Document number: P0194R0
Date: 2016-02-08
Project: Programming Language C++
Audience: Reflection (SG7) / EWG
Reply-to: Matúš Chochlík(chochlik@gmail.com), Axel Naumann (Axel.Naumann@cern.ch)

Static reflection (revision 4)

Matúš Chochlík, Axel Naumann

Abstract This paper is the follow-up to N3996, N4111 and N4451 and it is the fourth revision of the proposal to add static reflection to the C++ standard. It also introduces and briefly describes a partial, experimental implementation of this proposal.

Contents

1. Introduction

In this paper we propose to add native support for compile-time reflection to C++ by the means of compiler generated types providing basic metadata describing various program declarations. This paper introduces a new reflection operator and the initial subset of metaobject concepts which we assume to be essential and which will provide a good starting point for future extensions.

When finalized, these metaobjects, together with some additions to the standard library can later be used to implement other third-party libraries providing both compile-time and run-time, high-level reflection utilities.

Please refer to the previous papers [1, 2, 3, 4] for the motivation for this proposal, including some examples, design considerations and rationale. In [5] a complex use-case for a static reflection is described.

1.1. Revision history

- **Revision 4** – Further refines the concepts from N4111; prefixes the names of the metaobject operations with `get_`, adds new operations, replaces the metaobject category tags with new metaobject traits. Introduces a nested namespace `std::meta` which contains most of the reflection-related additions to the standard library. Rephrases definition of meta objects using Concepts Lite. Specifies the reflection operator name – `reflexpr`. Introduces an experimental implementation of the reflection operator in clang. Drops the context-dependent reflection from N4111 (will be re-introduced later).
- **Revision 3** (N4451 [3]) – Incorporates the feedback from the discussion about N4111 at the Urbana meeting, most notably reduces the set of metaobject concepts and refines their definitions, removes some of the additions to the standard library added in the previous revisions. Adds context-dependent reflection.

- **Revision 2** (N4111 [2]) – Refines the metaobject concepts and introduces a concrete implementation of their interface by the means of templates similar to the standard type traits. Describes some additions to the standard library (mostly meta-programming utilities), which simpilfy the use of the metaobjects. Answers some questions from the discussion about N3996 and expands the design rationale.
- **Revision 1** (N3996 [1]) – Describes the method of static reflection by the means of compiler-generated anonymous types. Introduces the first version of the metaobject concepts and some possiblities of their implementation. Also includes discussion about the motivation and the design rationale for the proposal.

2. Concept specification

We propose that the basic metadata describing a program written in C++ should be made available through a set of *anonymous* types – *metaobjects*, defined by the compiler and through related template classes implementing the interface of the metaobjects. At the moment these types should describe only the following program declarations: namespaces[1], types, typedefs, classes and their data members and enum types.

In the future, the set of metaobjects should be extended to reflect class inheritance, free functions, class member functions, templates, template parameters, enumerated values, possibly the C++ specifiers, etc.

The compiler should generate metadata for the program declarations in the currently processed translation unit, when requested by the invocation of the reflection operator. Members of ordered sets (sequences) of metaobjects, like scope members, parameters of a function, base classes, and so on, should be listed in the order of appearance in the processed translation unit.

Since we want the metadata to be available at compile-time, different base-level declarations should be reflected by *statically different* metaobjects and thus by *different* types. For example a metaobject reflecting the global scope namespace should be a different *type* than a metaobject reflecting the `std` namespace[2], a metaobject reflecting the `int` type should have a different type then a metaobject reflecting the `double` type, etc.

This section describes a set of metaobject concepts and their requirements, traits for metaobject classification and operations providing the individual bits of meta-data.

Unless stated otherwise, all proposed named templates described below should go into the `std::meta` nested namespace in order to contain reflection-related definitions and to help avoiding potential name conflicts.

2.1. StringConstant

`StringConstant` represents a compile-time character string type.

```
struct StringConstant {
    typedef const char value_type[N+1];
    static constexpr const char value[N+1];
    typedef StringConstant type;
};
```

[1] in a limited form
[2] this means that they should be distinguishable for example by the `std::is_same` type trait

This concept could be replaced by the `basic_string_constant` from N4236.

2.2. Meta-Object

A *Meta-Object* is a stateless anonymous type generated by the compiler (at the request of the programmer via the invocation of the `reflexpr` operator), providing metadata reflecting a specific program declaration.

Instances of metaobjects cannot be constructed, i.e. metaobject types do not have any constructors. If the need for instantiation of metaobject types arises in practical use, then this requirement can be added in the future.

2.2.1. `is_metaobject`

In order to distinguish between regular types and metaobjects generated by the compiler, the `is_metaobject` trait should be added directly to the `std` namespace as one of the type traits.

```
template <typename T>
struct is_metaobject : integral_constant<bool, ...> { };

template <typename T>
constexpr bool is_metaobject_v = is_metaobject<T>::value;
```

The expression `is_metaobject<X>::value` should be `true` if `X` is a metaobject generated by the compiler, otherwise it should be `false`.

The `is_metaobject` trait should be defined in the standard `<type_traits>` header.

Several additional trait templates should be defined in the nested namespace `std::meta` to provide further information about metaobjects. These traits are listed below together with their related metaobject concepts and should be defined in a new `<reflexpr>` header file.

2.2.2. Definition

```
namespace meta {

  template <typename T>
  concept bool Object = is_metaobject_v<T>;

} // namespace meta
```

The following operations are defined for each type satisfying the *Meta-Object* concept.

2.2.3. `reflects_same`

indicates if two metaobjects reflect the same base-level declaration.

```
namespace meta {

  template <Object T1, Object T2>
  struct reflects_same : integral_constant<bool, ...> { };
  template <Object T1, Object T2>
  constexpr auto reflects_same_v = reflects_same<T1, T2>::value;
```

```
} // namespace meta
```

2.2.4. `get_source_location`

returns the source location info of the declaration of a base-level program declaration reflected by a *Meta-Object*.

```
namespace meta {

  template <Object T>
  struct get_source_location : source_location { };

} // namespace meta
```

The returned instance of `std::source_location` should be `constexpr`, and even the source file name and function name strings should be compile-time constants.

The source information for built-in types and other such implicit declarations which are declared internally by the compiler should return an empty string as the function name and source file path and zero as the source file line and column.

2.3. Meta-ObjectSequence

A *Meta-ObjectSequence* represents an ordered sequence of *Meta-Object*s.

2.3.1. `is_sequence`

The `meta::is_sequence` trait indicates if the *Meta-Object* passed as argument is a *Meta-ObjectSequence*.

```
namespace meta {

  template <Object T>
  struct is_sequence : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_sequence_v = is_sequence<T>::value;

} // namespace meta
```

2.3.2. Definition

```
namespace meta {

  template <Object T>
  concept bool ObjectSequence = is_sequence_v<T>;

} // namespace meta
```

2.3.3. `get_size`

returns a number of elements in the sequence.

```
namespace meta {
```

```
  template <ObjectSequence T>
  struct get_size : integral_constant<size_t, ...> { };
  template <ObjectSequence T>
  constexpr auto get_size_v = get_size<T>::value;

} // namespace meta
```

2.3.4. get_element

returns the i-th *Meta-Object* in a *Meta-ObjectSequence*.

```
namespace meta {

  template <ObjectSequence T1, size_t Index>
  struct get_element
  {
      typedef /* generated by the compiler */ type;
  };

  template <ObjectSequence T1, size_t Index>
  using get_element_t = typename get_element<T1, Index>::type;

} // namespace meta
```

Note that `get_element<...>::type` must conform to the *Meta-Object* concept.

2.4. Meta-Named

A *Meta-Named* is a *Meta-Object* reflecting a named base-level declaration (namespace, type, variable, function, etc.).

2.4.1. has_name

The `meta::has_name` trait indicates if the *Meta-Object* passed as argument is a *Meta-Named*.

```
namespace meta {

  template <Object T>
  struct has_name : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool has_name_v = has_name<T>::value;

} // namespace meta
```

2.4.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Named = has_name_v<T>;

} // namespace meta
```

2.4.3. `get_name`

returns the basic name of the a named declaration reflected by a *Meta-Named*.

```
namespace meta {

  template <Named T>
  struct get_name : StringConstant { };
  template <Named T>
  constexpr auto get_name_v = get_name<T>::value;

} // namespace meta
```

The `get_name` operation returns the type, alias, namespace, etc. name without any qualifiers, asterisks[3], ampersands[4], angle or square brackets[5], etc.

For example:

```
get_name_v<reflexpr(std::vector<int>)>; // "vector"
get_name_v<reflexpr(volatile std::size_t* [10])>; // "size_t"
get_name_v<reflexpr(std::vector<int>* (*)(std::vector<double>&))>; // ""
get_name_v<reflexpr(std::vector<int>* (*pfoo)(std::vector<double>&))>; // "pfoo"
get_name_v<reflexpr(std::vector<int>* foo(std::vector<double>&))>; // "foo"
get_name_v<reflexpr(std::chrono)>; // "chrono"
get_name_v<reflexpr(std::nothrow)>; // "nothrow" (in the future)
get_name_v<reflexpr(std::string::npos)>; // "npos" (in the future)
```

2.5. Meta-Typed

A *Meta-Typed* is a *Meta-Object* reflecting base-level declaration with a type (for example a variable).

2.5.1. `has_type`

The `meta::has_type` trait indicates if the *Meta-Object* passed as argument is a *Meta-Typed*.

```
namespace meta {

  template <Object T>
  struct has_type : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool has_type_v = has_type<T>::value;

} // namespace meta
```

2.5.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Typed = has_type_v<T>;
```

[3] in case of pointers
[4] in case of references
[5] in case of templates or arrays

```
} // namespace meta
```

2.5.3. `get_type`

returns the *Meta-Type* reflecting the type of base-level declaration with a type reflected by a *Meta-Typed*.

```
namespace meta {

  template <Typed T>
  struct get_type
  {
      typedef /* generated by the compiler */ type;
  };

  template <Typed T>
  using get_type_t = typename get_type<T>::type;

} // namespace meta
```

Note that `get_type<...>::type` must conform to the *Meta-Type* concept.

2.6. Meta-Scoped

A *Meta-Scoped* is a *Meta-Object* reflecting base-level declaration nested inside of a scope.

2.6.1. `has_scope`

The `meta::has_scope` trait indicates if the *Meta-Object* passed as argument is a *Meta-Scoped*.

```
namespace meta {

  template <Object T>
  struct has_scope : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool has_scope_v = has_scope<T>::value;

} // namespace meta
```

2.6.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Scoped = has_scope_v<T>;

} // namespace meta
```

2.6.3. `get_scope`

returns the *Meta-Scope* reflecting the scope of a scoped declaration reflected by a *Meta-Scoped*.

```
namespace meta {
```

```
  template <Scoped T>
  struct get_scope
  {
      typedef /* generated by the compiler */ type;
  };

  template <Scoped T>
  using get_scope_t = typename get_scope<T>::type;

} // namespace meta
```

Note that `get_scope<...>::type` must conform to the *Meta-Scope* concept.

2.7. Meta-Scope

A *Meta-Scope* is a *Meta-Object* which is usually also a *Meta-Named* and possibly a *Meta-Scoped* reflecting a scope.

2.7.1. `is_scope`

The `meta::is_scope` trait indicates if the *Meta-Object* passed as argument is a *Meta-Scope*.

```
namespace meta {

  template <Object T>
  struct is_scope : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_scope_v = is_scope<T>::value;

} // namespace meta
```

2.7.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Scope = is_scope_v<T>;

} // namespace meta
```

2.8. Meta-Alias

A *Meta-Alias* is a *Meta-Named* reflecting a type or namespace alias.

2.8.1. `is_alias`

The `meta::is_alias` trait indicates if the *Meta-Object* passed as argument is a *Meta-Alias*.

```
namespace meta {
```

```
  template <Object T>
  struct is_alias : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_alias_v = is_alias<T>::value;

} // namespace meta
```

2.8.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Alias = Named<T> && is_alias_v<T>;

} // namespace meta
```

2.8.3. `get_aliased`

returns the *Meta-Named* reflecting the original declaration of a type or namespace alias reflected by a *Meta-Alias*.

```
namespace meta {

  template <Alias T>
  struct get_aliased
  {
      typedef /* generated by the compiler */ type;
  };

  template <Alias T>
  using get_aliased_t = typename get_aliased<T>::type;

} // namespace meta
```

Note that `get_aliased<...>::type` must conform to the *Meta-Named* concept.

2.9. Meta-Linkable

A *Meta-Linkable* is a *Meta-Named* and a *Meta-Scoped* reflecting declaration with storage duration and/or linkage.

2.9.1. `is_linkable`

The `meta::is_linkable` trait indicates if the *Meta-Object* passed as argument is a *Meta-Linkable*.

```
namespace meta {

  template <Object T>
  struct is_linkable : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_linkable_v = is_linkable<T>::value;

} // namespace meta
```

2.9.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Linkable = Named<T> && Scoped<T> && is_linkable_v<T>;

} // namespace meta
```

2.9.3. `is_static`

returns whether the declaration with storage duration and/or linkage reflected by a *Meta-Linkable* was declared with the static specifier.

```
namespace meta {

  template <Linkable T>
  struct is_static : integral_constant<bool, ...> { };
  template <Linkable T>
  constexpr auto is_static_v = is_static<T>::value;

} // namespace meta
```

2.10. Meta-ClassMember

A *Meta-ClassMember* is a *Meta-Scoped* reflecting a class member.

2.10.1. `is_class_member`

The `meta::is_class_member` trait indicates if the *Meta-Object* passed as argument is a *Meta-ClassMember*.

```
namespace meta {

  template <Object T>
  struct is_class_member : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_class_member_v = is_class_member<T>::value;

} // namespace meta
```

2.10.2. Definition

```
namespace meta {

  template <Object T>
  concept bool ClassMember = Scoped<T> && is_class_member_v<T> && Class<get_scope_t<T>>;

} // namespace meta
```

2.10.3. `is_public`

returns whether the class member reflected by a *Meta-ClassMember* was declared with public access.

```
namespace meta {

  template <ClassMember T>
  struct is_public : integral_constant<bool, ...> { };
  template <ClassMember T>
  constexpr auto is_public_v = is_public<T>::value;

} // namespace meta
```

The `is_public` trait only indicates whether we are breaking encapsulation by looking at the class member. The exact access type will be reflected by a *Meta-Specifier* which will be introduced in a future revision of this proposal.

2.11. Meta-GlobalScope

A *Meta-GlobalScope* is a *Meta-Scope* reflecting the global scope.

2.11.1. `is_global_scope`

The `meta::is_global_scope` trait indicates if the *Meta-Object* passed as argument is a *Meta-GlobalScope*.

```
namespace meta {

  template <Object T>
  struct is_global_scope : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_global_scope_v = is_global_scope<T>::value;

} // namespace meta
```

2.11.2. Definition

```
namespace meta {

  template <Object T>
  concept bool GlobalScope = Scope<T> && is_global_scope_v<T>;

} // namespace meta
```

2.12. Meta-Namespace

A *Meta-Namespace* is a *Meta-Named*, a *Meta-Scoped* and a *Meta-Scope* reflecting a namespace.

2.12.1. `is_namespace`

The `meta::is_namespace` trait indicates if the *Meta-Object* passed as argument is a *Meta-Namespace*.

```
namespace meta {
```

```
  template <Object T>
  struct is_namespace : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_namespace_v = is_namespace<T>::value;
} // namespace meta
```

2.12.2. Definition

```
namespace meta {
  template <Object T>
  concept bool Namespace = Named<T> && Scope<T> && Scoped<T> && is_namespace_v<T>;
} // namespace meta
```

2.13. Meta-NamespaceAlias

A *Meta-NamespaceAlias* is a *Meta-Namespace* and a *Meta-Alias* reflecting a namespace alias.

2.13.1. Definition

```
namespace meta {
  template <Object T>
  concept bool NamespaceAlias = Namespace<T> && Alias<T> && Namespace<get_aliased_t<T>>;
} // namespace meta
```

2.14. Meta-Type

A *Meta-Type* is a *Meta-Named* and a *Meta-Scoped* reflecting a type.

2.14.1. `is_type`

The `meta::is_type` trait indicates if the *Meta-Object* passed as argument is a *Meta-Type*.

```
namespace meta {
  template <Object T>
  struct is_type : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_type_v = is_type<T>::value;
} // namespace meta
```

2.14.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Type = Named<T> && Scoped<T> && is_type_v<T>;

} // namespace meta
```

2.14.3. `get_reflected_type`

returns the the base-level type reflected by a *Meta-Type*.

```
namespace meta {

  template <Type T>
  struct get_reflected_type
  {
      typedef /* generated by the compiler */ type;
  };

  template <Type T>
  constexpr auto get_reflected_type_v = get_reflected_type<T>::value;

} // namespace meta
```

2.15. Meta-TypeAlias

A *Meta-TypeAlias* is a *Meta-Type* and a *Meta-Alias* reflecting a typedef, a type alias or a substituted template type parameter.

2.15.1. Definition

```
namespace meta {

  template <Object T>
  concept bool TypeAlias = Type<T> && Alias<T> && Type<get_aliased_t<T>>;

} // namespace meta
```

If the reflected type alias or typedef refers to a class then the reflecting *Meta-TypeAlias* is also a *Meta-Class*, if the reflected alias refers to an enum then the *Meta-TypeAlias* is also a *Meta-Enum*.

2.16. Meta-Class

A *Meta-Class* is a *Meta-Type* and a *Meta-Scope* reflecting a class, struct or union.

2.16.1. `is_class`

The `meta::is_class` trait indicates if the *Meta-Object* passed as argument is a *Meta-Class*.

```
namespace meta {

  template <Object T>
  struct is_class : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_class_v = is_class<T>::value;

} // namespace meta
```

2.16.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Class = Type<T> && Scope<T> && is_class_v<T>;

} // namespace meta
```

2.16.3. `get_data_members`

returns a *Meta-ObjectSequence* of metaobjects reflecting the public data members of a class reflected by a *Meta-Class*.

```
namespace meta {

  template <Class T>
  struct get_data_members
  {
      typedef /* generated by the compiler */ type;
  };

  template <Class T>
  using get_data_members_t = typename get_data_members<T>::type;

} // namespace meta
```

Note that `get_data_members<...>::type` must conform to the *Meta-ObjectSequence* concept.

2.16.4. `get_all_data_members`

returns a *Meta-ObjectSequence* of metaobjects reflecting all (including the private and protected) data members of a class reflected by a *Meta-Class*.

```
namespace meta {

  template <Class T>
  struct get_all_data_members
  {
      typedef /* generated by the compiler */ type;
  };

  template <Class T>
  using get_all_data_members_t = typename get_all_data_members<T>::type;
```

```
} // namespace meta
```

Note that `get_all_data_members<...>::type` must conform to the *Meta-ObjectSequence* concept.

The *Meta-ObjectSequence* returned by `get_data_members` and `get_all_data_members` should *not* include metaobjects reflecting inherited data members.

2.17. Meta-Enum

A *Meta-Enum* is a *Meta-Type* and possibly also a *Meta-Scope* reflecting an enum or a scoped enum.

2.17.1. `is_enum`

The `meta::is_enum` trait indicates if the *Meta-Object* passed as argument is a *Meta-Enum*.

```
namespace meta {

  template <Object T>
  struct is_enum : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_enum_v = is_enum<T>::value;

} // namespace meta
```

2.17.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Enum = Type<T> && is_enum_v<T>;

} // namespace meta
```

2.18. Meta-EnumClass

A *Meta-EnumClass* is a *Meta-Enum* and a *Meta-Scope* reflecting a scoped, strongly-typed enumeration.

2.18.1. Definition

```
namespace meta {

  template <Object T>
  concept bool EnumClass = Enum<T> && Scope<T>;

} // namespace meta
```

2.19. Meta-Variable

A *Meta-Variable* is a *Meta-Named*, a *Meta-Typed* and a *Meta-Linkable* reflecting a variable[6].

[6] At the moment only class data members fall into this category, but variable reflection should be introduced in a future proposal

2.19.1. `is_variable`

The `meta::is_variable` trait indicates if the *Meta-Object* passed as argument is a *Meta-Variable*.

```
namespace meta {

  template <Object T>
  struct is_variable : integral_constant<bool, ... > { };
  template <Object T>
  constexpr bool is_variable_v = is_variable<T>::value;

} // namespace meta
```

2.19.2. Definition

```
namespace meta {

  template <Object T>
  concept bool Variable = Named<T> && Typed<T> && Linkable<T> && is_variable_v<T>;

} // namespace meta
```

2.19.3. `get_pointer`

returns a pointer to the a variable reflected by a *Meta-Variable*. If the variable is a class member then the pointer is a class data member pointer, otherwise it is a plain pointer.

```
namespace meta {

  template <Variable T>
  struct get_pointer
  {
      typedef conditional_t<
          is_class_member_v<T> && !is_static_v<T>,
          get_reflected_type_t<get_type_t<T>>
          get_reflected_type_t<get_scope_t<T>>::*,
          get_reflected_type_t<get_type_t<T>>*
      > value_type;

      static const value_type value;
  };

  template <Variable T>
  const auto get_pointer_v = get_pointer<T>::value;

} // namespace meta
```

2.20. Meta-DataMember

A *Meta-DataMember* is a *Meta-Variable* and a *Meta-ClassMember* reflecting a class data member.

2.20.1. Definition

```
namespace meta {

  template <Object T>
  concept bool DataMember = Variable<T> && ClassMember<T>;

} // namespace meta
```

3. Reflection operator

The metaobjects reflecting some program declaration `X` should be made available to the user by the means of a new operator `reflexpr`. More precisely, the reflection operator should return a type[7] conforming to a particular metaobject concept, depending on the argument.

The operand `X` of the `reflexpr` operator can at the moment be any of the following[8]:

- No expression or `::` – The global scope, the returned metaobject is a *Meta-GlobalScope*.
- *Namespace name* – (`std`) the returned metaobject is a *Meta-Namespace*.
- *Namespace alias name* – the returned metaobject is a *Meta-NamespaceAlias*.
- *Type name* – (`long double`) the returned metaobject is a *Meta-Type*.
- *Typedef name* – (`std::size_t` or `std::string`) the returned metaobject is a *Meta-TypeAlias*.
- *Class name* – (`std::thread` or `std::map<unsigned, float>`) the returned metaobject is a *Meta-Class*.
- *Enum name* – the returned metaobject is a *Meta-Enum*.
- *Enum class name* – (`std::launch`) the returned metaobject is a *Meta-EnumClass*.

3.1. Redeclarations

The meta data queried by `reflexpr` depends on the point of invocation of the operator; only declarations in front of the invocation will be visible. Subsequent invocations are independent of prior invocations, as if all compiler generated types were unique to each reflexpr invocation.

```
struct foo;
using meta_foo_fwd1 = reflexpr(foo);
constexpr size_t n1 = get_size_v<get_data_members_t<meta_foo_fwd1>>; // 0

struct foo;
using meta_foo_fwd2 = reflexpr(foo);
constexpr size_t n2 = get_size_v<get_data_members_t<meta_foo_fwd2>>; // 0

constexpr bool b1 = is_same_v<meta_foo_fwd1, meta_foo_fwd2>; // unspecified
constexpr bool b2 = meta::reflects_same_v<meta_foo_fwd1, meta_foo_fwd2>; // true

struct foo { int a,b,c,d; };
```

[7] Like the `decltype` operator
[8] Future proposals may add new operands like variables, overloaded functions, constructors, etc.

```
using meta_foo = reflexpr(foo);
constexpr size_t n3 = get_size_v<get_data_members_t<meta_foo>>; // 4

constexpr bool b3 = is_same_v<meta_foo_fwd1, meta_foo>; // false
constexpr bool b4 = meta::reflects_same_v<meta_foo_fwd1, meta_foo>; // true
```

3.2. Required header

If the header `<reflexpr>` is not included prior to a use of `reflexpr`, the program is ill-formed[9]. This header file defines the metaobject trait templates[10] and the templates implementing the operations on metaobjects as described above.

3.3. Future extensions

In a future proposal an additional, new variant of the `reflexpr` operator – `reflexpr(X, Concept)` could be introduced. This operator would return a pack of types, conforming to the *Meta-Object* concept, reflecting members of the base-level declaration `X` which conform to the specified `Concept`. The parameter `X` is the same as in the single argument version, `Concept` is the name of a metaobject concept.

For example:

```
typedef std::tuple<reflexpr(std, std::meta::Typedef)...> meta_std_typedefs;
```

4. Experimental implementation

A fork of the clang compiler with a partial experimental implementation of this proposal can be found on github [6]. The modified compiler can be built by following the instructions listed in [7], but instead of checking out the official clang repository, the sources on the `mirror-reflection` branch of the modified repository should be used.

The implementation required changes (mostly additions) to roughly 3500 lines of code in the compiler plus circa 540 lines in the `<reflexpr>` header.

All examples listed in Appendix B are working with the modified compiler.

This particular implementation works by generating and adding new `CXXRecordDecl`s, with member typedefs and member constants providing basic meta-data, to the AST. The templates implementing the metaobject operations are referencing and occasionally transforming the members of the metaobject types.

The `is_metaobject` trait is implemented by extending the existing clang's type-trait framework and adding the `__is_metaobject` builtin operator.

The *Meta-ObjectSequence* operations, namely `get_size` and `get_element` are implemented lazily by using the new `__reflexpr_size` and `__reflexpr_element` operators.

No rigorous measurements were made, but the implementation is not expected to incur any noticeable overhead in terms of memory footprint or processing times[11] when compiling programs, which do not use reflection.

[9] Similar to the `<typeinfo>` header being required for the `typeid` operator.
[10] Except for `is_metaobject`.
[11] Besides the fact that several new tokens were added.

Also note that this specification underwent some last-minute changes, which were not incorporated into the implementation yet.

5. Impact on the standard

This proposal requires the addition of a new operator `reflexpr` which *may* cause conflicts with identifiers in existing code.

For what it's worth, we have performed a quick analysis on 994 third-party, open-source repositories hosted on `http://github.com/`[12], where we counted identifiers in the C++ source files. We have found 646 313 149 instances of 7 903 042 *distinct* words matching the C++ identifier rules. We did not find *any* occurence of "reflexpr".

No other changes to the core language are required, especially no new rules for template parameters. The metaobjects can be implemented by (basically) regular types and the interface is implemented by regular templates. An implementation with lazy evaluation may require the addition of several compiler builtin operators similar to those already used for the type-traits, but other implementations are possible.

5.1. Alternative spelling of the operator

Alternatively the `constexpr(X)` or `std::reflexpr(X)` syntax could be used instead of unscoped `reflexpr(X)` to further minimise possible name conflicts with existing identifiers.

6. Known issues

- Reflection of `union`s – should we add a *Meta-Union* or handle unions together with classes.
- Should *Meta-GlobalScope* also conform to the *Meta-Namespace* concept?
- Since we can have different metaobjects reflecting different redeclarations of the same thing, there should be a way to traverse the chain of these metaobjects, in the direction from the "newest" to the "oldest".
- Interaction with generalized attributes.

7. References

[1] Chochlík M., N3996 - Static reflection, 2014, `https://isocpp.org/files/papers/n3996.pdf`.

[2] Chochlík M., N4111 - Static reflection (revision 2), 2014, `https://isocpp.org/files/papers/n4111.pdf`.

[3] Chochlík M., N4451 - Static reflection (revision 3), 2015, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4451.pdf`.

[4] Chochlík M., N4452 - A case for strong static reflection, 2015, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4452.pdf`.

[12] The main branches of original repositories, not forks.

[5] Chochlík M., Implementing the Factory pattern with the help of reflection, (pre-print), http://www.researchgate.net/publication/269668943_IMPLEMENTING_THE_FACTORY_PATTERN_WITH_THE_HELP_OF_REFLECTION.

[6] Experimental implementation of reflexpr, https://github.com/matus-chochlik/clang/tree/mirror-reflection.

[7] Getting Started: Building and Running Clang, http://clang.llvm.org/get_started.html.

Appendix

A. Diagrams

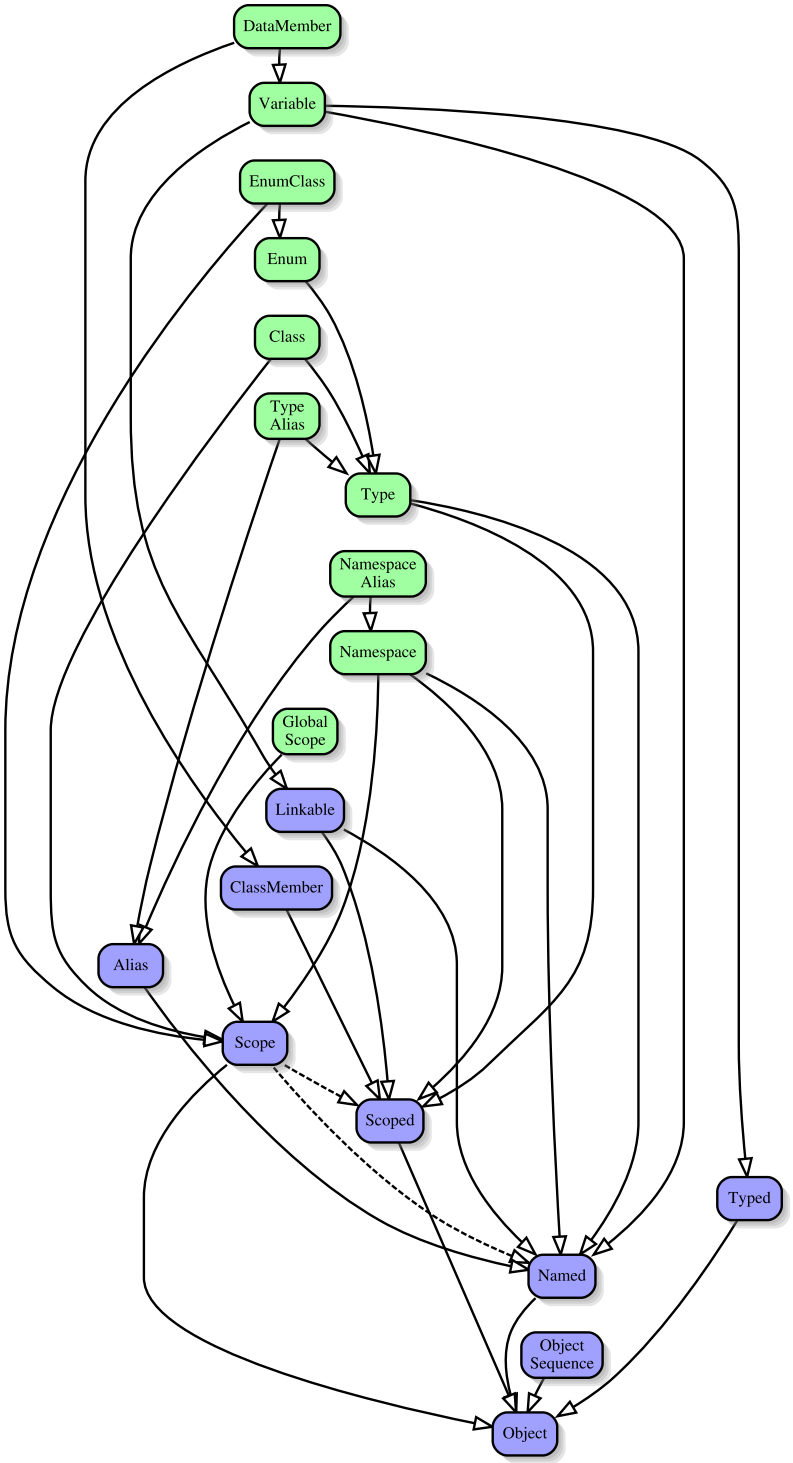

Figure 1: Metaobject concept inheritance hierarchy

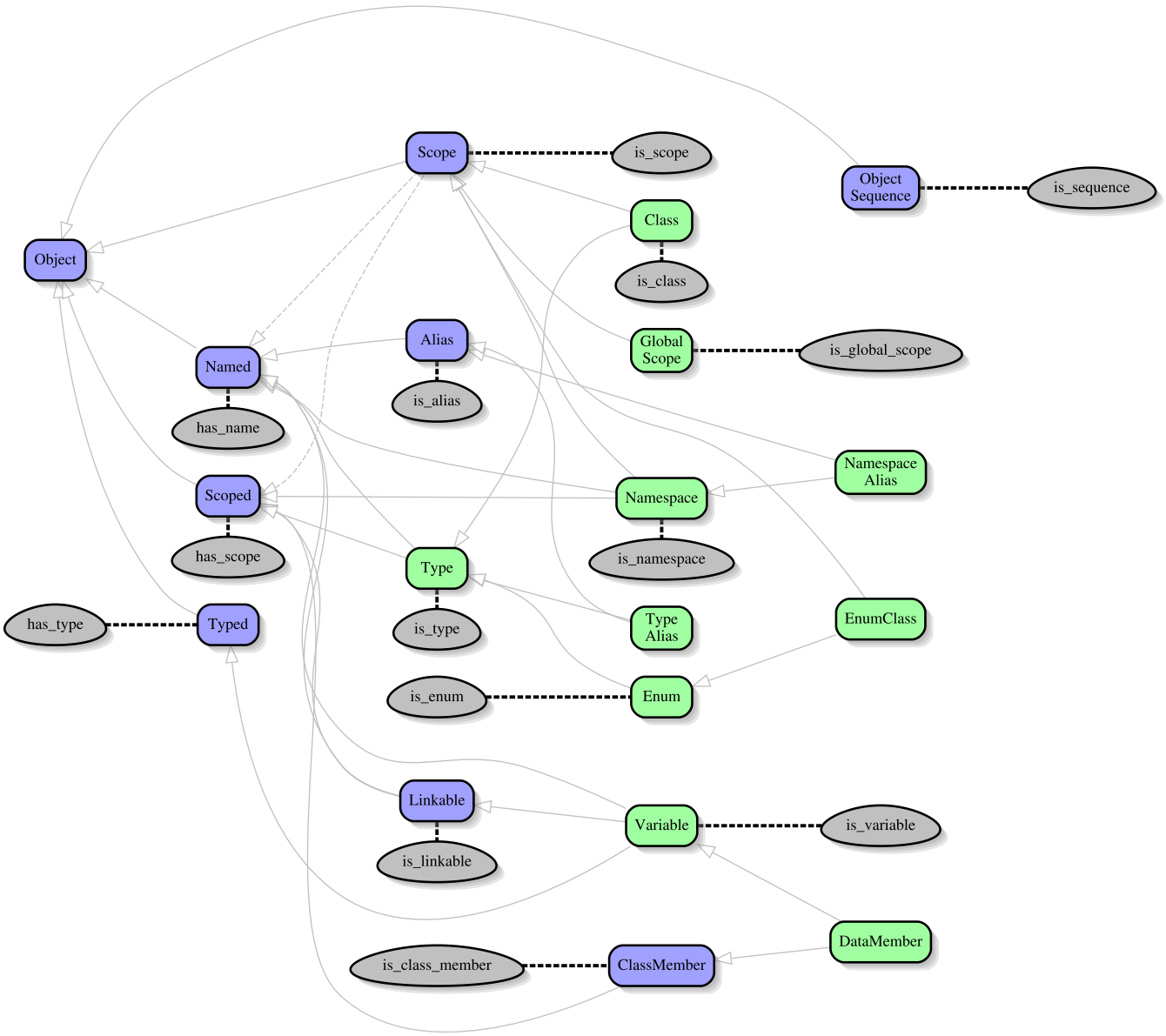

Figure 2: Metaobject traits

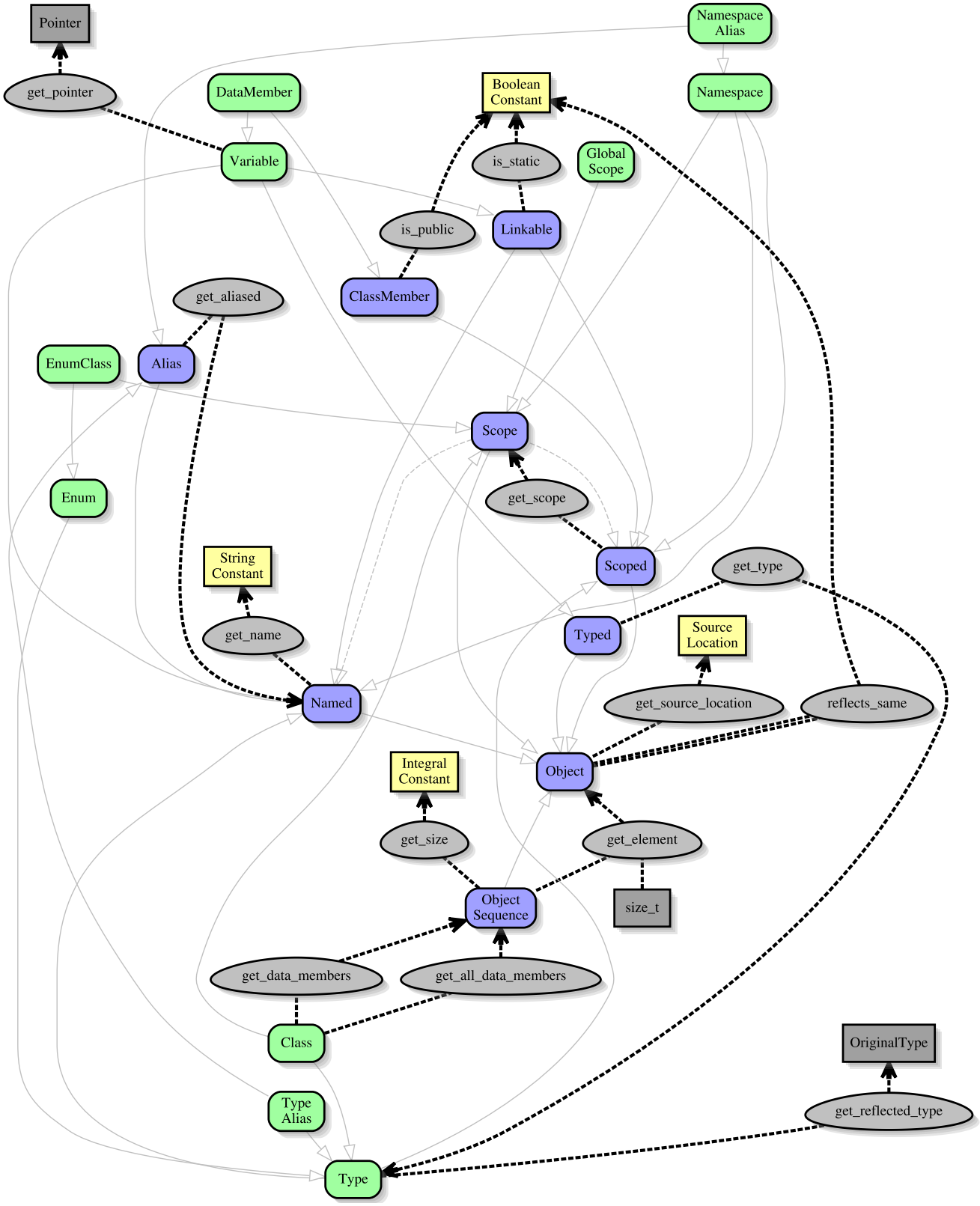

Figure 3: Metaobject operations

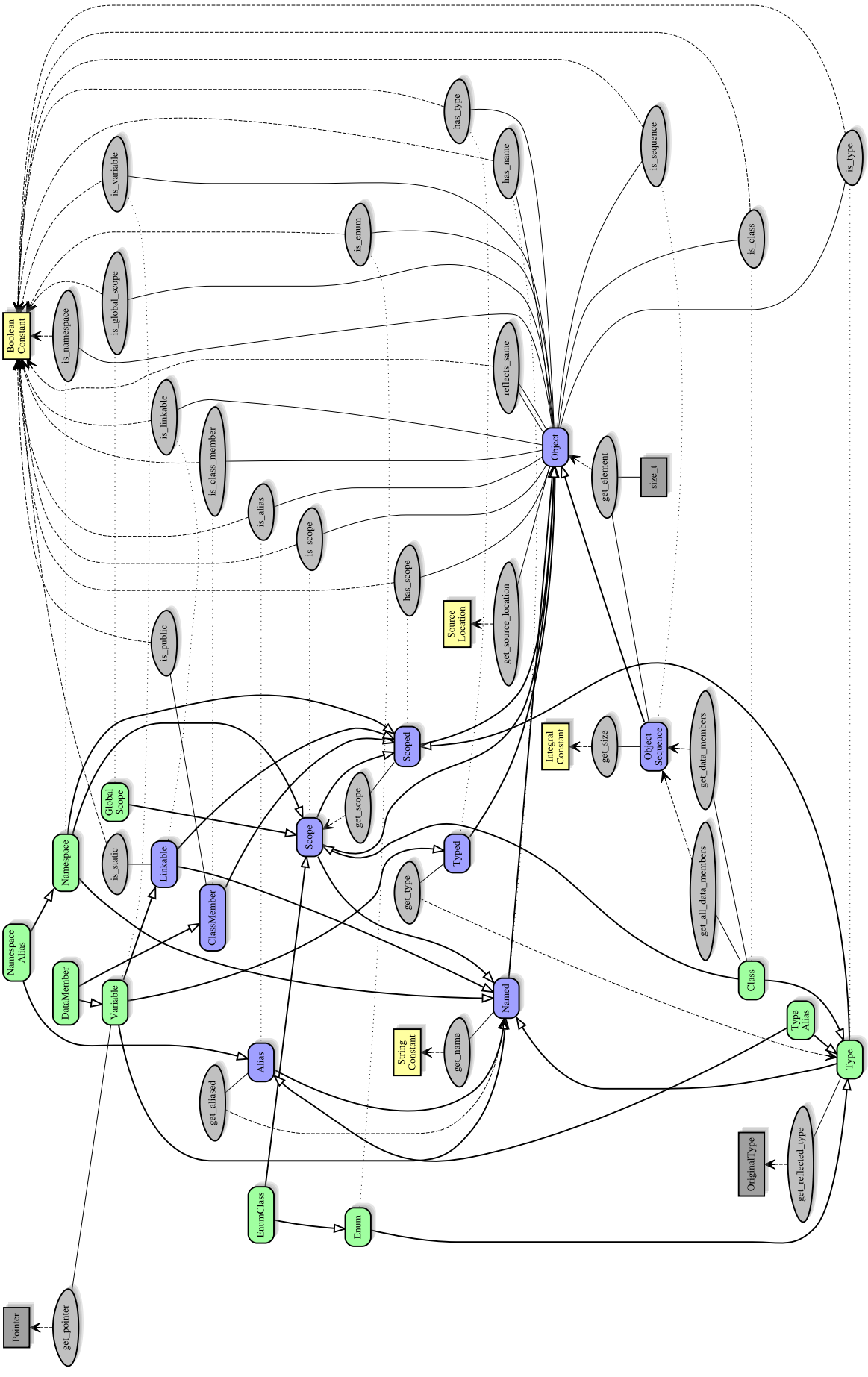

Figure 4: Overview

B. Examples of usage

B.1. Scope reflection

```
#include <reflexpr>
#include <iostream>

namespace foo {

struct bar
{
    typedef int baz;
};

} // namespace foo

typedef long foobar;

int main(void)
{
    using namespace std;

    typedef reflexpr(int) meta_int;
    typedef reflexpr(foo::bar) meta_foo_bar;
    typedef reflexpr(foo::bar::baz) meta_foo_bar_baz;
    typedef reflexpr(foobar) meta_foobar;

    static_assert(meta::has_scope_v<meta_int>, "");
    static_assert(meta::has_scope_v<meta_foo_bar>, "");
    static_assert(meta::has_scope_v<meta_foo_bar_baz>, "");
    static_assert(meta::has_scope_v<meta_foobar>, "");

    typedef meta::get_scope_t<meta_int> meta_int_s;
    typedef meta::get_scope_t<meta_foo_bar> meta_foo_bar_s;
    typedef meta::get_scope_t<meta_foo_bar_baz> meta_foo_bar_baz_s;
    typedef meta::get_scope_t<meta_foobar> meta_foobar_s;

    static_assert(meta::is_scope_v<meta_int_s>, "");
    static_assert(meta::is_scope_v<meta_foo_bar_s>, "");
    static_assert(meta::is_scope_v<meta_foo_bar_baz_s>, "");
    static_assert(meta::is_scope_v<meta_foobar_s>, "");

    static_assert(meta::is_namespace_v<meta_int_s>, "");
    static_assert(meta::is_global_scope_v<meta_int_s>, "");
    static_assert(meta::is_namespace_v<meta_foo_bar_s>, "");
    static_assert(!meta::is_global_scope_v<meta_foo_bar_s>, "");
    static_assert(meta::is_type_v<meta_foo_bar_baz_s>, "");
    static_assert(meta::is_class_v<meta_foo_bar_baz_s>, "");
    static_assert(!meta::is_namespace_v<meta_foo_bar_baz_s>, "");
    static_assert(meta::is_namespace_v<meta_foobar_s>, "");
```

```
    static_assert(meta::is_global_scope_v<meta_foobar_s>, "");
    static_assert(!meta::is_class_v<meta_foobar_s>, "");

    cout << meta::get_name_v<meta_foo_bar_baz> << endl;
    cout << meta::get_name_v<meta_foo_bar_baz_s> << endl;
    cout << meta::get_name_v<meta_foo_bar_s> << endl;

    return 0;
}
```

Output:

```
baz
bar
foo
```

B.2. Namespace reflection

```
#include <reflexpr>
#include <iostream>

namespace foo { namespace bar { } }

namespace foobar = foo::bar;

int main(void)
{
    using namespace std;

    typedef reflexpr(foo) meta_foo;

    static_assert(is_metaobject_v<meta_foo>, "");

    static_assert(!meta::is_global_scope_v<meta_foo>, "");
    static_assert(meta::is_namespace_v<meta_foo>, "");
    static_assert(!meta::is_type_v<meta_foo>, "");
    static_assert(!meta::is_alias_v<meta_foo>, "");

    static_assert(meta::has_name_v<meta_foo>, "");
    static_assert(meta::has_scope_v<meta_foo>, "");
    cout << meta::get_name_v<meta_foo> << endl;

    typedef reflexpr(foo::bar) meta_foo_bar;

    static_assert(is_metaobject_v<meta_foo_bar>, "");

    static_assert(!meta::is_global_scope_v<meta_foo_bar>, "");
    static_assert(meta::is_namespace_v<meta_foo_bar>, "");
    static_assert(!meta::is_type_v<meta_foo_bar>, "");
    static_assert(!meta::is_alias_v<meta_foo_bar>, "");
```

```
    static_assert(meta::has_name_v<meta_foo_bar>, "");
    static_assert(meta::has_scope_v<meta_foo_bar>, "");
    cout << meta::get_name_v<meta_foo_bar> << endl;

    typedef reflexpr(foobar) meta_foobar;

    static_assert(is_metaobject_v<meta_foobar>, "");

    static_assert(!meta::is_global_scope_v<meta_foobar>, "");
    static_assert(meta::is_namespace_v<meta_foobar>, "");
    static_assert(!meta::is_type_v<meta_foobar>, "");
    static_assert(meta::is_alias_v<meta_foobar>, "");

    static_assert(meta::has_name_v<meta_foobar>, "");
    static_assert(meta::has_scope_v<meta_foobar>, "");
    cout << meta::get_name_v<meta_foobar> << " a.k.a ";
    cout << meta::get_name_v<meta::get_aliased_t<meta_foobar>> << endl;

    return 0;
}
```

Output:

```
foo
bar
foobar a.k.a bar
```

B.3. Type reflection

```
#include <reflexpr>
#include <iostream>

int main(void)
{
    using namespace std;

    typedef reflexpr(unsigned) meta_unsigned;

    static_assert(is_metaobject_v<meta_unsigned>, "");
    static_assert(meta::is_type_v<meta_unsigned>, "");
    static_assert(!meta::is_alias_v<meta_unsigned>, "");

    static_assert(is_same_v<
        meta::get_reflected_type_t<meta_unsigned>,
        unsigned
    >, "");

    static_assert(meta::has_name_v<meta_unsigned>, "");
    cout << meta::get_name_v<meta_unsigned> << endl;

    typedef reflexpr(unsigned*) meta_ptr_unsigned;
```

```
    static_assert(meta::has_name_v<meta_ptr_unsigned>, "");
    cout << meta::get_name_v<meta_ptr_unsigned> << endl;

    return 0;
}
```

Output:

```
unsigned int
unsigned int
```

B.4. Typedef reflection

```
#include <reflexpr>
#include <iostream>

namespace foo {

typedef int bar;
using baz = bar;

} // namespace foo

int main(void)
{
    using namespace std;

    typedef reflexpr(foo::baz) meta_foo_baz;

    static_assert(is_metaobject_v<meta_foo_baz>, "");
    static_assert(meta::is_type_v<meta_foo_baz>, "");
    static_assert(meta::is_alias_v<meta_foo_baz>, "");

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

    static_assert(meta::has_name_v<meta_foo_baz>, "");
    cout << meta::get_name_v<meta_foo_baz> << endl;

    typedef meta::get_aliased_t<meta_foo_baz> meta_foo_bar;

    static_assert(is_metaobject_v<meta_foo_bar>, "");
    static_assert(meta::is_type_v<meta_foo_bar>, "");
    static_assert(meta::is_alias_v<meta_foo_bar>, "");

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

```
    static_assert(meta::has_name_v<meta_foo_bar>, "");
    cout << meta::get_name_v<meta_foo_bar> << endl;

    typedef meta::get_aliased_t<meta_foo_bar> meta_int;

    static_assert(is_metaobject_v<meta_int>, "");
    static_assert(meta::is_type_v<meta_int>, "");
    static_assert(!meta::is_alias_v<meta_int>, "");

    static_assert(is_same_v<
        meta::get_reflected_type_t<meta_int>,
        int
    >, "");

    static_assert(meta::has_name_v<meta_int>, "");
    cout << meta::get_name_v<meta_int> << endl;

    return 0;
}
```

Output:

```
baz
bar
int
```

B.5. Class alias reflection

```
#include <reflexpr>
#include <iostream>

struct foo { };
using bar = foo;

int main(void)
{
    using namespace std;

    typedef reflexpr(bar) meta_bar;

    static_assert(is_metaobject_v<meta_bar>, "");
    static_assert(meta::is_type_v<meta_bar>, "");
    static_assert(meta::is_class_v<meta_bar>, "");
    static_assert(meta::is_alias_v<meta_bar>, "");

    static_assert(is_same_v<meta::get_reflected_type_t<meta_bar>, bar>, "");

    static_assert(meta::has_name_v<meta_bar>, "");
    cout << meta::get_name_v<meta_bar> << endl;
```

```
    typedef meta::get_aliased_t<meta_bar> meta_foo;

    static_assert(is_metaobject_v<meta_foo>, "");
    static_assert(meta::is_type_v<meta_foo>, "");
    static_assert(meta::is_class_v<meta_foo>, "");
    static_assert(!meta::is_alias_v<meta_foo>, "");

    static_assert(is_same_v<meta::get_reflected_type_t<meta_foo>, foo>, "");

    static_assert(meta::has_name_v<meta_foo>, "");
    cout << meta::get_name_v<meta_foo> << endl;

    return 0;
}
```

Output:

```
bar
foo
```

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

B.6. Class data members (1)

```
#include <reflexpr>
#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;

    typedef reflexpr(foo) meta_foo;

    // (public) data members
    typedef meta::get_data_members_t<meta_foo> meta_data_mems;

    static_assert(is_metaobject_v<meta_data_mems>, "");
    static_assert(meta::is_sequence_v<meta_data_mems>, "");

    cout << meta::get_size_v<meta_data_mems> << endl;
```

```
    // 0-th (public) data member
    typedef meta::get_element_t<meta_data_mems, 0> meta_data_mem0;

    static_assert(is_metaobject_v<meta_data_mem0>, "");
    static_assert(meta::is_variable_v<meta_data_mem0>, "");
    static_assert(meta::has_type_v<meta_data_mem0>, "");

    cout << meta::get_name_v<meta_data_mem0> << endl;

    // 2-nd (public) data member
    typedef meta::get_element_t<meta_data_mems, 2> meta_data_mem2;

    static_assert(is_metaobject_v<meta_data_mem2>, "");
    static_assert(meta::is_variable_v<meta_data_mem2>, "");
    static_assert(meta::has_type_v<meta_data_mem2>, "");

    cout << meta::get_name_v<meta_data_mem2> << endl;

    // all data members
    typedef meta::get_all_data_members_t<meta_foo> meta_all_data_mems;

    static_assert(is_metaobject_v<meta_all_data_mems>, "");
    static_assert(meta::is_sequence_v<meta_all_data_mems>, "");

    cout << meta::get_size_v<meta_all_data_mems> << endl;

    // 0-th (overall) data member
    typedef meta::get_element_t<meta_all_data_mems, 0> meta_all_data_mem0;

    static_assert(is_metaobject_v<meta_all_data_mem0>, "");
    static_assert(meta::is_variable_v<meta_all_data_mem0>, "");
    static_assert(meta::has_type_v<meta_all_data_mem0>, "");

    cout << meta::get_name_v<meta_all_data_mem0> << endl;

    // 3-rd (overall) data member
    typedef meta::get_element_t<meta_all_data_mems, 3> meta_all_data_mem3;

    static_assert(is_metaobject_v<meta_all_data_mem3>, "");
    static_assert(meta::is_variable_v<meta_all_data_mem3>, "");
    static_assert(meta::has_type_v<meta_all_data_mem3>, "");

    cout << meta::get_name_v<meta_all_data_mem3> << endl;

    return 0;
}
```

This produces the following output:

```
4
```

```
b
y
7
_i
x
```

B.7. Class data members (2)

```
#include <reflexpr>
#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 meta::get_element_t<Metaobjects, I> metaobj;

    cout    << I << ": "
        << (meta::is_public_v<metaobj>?"public":"non-public")
        << " "
        << (meta::is_static_v<metaobj>?"static":"")
        << " "
        << meta::get_name_v<meta::get_type_t<metaobj>>
        << " "
        << meta::get_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<
            meta::get_size_v<Metaobjects>
        >()
    );
}

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

    cout<< "Public data members of " << meta::get_name_v<MetaClass> << endl;

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

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

    cout << "All data members of " << meta::get_name_v<MetaClass> << endl;

    do_print_data_members<meta::get_all_data_members_t<MetaClass>>();
}

int main(void)
{
    print_data_members<reflexpr(foo)>();
    print_all_data_members<reflexpr(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
```

B.8. Class data members (3)

```
#include <reflexpr>
#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;

    typedef reflexpr(A) meta_A;
    typedef reflexpr(B) meta_B;
    typedef reflexpr(C) meta_C;

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

    cout << meta::get_size_v<meta::get_all_data_members_t<meta_A>> << endl;
    cout << meta::get_size_v<meta::get_all_data_members_t<meta_B>> << endl;
    cout << meta::get_size_v<meta::get_all_data_members_t<meta_C>> << endl;

    return 0;
}
```

This program produces the following output:

```
1
0
```

```
1
1
1
1
```

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

B.9. Simple serialization to JSON

```
#include <reflexpr>
#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)
{
    typedef std::meta::get_element_t<Metaobjects, I> meta_F;

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

    out << '"' << std::meta::get_name_v<meta_F> << "\": ";

    value_to_json(out, (v.*std::meta::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 << "{";

    typedef std::meta::get_all_data_members_t<MO> meta_DMs;
    fields_to_json<meta_DMs>(
        out, v,
        std::make_index_sequence<
            std::meta::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)
{
    typedef reflexpr(T) meta_T;
    return reflected_to_json<meta_T>(
        out, v,
        std::meta::is_class_t<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;
}
```

This program 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}}}
```