Basic overview . . . . .	11
4.2	Design considerations . . . . .	12
4.2.1	Completeness and reusability . . . . .	12
4.2.2	Consistency . . . . .	13
4.2.3	Encapsulation . . . . .	13
4.2.4	Stratification . . . . .	13
4.2.5	Meta-level reification . . . . .	13

4.2.6	Ontological correspondence . . . . .	14
4.2.7	Efficiency . . . . .	14
4.2.8	Ease of use . . . . .	14
4.2.9	Integration . . . . .	15
4.2.10	Extensibility . . . . .	15
4.3	Compile-time vs. run-time reflection . . . . .	16
4.4	Design evaluation . . . . .	16
<b>5</b>	<b>Use cases and examples</b>	<b>17</b>
5.1	Portable (type) names . . . . .	17
5.2	Logging . . . . .	17
5.3	Implementing common functions and operators . . . . .	20
5.4	Enumerator to string and vice versa . . . . .	23
5.5	Simple serialization . . . . .	25
5.6	Structure data member transformations . . . . .	27
5.7	Simple examples of usage . . . . .	29
5.7.1	Scope reflection . . . . .	29
5.7.2	Namespace reflection . . . . .	30
5.7.3	Type reflection . . . . .	32
5.7.4	Typedef reflection . . . . .	32
5.7.5	Class alias reflection . . . . .	34
5.7.6	Class data members (1) . . . . .	35
5.7.7	Class data members (2) . . . . .	36
5.7.8	Class data members (3) . . . . .	38
<b>6</b>	<b>Future extensions</b>	<b>39</b>
6.1	Additional utilities . . . . .	40
6.1.1	<code>for_each</code> . . . . .	40
6.1.2	<code>unpack_sequence_if</code> . . . . .	40
6.2	Reversing reflection . . . . .	41
6.3	Generating identifiers programmatically . . . . .	43
6.3.1	Identifier from a generic compile-time string . . . . .	44
6.3.2	Identifier formatting . . . . .	44
6.3.3	<code>named_data_member</code> and <code>named_member_typedef</code> . . . . .	45
6.4	Variadic composition . . . . .	46
6.5	Specifier reflection . . . . .	47
6.6	Context-dependent reflection . . . . .	48
6.6.1	Namespaces . . . . .	48
6.6.2	Classes . . . . .	48
6.6.3	Functions . . . . .	49
6.6.4	Templates . . . . .	50
6.7	Metaobject unique ID . . . . .	51
<b>7</b>	<b>Issues</b>	<b>52</b>
7.1	Interaction with attributes . . . . .	52
7.2	Interaction with concepts . . . . .	53
7.3	Interaction with modules . . . . .	53

<b>8 Experimental implementation</b>	<b>53</b>
<b>9 The Mirror reflection utilities</b>	<b>53</b>
<b>10 Frequently asked questions</b>	<b>53</b>
10.1 Why metaobjects, why not reflect directly with type traits? . . . . .	53
10.2 OK, but why not reflect directly with several different operators? . . . . .	54
10.3 Creating a separate type for each metaobject is so heavyweight. . . . .	55
10.4 There’s already another expression for that! . . . . .	55
10.5 All these additional options make it hard for the novices. . . . .	57
10.6 Why are the metaobjects anonymous? . . . . .	58
10.7 Reflection violating access restrictions to class members? . . . . .	59
10.8 We need to get around access restrictions, but not in reflection. . . . .	59
10.9 Why do we need typedef reflection? . . . . .	59
10.10 Why <i>ObjectSequences</i> ? Why not replace them with typelists? . . . . .	60
10.11 Why not reflect with constexpr instances of the same type? . . . . .	61
10.12 Why not return fully qualified names? . . . . .	62
10.13 What about the use cases from the committee’s CFP? . . . . .	62
<b>11 Acknowledgments</b>	<b>62</b>
<b>Appendix</b>	<b>64</b>

## 1 Introduction

This paper accompanies the P0194R3 paper which we want to keep brief, technical and to the point and which will eventually result in the final wording to be included into the standard, if this proposal is accepted. There is also a short introductory paper – P0578 which gives a concise introduction and overview of our reflection proposal.

We are writing this paper with several goals in mind:

- To define and explain the reflection-related terminology.
- To keep a written record of the rationale behind the design of the proposed static reflection facility.
- To keep the answers to the frequently asked questions about the decisions we’ve made in this proposal in one place so that we can avoid having to write them over and over from scratch in various discussions<sup>1</sup>.
- To enumerate and describe the use cases for the various features which we included in the proposal.
- To provide concrete examples of usage.
- To discuss the possibilities of the evolution of reflection in the future.

---

<sup>1</sup>Also to help us remember what the answers were.

The text of this paper includes several revised, updated and extended parts from the previous papers: N3996, N4111, N4451, N4452 and previous revisions of P0385.

There is also an experimental implementation of the P0194R2 proposal described in greater detail in section 8 and the *Mirror* reflection library built on top of the proposed static reflection facility described in section 9.

## 1.1 Terminology

In order to avoid confusion about terminology, this section provides definitions for several important terms used throughout the text of the paper and explains what we mean when using them in the context of this paper.

### 1.1.1 Base-level, meta-level

When speaking generally, the *meta-level* is some higher level of abstraction conceptually describing a lower, *base-level* which is the primary subject of our endeavors.

In the context of this paper the base-level is the structure of a C++ program. The meta-level is an abstraction partially describing that structure, mainly the declarations of the program.

### 1.1.2 Metadata

*Metadata* is generally a piece of data conceptually describing some other, *primary* data.

In the context of this paper, metadata is data providing information about the base-level structure of a program.

*Static metadata* is metadata which can be manipulated or reasoned about at compile-time by the compiler.

The metadata itself has its own structure. For example metadata describing the base-level declaration of a class from a C++ program includes;

- the name of the class,
- its scope,
- list of its base classes,
- list of data members,
- list of nested types like type aliases, classes and enumerations,
- the class key specifier, the access specifier,
- source location information,
- etc.

### 1.1.3 Metaprogramming

*Metaprogramming* is a kind of programming with the ability to treat and manipulate other programs as data, so both the input and the output of a metaprogram is usually a program. The language in which metaprograms are written is called the *metalanguage* and it can be a different or the same language as the one used to write the primary program.

C++ metaprogramming can be done both in an external language<sup>2</sup>, in a C++ compiler plug-in, or in C++ itself.

In C++ metaprogramming usually takes the form of an external *source code generator*<sup>3</sup>, or the form of *preprocessor*, *template* or *constexpr* metaprogramming.

With template metaprogramming we use C++’s type system as a standalone functional programming language “interpreted” by a C++ compiler, with “variables” being represented by types (or compile-time constants), “data structures” by instantiations of templates, “subroutines” by class templates or template aliases, and algorithms or “programs” by compositions of the above.

Unless stated otherwise when we say “metaprogramming” in the following text, we mean template metaprogramming<sup>4</sup>.

### 1.1.4 Reflection

In the context of computer science, the term *reflection* refers to the ability of a program to examine and possibly modify its own structure and/or behavior.

When combined with metaprogramming, this can include modification of the existing or the definition of new data structures, doing changes to algorithms or changing the way a program code is interpreted<sup>5</sup>.

For the purpose of this paper – *reflection is the process of obtaining metadata*.

In the future the meaning can be expanded to include modification of the program in ways exceeding the capabilities of current template metaprogramming<sup>6</sup>.

### 1.1.5 First-class object, second-class object

First-class objects are also known as first-class *citizens*, *types*, *entities* or *values*. In the context of programming language design they describe an entity that satisfies the following:

- Can be stored in a named variable or a data structure.
- Can be passed as an argument to a subroutine.
- Can be returned as a result of a subroutine.
- Has an intrinsic identity making the entity unique and distinguishable.

---

<sup>2</sup>For example a C++ code generator written in Python or Bash.

<sup>3</sup>Like Qt’s MOC.

<sup>4</sup>Although we envision that a constexpr-metaprogramming interface will also be added later in the future.

<sup>5</sup>Mostly in interpreted languages.

<sup>6</sup>For example generating new class data members.

Since this paper deals with compile-time static reflection and its use in compile-time metaprogramming, we will be talking about first- or second-class citizens in this regard. For the purpose of this paper a first-class object is something that we can distinguish and reason about at compile-time and what can be passed around as “data” in metaprograms – something that can be a template parameter.

This means that a type, a template instance or a compile-time integral constant is for our purposes a first-class object:

```
// compile-time "values"
struct value_a { };
struct value_b { };

// a compile-time "subroutine"
template <typename Param>
struct identity {
    using result = Param;
};

// "values" are equality comparable and they are distinguishable
assert(!std::is_same_v<value_a, value_b>);
// and can be used for overloading
assert(!std::is_same_v<identity<value_a>, identity<value_b>>);

// they can be stored in named "variables" ...
using x = value_a;

// ... while maintaining their identity
assert(std::is_same_v<value_a, x>);

// they can be passed as parameters to subroutines
identity<value_a>;
identity<value_b>;

// they can be returned from subroutines
using y = identity<value_a>::result;
using z = identity<value_b>::result;

// ... while still maintaining their identity
assert(std::is_same_v<value_a, y>);
assert(std::is_same_v<value_b, z>);
// and still being distinguishable
assert(!std::is_same_v<y, z>);
```

On the other hand a namespace, a type alias or a template parameter do not have some or any of these properties.

```
// other entities
namespace std { }
namespace foo = std;
```

```
using bar = unsigned;
using baz = unsigned;

template <typename Param>
struct identity {
    using result = Param;
};

// they are not distinguishable
std::is_same_v<bar, baz>;
// and cannot be used for overloading
std::is_same_v<identity<bar>, identity<baz>>;

// they are not even comparable
std::is_same_v<std, foo>;

// they cannot be passed as arguments to subroutines
identity<std>;
identity<foo>;

// nor returned as a result
namespace y = identity<foo>::result;

// those which can be returned ...
using z = identity<bar>::result;
// ... do not maintain their unique identity
std::is_same_v<z, baz>;
```

From the above follows, that a *second-class object* is everything else that is not a first-class object, for example a namespace or a typedef.

### 1.1.6 Reification

Generally speaking *reification* or “thingification”<sup>7</sup>, is making something real, bringing it into being as an entity with its own identity, or making something concrete.

In regard to programming languages, reification is often defined as making an entity in the language a first-class object. So in the context of C++ template metaprogramming a type is reified, but a namespace or a specifier is not.

## 2 Revision history

### 2.1 P0385R0

The initial revision of the design paper accompanying the P0194 papers.

---

<sup>7</sup>From latin “rei”, the dative form of “res” – a thing.

## 2.2 P0385R1

This revision brought the following updates reflecting the changes made in the P0194R2 paper.

- Based on the feedback from Oulu, the specification of typedef and type alias reflection has been simplified. We no longer require reflection to be aware of the whole chain of type aliases especially not in the context of templates, each type alias only references the underlying type with unique identity at the base level.
- Unlike in the previous revision it is now possible to create variables with *metaobject* types.
- Section 6.1.1 describing the `for_each` function planned to be included in future revisions has been revised.
- The *reverse reflection operator* has been renamed to `unreflexpr`.
- The *identifier-generating operator* has been renamed to `idreflexpr`.
- The mechanism for enumerating public-only vs. all (including non-public ones) class members has been changed. Now the “basic” operations like `get_data_members`, `get_member_types`, etc. return all members, and the `get_public_data_members`, `get_public_member_types`, return only the public class members.
- All examples have been revised and extended according to the changes listed above.
- New examples have been added to section 4.2 describing the design considerations.
- A description of the implementation has been added (section 8).
- A brief introduction to the *Mirror reflection utilities* has been added (section 9).

## 2.3 P0385R2 – current revision

We have made several big changes in revision R3 of the P0194 paper:

- We introduced the `$` character to the basic source character set as a prefix for all reflection-related operators – both the primary reflection operator which is already part of the wording and for any which may be added in the future.
- The reflection-related operators have been renamed
  - from `reflexpr` to `$reflect`,
  - from `unreflexpr` to `$unreflect` and
  - from `idreflexpr` to `$identifier`.
- The reflection-related namespace has been renamed from `std::meta` to `std::reflect` in order to avoid confusion and conflicts with any potential metaprogramming library which could be added in the future.
- The reflection-related header file has been renamed from `<reflexpr>` to `<reflect>`.
- Direct reflection of specifiers (previously `reflexpr(public)`) and metaobjects reflecting specifiers (conforming to the `Specifier` concept) have been removed temporarily and their functionality has been partially replaced with boolean traits.



- The metaobject concept hierarchy has been slightly changed; in addition to the `Specifier`, the `TagType` concept has also been removed, its operations were taken over by `Enum` and `Record` concepts. The `Reversible` concept has also been removed for the time being.
- The order of template parameters for `MetaSequence` operations has been matched to similar operations of `std::tuple`.
- The suffix `_m` was reverted back to `_t` for all metaobject-returning operations.

In this revision of P0385 we made the following changes:

- Some of the use-cases were removed for the sake of brevity, please see the previous revisions of this paper.
- The examples have been updated to match the wording from P0194R3.

### 3 Motivation

Generic programming and metaprogramming supported by reflection can be valuable tools in the implementation of an extensive range of various use cases or programming patterns, including but not limited to:

- serialization or conversion of data from the native C++ representation into a standard or custom, text-based or binary format like XML, JSON, XDR, ASN1, etc.,
- (re-)construction of instances of both “atomic” and structured types from external data representations, like those listed above, or from the data stored in a relational database, or from data entered by a user through a user interface, or queried through a web service API,
- automated implementation of comparison operations or hash functions for structured types,
- automated generation of user interface elements,
- automatic generation of a relational schema from the application object model and object-relational mapping (ORM),
- support for scripting or implementation of input data parsers,
- support for remote procedure calls (RPC) / remote method invocation (RMI),
- inspection and manipulation of existing objects via a user interface or a web service,
- visualization of objects or structured data and their relationships,
- automatic or semi-automatic implementation of certain software design patterns, for example the factory pattern,
- implementation of cross-cutting aspects like debugging, logging, profiling, access control, etc.,
- implementation of *array-of-structures* to *structure-of-arrays* transformations,
- implementation of source code generators.

Some of the use cases listed above are described in more detail in section 5, and also in [2, 7].

There are several approaches to the implementation of the mentioned functionality. The most basic, 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 and organize programming language elements<sup>8</sup> in a very uniform way which could be transcribed into a metaprogram<sup>9</sup>.

This leads to the second, heavily used approach: preprocessing and parsing of the program source text by a usually very specific external program like, a documentation generation tool, an interface definition language compiler for a RPC/RMI framework, a web service interface generator, a rapid application development environment with a form designer, etc., resulting in additional program source code, which is then integrated into the project and 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 and third, there is a lot of repeated code related to the parsing, the representation and the manipulation of the input program source.

Another way to automate these tasks is to use reflection and metaprogramming.

For example if we want to log the execution of a function, reflection may be used as a source of metadata:

```
template <typename T>
T min(const T& a, const T& b) {
    log()    << "function: min<"
            << get_base_name_v<get_aliased_t<$reflect(T)>>
            << ">("
            << get_base_name_v<$reflect(a)> << ": "
            << get_base_name_v<get_aliased_t<get_type_t<$reflect(a)>>>
            << " = " << a
            << get_base_name_v<$reflect(b)> << ": "
            << get_base_name_v<get_aliased_t<get_type_t<$reflect(b)>>>
            << " = " << b
            << ")" << std::endl

    return a<b?a:b;
}
```

Calling the min function:

```
double m = min(12.34, 23.45);
```

would produce the following log entry:

```
function: min<double>(a: double = 12.34, b: double = 23.45)
```

---

<sup>8</sup>types, structures, containers, functions, constructors, class member variables, enumerated values, etc.

<sup>9</sup>with varying level of complexity

## 4 Design

### 4.1 Basic overview

As the introduction briefly mentions, the metadata reflecting base-level program declarations has its own structure. One way to maintain this structure and to organize the individual, but related pieces of metadata reflecting for example the structure of a class is to compose them into *metaobjects*.

In P0194 we propose to add support for compile-time reflection to C++ by the means of lightweight, compiler-generated types – *metaobjects*, providing metadata describing various base-level program declarations.

The Metaobjects themselves are opaque, without any visible internal structure. Values of such a Metaobject type are default- and copy-constructible.

Their primary purpose is to give a first-class identity to the reflected entity<sup>10</sup>, so that we can pass it as an argument or a return value in metaprograms and to separate the reflection of a declaration from the querying of metadata<sup>11</sup>.

We introduce a new reflection operator – `$reflect` which returns a metaobject type reflecting its operand.

For example:

```
using meta_global_scope    = $reflect(::);           // GlobalScope
using meta_int              = $reflect(int);         // Type
using meta_std              = $reflect(std);         // Namespace
using meta_std_size_t      = $reflect(std::size_t);  // Type and Alias
using meta_std_thread       = $reflect(std::thread); // Class
using meta_std_pair        = $reflect(std::pair<int, int>); // Class
using meta_std_launch      = $reflect(std::launch); // Enum
using meta_std_launch_async = $reflect(std::launch::async); // Enumerator
```

or in the future revisions;

```
using meta_std_pair        = $reflect(std::pair);    // Template
using meta_std_sin         = $reflect(std::sin);     // OverloadedFunction
```

etc.

Since there are many different kinds of base-level declarations, the metaobjects reflecting them are modeling various *metaobject concepts*, which also serve to classify metaobjects and to indicate whether a metaobject has or has not a particular property;

```
static_assert(reflect::Named<$reflect(std)>);
static_assert(reflect::ScopeMember<$reflect(int)>);
static_assert(reflect::Scope<$reflect(std::string)>);
static_assert(reflect::Alias<$reflect(std::string::size_type)>);
```

or if it falls into a particular category:

---

<sup>10</sup>Namespace, type alias, function, parameter, specifier, etc.

<sup>11</sup>Which will happen very often in the more complex use cases

```
static_assert(reflect::GlobalScope<$reflect(::)>);  
static_assert(reflect::Namespace<$reflect(std)>);  
static_assert(reflect::Type<$reflect(int)>);  
static_assert(reflect::Class<$reflect(std::string)>);
```

The individual pieces of metadata can be obtained from a metaobject by using one of the class templates which comprise its interface.

Some of this metadata like the class name or number of base classes is provided as compile-time constant values, some as base-level types and some in the form of other metaobjects, like metaobjects reflecting declaration scope or a sequence of metaobjects reflecting class members, etc.:

```
using meta_str = $reflect(std::string);  
  
get_base_name_v<meta_str>;           // a compile-time constant string  
get_reflected_type_t<meta_str>;     // a base-level type: std::string  
get_scope_t<meta_str>;              // another metaobject reflecting the scope  
get_data_members_t<meta_str>;       // a metaobject-sequence containing other metaobjects
```

P0194R3 also defines the initial subset of metaobject concepts which we assume to be essential and which will provide a good starting point for future extensions.

## 4.2 Design considerations

The proposed static reflection facility has been designed with the following considerations and goals in mind. Note that some of the principles listed here apply only to the whole reflection facility as it is envisioned to look in the future, not to the initial, limited subset from P0194R3.

### 4.2.1 Completeness and reusability

The metadata provided by reflection is reusable in many situations and for many different purposes. It does not focus on nor is limited only to the simple and immediately obvious use cases. New use cases which we are not aware of at this moment, may emerge in the future. So having or not having a compelling use case for a particular feature is a factor in the decision whether to include it, but it should not be the most important one.

When completed, the proposed reflection facility will provide as much useful metadata as possible, reflecting various base-level declarations like types, namespaces, variables, functions, templates, specifiers<sup>12</sup> and will provide access to scope members, base classes, etc.

This will make the compiler-assisted reflection<sup>13</sup> a useful tool in a wide range of scenarios during both compile-time and run-time and under various paradigms<sup>14</sup> depending on the application needs.

---

<sup>12</sup>Like constness, storage-class, access, etc. specifiers

<sup>13</sup>Either by itself or serving as the foundation for other standard or third-party libraries.

<sup>14</sup>Object-oriented, functional, etc.

### 4.2.2 Consistency

The reflection facility as a whole is consistent, instead of being composed of several ad-hoc, individually-designed parts. This makes its interface more tidy, coherent and easier to learn and teach.

### 4.2.3 Encapsulation

The metadata is not exposed directly to the user by many different compiler built-ins, operators or special expressions. Instead it is accessible through conceptually well-defined interfaces, inspired by the existing *type-traits*, already present in the C++ standard template library.

### 4.2.4 Stratification

Reflection is non-intrusive and the metaobjects are separated from the base-level language declarations which they reflect:

```
float foo = 42.f;
```

```
using m_std = $reflect(std);  
using m_int = $reflect(int);  
using m_foo = $reflect(foo);
```

instead of for example

```
using m_std = std::reflect();  
using m_int = int::reflect();  
using m_foo = foo::reflect();
```

This is achieved by using the reflection operator which hides most of the “magic”, isolates reflection from the rest of the language, for example by allowing to pass expressions not valid elsewhere as operands.

It is also similar to the syntax of other existing operators in C++:

```
float foo = 42.f;
```

```
using foo_type = decltype(foo);  
using meta_foo = $reflect(foo);
```

### 4.2.5 Meta-level reification

The metaobjects giving a first-class identity to declarations which are only second-class in base-level C++, allow to partially “reify” namespaces, typedefs, specifiers, etc., without actually making them first-class citizens. This in turn allows to pass their reflections around metaprograms even if it was not possible with the base-level declarations.

```
template <typename X>  
struct my_meta_algorithm {
```

```
    using result = /* ... */;  
};  
  
using x = my_meta_algorithm<$reflect(std)>::result; // OK  
  
using y = my_meta_algorithm<std>::result; // Error
```

#### 4.2.6 Ontological correspondence

The meta-level corresponds to the ontology of the base-level C++ language which it reflects. This basically means that all important existing language features<sup>15</sup> will eventually be reflected by appropriate metaobjects, but new ones not having an equivalent in the base-level language<sup>16</sup> will not be invented.

Ontological correspondence is one of the main factors driving the definition of the individual metaobject concepts and the design of their interfaces.

#### 4.2.7 Efficiency

The proposed reflection is fine grained as much as possible. Things that are not needed for a particular application, are not compiled into its program code nor result in increased compiler footprint or compilation times.

The proposed reflection facility makes a completely lazy implementation of metaobjects possible. Metaobjects are created only when requested and the reflection operator is able to generate very lightweight types providing internal links back to the reflected declaration. The actual metadata<sup>17</sup> is materialized only when requested by the programmer via the templates which act as the metaobject interface.

#### 4.2.8 Ease of use

Although reflection-based metaprogramming should allow to implement very complicated meta-algorithms, we try to adhere to the principle that things should be kept as simple as possible, but not simpler<sup>18</sup>.

This can be achieved by having a solid and powerful compiler-assisted reflection as the foundation and by implementing a simplifying facade on top of it once the common use-cases are identified.

Utilities such as *enum-to-string*, *string-to-enum* or *get-type-name*, etc., can be implemented on top of reflection instead of each of them using their own “dedicated compiler magic”.

---

<sup>15</sup>Within reason, we certainly do not want to reflect every token in a C++ program.

<sup>16</sup>At least conceptually.

<sup>17</sup>Like a compile-time string containing the identifier, the list of metaobjects reflecting class members or the scope of a declaration, etc.

<sup>18</sup>Credits to whoever said that.

## 4.2.9 Integration

The proposed reflection facility is easily usable with the existing introspection utilities<sup>19</sup> already provided by the standard library and by other third-party libraries.

For example:

```
using meta_int = $reflect(int);
static_assert(is_integral_v<reflect::get_reflected_type_t<meta_int>>);
```

## 4.2.10 Extensibility

It is important be able to gradually add new features and to allow reflecting new declaration kinds in the future without introducing breaking changes. The metaobjects make this goal easily achievable.

If we want to add reflection of previously unsupported declaration kind, for example the reflection of functions, we define a new *metaobject concept* like ***Function*** and ***OverloadedFunction***.

```
namespace reflect {
    template <Object T>
    concept bool Function =
        Named<T> && ScopeMember<T> && Typed<T>
        __metaobject_reflects_function(T);

    template <Object T>
    concept bool OverloadedFunction =
        Named<T> && ScopeMember<T> &&
        __metaobject_reflects_overloaded_function(T);
} // namespace reflect
```

Then we need to extend the list of expressions which can appear as operands to `$reflect` to include specifiers, so that the following is valid code:

```
using meta_func = $reflect(std::sin);
```

To extend the interface of an existing metaobject by adding a new operation returning for example the elaborated type specifier of a class we need to add a new template like:

```
namespace reflect {
    template <Class T>
    struct get_member_functions {
        using type = /* unspecified-metaobject */;
    };

    template <OverloadedFunction T>
    struct get_overloads {
        using type = /* unspecified-metaobject */;
    };
}
```

---

<sup>19</sup>Like the *type-traits* or *typeid* and `std::type_info`.

```
template <Function T>
struct get_parameters {
    using type = /* unspecified-metaobject */;
};
} // namespace reflect
```

### 4.3 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<sup>20</sup> 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 more possibilities for the compiler to do optimizations.

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

### 4.4 Design evaluation

The proposed reflection facility – assuming it is fully completed – will have the following properties:

- covers many different use cases,
- is fairly powerful and expressive,
- is non-intrusive,
- is fine-grained,
- allows for efficient implementations,
- allows to manipulate and reason about all provided metadata at compile-time,
- gives the metadata a structure by arranging it into metaobjects,
- makes the metaobjects first-class entities allowing to pass representations of second-class base-level language entities around metaprograms as arguments and return values and store them in named “variables”<sup>21</sup>,
- it can serve as the foundation for other, compile-time or run-time reflection utilities implementing other interfaces or façades aimed at various paradigms or use cases,
- contains and isolates all the required changes within the reflection operator(s),

---

<sup>20</sup>already taken care of by `type_traits`

<sup>21</sup>We cannot stress the importance of this feature enough.



- limits the impact on existing code by prefixing the names of new operators with the \$ sign and by putting the concepts and templates into the nested `std::reflect` namespace,
- does not require any other changes to the core language, especially no new rules for template parameters.

## 5 Use cases and examples

Note that some of the examples listed in this section use features which are not part of the initial reflection specification, but which are planned as future additions.

Also please see the previous papers [2, 7] as well as [4] which describe several other important use-cases that were omitted here for the sake of brevity.

### 5.1 Portable (type) names

One of the notorious problems of `std::type_info` is that the string returned by its `name` member function is not standardized and is not even guaranteed to return any meaningful, unique human-readable string, at least not without de-mangling, which is platform specific. Furthermore the returned string is not `constexpr` and cannot be reasoned about at compile-time and is applicable only to types. One other problem with `typeid` that it is not always aware of `typedefs`. In some cases we would like to obtain the alias name, instead of the “real” name of a type or a class member or function parameter.

The ability to uniquely map any type used in a program to a human-readable, portable, compile-time string has several use-cases described in this paper.

The *Named* concept reflects named language constructs and provides the `get_base_name` operation returning their basic name without any qualifiers or decorations. This can be with the help of metaprogramming turned into a fully-qualified name. The `get_display_name` operation returns a user-friendly implementation-dependent string containing the full name of a declaration, possibly retaining type alias names.

### 5.2 Logging

When tracing and logging the execution of functions<sup>22</sup> it is sometimes desirable to also include the names of the parameter types or even the names of the parameters and other variables.

The best we can do with just the `std::type_info` is the following:

```
#if __PLATFORM_ABC__
std::string demangled_type_name(const char*) { /* implementation 1 */ }
#else if __PLATFORM_MNO__
std::string demangled_type_name(const char*) { /* implementation 2 */ }
#else if __PLATFORM_XYZ__
std::string demangled_type_name(const char*) { /* implementation N */ }
```

---

<sup>22</sup>especially template functions

```
#else
std::string demangled_type_name(const char* mangled_name) {
    // don't know how to demangle this; let's try our luck
    return mangled_name;
}
#endif

template <typename T>
T min(const T& a, const T& b) {
    log()    << "min<"
            << demangled_type_name(typeid(T).name())
            << ">(" << a << ", " << b << ") = ";

    T result = a<b?a:b;

    log()    << result << std::endl;

    return result;
}
```

Which may or may not work, depending on the platform.

With the help of reflection we can do:

```
template <typename T>
T min(const T& a, const T& b) {
    log()    << "min<"
            << get_base_name_v<$reflect(T)>
            << ">(" << a << ", " << b << ") = ";

    T result = a<b?a:b;

    log()    << result << std::endl;

    return result;
}
```

The `__PRETTY_FUNCTION__` macro generated by the compiler could be also used in this case, but the format of the string which this macro expands into is not customizable (which may be necessary for logs formatted in XML, JSON, etc).

A more elaborate output containing also the parameter names, type names and values can be achieved by using reflection:

```
template <typename T>
T min(const T& a, const T& b) {
    log()    << "function: min<"
            << get_display_name_v<get_aliased_t<$reflect(T)>>
            << ">("
            << get_base_name_v<$reflect(a)> << ": "
            << get_display_name_v<get_aliased_t<get_type_t<$reflect(a)>>>
```

```
<< " = " << a
<< get_base_name_v<$reflect(b)> << ": "
<< get_display_name_v<get_aliased_t<get_type_t<$reflect(b)>>>
<< " = " << b
<< ")" << std::endl
```

```
T result = a<b?a:b;
```

```
log() << get_base_name_v<$reflect(result)> << ": "
<< get_display_name_v<get_aliased_t<get_type_t<$reflect(result)>>>
<< " = " << result << std::endl;
```

```
return result;
```

```
}
```

Calling

```
double x = 12.34;
double y = 23.45;
double z = min(x, y);
```

would produce the following log entries:

```
function: min<double>(a: double = 12.34, b: double = 23.45)
result: double = 12.34
```

It is true that the lines:

```
<< get_base_name_v<$reflect(a)> << " = "
<< get_base_name_v<$reflect(b)> << " = "
```

could be replaced by preprocessor stringization

```
<< BOOST_PP_STRINGIZE(a) << ": "
<< BOOST_PP_STRINGIZE(b) << ": "
```

or just hard coded strings, like

```
<< "a: "
<< "b: "
```

but the compiler would not force the programmer to change the macro parameter or the content of the string the if the parameters `a` and `b` were renamed for example to `first` and `second`. On the other hand, it *would* enforce the change if reflection was used.

Furthermore, with the *Function* concept and the context-dependent reflection features, even more would be possible; The function name and even the parameter names can be obtained from reflection and encapsulated into a function.

```
template <typename MetaFunction, typename ... P>
void log_function_exec(MetaFunction, const std::tuple<P&...& params) {
    log() << "function: "
        << get_display_name_v<MetaFunction>
        << std::endl;
```

```
// obtain the MetaParameter(s) from the MetaFunction
// and print them pairwise with the values from params.
for_each<get_parameters<MetaFunction>>(
    [&params](auto meta_param) {
        using MP = decltype(meta_param)::type;
        log() << get_base_name_v<MP> << ": "
            << std::get<get_position_v<MP>>(params)
            << std::endl;
    }
);
}

template <typename T>
T min(T a, T b) {
    log_function_exec($reflect(this::function), std::tie(a, b));
    /* ... */
}

template <typename T>
T max(T a, T b) {
    log_function_exec($reflect(this::function), std::tie(a, b));
    /* ... */
}

template <typename T>
T avg(T a, T b) {
    log_function_exec($reflect(this::function), std::tie(a, b));
    /* ... */
}
```

Logging is one of the use cases where *display names* are really useful. Consider for example that we want to log the execution of a function with the following signature:

```
std::string::iterator foo(std::string::iterator iter);
```

The log is much cleaner and informative if it contains the typedef name `iterator` instead of `const char*` on one platform, `__normal_iterator<pointer, std::basic_string<char>>` on another and something completely different on yet another. On some platforms the definition of `iterator` may even vary between build modes<sup>23</sup> which may lead to confusion.

### 5.3 Implementing common functions and operators

This use case was part of the “targeted use cases” in the committee’s call for compile-time reflection proposals [9]:

*There are many functions that generally consist of boilerplate code, performing some action for each member of a class. Such functions include equality operators, comparison operators, serialization*

---

<sup>23</sup>32-bit vs. 64-bit or debug vs. release

*functions, hash functions and swap functions.*

In other words for arbitrary structured type, for example:

```
struct S {
    int i;
    long l;
    float f;
};
```

we want to create equality or non-equality comparison function like:

```
bool S_equal(const S& a, const S& b) {
    bool result = true;
    result &= a.i == b.i;
    result &= a.l == b.l;
    result &= a.f == b.f;
    return result;
}
```

```
bool S_not_equal(const S& a, const S& b) {
    bool result = false;
    result |= a.i != b.i;
    result |= a.l != b.l;
    result |= a.f != b.f;
    return result;
}
```

or a hash function:

```
std::size_t S_hash(const S& a) {
    std::size_t result = 0u;
    result ^= std::hash<int>()(a.i);
    result ^= std::hash<long>()(a.l);
    result ^= std::hash<float>()(a.f);
    return result;
}
```

This is one of the many use cases where the `for_each` function described in section 6.1.1 comes in handy. The above could be implemented along the lines of:

```
template <typename T>
struct compare_data_members {
    const T& a;
    const T& b;
    bool& result;

    template <reflect::Object MetaDataMember>
    void operator()(MetaDataMember) const {
        auto mem_ptr = reflect::get_pointer_v<MetaDataMember>;
        result &= a.*mem_ptr == b.*mem_ptr;
    }
}
```

```
};

template <typename T>
bool generic_equal(const T& a, const T& b) {
    using metaT = $reflect(T);
    bool result = true;

    reflect::for_each<reflect::get_data_members_t<metaT>>(
        compare_data_members<T>{a, b, result}
    );

    return result;
}
```

If the reversible reflection feature described in section 6.2 was implemented then the helper could take advantage of it:

```
template <typename T>
struct compare_data_members {
    const T& a;
    const T& b;
    bool& result;

    template <reflect::Object MetaDataMem>
    void operator()(MetaDataMem) const {
        result &= a.$unreflect(MetaDataMem) == b.$unreflect(MetaDataMem);
    }
};
```

The helper could also be implemented by using a lambda function:

```
template <typename T>
std::size_t generic_hash(const T& a) {
    std::size_t result = 0u;

    reflect::for_each<reflect::get_data_members_t<$reflect(T)>>(
        [&result,&a](auto meta_dm) {
            using MetaDataMem = decltype(meta_dm)::type;
            using MetaT = reflect::get_type_t<MetaDataMem>;

            using T = reflect::get_reflected_type_t<MetaT>;
            // or T = $unreflect(metaT);

            auto mem_ptr = reflect::get_pointer_v<MetaDataMem>;

            result ^= std::hash<T>(a.*mem_ptr);
            // or ^= std::hash<T>(a.$unreflect(MetaDataMem));
        }
    );
};
```

```
    return result;
}
```

## 5.4 Enumerator to string and vice versa

This is another use case from the “targeted use cases” in the committee’s call for compile-time reflection proposals [9].

The goal is to automate the implementation of functions which for a given enumeration value, return the name of the enumeration value:

```
enum class E {
    a, b, c, d, e, f
};

string E_to_string(E value) {
    switch(value) {
        case E::a: return "a";
        case E::b: return "b";
        case E::c: return "c";
        case E::d: return "d";
        case E::e: return "e";
        case E::f: return "f";
    }
    return {};
}
```

or the other way around:

```
E string_to_E(const string& name) {
    if(name == "a") return E::a;
    if(name == "b") return E::b;
    if(name == "c") return E::c;
    if(name == "d") return E::d;
    if(name == "e") return E::e;
    if(name == "f") return E::f;

    // or throw here
    return {};
}
```

The Mirror reflection library [5] has several versions of enum-to-string and string-to-enum, based on top of the clang implementation of P0194R2. The following code snippet is adjusted to match the new wording:

```
template <typename Enum>
class enum_to_string {
private:
    template <typename ... MEC>
    struct _hlpr {
```

```
    static void _eat(bool ...) { }

    static auto _make_map(void) {
        using namespace std;

        map<Enum, string> res;
        _eat(res.emplace(
            reflect::get_constant_v<MEC>,
            string(reflect::get_base_name<MEC>())
        ).second...);
        return res;
    }
};
public:
    const std::string& operator()(Enum e) const {
        using namespace std;

        using ME = $reflect(Enum);
        using hlpr = reflect::unpack_sequence_t<
            reflect::get_enumerators_t<ME>,
            _hlpr
        >;
        static auto m = hlpr::_make_map();
        return m[e];
    }
};
```

and the string\_to\_enum utility:

```
template <typename Enum>
class string_to_enum {
private:
    template <typename ... MEC>
    struct _hlpr {
        static void _eat(bool ...) { }

        static auto _make_map(void) {
            using namespace std;

            map<string, Enum> res;
            _eat(res.emplace(
                string(reflect::get_base_name<MEC>()),
                reflect::get_constant_v<MEC>
            ).second...);
            return res;
        }
    };
public:
    Enum operator()(const std::string& s) const {
```



```
    using namespace std;

    using ME = $reflect(Enum);
    using hlpr = reflect::unpack_sequence_t<
        reflect::get_enumerators_t<ME>,
        _hlpr
    >;
    static auto m = hlpr::_make_map();
    auto p = m.find(s);
    if(p == m.end()) {
        throw runtime_error("Invalid enumerator name");
    }
    return p->second;
}
};
```

## 5.5 Simple serialization

We need to serialize instances of assorted classes into a structured external format like XML, JSON, XDR or even into a format like Graphviz dot for the purpose of creating a visualization of a static class or dynamic object hierarchy or graph.

Reflection makes this task trivial:

```
#include <reflect>
#include <iostream>

template <typename T>
std::ostream& value_to_json(std::ostream& out, const T& v);

template <typename ... T>
void eat(T ... ) { }

template <typename Metaobjects, std::size_t I, typename T>
int field_to_json(std::ostream& out, const T& v) {
    using meta_F = std::reflect::get_element_t<Metaobjects, I>;

    if(I > 0) out << ", ";

    out << '"' << std::reflect::get_base_name_v<meta_F> << "\"": ";

    value_to_json(out, (v.*std::reflect::get_pointer_v<meta_F>));

    return 0;
}

template <typename Metaobjects, std::size_t ... I, typename T>
void fields_to_json(std::ostream& out, const T& v, std::index_sequence<I...>) {
```

```
    eat(field_to_json<Metaobjects, I>(out, v)...);
}

template <typename MO, typename T>
std::ostream& reflected_to_json(std::ostream& out, const T& v, std::true_type) {
    out << "{";

    using meta_DMs = std::reflect::get_data_members_t<MO>;
    fields_to_json<meta_DMs>(
        out, v,
        std::make_index_sequence<
            std::reflect::get_size_v<meta_DMs>
        >()
    );

    out << "}";
    return out;
}

template <typename MO, typename T>
std::ostream& reflected_to_json(std::ostream& out, const T& v, std::false_type) {
    return out << v;
}

template <typename T>
std::ostream& value_to_json(std::ostream& out, const T& v) {
    using meta_T = $reflect(T);
    return reflected_to_json<meta_T>(
        out, v,
        std::reflect::Record<meta_T>()
    );
};

struct point { float x, y, z; };
struct triangle { point a, b, c; };

struct tetrahedron {
    triangle base;
    point apex;
};

int main(void) {
    using namespace std;

    tetrahedron t{
        {{0.f,0.f,0.f}, {1.f,0.f,0.f}, {0.f,0.f,1.f}},
        {0.f,1.f,0.f}
    };
};
```

```
std::cout << "\\t\\": ";
value_to_json(std::cout, t);
std::cout << "]" << std::endl;

return 0;
}
```

The above is a variation of an already working code from the implementation of P0194R2, which produces the following output:

```
{"t": {"base": {"a": {"x": 0, "y": 0, "z": 0}, "b": {"x": 1, "y": 0, "z": 0}, \
"c": {"x": 0, "y": 0, "z": 1}}, "apex": {"x": 0, "y": 1, "z": 0}}}
```

## 5.6 Structure data member transformations

We need to create a new structure which has data members with the same or similar names as an original structure, but we need to change some of the properties of the data members, usually their types.

For example we need to transform a structure like:

```
struct foo {
    bool b;
    char c;
    double d;
    float f;
    string s;
};
```

into

```
struct rdbms_table_placeholder_foo {
    column_placeholder_t<bool> b;
    column_placeholder_t<char> c;
    column_placeholder_t<double> d;
    column_placeholder_t<float> f;
    column_placeholder_t<string> s;
};
```

or create a structure-of-arrays, which was one of the “targeted use cases” from the committee’s CFP [9]:

```
struct soa_foo {
    vector<bool> vec_b;
    vector<char> vec_c;
    vector<double> vec_d;
    vector<float> vec_f;
    vector<string> vec_s;
};
```

The primary obstacle to implementing this use case with the help of reflection is at the moment the fact that we do not have the ability to create C++ identifiers “programmatically”, at least not without the help of the preprocessor.

We either have to add the ability to create identifiers from compile-time strings which may be fairly complicated, or look for some simpler workarounds.

First, let’s say that we have a magic operator, like `$identifier` as described in 6.3.1 , which “creates” an identifier from a compile-time string, so that for example:

```
$identifier("long") $identifier("foo")(int i, int j);
```

would be equivalent to

```
long foo(int i, int j);
```

To copy a name of another declaration from a metaobject reflecting it, we can use the `get_base_name` operation, which returns a `constexpr` array of chars:

```
struct bar {
    int foo;
};
```

```
using meta_bar_foo = $reflect(bar::foo);
```

```
long $identifier(reflect::get_base_name_v<meta_bar_foo>)(int i, int j);
```

In order to do anything more complex than just copying the identifiers of other declarations we would need a compile-time string manipulation utility implementing for example functions for compile-time string concatenation, like `ct_concat`:

```
int $identifier(ct_concat("get_", reflect::get_base_name_v<meta_bar_foo>))(const bar&);
```

which would be equivalent to:

```
int get_foo(const bar&);
```

With the help of multiple inheritance and the `unpack_sequence` helper template described in section 6.1.2, we can create a new structure that is nearly equivalent to `soa_foo` via metaprogramming:

```
template <typename MetaDataMember>
struct soa_single_member {
    // vector<T> Xs;
    vector<reflect::get_reflected_type_t<reflect::get_type_t<MetaDataMember>>>
        $identifier(ct_concat(reflect::get_base_name_v<MetaDataMember>, "s"));

    /* constructors, forwarding parameters to the vector, ... */
};
```

```
template <typename ... MetaDataMembers>
struct soa_inherit_all
: soa_single_member<MetaDataMembers>... {
    /* constructors forwarding parameters to the inherited soa_single_members */
};
```

```
template <typename T>
struct soa
: reflect::unpack_sequence_t<
  reflect::get_data_members_t<$reflect(T)>,
  soa_inherit_all
> {
/* constructors forwarding parameters to soa_inherit_all */
};

using soa_foo = soa<foo>;
```

For a more detailed discussion on the possibilities of identifier generation see section 6.3 and also section 6.4.

## 5.7 Simple examples of usage

This section shows several simple examples of usage working with the experimental implementation of P0194R2, but transformed to use the new wording from P0194R3.

### 5.7.1 Scope reflection

```
#include <reflect>
#include <iostream>

namespace foo {

struct bar {
  typedef int baz;
};

} // namespace foo

typedef long foobar;

int main(void) {
  using namespace std;
  using namespace std::reflect;

  using meta_int          = $reflect(int);
  using meta_foo_bar      = $reflect(foo::bar);
  using meta_foo_bar_baz = $reflect(foo::bar::baz);
  using meta_foobar       = $reflect(foobar);

  static_assert( ScopeMember<meta_int>);
  static_assert( ScopeMember<meta_foo_bar>);
  static_assert( ScopeMember<meta_foo_bar_baz>);
```

```
static_assert( ScopeMember<meta_foobar>);

using meta_int_s      = get_scope_t<meta_int>;
using meta_foo_bar_s  = get_scope_t<meta_foo_bar>;
using meta_foo_bar_baz_s = get_scope_t<meta_foo_bar_baz>;
using meta_foobar_s   = get_scope_t<meta_foobar>;

static_assert( Scope<meta_int_s>);
static_assert( Scope<meta_foo_bar_s>);
static_assert( Scope<meta_foo_bar_baz_s>);
static_assert( Scope<meta_foobar_s>);

static_assert( Namespace<meta_int_s>);
static_assert( GlobalScope<meta_int_s>);
static_assert( Namespace<meta_foo_bar_s>);
static_assert(!GlobalScope<meta_foo_bar_s>);
static_assert( Type<meta_foo_bar_baz_s>);
static_assert( Class<meta_foo_bar_baz_s>);
static_assert(!Namespace<meta_foo_bar_baz_s>);
static_assert( Namespace<meta_foobar_s>);
static_assert( GlobalScope<meta_foobar_s>);
static_assert(!Class<meta_foobar_s>);

cout << get_base_name_v<meta_foo_bar_baz> << endl;
cout << get_base_name_v<meta_foo_bar_baz_s> << endl;
cout << get_base_name_v<meta_foo_bar_s> << endl;

return 0;
}
```

Output:

```
baz
bar
foo
```

## 5.7.2 Namespace reflection

```
#include <reflect>
#include <iostream>

namespace foo { namespace bar { } }

namespace foobar = foo::bar;

int main(void) {
    using namespace std;
    using namespace std::reflect;
}
```

```
using meta_foo = $reflect(foo);

static_assert( Object<meta_foo>);

static_assert(!GlobalScope<meta_foo>);
static_assert( Namespace<meta_foo>);
static_assert(!Type<meta_foo>);
static_assert(!Alias<meta_foo>);

static_assert( Named<meta_foo>);
static_assert( ScopeMember<meta_foo>);
cout << get_base_name_v<meta_foo> << endl;

using meta_foo_bar = $reflect(foo::bar);

static_assert( Object<meta_foo_bar>);

static_assert(!GlobalScope<meta_foo_bar>);
static_assert( Namespace<meta_foo_bar>);
static_assert(!Type<meta_foo_bar>);
static_assert(!Alias<meta_foo_bar>);

static_assert( Named<meta_foo_bar>);
static_assert( ScopeMember<meta_foo_bar>);
cout << get_base_name_v<meta_foo_bar> << endl;

using meta_foobar = $reflect(fooobar);

static_assert( Object<meta_foobar>);

static_assert(!GlobalScope<meta_foobar>);
static_assert( Namespace<meta_foobar>);
static_assert(!Type<meta_foobar>);
static_assert( Alias<meta_foobar>);

static_assert( Named<meta_foobar>);
static_assert( ScopeMember<meta_foobar>);
cout << get_base_name_v<meta_foobar> << " a.k.a ";
cout << get_base_name_v<get_aliased_t<meta_foobar>> << endl;

return 0;
}
```

Output:

```
foo
bar
fooobar a.k.a bar
```

### 5.7.3 Type reflection

```
#include <reflect>
#include <iostream>

int main(void) {
    using namespace std;
    using namespace std::reflect;

    using meta_unsigned = $reflect(unsigned);

    static_assert( Object<meta_unsigned>);
    static_assert( Type<meta_unsigned>);
    static_assert(!Alias<meta_unsigned>);

    static_assert( is_same_v<
        get_reflected_type_t<meta_unsigned>,
        unsigned
    >);

    static_assert( Named<meta_unsigned>);
    cout << get_base_name_v<meta_unsigned> << endl;

    using meta_ptr_unsigned = $reflect(unsigned*);
    static_assert( Named<meta_ptr_unsigned>);
    cout << get_base_name_v<meta_ptr_unsigned> << endl;

    return 0;
}
```

Output:

```
unsigned int
unsigned int
```

### 5.7.4 Typedef reflection

```
#include <reflect>
#include <iostream>

namespace foo {

    typedef int bar;
    using baz = bar;

} // namespace foo

int main(void) {
```



```
using namespace std;
using namespace std::reflect;

using meta_foo_baz = $reflect(foo::baz);

static_assert( Object<meta_foo_baz>);
static_assert( Type<meta_foo_baz>);
static_assert( Alias<meta_foo_baz>);

static_assert( is_same_v<
    get_reflected_type_t<meta_foo_baz>,
    foo::baz
>);

static_assert( Named<meta_foo_baz>);
cout << get_base_name_v<meta_foo_baz> << endl;

using meta_foo_bar = get_aliased_t<meta_foo_baz>;

static_assert( Object<meta_foo_bar>);
static_assert( Type<meta_foo_bar>);
static_assert( Alias<meta_foo_bar>);

static_assert( is_same_v<
    get_reflected_type_t<meta_foo_bar>,
    foo::bar
>);

static_assert( Named<meta_foo_bar>);
cout << get_base_name_v<meta_foo_bar> << endl;

using meta_int = get_aliased_t<meta_foo_bar>;

static_assert( Object<meta_int>);
static_assert( Type<meta_int>);
static_assert(!Alias<meta_int>);

static_assert( is_same_v<
    get_reflected_type_t<meta_int>,
    int
>);

static_assert( Named<meta_int>);
cout << get_base_name_v<meta_int> << endl;

return 0;
}
```

Output:

```
baz
bar
int
```

### 5.7.5 Class alias reflection

```
#include <reflect>
#include <iostream>

struct foo { };
using bar = foo;

int main(void) {
    using namespace std;
    using namespace std::reflect;

    using meta_bar = $reflect(bar);

    static_assert( Object<meta_bar>);
    static_assert( Type<meta_bar>);
    static_assert( Class<meta_bar>);
    static_assert( Alias<meta_bar>);

    static_assert( is_same_v<get_reflected_type_t<meta_bar>, bar>);

    static_assert( Named<meta_bar>);
    cout << get_base_name_v<meta_bar> << endl;

    using meta_foo = get_aliased_t<meta_bar>;

    static_assert( Object<meta_foo>);
    static_assert( Type<meta_foo>);
    static_assert( Class<meta_foo>);
    static_assert(!Alias<meta_foo>);

    static_assert( is_same_v<get_reflected_type_t<meta_foo>, foo>);

    static_assert( Named<meta_foo>);
    cout << get_base_name_v<meta_foo> << endl;

    return 0;
}
```

Output:

```
bar
```

foo

Note that `meta_bar` is both an *Alias* and a *Class*.

### 5.7.6 Class data members (1)

```
#include <reflect>
#include <iostream>

struct foo {
private:
    int _i, _j;
public:
    static constexpr const bool b = true;
    float x, y, z;
private:
    static double d;
};

int main(void) {
    using namespace std;
    using namespace std::reflect;

    using meta_foo = $reflect(foo);

    // public data members
    using meta_data_mems = get_public_data_members_t<meta_foo>;

    static_assert(Object<meta_data_mems>);
    static_assert(ObjectSequence<meta_data_mems>);

    cout << get_size_v<meta_data_mems> << endl;

    // 0-th (public) data member
    using meta_data_mem0 = get_element_t<meta_data_mems, 0>;

    static_assert(Object<meta_data_mem0>);
    static_assert(Variable<meta_data_mem0>);
    static_assert(Typed<meta_data_mem0>);

    cout << get_base_name_v<meta_data_mem0> << endl;

    // 2-nd (public) data member
    using meta_data_mem2 = get_element_t<meta_data_mems, 2>;

    static_assert(Object<meta_data_mem2>);
    static_assert(Variable<meta_data_mem2>);
```

```
static_assert(Typed<meta_data_mem2>);

cout << get_base_name_v<meta_data_mem2> << endl;

// all data members
using meta_all_data_mems = get_data_members_t<meta_foo>;

static_assert(Object<meta_all_data_mems>);
static_assert(ObjectSequence<meta_all_data_mems>);

cout << get_size_v<meta_all_data_mems> << endl;

// 0-th (overall) data member
using meta_all_data_mem0 = get_element_t<meta_all_data_mems, 0>;

static_assert(Object<meta_all_data_mem0>);
static_assert(Variable<meta_all_data_mem0>);
static_assert(Typed<meta_all_data_mem0>);

cout << get_base_name_v<meta_all_data_mem0> << endl;

// 3-rd (overall) data member
using meta_all_data_mem3 = get_element_t<meta_all_data_mems, 3>;

static_assert(Object<meta_all_data_mem3>);
static_assert(Variable<meta_all_data_mem3>);
static_assert(Typed<meta_all_data_mem3>);

cout << get_base_name_v<meta_all_data_mem3> << endl;

return 0;
}
```

This produces the following output:

```
4
b
y
7
_i
x
```

### 5.7.7 Class data members (2)

```
#include <reflect>
#include <iostream>

struct foo {
```

```
private:
    int _i, _j;
public:
    static constexpr const bool b = true;
    float x, y, z;
private:
    static double d;
};

template <typename ... T>
void eat(T ... ) { }

template <typename Metaobjects, std::size_t I>
int do_print_data_member(void) {
    using namespace std;

    typedef reflect::get_element_t<Metaobjects, I> metaobj;

    cout    << I << ": "
            << (reflect::is_public_v<metaobj>?"public":"non-public")
            << " "
            << (reflect::is_static_v<metaobj>?"static":"" )
            << " "
            << reflect::get_base_name_v<reflect::get_type_t<metaobj>>
            << " "
            << reflect::get_base_name_v<metaobj>
            << endl;

    return 0;
}

template <typename Metaobjects, std::size_t ... I>
void do_print_data_members(std::index_sequence<I...>) {
    eat(do_print_data_member<Metaobjects, I>()...);
}

template <typename Metaobjects>
void do_print_data_members(void) {
    using namespace std;

    do_print_data_members<Metaobjects>(
        make_index_sequence<
            reflect::get_size_v<Metaobjects>
        >()
    );
}

template <typename MetaClass>
```

```
void print_data_members(void) {
    using namespace std;

    cout<< "Public data members of " << reflect::get_base_name_v<MetaClass> << endl;

    do_print_data_members<reflect::get_public_data_members_t<MetaClass>>();
}

template <typename MetaClass>
void print_all_data_members(void) {
    using namespace std;

    cout << "All data members of " << reflect::get_base_name_v<MetaClass> << endl;

    do_print_data_members<reflect::get_data_members_t<MetaClass>>();
}

int main(void) {
    print_data_members<$reflect(foo)>();
    print_all_data_members<$reflect(foo)>();

    return 0;
}
```

This program produces the following output:

```
Public data members of foo
0: public static bool b
1: public float x
2: public float y
3: public float z
All data members of foo
0: non-public int _i
1: non-public int _j
2: public static bool b
3: public float x
4: public float y
5: public float z
6: non-public static double d
```

### 5.7.8 Class data members (3)

```
#include <reflect>
#include <iostream>

struct A {
    int a;
};
```

```
class B : public A {
private:
    bool b;
};

class C : public B {
public:
    char c;
};

int main(void) {
    using namespace std;
    using namespace std::reflect;

    using meta_A = $reflect(A);
    using meta_B = $reflect(B);
    using meta_C = $reflect(C);

    cout << get_size_v<get_public_data_members_t<meta_A>> << endl;
    cout << get_size_v<get_public_data_members_t<meta_B>> << endl;
    cout << get_size_v<get_public_data_members_t<meta_C>> << endl;

    cout << get_size_v<get_data_members_t<meta_A>> << endl;
    cout << get_size_v<get_data_members_t<meta_B>> << endl;
    cout << get_size_v<get_data_members_t<meta_C>> << endl;

    return 0;
}
```

This program produces the following output:

```
1
0
1
1
1
1
1
```

Note that neither the result of `get_data_members` nor the result of `get_public_data_members` includes the inherited data members.

## 6 Future extensions

In this section we would like to discuss possible future extensions and the evolution of reflection.

It is important to emphasize that the features discussed in this section *are not* part of the initial implementation of reflection as proposed in the P0194Rx papers.

## 6.1 Additional utilities

In this section we describe several useful metaprogramming utilities which could be added later to the standard library to simplify the use of the basic reflection primitives in certain cases.

### 6.1.1 for\_each

The `for_each` function should sequentially call a specified `UnaryFunction` on instances of metaobjects from a *ObjectSequence*. It should *be equivalent* to the following:

```
namespace reflect {  
    template <typename MetaObjectSequence, typename UnaryFunction>  
    void for_each(UnaryFunction func) {  
        func(Metaobject1{});  
        func(Metaobject2{});  
        /* ... */  
        func(MetaobjectN{});  
    }  
} // namespace reflect
```

For example:

```
reflect::for_each<reflect::get_data_members_t<$reflect(my_class)>>(  
    [](auto idmo) {  
        using MO = decltype(idmo);  
        ...  
    }  
);
```

### 6.1.2 unpack\_sequence\_if

We already have the `unpack_sequence` template which instantiates the provided variadic template with metaobjects from a *ObjectSequence*.

```
namespace reflect {  
    template <typename MetaObjectSequence, template <typename ...> class Template>  
    struct unpack_sequence {  
        typedef Template<Metaobject1, Metaobject2, ... MetaobjectN> type;  
    };  
  
    template <typename MetaObjectSequence, template <typename ...> class Template>  
    using unpack_sequence_t =  
        typename unpack_sequence<MetaobjectSequence, Template>::type;  
} // namespace reflect
```

We could also add the following variant of `unpack_sequence` to be able to unpack only metaobjects satisfying a given predicate:



```
namespace reflect {  
    template <  
        typename MetaObjectSequence,  
        template <typename ...> class Template,  
        template <typename> class Predicate  
    >  
    struct unpack_sequence_if {  
        typedef Template<...> type;  
    };  
  
    template <  
        typename MetaObjectSequence,  
        template <typename ...> class Template,  
        template <typename> class Predicate  
    >  
    using unpack_sequence_if_t =  
        typename unpack_sequence_if<  
            MetaobjectSequence,  
            Template,  
            Predicate  
        >::type;  
} // namespace reflect
```

## 6.2 Reversing reflection

The `$reflect` operator allows to obtain the meta-level representation of a base-level declaration. But what if we wanted to do the opposite: to get back the original declaration from a metaobject?

This of course would make sense only for a subset of metaobjects, so let's define a new concept like *Reversible*.

One way to achieve this would be to add new operator – `$unreflect`.

Let's look at some possible use cases for this feature. The most trivial one is to replace the `get_reflected_type` template from the interface of *Type*:

```
using MT = $reflect(int);  
  
// ... lots of other code here ...  
  
// Now we arrived to a point where we need  
// to get the base-level type reflected by 'MT'.  
  
// Instead of using get_reflected_type ...  
using T = get_reflected_type_t<MT>;  
  
// we could do:  
static_assert(reflect::Reversible<MT> && reflect::Type<MT>);
```

```
using T = $unreflect(MT);
static_assert(is_same_v<T, int>);
```

But that's far from being the only use case. If we could get back to the original namespace or variable;

```
namespace foo {
    std::string str;

    void bar(const std::string&);
} // namespace foo

using MV = $reflect(foo::str); // Variable

// ... lots of code here ...
// using namespace foo;
using namespace $unreflect(get_scope_t<MV>);

// [foo::]bar(foo::str);
bar($unreflect(MV));
```

or even selecting whole namespaces (meta-)programmatically;

```
namespace foo {
    void func1(const std::string&);
    void func2(int, int, int);
    int func3(long, short, bool);
} // namespace foo

namespace bar {
    void func1(const std::string&);
    void func2(int, int, int);
    int func3(long, short, bool);
} // namespace bar

template <typename MN>
void algorithm(const string& str, int i, int j) {
    // [foo|bar]::func1
    $unreflect(MN)::func1(str);

    // [foo|bar]::func2([foo|bar]::func3(i*j, 64, i == j), i, j);
    $unreflect(MN)::func2($unreflect(MN)::func3(i*j, 64, i == j), i, j);
}

void func(const string& str, int i, int j, bool want_foo) {
    if(want_foo) {
        algorithm<$reflect(foo)>(str, i, j);
    } else {
        algorithm<$reflect(bar)>(str, i, j);
    }
}
```

```
    }  
}
```

or get back to the original function;

```
namespace foo {  
    void bar(const std::string&);  
} // namespace foo  
  
using MF = $reflect(foo::bar); // Function  
  
// ... lots of code here ...  
// foo::bar("hello");  
$unreflect(MF)("hello");
```

or back to the original class data member;

```
struct my_struct {  
    int i;  
    float f;  
};  
  
// DataMember  
using MDM = $reflect(foo::bar::i);
```

```
my_struct x {123, 45.67f};
```

```
assert(x.i == 123);  
// x.i = 234;  
x.$unreflect(MDM) = 234;  
assert(x.i == 234);
```

or go back to the original template;

```
// Template  
using MTpl = $reflect(std::pair);  
  
// std::pair<int, std::string> p;  
$unreflect(MTpl)<int, std::string> p{10, "Hi!"};
```

and so on.

Combined with the fact that all metaobjects are first-class entities this would make a very powerful feature, which would essentially allow to reify all reflectable declarations at the meta-level, even those which are second-class without actually reifying them at the base-level<sup>24</sup>

### 6.3 Generating identifiers programmatically

Some of the use cases from the committee's CFP, for example the one described in section 5.6, depend on the ability to generate identifiers programmatically.

---

<sup>24</sup>At the cost of the round-trip through the meta-level.

### 6.3.1 Identifier from a generic compile-time string

We will start by the most powerful, but also the most difficult to implement option – an operator, say `$identifier(const char (&)[N])`<sup>25</sup>, which “creates” an identifier from an arbitrary compile-time string, so that:

```
$identifier("std")::$identifier("size_t")
$identifier("foo")(int $identifier("i"), int j);
```

would be equivalent to

```
std::size_t foo(int i, int j);
```

Remember that the string-returning operations defined on the named metaobjects, for example the `reflect::get_base_name` operation, return `constexpr` arrays of `chars`. This means that we can use them to copy the name of a declaration, from a metaobject reflecting that declaration:

```
struct bar {
    int foo;
};
```

```
using meta_bar_foo = $reflect(bar::foo);
```

```
std::size_t $identifier(reflect::get_base_name_v<meta_bar_foo>)(int i, int j);
```

which is again equivalent to:

```
std::size_t foo(int i, int j);
```

It’s important to note, that in order to do anything more complex than just copying the identifiers of other declarations, we would also need some compile-time string manipulation utility. For example to create identifiers like `get_%` or `set_%` where `%` is a declaration name, we would need to have a function for compile-time string concatenation, like `ct_concat`:

```
using meta_bar_foo = $reflect(bar::foo);
```

```
int $identifier(ct_concat("get_", reflect::get_base_name_v<meta_bar_foo>))(const bar&);
```

which would be equivalent to:

```
int get_foo(const bar&);
```

While being very appealing and powerful, implementing this feature may also prove to be challenging, especially taking the different phases of compilation into account.

### 6.3.2 Identifier formatting

Another option is a simplified version of the operator `$identifier`, which would take a format string *literal* and optionally a pack of *Named* metaobjects:

```
struct bar {
    int foo;
```

---

<sup>25</sup>We do not insist on this particular name in case it should be implemented.

```
};

using meta_bar = $reflect(bar);
using meta_foo = $reflect(bar::foo);

$identifier("int")
$identifier("get_%1_%2", meta_bar, meta_foo)(const $identifier("%1", meta_bar)&);
```

which would result in:

```
int get_bar_foo(const bar&);
```

In this case we would only allow string *literals*, not arbitrary `constexpr` string-returning expressions and the identifier formatting would be handled inside of the compiler.

### 6.3.3 named\_data\_member and named\_member\_typedef

If we don't want to implement the operator `$identifier` or expose it to the users directly, the third option is to add new operations like `named_member_typedef` and `named_data_member` for *Named* metaobjects *equivalent* to the following:

```
template <MetaNamed MO, typename X>
struct named_member_typedef {
    struct type {
        // same as
        // typedef X $identifier(reflect::get_base_name_v<MO>);
        typedef X __builtin_identifier_of(MO);
    };
};

template <MetaNamed MO, typename X>
struct named_data_member {
    struct type {
        // same as
        // X $identifier(reflect::get_base_name_v<MO>);
        X __builtin_identifier_of(MO);

        template <typename ... P>
        type(P&& ... p)
            : __builtin_identifier_of(MO)(forward<P>(p)...)
        { }
    };
};

template <MetaNamed MO, typename X>
using named_member_typedef_t = typename named_member_typedef<MO, X>::type;

template <MetaNamed MO, typename X>
using named_data_member_t = typename named_data_member<MO, X>::type;
```

So for example the specializations of `named_member_typedef` and `named_data_member` for the metaobject reflecting `bar::foo` would look like this:

```
template <typename X>
struct named_member_typedef<$reflect(bar::foo), X> {
    struct type {
        typedef X foo;
    };
};

template <typename X>
struct named_data_member<$reflect(bar::foo), X> {
    struct type {
        X foo;

        template <typename ... P>
        type(P&& ... p)
            : foo(forward<P>(p)...)
        { }
    };
};
```

With some clever metaprogramming we can combine specializations of `named_data_member_t<...>` or `named_member_typedef_t<...>` into structures having member types or variables with names, matching other existing declarations.

Note that this option does not give use the ability to change the identifiers, just copy them and it's generally less powerful. On the other hand, implementing it may be much less complicated than the first two options.

## 6.4 Variadic composition

Another feature which<sup>26</sup> has the potential to greatly simplify the implementation of several important use cases, including the structure data member transformations, is *variadic composition*.

We already have variadic *inheritance* that is used in the implementation of tuples for example, but we don't have variadic *composition*.

The reason for this is simple – every class data member needs to have a name which is unique in the scope of its class and since we don't have the ability to generate identifiers, we cannot implement variadic composition.

For instance, the structure-of-arrays transformation described in section 5.6 could be simplified by using variadic composition, together with the “identifier formatting operator” described in section 6.3.2 and the `unpack_sequence` helper template described in section 6.1.2:

```
template <DataMember ... MDM>
struct soa_helper {
    std::vector<reflect::get_reflected_type_t<reflect::get_type_t<MDM>>>
```

---

<sup>26</sup>together with the ability to generate identifiers

```
    $identifier("vec_%1", MDM)...;
};

template <typename T>
using soa = reflect::unpack_sequence_t<
    reflect::get_data_members_t<$reflect(T)>,
    soa_helper
>;
```

With the “reversible reflection” feature described in section 6.2, the `soa_helper` could be implemented as:

```
template <DataMember ... MDM>
struct soa_helper {
    std::vector<$unreflect(reflect::get_type_t<MDM>)>
    $identifier("vec_%1", MDM)...;
};
```

## 6.5 Specifier reflection

There are several use-cases where the reflection of specifiers would be useful.

We might want for example to “copy” the storage class specifier of one variable to other variables

```
thread_local int dummy_v;
using m_storage = get_storage_class_specifier_t<$reflect(dummy_v)>;

$unreflect(m_storage) long foo_v;    // thread_local long foo_v;
$unreflect(m_storage) double bar_v; // thread_local double bar_v;
$unreflect(m_storage) short baz_v;   // thread_local short baz_v;
```

or the linkage specifier from one function to several other functions:

```
static inline void dummy_f(void);
using m_linkage = get_linkage_specifier_t<$reflect(dummy_f)>;

$unreflect(m_linkage) volatile int foo_f(int); // static volatile int foo_f(int);
$unreflect(m_linkage) long bar_f(short, short); // static long bar_f(short, short);
```

The open question at the moment is whether we should allow “direct” reflection of specifiers since some keywords like `inline` or `virtual` have several different meanings:

```
#ifdef IMPLEMENT
using m_linkage = $reflect(static);
#else
using m_linkage = $reflect(extern);
#endif

$unreflect(m_linkage) volatile int foo_f(int);
$unreflect(m_linkage) long bar_f(short, short);
```

## 6.6 Context-dependent reflection

One of the interesting potential future extensions of reflection is *context-dependent reflection*. Currently the operands of `$reflect` are names of the base-level entities which we want to reflect.

Context-dependent reflection would allow to obtain metadata based on the context in which the reflection operator is invoked, instead of on the declaration name.

This could take the form of either something simple like:

```
$reflect()
```

which would reflect the current scope or even the current position in the source code, or several new expressions could be added as valid operands for `$reflect`, allowing to reflect the current namespace, class, template or function as described below.

### 6.6.1 Namespaces

If the `this::namespace` expression was used as the operand of the reflection operator, then it would return a *Namespace* reflecting the innermost namespace inside of which the reflection operator was invoked.

For example:

```
using _meta_gs = $reflect(this::namespace);
```

reflects the global scope namespace and is equivalent to

```
using _meta_gs = $reflect(::);
```

For named namespaces:

```
namespace foo {
    using _meta_foo = $reflect(this::namespace);

    namespace bar {
        using _meta_foo_bar = $reflect(this::namespace);
    } // namespace bar
} // namespace foo
```

### 6.6.2 Classes

If the `this::class` expression was used as the argument of the reflection operator, then it would return a *Class* reflecting the class inside of which the reflection operator was invoked.

For example:

```
struct foo {
    const char* _name;
```



```
// reflects foo
using _meta_foo1 = $reflect(this::class);

foo(void)
: _name(get_base_name_v<$reflect(this::class)>())
{ }

void f(void) {
    // reflects foo
    using _meta_foo2 = $reflect(this::class);
}

double g(double, double);

struct bar {
    // reflects foo::bar
    using _meta_foo_bar = $reflect(this::class);
};

};

double foo::g(double a, double b) {
    // reflects foo
    using _meta_foo3 = $reflect(this::class);
    return a+b;
}

class baz {
private:
    using _meta_baz = $reflect(this::class);
};

using _meta_cls = $reflect(this::class); // error: not used inside of a class.
```

### 6.6.3 Functions

If the `this::function` expression was used as the argument of the reflection operator, then it would return a *Function* reflecting the function or operator inside of which the reflection operator was invoked.

For example:

```
void foobar(void) {
    // reflects this particular overload of the foobar function
    using _meta_foobar = $reflect(this::function);
}

int foobar(int i) {
    // reflects this particular overload of the foobar function
```

```
    using _meta_foobar = $reflect(this::function);
    return i+1;
}

class foo {
private:
    void f(void) {
        // reflects this particular overload of foo::f
        using _meta_foo_f = $reflect(this::function);
    }

    double f(void) {
        // reflects this particular overload of foo::f
        using _meta_foo_f = $reflect(this::function);
        return 12345.6789;
    }
public:
    foo(void) {
        // reflects this constructor of foo
        using _meta_foo_foo = $reflect(this::function);
    }

    friend bool operator == (foo, foo) {
        // reflects this operator
        using _meta_foo_eq = $reflect(this::function);
    }

    using _meta_fn = $reflect(this::function); // error
};

using _meta_fn = $reflect(this::function); // error
```

#### 6.6.4 Templates

If the `this::template` expression was used as the argument of the reflection operator, then it would return a *Template* reflecting the class (or function) template inside of which the reflection operator was invoked.

For example:

```
template <typename First, typename Second, typename Third>
struct triplet {
    // reflects the template not the instantiation
    using _meta_tpl = $reflect(this::template);

    // reflects the instantiation not the template
    using _meta_cls = $reflect(this::class);
};
```

## 6.7 Metaobject unique ID

In order to accommodate different programming paradigms, it may be advantageous to represent metaobjects as both types *and* optionally also as `constexpr values` and have the ability to convert between these representations.

The compile-time value representation of a metaobject could be defined as:

```
namespace reflect {  
    class object_uid {  
    public:  
        constexpr object_uid(void) noexcept;  
        constexpr object_uid(const object_uid&) noexcept;  
  
        friend constexpr  
        bool operator == (object_uid, object_uid) noexcept;  
  
        friend constexpr  
        bool operator != (object_uid, object_uid) noexcept;  
  
        friend constexpr  
        bool operator < (object_uid, object_uid) noexcept;  
    };  
} // namespace reflect
```

The actual identifier could internally be something like a constant value of `int`, `std::size_t` or `std::intptr_t` storing an index, a hash or an encoded (and possibly encrypted) pointer to the internal representation of the metaobject in the compiler or something similar. This would however be a hidden implementation detail.

The advantage of the `object_uid` is that we could represent heterogeneous metaobjects – different types, as `constexpr` instances of a homogeneous type and store them in `constexpr` data structures and use them with `constexpr` functions.

Instances of `object_uid` could be queried from a metaobject by a new operation, implemented by the following template:

```
namespace reflect {  
    template <Object T>  
    struct get_uid {  
        constexpr const object_uid value = /* ... */;  
    };  
  
    template <Object T>  
    constexpr object_uid get_uid_v = get_uid<T>::value;  
} // namespace reflect
```

As long as an instance of `reflect::object_uid` stayed `constexpr`, it would also be possible to convert it back to a metaobject (type), for example by using a new version of the reflection operator

`$reflect:`

```
// pseudocode  
constexpr Object $reflect(reflect::object_uid);
```

This feature would make the use of reflection more convenient in some use cases or under some paradigms and could help to simplify the implementation of the higher-level façades.

The experimental implementation described in section 8 already represents metaobjects as integral constants and makes the addition of the functionality described above trivial.

## 7 Issues

### 7.1 Interaction with attributes

**Q:** *Should we even reflect attributes? If so, how will the reflection of generalized attributes look like?*

Currently generalized attributes are used *mostly* as hints to the compiler. They can help the compiler to do some optimizations<sup>27</sup> or to indicate that the user really means to do something what in other circumstances could be a bug<sup>28</sup> that the compiler warns about, etc.

Some people argue that generalized attributes are not supposed to have any semantic effects – a compiler should be able to completely strip a program of all attributes and it should not have any effects on the behavior of the program.

Attributes could however be viewed more broadly as a generic mechanism for annotating declarations<sup>29</sup> in the code, without explicitly saying that the recipient can only be the compiler. There are valid use cases for user annotations and since the annotations usually provide additional conceptual (meta-)information about a declaration, reflection is a natural mechanism for accessing the annotations.

We favor the view that on the base-level the compiler should be able to strip attributes<sup>30</sup>, but it should reflect them on the meta-level on demand.

Having covered that, the reflection of the “simple” attributes like `[[attr1]]` or `[[namespace::attr2]]` could be pretty straightforward:

We add a *Attribute* concept, make it conform to *Named* and *Scoped* concepts and then add an operation to the interface of *Object* like `get_attributes`, returning a sequence of metaobjects reflecting the attributes on a base-level declaration.

The complication is that attributes can have nearly arbitrary parameters; `[[probably(true)]]`, `[[deprecated("reason")]]`, `[[visibility(hidden)]]`, `[[gnu::aligned(64)]]`. The reflection of such attributes and their arguments is currently unresolved.

---

<sup>27</sup>for example the `probably(true)` or `carries_dependency` attributes

<sup>28</sup>like the `fallthrough` or `maybe_unused` attributes

<sup>29</sup>Associating one or more constant values with the declarations.

<sup>30</sup>or to ignore attributes which it does not understand

## 7.2 Interaction with concepts

**Q:** *Are there any special interactions between compile-time reflection and concepts?*

Unresolved.

## 7.3 Interaction with modules

**Q:** *Are there any special interactions between compile-time reflection and modules?*

Unresolved.

# 8 Experimental implementation

A fork of the clang compiler with a partial, experimental implementation of P0194R2 can be found on github [3]. Please see [8] (P0194R2) for more details.

Due to time-constraints there is no implementation of P0194R3 available at the moment of writing of this paper.

# 9 The Mirror reflection utilities

In order to show the usability of the proposed reflection facility, we have implemented a set of higher level reflection utilities and examples built on top of it. The source code repository can be found in [6] and the documentation also including the build-instructions is in [5].

# 10 Frequently asked questions

## 10.1 Why metaobjects, why not reflect directly with type traits?

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

**A:** The most important reasons are the completeness and the scope of reflection. Type traits<sup>31</sup> work just with types. A reflection facility should however provide much more metadata. It should be able to reflect namespaces, functions, constructors, class inheritance, templates, specifiers, variables, etc.

Without drastically changing the rules specifying what can or cannot be a template parameter, we cannot properly reflect C++’s second-class declarations like namespaces.

This is also connected with the usability of reflection in metaprograms. Using types to reflect various base-level declarations, allows to pass a representation of a namespace, a constructor, a specifier or a parameter, around various “subroutines” of a metaprogram as a parameter or a return value.

---

<sup>31</sup>As they are defined now.

In order to achieve this by other means it would be necessary to make namespaces, etc. first-class objects or at least to allow them to become template arguments.

So with type-based metaobjects we can do for example the following:

```
template <typename Metaobject>
void foo(void) { // do something terribly useful }

foo<$reflect(int)>(); // works
foo<$reflect(std)>(); // works
foo<$reflect(std::string)>(); // works
foo<$reflect(std::string::size_type)>(); // works
foo<$reflect(std::size_t)>(); // works and is different from the above
foo<$reflect(std::pair)>(); // works
foo<$reflect(std::sqrt)>(); // works
foo<$reflect(static)>(); // works
```

meanwhile without metaobjects:

```
template <something Param>
void foo(void) { // do something terribly useful, if possible }

foo<int>(); // works
foo<std>(); // error?
foo<std::string>(); // works
foo<std::string::size_type>(); // works
foo<std::size_t>(); // works but same as the above
foo_tpl<std::pair>(); // works (if there are two versions of foo)
foo<std::sqrt>(); // error?
foo<static>(); // error?
```

## 10.2 OK, but why not reflect directly with several different operators?

**Q:** *Alright, we partially agree with the reasoning in the previous question, but why don't we use several different operators to get the various pieces of metadata (like declaration names, types, scopes, etc.) directly, skipping the metaobjects? For example we already have `decltype(expr)` so why not add, `declname(expr)` or `declscope(expr)`, etc.*

**A:** The main reason is that there are many base-level declarations with various properties which we want to reflect and having a separate operator for every one of them<sup>32</sup>, will require many new reserved keywords which can break existing code.

And still, what would the operator reflecting for example a scope return? In case of class scopes it could return a type, but what if we are asking about the scope of a global-scope or namespace-level declaration? We would either end up with something like the proposed metaobjects, or we would have to make some other (not type-based) representation of any possible scope. In case of the latter we would lose the ability to pass the representation of the scope as parameters to metaprograms.

---

<sup>32</sup>There may be even several dozen.

The same applies to every other operation returning an expression which does not have first-class identity in base-level C++.

It is possible that, the templates implementing the metaobject operations will internally use compiler built-ins<sup>33</sup>, but these can use compiler-specific reserved keywords and they will all operate on the metaobjects.

### 10.3 Creating a separate type for each metaobject is so heavyweight.

**Q:** *Isn't the creation of a new type for each metaobject too heavyweight? Won't it lead to unacceptable increases in the memory footprint and compilation times?*

**A:** This might have been a problem if the metaobjects were regular types and if the implementation was too coarse-grained or too eager.

For example if upon the invocation of `$reflect(std::string)`, the representations of all the metadata related to the `std::string` type<sup>34</sup> were generated immediately.

This is however *not* the case.

The metaobjects need to be types just enough so that they can be used as template arguments and types of default- and copy-constructible variables. They don't need to have any name, scope, virtual method tables, run-time type information, etc. All they need to have is a unique identity and their internal representations need to point to the internal representations of the declarations<sup>35</sup> which they reflect.

Our proposal allows very fine granularity. The result of `$reflect` can be a very lightweight type, as we just described and the individual “attributes” like the name, scope, members, specifiers, etc. related to the metaobject are materialized only when requested by one of the operations defined for that particular metaobject concept, like `get_base_name`, `get_scope`, `get_data_members`, etc.

### 10.4 There's already another expression for that!

**Q:** *There are situations where you can do something much more easily without reflection than with it, like `&variable` vs. `reflect::get_pointer_v<$reflect(variable)>` or `decltype(var)` vs. `reflect::get_reflected_type_t<reflect::get_type_t<$reflect(var)>>`, etc.*

**A:** Of course there are! There are also situations where the decision that you want to get a pointer to or a to get the type of a (reflected) variable can be separated from the actual site where you access (reflect) the variable by several layers of metaprogramming.

Consider for example the following pseudo-code:

```
template <typename T>
void handle_var(const char* name, const T* ptr) {
    /* ... */
}
```

---

<sup>33</sup>Just like some of the type traits are implemented.

<sup>34</sup>Like the compile-time representation of its name and the metaobjects reflecting its scope or its members.

<sup>35</sup>Which the compiler needs to maintain anyway, regardless of reflection.

```
template <MetaVariable MV>
void handle_meta_var(void) {
    // this may be true only in a fraction of cases
    if(some_logic()) {
        // we need the pointer just here
        handle_var(
            reflect::get_base_name_v<MV>,
            reflect::get_pointer_v<MV>
        );
    }
}

template <MetaClass MC>
void handle_meta_class(void);

template <MetaType MT>
void handle_meta_type(void);

template <MetaVariable MV>
void handle_meta_ns(void);

template <Metaobject MO>
void generic_func_1(void) {
    // this may be true only in a fraction of cases
    if(reflect::Variable<MO>) {
        handle_meta_var<MO>();
    } else if(reflect::Class<MO>) {
        handle_meta_class<MO>();
    } else if(reflect::Type<MO>) {
        handle_meta_type<MO>();
    } else if(reflect::Namespace<MO>) {
        handle_meta_ns<MO>();
    }
}

template <Metaobject MO>
void generic_func_2(void) {
    if(some_trait_v<MO>) {
        generic_func_1<MO>();
    }
    else ...
}

// ... generic_func_3 - generic_func_19 ...
// all eventually calling generic_func_1

template <Metaobject MO>
```



```
void generic_func_20(void) {
    if(some_logic()) {
        generic_func_19<MO>();
    } else {
        generic_func_15<MO>();
    }
}

void main(int argc, const char** argv) {
    std::string str;
    generic_func_20<$reflect(str)>();
    generic_func_18<$reflect(std)>();
    generic_func_19<$reflect(argc)>();
    generic_func_20<$reflect(std::pair)>();
    // etc.

    return 0;
}
```

In this case we actually need the pointer or a reference to a variable only in a fraction of cases, deep inside of the (meta-)program.

If we didn't have the `get_pointer` operation, we would have to get the pointer very early and pass it through the whole algorithm. Furthermore in many cases there would not be any meaningful pointer to get so we would have pass `nullptr`. Now imagine that there were more metaobjects, and potentially more pointers to pass around.

Having said this, we of course do not force anybody to use `get_pointer` when simply using the ampersand operator directly would be enough.

Both cases are valid and again you should pick the most appropriate option for your situation.

## 10.5 All these additional options make it hard for the novices.

**Q:** *This reflection proposal is too complex to be learned by beginning C++ programmers and the novices might start to use unnecessarily complicated expressions to do simple things, like doing `get_pointer_v<$reflect(x)>` instead of `&x`.*

**A:** A novice could equally do:

```
double a[2];
*(a+int(-exp(complex<double>(0, M_PI)).real())) = 123.456;
```

instead of

```
double a[2];
a[1] = 123.456;
```

Doing mistakes is a part of being a beginner in something.

C++ is a complex<sup>36</sup> language which means that the reflection mechanism also must be expressive

---

<sup>36</sup>No pun intended.

enough to capture the complexity of the base-level. However we do expect that once static reflection is available and the most common use cases are identified a more convenient and straightforward façade will be added on top of it for simple and common things, while maintaining the usability of reflection in more elaborate use cases.

## 10.6 Why are the metaobjects anonymous?

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

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

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

```
namespace std {  
  
    template <typename T>  
    class meta_type /* Model of MetaType */  
    { };  
  
} // namespace std
```

which could work with types, classes, etc., but would not work with namespaces, constructors, etc. (see also the Q/A above):

```
namespace std {  
  
    template <something X> // problem  
    class meta_constructor /* Model of Constructor */  
    { };  
  
    template <something X> // problem  
    class meta_namespace /* Model of Namespace */  
    { };  
  
} // namespace std  
  
typedef std::meta_namespace<std> meta_std; // problem
```

Instead of this, the metaobjects are anonymous and their (internal) identification is left to the compiler. From the user's point of view, the metaobject can be distinguished by the means of the metaobject traits.

## 10.7 Reflection violating access restrictions to class members?

**Q:** *Why do you allow reflection to bypass the class member access restrictions? This is like giving people nuclear weapons. I don't want people to have nuclear weapons [sic].*

**A:** This is a valid point, but there are several different ways to break access restrictions if the programmer wants to and we feel that restricting reflection only to public class members would severely limit its usefulness in implementing things like non-intrusive serialization, object-relational mapping, etc.

Having said that we also try to design the interface to allow the users to indicate that they only want the public data members, by providing two sets of operations where applicable (for example `get_data_members` vs. `get_public_data_members`).

Furthermore the `is_public` trait can be used to test if a class member is publicly accessible or not.

## 10.8 We need to get around access restrictions, but not in reflection.

**Q:** *OK, we understand that there are use cases where we need to get around access restrictions, but why do this in reflection? Why don't we use some other, unrelated, "magic" operator for that?*

**A:** In our opinion if want<sup>37</sup> to add a way of sneaking past access restrictions, then reflection is the perfect place to do it.

Reflection is like a way to look at a program from a higher dimension, not perceivable from the base-level itself. The access restrictions are in the base-level language for a reason and having to go through the meta-level to get around them seems appropriate.

Another reason for not using an operator – a reserved identifier, is to avoid causing conflicts with names in existing code.

## 10.9 Why do we need typedef reflection?

**Q:** *Why is it necessary to distinguish between types and typedefs or type aliases on the meta-level when they are not distinguishable on the base-level? Or why do we need to reflect on syntax rather than just on semantics?*

**A:** Our preferred answer is that it's better to have more information than less. Being able to distinguish typedefs or aliases on the meta-level, brings additional information about what the programmer meant, not just how it is implemented;

- `size_type` vs. `unsigned`,
- `rank_type` vs. `short`,
- `const_iterator` vs. `const Element*`,
- `height_cm` vs. `float`,
- `pid_t` vs. `int`,

---

<sup>37</sup>and we do!

- `sighandler_t` vs. `void (*)(int)`,
- etc.

If we reflect on typedefs or aliases we still know the underlying types, the opposite is not true.

Having this information is important<sup>38</sup> in several use cases.

- Debugging or trace messages (containing function signatures) are more readable and informative if they also contain the typedef names of parameter types, return values or variables, not just their underlying types.
- Cross-platform serialization or remote procedure calls may require typedef names instead of native type names which vary between platforms.

Having said this, we are aware of the compiler limitations in this area and we require the compilers to reflect on type aliases only in the reasonable cases.

## 10.10 Why *ObjectSequences*? Why not replace them with typelists?

**Q:** *Why do you define the **ObjectSequence** concept and its operations? There are type-lists proposed for the inclusion into the standard, why not use those?*

**A:** The reason why we use *ObjectSequences* is efficiency. Operations like `get_data_members` are currently designed to return metaobjects – very lightweight types representing a whole set of other metaobjects, without actually instantiating the elements eagerly, unlike typelists which would require that all contained metaobjects are generated as a part of the definition of the typelist. There are use cases where returning a typelist directly would be much less efficient than returning the lightweight metaobject sequence.

For example the user might want to test a big set of classes and find those, which have a precise number of members or find those which have multiple base classes or just get the metaobject reflecting only the first<sup>39</sup> data member of a class having hundreds of members:

```
get_element_t<get_data_members_t<$reflect(my_class)>, 0>;
```

This operation would involve the creation of two metaobjects:

1. the *ObjectSequence* and
2. the *DataMember* reflecting the first data member.

On the other hand if this operation returned a typelist of metaobjects, then all metaobjects would have to be generated, even if most of them were not used afterwards.

What we can do is to add an operation converting a *ObjectSequence* into a typelist on demand, which can be implemented trivially with the `reflect::unpack_sequence` template:

```
template <typename ... T>  
struct type_list { /* ... */};
```

---

<sup>38</sup>or at least very nice to have

<sup>39</sup>zero-th

```
template <ObjectSequence MOS>
using convert_to_list = unpack_sequence<MOS, type_list>;
```

## 10.11 Why not reflect with constexpr instances of the same type?

**Q:** *Why are the metaobjects implemented as distinct types and their interface as specializations of templates? Why not provide the metadata as constexpr instances of the same type? Then we could replace the metaobject sequences with arrays of constexpr instances and the templates with constexpr functions. Wouldn't the latter be more nice and efficient?*

**A:** In other words why don't we instead of:

```
using mo = $reflect(my_ns::my_class);
```

```
std::cout << reflect::get_base_name_v<mo> << std::endl;
std::cout << reflect::get_base_name_v<reflect::get_scope_t<mo>> << std::endl;
reflect::for_each<reflect::get_data_members_t<mo>>(my_func);
```

do something like one of the following:

A) `reflect::object mo = $reflect(my_ns::my_class);`

```
std::cout << reflect::get_base_name(mo) << std::endl;
std::cout << reflect::get_base_name(reflect::get_scope_t(mo)) << std::endl;
reflect::for_each(reflect::get_data_members_t(mo), my_func);
```

B) `reflect::object mo = $reflect(my_ns::my_class);`

```
std::cout << mo.get_base_name() << std::endl;
std::cout << mo.get_scope().get_base_name() << std::endl;
mo.get_data_members().for_each(my_func)
reflect::for_each(reflect::get_data_members(mo), my_func);
```

One of the reasons for the chosen representation of metaobjects and their interface was consistency with the already existing type traits and the established practices in metaprogramming<sup>40</sup>.

Having said that, there is nothing preventing us to implement a façade with similar syntax as either of the above<sup>41</sup>, on top of our chosen representation and we do plan to do that in the future. “Niceness” is generally a matter of taste and we want to accommodate various programming styles. See the various layers implemented by the Mirror reflection utilities [5], [1] for some examples.

In regard to efficiency, the types representing the metaobjects are very lightweight up to the point of being comparable to compile-time constants. Also the representation which we have chosen allows for a very fine granularity and metadata which are not queried don't have to be generated by the compiler. This may not be true if the metadata is represented as instances of `constexpr` structures, which must be initialized or would require some “magic” implementation.

Also note that the metaobjects reflecting the various members of a scope<sup>42</sup> can and will be heteroge-

---

<sup>40</sup>Yes we are aware of the Boost.Hana library and the metaprogramming paradigm which it brings.

<sup>41</sup>and even others

<sup>42</sup> namespaces, native types, structured types, enumerations, functions, etc.

neous. This means that we will either have to implement them only as homogeneous compile-time identifiers and implement their interface with “magic” `constexpr` functions very similar to our templates, or we cannot store them in an array.

Furthermore, representing metaobject sequences as arrays of such `constexpr` objects also requires all of the metaobjects in the sequence to be instantiated even if only a few of them will actually be used. See also the previous question for more details on this.

## 10.12 Why not return fully qualified names?

**Q:** *Why doesn't the `get_base_name` operation return fully qualified names instead of base names?*

**A:** This is because it is much easier to generate full names from base names by using metaprogramming, than to do the reverse – to parse the base name from a fully qualified and decorated name at compile-time.

Also there are several possibilities how the full name can be formatted; should it contain any whitespaces, if so where and how many?, should the `const` and `volatile` qualifiers be put before or after the type name?, should a complicated type name be split into several lines and indented, etc.

These details shouldn't be handled by the basic reflection facility, but by a higher-level library. See for example the *Mirror reflection utilities* [5].

## 10.13 What about the use cases from the committee's CFP?

**Q:** *How does this proposal handle the targeted use cases from the committee's Call for Compile-Time Reflection Proposals [9]?*

**A:** Some of these use cases are discussed in the sections 5.3, 5.4 and 5.6 of this paper.

## 11 Acknowledgments

Thanks to Ricardo Fabiano de Andrade, Roland Bock and Klaim-Joël Lamotte who provided valuable feedback, criticism and suggestions.

## References

- [1] CHOCHLÍK, M. Mirror c++ reflection library documentation (previous version). <http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/>, 2012.
- [2] CHOCHLÍK, M. A case for strong static reflection. Proposal for ISO/IEC JTC1 SC22 WG21 N4452, 2015.
- [3] CHOCHLÍK, M. Github.com: Implementation of the `constexpr` reflection operator. <https://github.com/matus-chochlik/clang/tree/reflexpr>, 2016.
- [4] CHOCHLÍK, M. Implementing the factory pattern with the help of reflection. *Computing and Informatics* 35, 3 (2016), 653–686.

- [5] CHOCHLÍK, M. Mirror c++ reflection library documentation (current version). <http://matus-chochlik.github.io/mirror/doc/html/>, 2016.
- [6] CHOCHLÍK, M. Mirror c++ reflection library source repository (current version). <https://github.com/matus-chochlik/mirror>, 2016.
- [7] CHOCHLÍK, M., NAUMANN, A., AND SANKEL, D. Static reflection: rationale, design and evolution. Proposal for ISO/IEC JTC1 SC22 WG21 P0385R1, 2016.
- [8] CHOCHLÍK, M., NAUMANN, A., AND SANKEL, D. Static reflection (rev. 6). Proposal for ISO/IEC JTC1 SC22 WG21 P0194R2, 2016.
- [9] SNYDER, J., AND CARRUTH, C. Call for compile-time reflection proposals. Tech. Rep. N3814, 2013.

# Appendix

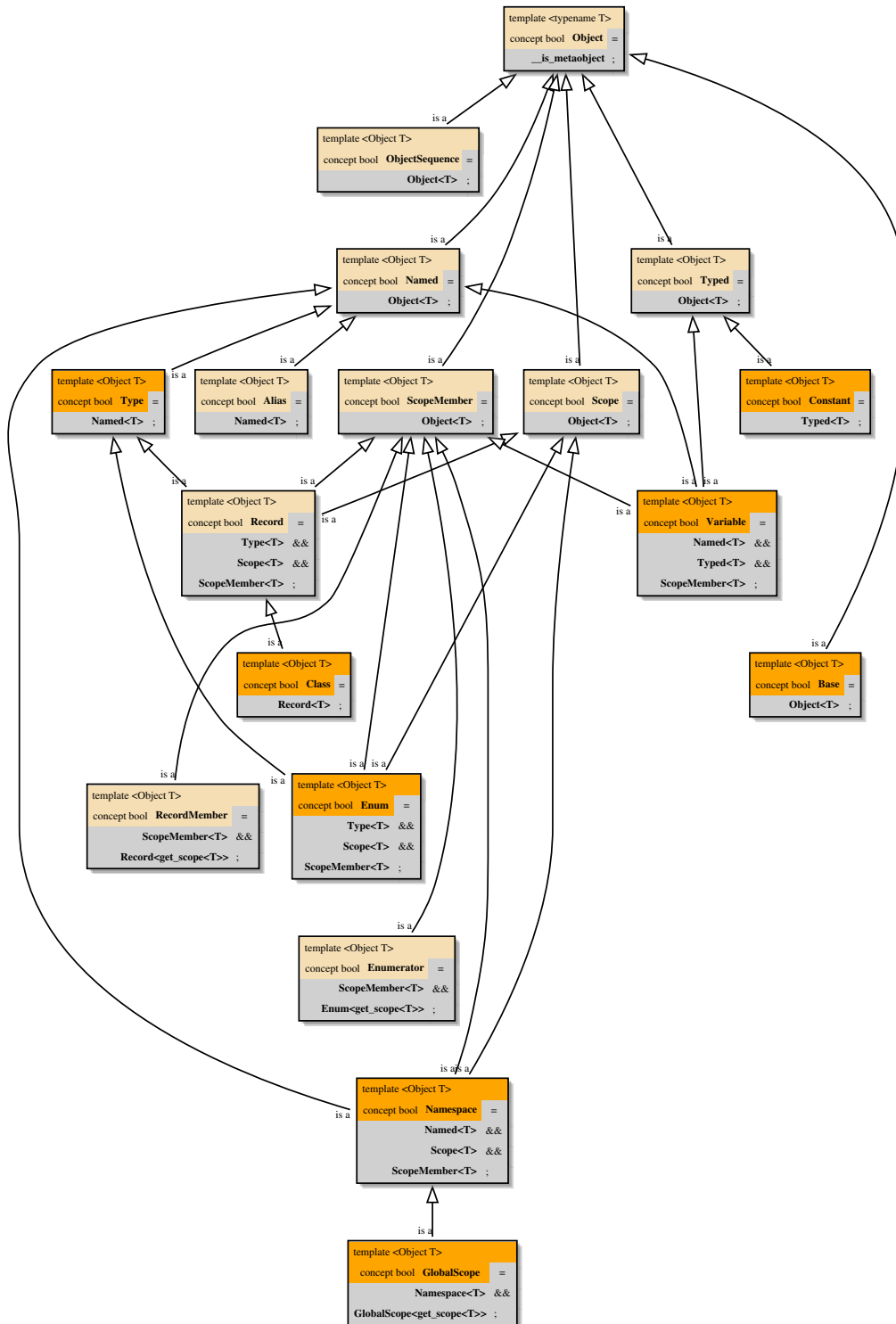


Figure 1: Metaobject concept hierarchy