Document number: P0385R1
Date: 2016-10-15
Project: Programming Language C++
Audience: Reflection (SG7) / EWG
Matúš Chochlík(chochlik@gmail.com),
Reply-to: Axel Naumann (axel@cern.ch),
David Sankel (camior@gmail.com)

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

1 Introduction

This paper accompanies the P0194Rx papers 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.

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[1].

[1] Also to help us remember what the answers were.

- 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.

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

There is also an experimental implementation of the P0194 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 elaborated type 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[2], in a C++ compiler plug-in, or in C++ itself.

In C++ metaprogramming usually takes the form of an external *source code generator*[3], 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.

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[4].

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[5].

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.

[2]For example a C++ code generator written in Python or Bash.
[3]Like Qt's MOC.
[4]Mostly in interpreted languages.
[5]For example generating new class data members.

- Has an intrinsic identity making the entity unique and distinguishable.

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"[6], 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.

[6] From latin "rei", the dative form of "res" – a thing.

2 Revision history

2.1 P0385R0

This is the initial revision of the design paper accompanying the P0194 papers.

2.2 P0385R1 – current revision

This revision brings the following updates reflecting[7] 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).

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,

[7] Pun not intended

- 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 source code generators.

Some of the use cases listed above are described in more detail in section 5.

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[8] in a very uniform way which could be transcribed into a metaprogram[9].

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. Metaprogramming is the tool for transforming one program into another based on some meta-algorithm and reflection provides the input data for that algorithm directly from the compiler without the need for an external source code parser.

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<"
```

[8] types, structures, containers, functions, constructors, class member variables, enumerated values, etc.
[9] with varying level of complexity

```
            << get_base_name_v<get_aliased_m<reflexpr(T)>>
            << ">("
            << get_base_name_v<reflexpr(a)> << ": "
            << get_base_name_v<get_aliased_m<get_type_m<reflexpr(a)>>>
            << " = " << a
            << get_base_name_v<reflexpr(b)> << ": "
            << get_base_name_v<get_aliased_m<get_type_m<reflexpr(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)
```

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 P0194R2 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[10], 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[11].

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

For example:

```
typedef reflexpr() meta_global_scope;
typedef reflexpr(int) meta_int;
typedef reflexpr(std) meta_std;
typedef reflexpr(std::size_t) meta_std_size_t;
```

[10] Namespace, type alias, function, parameter, specifier, etc.
[11] Which will happen very often in the more complex use cases

```
typedef reflexpr(std::thread) meta_std_thread;
typedef reflexpr(std::pair) meta_std_pair;
typedef reflexpr(std::launch) meta_std_launch;
typedef reflexpr(std::launch::async) meta_std_launch_async;
typedef reflexpr(static) meta_static;
typedef reflexpr(public) meta_public;
```

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

```
static_assert(meta::Named<reflexpr(std)>, "");
static_assert(meta::ScopeMember<reflexpr(int)>, "");
static_assert(meta::Scope<reflexpr(std::string)>, "");
static_assert(meta::Alias<reflexpr(std::string::size_type)>, "");
```

or if is falls into a particular category:

```
static_assert(meta::GlobalScope<reflexpr()>, "");
static_assert(meta::Namespace<reflexpr(std)>, "");
static_assert(meta::Type<reflexpr(int)>, "");
static_assert(meta::Class<reflexpr(std::string)>, "");
static_assert(meta::Specifier<reflexpr(virtual)>, "");
```

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 = reflexpr(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_m<meta_str>; // another metaobject reflecting the scope
get_data_members_m<meta_str>; // a metaobject-sequence containing other metaobjects
```

P0194R2 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 P0194R2.

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[12] and will provide access to scope members, base classes, etc.

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

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 = reflexpr(std);
using m_int = reflexpr(int);
using m_foo = reflexpr(foo);
```

instead of for example

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

[12] Like constness, storage-class, access, etc. specifiers
[13] Either by itself or serving as the foundation for other standard or third-party libraries.
[14] Object-oriented, functional, etc.

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 consistent with the syntax of other existing operators in C++:

```
float foo = 42.f;

using foo_type = decltype(foo);
using meta_foo = reflexpr(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, 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<reflexpr(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[15] will eventually be reflected by appropriate metaobjects, but new ones not having an equivalent in the base-level language[16] 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

[15] Within reason, we certainly do not want to reflect every token in a C++ program.
[16] At least conceptually.

lightweight types providing internal links back to the reflected declaration. The actual metadata[17] 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[18].

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".

4.2.9 Integration

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

For example:

```
using meta_int = reflexpr(int);
static_assert(is_integral_v<meta::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 ***Meta-Function*** and ***Meta-OverloadedFunction***.

```
namespace meta {

  template <Object T>
  concept bool Function =
      Named<T> && ScopeMember<T> && Typed<T>
      __metaobject_is_meta_function(T);

  template <Object T>
  concept bool OverloadedFunction =
      Named<T> && ScopeMember<T> &&
      __metaobject_is_meta_overloaded_function(T);
```

[17] Like a compile-time string containing the identifier, the list of metaobjects reflecting class members or the scope of a declaration, etc.
[18] Credits to whoever said that.
[19] Like the *type-traits* or `typeid` and `std::type_info`.

```
} // namespace meta
```

Then it is necessary to extend the list of expressions which can appear as operands to `reflexpr` to include specifiers, so that the following is valid code:

```
using meta_func = reflexpr(std::time);
```

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 meta {

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

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

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

} // namespace meta
```

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[20] and reflection also increases.

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

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

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

[20] already taken care of by `type_traits`

4.4 Design evaluation

4.4.1 The good

The proposed reflection facility[21]:

- 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"[22],
- 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,
- limits the impact on existing code by adding only a single reserved keyword,
- does not require any other changes to the core language, especially no new rules for template parameters.

4.4.2 The bad

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 of C++ projects, hosted on `http://github.com/`[23], 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* occurrence of "reflexpr".

[21] assuming it is fully completed
[22] We cannot stress the importance of this feature enough.
[23] The main branches of original repositories, not forks.

4.4.3 The ugly

The complexity of the proposal[24] makes it too verbose for certain simple use cases or it may be difficult to learn for beginners.

On the other hand we do not want to trade its usefulness for "simplicity" and as already said simplifying wrappers aimed at trivial use cases can and *will* be devised and added to the standard library. See also the discussion in the FAQ section.

Representing the metaobjects as types may be suboptimal in some simple use cases. For example:

```
std::string(std::meta::get_base_name_v<reflexpr(T)>);
```

However, the compilers already are doing many other AST transformations, optimizations and elisions, and their developers have many tricks at their disposal to make the above as efficient as say:

```
std::string(operator_get_base_name_of(T));
```

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.

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 `typedef`s. 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 ***Meta-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.

[24] stemming from the complexity of the base-level language

5.2 Logging

When tracing and logging the execution of functions[25] 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 */ }
#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<reflexpr(T)>
            << ">(" << a << ", " << b << ") = ";

    T result = a<b?a:b;

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

[25] especially template functions

```
    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_m<reflexpr(T)>>
            << ">("
            << get_base_name_v<reflexpr(a)> << ": "
            << get_display_name_v<get_aliased_m<get_type_m<reflexpr(a)>>>
            << " = " << a
            << get_base_name_v<reflexpr(b)> << ": "
            << get_display_name_v<get_aliased_m<get_type_m<reflexpr(b)>>>
            << " = " << b
            << ")" << std::endl

    T result = a<b?a:b;

    log()   << get_base_name_v<reflexpr(result)> << ": "
            << get_display_name_v<get_aliased_m<get_type_m<reflexpr(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<reflexpr(a)> << " = "
    << get_base_name_v<reflexpr(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 ***Meta-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(reflexpr(this::function), std::tie(a, b));
    /* ... */
}

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

```
template <typename T>
T avg(T a, T b)
{
    log_function_exec(reflexpr(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[26] which may lead to confusion.

5.3 Generation of common functions

This use case was part of the "targeted use cases" in the committee's call for compile-time reflection proposals [8]:

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 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;
```

[26] 32-bit vs. 64-bit or debug vs. release

```
    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 <meta::Object MetaDataMember>
    void operator()(MetaDataMember) const
    {
        auto mem_ptr = meta::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 = reflexpr(T);
    bool result = true;

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

    return result;
}
```

If the reversible reflection feature described in section 6.3 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 <meta::Object MetaDataMem>
    void operator()(MetaDataMem) const
    {
        result &= a.unreflexpr(MetaDataMem) == b.unreflexpr(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;

    meta::for_each<meta::get_data_members_m<reflexpr(T)>>(
        [&result,&a](auto meta_dm)
        {
            using MetaDataMem = decltype(meta_dm)::type;
            using MetaT = meta::get_type_m<MetaDataMem>;

            using T = meta::get_reflected_type_t<MetaT>;
            // or T = unreflexpr(metaT);

            auto mem_ptr = meta::get_pointer_v<MetaDataMem>;

            result ^= std::hash<T>(a.*mem_ptr);
            // or  ^= std::hash<T>(a.unreflexpr(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 [8].

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] shows a possible implementation of the `enum_to_string` utility,

```
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(
```

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

        using ME = reflexpr(Enum);
        using hlpr = meta::unpack_sequence_t<
            meta::get_enumerators_m<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(meta::get_base_name<MEC>()),
                meta::get_constant_v<MEC>
            ).second...);
            return res;
        }
    };
public:
    Enum operator()(const std::string& s) const
    {
        using namespace std;
```

```
        using ME = reflexpr(Enum);
        using hlpr = meta::unpack_sequence_t<
            meta::get_enumerators_m<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 <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_m<Metaobjects, I> meta_F;

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

    out << '"' << std::meta::get_base_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_data_members_m<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::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 an already working code and it 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 Cross-cutting aspects

We need to execute the same action or a set of unrelated actions at the entry of or at the exit from the body of each function from a set of multiple functions meeting some criteria every time one of them is called.

The actions may be related to logging, debugging, profiling, but also to access control[27], etc.

The condition which selects the functions for which the action is invoked might be something like:

- each member function of a particular class,
- each function defined in some namespace,
- each function returning values of a particular type or having a particular set of parameters,
- each function whose name matches a pattern,
- each function declared in a particular source file,
- and so on and various combinations of the above.

It may not be possible to tell in advance the relations between the aspects and the individual functions or these relations may vary for different builds or build configurations. Furthermore we want to be able to quickly change the assignment of actions to functions in one place instead of going through the whole project source which may consists of dozens or even hundreds of files.

We want for example temporarily enable logging of the entry and exit of each member function of class `foo`, or we need to count the number of invocations of functions defined in the `bar` namespace with names not starting with an underscore, or we want to throw the `not_logged_in` exception at the entry of each member function of class `secure` if the global function `user_logged_in` returns `false`.

Without reflection something like this could be implemented in the following way:

[27] Not allowing users to execute operations for which they are not authorized.

```
class logging_aspect
{
public:
    template <typename ... P>
    logging_aspect(const char* func_name, P&&...)
    {
        // log entry to std::clog
    }

    ~logging_aspect(void)
    {
        // log exit
    }
};

class profiling_aspect
{
    /* ... */
};

class authorization_aspect
{
public:
    template <typename ... P>
    authorization_aspect(const char* func_name, P&&...)
    {
        if(contains(func_name, "secure"))
        {
            if(!::is_user_logged_in())
            {
                throw not_logged_in(func_name);
            }
        }
    }
};

template <typename RV, typename ... P>
class func_aspects
 : logging_aspect
 , profiling_aspect
 , authorization_aspect
/* ... etc. ... */
{
public:
    func_aspects(
        const char* name,
        const char* file,
        unsigned line,
```

```
        P&&... args
    ): logging_aspect(name, file, line, args...)
     , profiling_aspect(name, file, line, args...)
     , authorization_aspect(name, file, line, argc...)
    /* ... etc. ... */
    { }
};

template <typename RV, typename ... P>
func_aspects<RV, P...>
make_func_aspects(
    const char* name,
    const char* file,
    unsigned line,
    P&&...args
);


void func1(int a, int b)
{
    const auto _fa = make_func_aspects<void>(
        __func__,
        __FILE__,
        __LINE__,
        a, b
    );
    /* function body */
}

double func2(double a, float b, long c)
{
    const auto _fa = make_func_aspects<double>(
        __func__,
        __FILE__,
        __LINE__,
        a, b, c
    );
    /* function body */
}

namespace foo {

long func3(int x)
{
    const auto _fa = make_func_aspects<long>(
        __func__,
        __FILE__,
        __LINE__,
```

```
        x
    );
    /* function body */
}

} // namespace foo
```

Obviously this is very repetitive and it can get quite tedious and error-prone to supply all this information to the aspects in each function manually. Also if the signature or the name of the function changes the construction of the `func_aspects` instance must be updated accordingly. With the help of reflection things can be simplified considerably:

```
template <typename MetaFunction, typename Enabled>
class logging_aspect_impl;

template <typename MetaFunction>
class logging_aspect_impl<MetaFunction, false_type>
{ };

template <typename MetaFunction>
class logging_aspect_impl<MetaFunction, true_type>
{
public:
    logging_aspect_impl(void)
    {
        clog
            << get_base_name_v<MetaFunction>
            << "("
            /* ... */
            << ")"
            << endl;
    }
};

template <typename MetaFunction>
constexpr bool logging_enabled =
    is_same_v<
        reflexpr(std),
        get_scope_m<MetaFunction>
    > && is_same_v<
        std::string,
        get_reflected_type_t<get_result_m<MetaFunction>>
    > &&
    /* ... etc. ... */
;

template <typename MetaFunction>
using logging_aspect =
    logging_aspect_impl<
```

```
        MetaFunction,
        logging_enabled<MetaFunction>
    >;

template <typename MetaFunction, typename Enabled>
class authorization_aspect_impl;

template <typename MetaFunction>
class authorization_aspect_impl<MetaFunction, false_type>
{ };

template <typename MetaFunction>
class authorization_aspect_impl<MetaFunction, true_type>
{
public:
    authorization_aspect_impl(void)
    {
        if(!::is_user_logged_in())
        {
            throw not_authorized(
                full_name<MetaFunction>()
            );
        }
    }
};

template <typename MetaFunction>
struct autorization_enabled
 : integral_constant<
    bool,
    is_base_of<
        reflexpr(foo::bar),
        get_scope_m<MetaFunction>
    > && constexpr_starts_with(
        get_base_name_v<MetaFunction>,
        "secure_"
    ) &&
    /* ... etc. ... */
;

template <typename MetaFunction>
using authorization_aspect =
    authorization_aspect_impl<
        MetaFunction,
        typename authorization_enabled<MetaFunction>::type
    >;

template <typename MetaFunction>
```

```
class func_aspects
 : logging_aspect<MetaFunction>
 , profiling_aspect<MetaFunction>
 , authorization_aspect<MetaFunction>
/* ... etc. ... */
{
public:
};

void func1(int a, int b)
{
    const func_aspects<reflexpr(this::function)> _fa;
    /* function body */
}

double func2(double a, float b, long c)
{
    const func_aspects<reflexpr(this::function)> _fa;
    /* function body */
}

namespace foo {

long func3(int x)
{
    const func_aspects<reflexpr(this::function)> _fa;
    /* function body */
}

} // namespace foo
```

In this case the same expression is used in all functions regardless of their name and signature and the aspects get all the information they require from the metaobject reflecting the function. All the data obtained from the metaobjects is available at compile-time so various specializations of the aspect classes can be implemented as required.

This same technique could also be used with instances of classes:

```
template <typename MetaClass>
class class_aspects
 : logging_aspects<MetaClass>
/* ... etc. ... */
{
public:
    class_aspects(get_reflected_type_t<MetaClass>* that);
};

class cls1
{
```

```
private:
    int member1;
    /* ... other members ... */
    class_aspect<reflexpr(this::class)> _ca;
public:
    cls1(void)
     : member1(...)
     , _ca(this)
    { }
};
```

Class aspects like these could also be used for logging, monitoring of object instantiation, resource leak detection, etc.

5.7 Implementing the factory pattern

The purpose of the *Factory* pattern is to separate its caller, who requires a new instance of a *Product* type, from the details of this instance's construction. The caller only supplies the input data to the factory and collects the new instance. There are several aspects that need to be considered when designing and implementing a factory.

The input data for the construction of an instance of the *Product* can be stored in an external representation (an XML fragment, a RDBS database dataset, a JSON document, etc.) or even entered by the user through a GUI oron the command-line and so on, and would need to be converted into a native C++ representation. The new instance also might be constructed as a copy of another already existing *prototype* instance of the same type sitting in an object pool.

The product may be polymorphic and the exact type may not even be known to the user. It may have one or several constructors, each of which may require a different set of arguments. It may or may not have constructors with a specific signature, for example a default constructor.

A default constructor does not make sense for many types and requiring it just because the type will be used with a factory is problematic[28]. Consider for example what a "default" instance of `person` or `address` would look like – it would not have any meaning at all. Thus well-designed factories should not depend on the presence of constructors with specific signatures.

Furthermore it might be desirable, that the constructor used to construct a particular instance is picked based on the available input data which is known only at run-time, but not when the factory is designed and implemented.

Let's consider the implementation of a factory for a rather simple `point` class, representing a point in 3-dimensional space:

```
struct point
{
    double _x, _y, _z;

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

[28] or even impossible with third-party code

```
    point(double w): _x(w), _y(w), _z(w) { }

    point(void): _x(0.0), _y(0.0), _z(0.0) { }

    point(const point&) = default;

    // ... other declarations
};
```

A naive hand-coded implementation, of a factory constructing `point`s from some `Data` type (for example an XML node) might look like this:

```
class point_factory
{
private:
    unsigned pick_constructor(Data data)
    {
        // somehow examine the data and pick
        // the most suitable constructor of the point class
    }

    double extract(Data data, string param)
    {
        // somehow extract and convert the value
        // of a named parameter from the data
    }
public:
    point create(Data data)
    {
        switch(pick_constructor(data))
        {
            case 0: return point();

            case 1: return point(extract(data, "w"));

            case 2: return point(
                        extract(data, "x"),
                        extract(data, "y"),
                        extract(data, "z")
                    );

            default: throw exception(...);
        }
    }
};
```

Now suppose that there is some pool of existing `point` objects and let's extend the factory to use this pool and return copies if applicable:

```
extern pool_of<point>& point_pool;
```

```
class point_factory
{
private:
    unsigned pick_constructor(Data data)
    {
        // same as before but also allow the copy
        // constructor to be picked if the data says so
    }

    double extract(Data data, string param);
public:
    point create(Data data)
    {
        switch(pick_constructor(data))
        {
            // same as before, but add a new case
            // returning copies from the pool

            case 3: return point_pool.get(data);

            default: throw exception(...);
        }
    }
};
```

When looking at the hand-coded factories above, it is obvious that implementing and maintaining[29] factories for several dozens of classes in a larger application is a highly repetitive, tedious and possibly error-prone process and at least partial automation is desirable.

Factory classes must generally handle several tasks which fit into two distinct and nearly orthogonal categories:

- *Product type-related*
  - *Constructor description* – providing the metadata describing the individual constructors, their parameters, etc.
  - *Constructor dispatching* – calling the selected constructor. with the supplied arguments which results in a new instance of the product type.
- *Input data representation-related*
  - *Input data validation* – checking if the input data match the available constructors.
  - *Constructor selection* – examining the input data, comparing it to the metadata describing product's constructors and determining which constructor should be called.
  - *Getting the argument values* – determining where the argument values should come from and getting them:

[29] as the constructed types evolve and change

* *Conversion from the external representation* – this usually applies to intrinsic C++ types, but complex types could be converted directly too.
* *Recursive construction by using another factory* – this usually requires some form of cooperation between the parent and its child factories and it means that all the tasks discussed here must be repeated also for the recursively constructed parameter(s).
* *Copying an existing instance* – for example from an object pool.

Parts from each category can be combined with parts from the other to create new factories which promotes code re-usability. Factories constructing instances of a single product from various data representations share the product-related components and factories constructing instances of various product types from a single input data representation share the input-data-related parts. This approach has several advantages like better maintainability or the ability to develop the components separately and combine them later via metaprogramming.

If the input data for a metaprogram generating the factory class, that is the metadata describing the Product type[30] can be obtained by using compile-time reflection then new factory classes can be generated automatically for nearly arbitrary type provided that the input data type-related parts are implemented.

The scope of this paper does not allow to fully explain the implementation of the factory generators. Please see [4] for further details.

5.8 SQL schema generation

We need to create an SQL/DDL (data definition language) script for creating a schema with tables which will be storing the values of all structures in namespace C++ `foo` having names starting with `persistent_`:

```
const char* translate_to_sql(const std::string& type_name)
{
    if(type_name == "int")
        return "INTEGER";
    if(type_name == "float")
        return "FLOAT";
    /* .. etc. */
}

struct create_table_column_from
{
    template <typename MetaMemVar>
    void operator()(MetaMemVar)
    {
        if(!meta::Variable<MetaMemVar>) return;

        std::cout << meta::get_base_name_v<MetaMemVar> << " ";
```

[30] specifically the constructors of Product

```
        std::cout << translate_to_sql(
            meta::get_base_name_v<meta::get_type_m<MetaMemVar>>
        );

        if(starts_with(get_base_name_v<MetaMemVar>, "id_"))
        {
            std::cout << " PRIMARY KEY";
        }
        std::cout << "," << std::endl;
    }
};

struct create_table_from
{
    const char* schema_name;

    template <typename MetaClass>
    void operator()(MetaClass)
    {
        if(!meta::Class<MetaClass>) return;

        if(!starts_with(
            get_base_name_v<MetaClass>,
            "persistent_"
        )) return;

        std::cout << "CREATE TABLE "
                  << schema_name << "."
          << strip_prefix("persistent_", get_base_name_v<MetaClass>)
          << "(" << std::endl;

        meta::for_each<meta::get_data_members_m<MetaClass>>(
            create_table_column_from()
        );

        std::cout << ");"
    }
};

struct create_schema_from
{
    template <typename MetaNamespace>
    void operator()(MetaNamespace)
    {
        std::cout << "CREATE SCHEMA "
          << get_base_name_v<MetaNamespace>
          << ";" << std::endl;
```

```
        meta::for_each<meta::get_member_types<MetaNamespace>>(
            create_table_from{meta::get_base_name_v<MetaNamespace>}
        );
    }
};

int main(void)
{
    create_schema_from create;
    create(reflexpr(foo){});
    return 0;
}
```

Furthermore reflection can be used to implement actual object-relational mapping, together with a library like `SOCI`, `ODBC`, `libpq` or similar as shown for example in [2].

5.9 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 rdbs_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 [8]:

```
struct soa_foo
{
    vector<bool> bs;
```

```
    vector<char> cs;
    vector<double> sd;
    vector<float> fs;
    vector<string> ss;
};
```

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 `idreflexpr` as described in 6.4.1 , which "creates" an identifier from a compile-time string, so that for example:

```
idreflexpr("long") idreflexpr("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 `char`s:

```
struct bar
{
    int foo;
};

using meta_bar_foo = reflexpr(bar::foo);

long idreflexpr(meta::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 idreflexpr(ct_concat("get_", meta::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<meta::get_reflected_type_t<meta::get_type_m<MetaDataMember>>>
        idreflexpr(ct_concat(meta::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
 : meta::unpack_sequence_t<
    meta::get_data_members_m<reflexpr(T)>,
    soa_inherit_all
> { /* constructors forwarding parameters to soa_inherit_all */ };

using soa_foo = soa<foo>;
```

If we don't want to implement the magic operator `idreflexpr`, we could instead add a new template `named_data_member` for ***Meta-Named*** metaobjects, described in greater detail in section 6.4.3:

```
template <MetaNamed MO, typename X>
struct named_data_member;
```

A specialization of `named_data_member` for a metaobject reflecting for example `foo::c`, would look like this:

```
template <typename X>
struct named_data_member<reflexpr(foo::c), X>
{
    struct type
    {
        X c;

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

We can then combine the specializations of `named_data_member<...>::type` via multiple inheritance as before:

```
template <typename MetaDataMember>
using soa_single_member = meta::named_data_member_m<
    MetaDataMember,
    vector<meta::get_reflected_type_t<meta::get_type_m<MetaDataMember>>>
>;

template <typename ... MetaDataMember>
struct soa_inherit_all
 : soa_single_member<MetaDataMembers>...
```

```
{ /* constructors forwarding parameters to the inherited soa_single_members */ };

template <typename T>
struct soa
 : meta::unpack_sequence_t<
    meta::get_data_members_m<reflexpr(T)>,
    soa_inherit_all
> { /* constructors forwarding parameters to soa_inherit_all */ };

using soa_foo = soa<foo>;
```

Note that with this workaround we don't have the ability to change the name of the data members from `c` to `cs`, etc., but the `named_data_member` template is potentially much simpler to implement.

For a more detailed discussion on the possibilities of identifier generation see section 6.4 and also section 6.5.

5.10 Simple examples of usage

This section shows several simple, but complete examples of usage already working with the experimental implementation of the initial specification.

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

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

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

    static_assert(meta::Namespace<meta_int_s>, "");
    static_assert(meta::GlobalScope<meta_int_s>, "");
    static_assert(meta::Namespace<meta_foo_bar_s>, "");
    static_assert(!meta::GlobalScope<meta_foo_bar_s>, "");
    static_assert(meta::Type<meta_foo_bar_baz_s>, "");
    static_assert(meta::Class<meta_foo_bar_baz_s>, "");
    static_assert(!meta::Namespace<meta_foo_bar_baz_s>, "");
    static_assert(meta::Namespace<meta_foobar_s>, "");
    static_assert(meta::GlobalScope<meta_foobar_s>, "");
    static_assert(!meta::Class<meta_foobar_s>, "");

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

    return 0;
}
```

Output:

```
baz
bar
foo
```

5.10.2 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::GlobalScope<meta_foo>, "");
    static_assert(meta::Namespace<meta_foo>, "");
    static_assert(!meta::Type<meta_foo>, "");
    static_assert(!meta::Alias<meta_foo>, "");

    static_assert(meta::Named<meta_foo>, "");
    static_assert(meta::ScopeMember<meta_foo>, "");
    cout << meta::get_base_name_v<meta_foo> << endl;

    typedef reflexpr(foo::bar) meta_foo_bar;

    static_assert(is_metaobject_v<meta_foo_bar>, "");

    static_assert(!meta::GlobalScope<meta_foo_bar>, "");
    static_assert(meta::Namespace<meta_foo_bar>, "");
    static_assert(!meta::Type<meta_foo_bar>, "");
    static_assert(!meta::Alias<meta_foo_bar>, "");

    static_assert(meta::Named<meta_foo_bar>, "");
    static_assert(meta::ScopeMember<meta_foo_bar>, "");
    cout << meta::get_base_name_v<meta_foo_bar> << endl;

    typedef reflexpr(foobar) meta_foobar;

    static_assert(is_metaobject_v<meta_foobar>, "");

    static_assert(!meta::GlobalScope<meta_foobar>, "");
    static_assert(meta::Namespace<meta_foobar>, "");
    static_assert(!meta::Type<meta_foobar>, "");
    static_assert(meta::Alias<meta_foobar>, "");

    static_assert(meta::Named<meta_foobar>, "");
    static_assert(meta::ScopeMember<meta_foobar>, "");
    cout << meta::get_base_name_v<meta_foobar> << " a.k.a ";
    cout << meta::get_base_name_v<meta::get_aliased_m<meta_foobar>> << endl;

    return 0;
}
```

Output:

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

5.10.3 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::Type<meta_unsigned>, "");
    static_assert(!meta::Alias<meta_unsigned>, "");

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

    static_assert(meta::Named<meta_unsigned>, "");
    cout << meta::get_base_name_v<meta_unsigned> << endl;

    typedef reflexpr(unsigned*) meta_ptr_unsigned;
    static_assert(meta::Named<meta_ptr_unsigned>, "");
    cout << meta::get_base_name_v<meta_ptr_unsigned> << endl;

    return 0;
}
```

Output:

```
unsigned int
unsigned int
```

5.10.4 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::Type<meta_foo_baz>, "");
    static_assert(meta::Alias<meta_foo_baz>, "");

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

    static_assert(meta::Named<meta_foo_baz>, "");
    cout << meta::get_base_name_v<meta_foo_baz> << endl;

    typedef meta::get_aliased_m<meta_foo_baz> meta_foo_bar;

    static_assert(is_metaobject_v<meta_foo_bar>, "");
    static_assert(meta::Type<meta_foo_bar>, "");
    static_assert(meta::Alias<meta_foo_bar>, "");

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

    static_assert(meta::Named<meta_foo_bar>, "");
    cout << meta::get_base_name_v<meta_foo_bar> << endl;

    typedef meta::get_aliased_m<meta_foo_bar> meta_int;

    static_assert(is_metaobject_v<meta_int>, "");
    static_assert(meta::Type<meta_int>, "");
    static_assert(!meta::Alias<meta_int>, "");

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

    static_assert(meta::Named<meta_int>, "");
```

```
    cout << meta::get_base_name_v<meta_int> << endl;

    return 0;
}
```

Output:

```
baz
bar
int
```

5.10.5 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::Type<meta_bar>, "");
    static_assert(meta::Class<meta_bar>, "");
    static_assert(meta::Alias<meta_bar>, "");

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

    static_assert(meta::Named<meta_bar>, "");
    cout << meta::get_base_name_v<meta_bar> << endl;

    typedef meta::get_aliased_m<meta_bar> meta_foo;

    static_assert(is_metaobject_v<meta_foo>, "");
    static_assert(meta::Type<meta_foo>, "");
    static_assert(meta::Class<meta_foo>, "");
    static_assert(!meta::Alias<meta_foo>, "");

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

    static_assert(meta::Named<meta_foo>, "");
    cout << meta::get_base_name_v<meta_foo> << endl;

    return 0;
```

```
}
```

Output:

```
bar
foo
```

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

5.10.6 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_public_data_members_m<meta_foo> meta_data_mems;

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

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

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

    static_assert(is_metaobject_v<meta_data_mem0>, "");
    static_assert(meta::Variable<meta_data_mem0>, "");
    static_assert(meta::Typed<meta_data_mem0>, "");

    cout << meta::get_base_name_v<meta_data_mem0> << endl;
```

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

    static_assert(is_metaobject_v<meta_data_mem2>, "");
    static_assert(meta::Variable<meta_data_mem2>, "");
    static_assert(meta::Typed<meta_data_mem2>, "");

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

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

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

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

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

    static_assert(is_metaobject_v<meta_all_data_mem0>, "");
    static_assert(meta::Variable<meta_all_data_mem0>, "");
    static_assert(meta::Typed<meta_all_data_mem0>, "");

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

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

    static_assert(is_metaobject_v<meta_all_data_mem3>, "");
    static_assert(meta::Variable<meta_all_data_mem3>, "");
    static_assert(meta::Typed<meta_all_data_mem3>, "");

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

    return 0;
}
```

This produces the following output:

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

5.10.7 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_m<Metaobjects, I> metaobj;

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

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

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

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

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

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

    do_print_data_members<meta::get_data_members_m<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
```

5.10.8 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_public_data_members_m<meta_A>> << endl;
    cout << meta::get_size_v<meta::get_public_data_members_m<meta_B>> << endl;
    cout << meta::get_size_v<meta::get_public_data_members_m<meta_C>> << endl;

    cout << meta::get_size_v<meta::get_data_members_m<meta_A>> << endl;
    cout << meta::get_size_v<meta::get_data_members_m<meta_B>> << endl;
    cout << meta::get_size_v<meta::get_data_members_m<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_public_data_members` includes the inherited data members.

6 The unpredictable future

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 ***Meta-ObjectSequence***. It should *be equivalent* to the following:

```
namespace meta {

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

} // namespace meta
```

For example:

```
meta::for_each<meta::get_data_members_m<reflexpr(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 ***Meta-ObjectSequence***.

```
namespace meta {

  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 meta
```

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

```
namespace meta {

  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 meta
```

6.2 Context-dependent reflection

One of the interesting potential future extensions of reflection is *context-dependent reflection*. Currently the operands of `reflexpr` 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.

Several new expressions would have to be added as valid operands for `reflexpr`, allowing to reflect the "current" namespace, class or even function.

6.2.1 Namespaces

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

For example:

```
typedef reflexpr(this::namespace) _meta_gs;
```

reflects the global scope namespace and is equivalent to

```
typedef reflexpr(::) _meta_gs;
```

For named namespaces:

```
namespace foo {

  typedef reflexpr(this::namespace) _meta_foo;

  namespace bar {

    typedef reflexpr(this::namespace) _meta_foo_bar;

    } // namespace bar

} // namespace foo
```

6.2.2 Classes

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

For example:

```
struct foo
{
    const char* _name;

    // reflects foo
    typedef reflexpr(this::class) _meta_foo1;
```

```
    foo(void)
     : _name(get_base_name_v<reflexpr(this::class)>())
    { }

    void f(void)
    {
        // reflects foo
        typedef reflexpr(this::class) _meta_foo2;
    }

    double g(double, double);

    struct bar
    {
        // reflects foo::bar
        typedef reflexpr(this::class) _meta_foo_bar;
    };
};

double foo::g(double a, double b)
{
    // reflects foo
    typedef reflexpr(this::class) _meta_foo3;
    return a+b;
}

class baz
{
private:
    typedef reflexpr(this::class) _meta_baz;
};

typedef reflexpr(this::class); // error: not used inside of a class.
```

6.2.3 Functions

If the `this::function` expression was used as the argument of the reflection operator, then it would return a ***Meta-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
    typedef reflexpr(this::function) _meta_foobar;
}
```

```
int foobar(int i)
{
    // reflects this particular overload of the foobar function
    typedef reflexpr(this::function) _meta_foobar;
    return i+1;
}

class foo
{
private:
    void f(void)
    {
        // reflects this particular overload of foo::f
        typedef reflexpr(this::function) _meta_foo_f;
    }

    double f(void)
    {
        // reflects this particular overload of foo::f
        typedef reflexpr(this::function) _meta_foo_f;
        return 12345.6789;
    }
public:
    foo(void)
    {
        // reflects this constructor of foo
        typedef reflexpr(this::function) _meta_foo_foo;
    }

    friend bool operator == (foo, foo)
    {
        // reflects this operator
        typedef reflexpr(this::function) _meta_foo_eq;
    }

    typedef reflexpr(this::function) _meta_fn; // error
};

typedef reflexpr(this::function) _meta_fn; // error
```

6.2.4 Templates

If the `this::template` expression was used as the argument of the reflection operator, then it would return a ***Meta-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 = reflexpr(this::template);

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

6.3 Reversing reflection

The `reflexpr` 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 ***Meta-Reversible***.

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

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 ***Meta-Type***:

```
using MT = reflexpr(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(meta::Reversible<MT> && meta::Type<MT>, "");
using T = unreflexpr(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 = reflexpr(foo::str); // Meta-Variable

// ... lots of code here ...
```

```
// using namespace foo;
using namespace unreflexpr(get_scope_m<MV>);

// [foo::]bar(foo::str);
bar(unreflexpr(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
    unreflexpr(MN)::func1(str);

    // [foo|bar]::func2([foo|bar]::func3(i*j, 64, i == j), i, j);
    unreflexpr(MN)::func2(unreflexpr(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<reflexpr(foo)>(str, i, j);
    }
    else
    {
        algorithm<reflexpr(bar)>(str, i, j);
    }
}
```

or get back to the original function;

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

using MF = reflexpr(foo::bar); // Meta-Function
```

```
// ... lots of code here ...
//   foo::bar("hello");
unreflexpr(MF)("hello");
```

or back to the original class data member;

```
struct my_struct
{
    int i;
    float f;
};

// Meta-DataMember
using MDM = reflexpr(foo::bar::i);

my_struct x {123, 45.67f};

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

or go back to the original template;

```
// Meta-Template
using MTpl = reflexpr(std::pair);

// std::pair<int, std::string> p;
unreflexpr(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[31]

6.4 Generating identifiers programmatically

Some of the use cases from the committee's CFP, for example the one described in section 5.9, depend on the ability to generate identifiers programmatically.

6.4.1 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 `idreflexpr(const char (&)[N])`[32], which "creates" an identifier from an arbitrary compile-time string, so that:

[31] At the cost of the round-trip through the meta-level.
[32] We do not insist on this particular name in case it should be implemented.

```
idreflexpr("std")::idreflexpr("size_t")
idreflexpr("foo")(int idreflexpr("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 `meta::get_base_name` operation, return `constexpr` arrays of `char`s. 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 = reflexpr(bar::foo);

std::size_t idreflexpr(meta::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 = reflexpr(bar::foo);

int idreflexpr(ct_concat("get_", meta::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.

6.4.2 Identifier formatting

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

```
struct bar
{
    int foo;
};

using meta_bar = reflexpr(bar);
using meta_foo = reflexpr(bar::foo);
```

```
idreflexpr("int")
idreflexpr("get_%1_%2", meta_bar, meta_foo)(const idreflexpr("%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.

6.4.3 `named_data_member` and `named_member_typedef`

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

```
template <MetaNamed MO, typename X>
struct named_member_typedef
{
    struct type
    {
        // same as
        // typedef X idreflexpr(meta::get_base_name_v<MO>);
        typedef X __builtin_idreflexpr_of(MO);
    };
};

template <MetaNamed MO, typename X>
struct named_data_member
{
    struct type
    {
        // same as
        // X idreflexpr(meta::get_base_name_v<MO>);
        X __builtin_idreflexpr_of(MO);

        template <typename ... P>
        type(P&& ... p)
         : __builtin_idreflexpr_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<reflexpr(bar::foo), X>
{
    struct type
    {
        typedef X foo;
    };
};

template <typename X>
struct named_data_member<reflexpr(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.

6.5 Variadic composition

Another feature which[33] 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 5.9 could be simplified by using variadic composition, together with the "identifier formatting operator" described in section 6.4.2 and the `unpack_sequence` helper template described in section 6.1.2:

[33] together with the ability to generate identifiers

```
template <DataMember ... MDM>
struct soa_helper
{
    std::vector<meta::get_reflected_type_t<meta::get_type_m<MDM>>>
    idreflexpr("vec_%1", MDM)...;
};

template <typename T>
using soa = meta::unpack_sequence_t<
    meta::get_data_members_m<reflexpr(T)>,
    soa_helper
>;
```

With the "reversible reflection" feature described in section 6.3, the `soa_helper` could be implemented as:

```
template <DataMember ... MDM>
struct soa_helper
{
    std::vector<unreflexpr(meta::get_type_m<MDM>)>
    idreflexpr("vec_%1", MDM)...;
};
```

6.6 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 meta {

  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 meta
```

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 meta {

  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 meta
```

As long as an instance of `meta::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 `reflexpr`:

```
// pseudocode
constexpr Object reflexpr(meta::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.

6.7 Future extensions of the reflection operator

Besides the primary function of the `reflexpr` operator – reflecting, base-level declarations, there are several potential additional uses for it, or for related operators like (`unreflexpr` and `idreflexpr`), some of which are also mentioned in other sections. These are briefly reiterated here:

- context-dependent reflection, described in section 6.2,
- reverse reflection, described in section 6.3,
- generating identifiers, described in section 6.4.2,
- turning a `constexpr` metaobject unique-identifier back to the metaobject type as described in section 6.6.

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[34] or to indicate that the user really means to do something what in other circumstances could be a bug[35] 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[36] 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[37], 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 ***Meta-Attribute*** concept, make it conform to ***Meta-Named*** and ***Meta-Scoped*** concepts and then add an operation to the interface of ***Meta-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.

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.

[34] for example the `probably(true)` or `carries_dependency` attributes
[35] like the `fallthrough` or `maybe_unused` attributes
[36] Associating one or more constant values with the declarations.
[37] or to ignore attributes which it does not understand

8 Experimental implementation

A fork of the clang compiler with a initial, partial, experimental implementation of this proposal can be found on github [3]. The modified compiler can be built by following the instructions listed in [1], but instead of checking out the official clang repository, the sources on the `reflexpr` branch of the modified repository should be used.

The reflection implementation represents the metaobjects as compile-time constants wrapped in a class template and implements the metaobject operations with the help of compiler built-in operators taking the metaobject ID values as arguments.

It adds a new native integral `__metaobject_id` type which has the same bit-width as the `std::uintptr_t` type on the platform for which the compiler is built (this allows values of `__metaobject_id` to represent pointers in the compiler). Unlike other integral types, `__metaobject_id` does not have any arithmetic operators and no conversions to other integral types.

The metaobjects are represented as specializations of the `std::__metaobject` template:

```
namespace std {

template <__metaobject_id MoId>
struct __metaobject
{
        __metaobject() = default;
        __metaobject(const metaobject&) = default;
};

} // namespace std
```

The metaobject ID can be obtained from a metaobject (type) by using the internal helper template `__unwrap_id`:

```
namespace std {

template <typename T>
struct __unwrap_id;

template <__metaobject_id MoId>
struct __unwrap_id<__metaobject<MoId>>
{
        static constexpr __metaobject_id value = MoId;
};

template <typename T>
constexpr __metaobject_id __unwrap_id_v = __unwrap_id<T>::value;

} // namespace std
```

The proposed `std::is_metaobject` trait can then be implemented as follows:

```
namespace std {
```

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

template <__metaobject_id MoId>
struct is_metaobject<__metaobject<MoId>> : true_type { }

} // namespace std
```

The invocation of the `reflexpr` operator yields specializations of the `std::__metaobject` template, where the value of the `MoId` argument is a handle to a representation of the metaobject inside of the compiler, as described below.

The metaobject operations and concepts are implemented with the help of compiler built-ins which take one or several arguments where at least one which is a metaobject ID (i.e. a compile-time constant of the `__metaobject_id` type).

These built-ins are then used in the definition of the class templates, template aliases and concepts which implement the public interface of the metaobject.

The built-ins can yield compile-time constant values of the following types:

- *boolean constants* – used mostly by the metaobject concept definitions like `std::meta::Named`, `std::meta::ScopeMember`, etc. and operations like `std::meta::is_anonymous`.
- *integral constants* – for operations like `std::meta::get_source_line` or `std::meta::get_size` and `std::meta::get_constant`.
- *pointer constants* – for operations like `std::meta::get_pointer`.
- *c-string literals* – for operations like `std::meta::get_base_name`.
- *metaobject IDs* – for operations like `std::meta::get_scope`, `std::meta::get_base_classes` or `std::meta::get_element`.

So for example the Named concept is implemented with the help of the `__metaobject_is_meta_named` compiler built-in which returns a boolean compile time constant (equivalent to the following pseudocode):

```
constexpr bool __metaobject_is_meta_named(__metaobject_id moid);
```

The concept as defined in the specification can be then implemented in the following manner:

```
namespace std {
namespace meta {

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

template <Object T>
concept bool Named = __metaobject_is_meta_named(__unwrap_id<T>);

} // namespace meta
} // namespace std
```

The `get_base_name` operation is implemented with the help of the `__metaobject_get_base_name` built-in resulting in a `constexpr char` array:

```
constexpr const char [] __metaobject_get_base_name(__metaobject_id moid);

namespace std {
namespace meta {

template <Named T>
struct get_base_name
{
        constexpr const char value[] =
                __metaobject_get_base_name(__unwrap_id_v<T>);
};

} // namespace meta
} // namespace std
```

There is also the `std::meta::get_reflected_type` operation which instead of a compile-time constant value, yields a type. This operation is implemented by the `__unrefltype` built-in operator:

```
unspecified-type __unrefltype(__metaobject_id);
```

The template `get_reflected_type` is then implemented as follows:

```
namespace std {
namespace meta {

template <Type T>
struct get_reflected_type
{
        using type = __unrefltype(__unwrap_id_v<T>);
};

} // namespace meta
} // namespace std
```

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[38] 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.

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

meanwhile without metaobjects:

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

foo<int>(); // works
foo<std>(); // error?
```

[38] As they are defined now.

```
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[39], 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 loose the ability to pass the representation of the scope as parameters to metaprograms.

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[40], 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 `reflexpr(std::string)`, the representations of all the metadata related to the `std::string` type[41] 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

[39] There may be even several dozen.
[40] Just like some of the type traits are implemented.
[41] Like the compile-time representation of its name and the metaobjects reflecting its scope or its members.

and their internal representations need to point to the internal representations of the declarations[42] which they reflect.

Our proposal allows very fine granularity. The result of `reflexpr` 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 Creating a separate type for each string is so heavyweight.

Q: *Isn't the creation of a new type for each string returned for example from the* `get_base_name` *operation too heavyweight? Won't that lead to unacceptable increases in the memory footprint and compilation times?*

A: The answer is that we *do not* insist on creating a separate type for each string returned by reflection. What we insist on is that we should have the ability to reason about and manipulate the strings at compile-time. A static, `constexpr`-initialized, zero-terminated array of `char`s gives us this ability.

So for example the implementation of the `get_base_name` operation should *be equivalent* to the following:

```
template <Named T>
struct get_base_name
{
    typedef const char value_type[N+1];
    static constexpr const char value[N+1] = {...,'\0'};
};
```

Some members of the committee suggested that for example the `basic_string_constant` from N4121 [9] or N4236 [7] or whichever compile-time string representation eventually makes it to the standard could be used to implement the `get_base_name` operation, but this in just an option not a requirement for our part. We trust in the ability of the compiler vendors to pick and implement the best option and to use all the tricks at their disposal to make expressions like;

```
std::string(std::meta::get_base_name_v<reflexpr(std::pair)>);
```

as efficient as possible.

10.5 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.* `meta::get_pointer_v<reflexpr(variable)>` *or* `decltype(var)` *vs.* `meta::get_reflected_type_t<meta::get_type_m<reflexpr(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 metaprogram templates.

[42] Which the compiler needs to maintain anyway, regardless of reflection.

Consider for example the following pseudo-code:

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

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(
            meta::get_base_name_v<MV>,
            meta::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(meta::Variable<MO>)
    {
        handle_meta_var<MO>();
    }
    else if(meta:Class<MO>)
    {
        handle_meta_class<MO>();
    }
    else if(meta:Type<MO>)
    {
        handle_meta_type<MO>();
    }
    else if(meta: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<reflexpr(str)>();
    generic_func_18<reflexpr(std)>();
    generic_func_19<reflexpr(argc)>();
    generic_func_20<reflexpr(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.

10.6 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<reflexpr(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[43] language which means that the reflection mechanism also must be expressive 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.

10.7 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
```

[43] No pun intended.

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 Meta-Constructor */
    { };

    template <something X> // problem
    class meta_namespace /* Model of Meta-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.

10.8 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.

10.9 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[44] 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.

[44] and we do!

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

10.10 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 then 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`,
- `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[45] in several use cases.

- Debugging or trace messages (containing function signatures) are much 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.

10.11 Why Meta-ObjectSequences? Why not replace them with typelists?

Q: *Why do you define the* **Meta-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 ***Meta-ObjectSequence***s 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.

[45] or at least very nice to have

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[46] data member of a class having hundreds of members:

```
get_element_m<get_data_members_m<reflexpr(my_class)>, 0>;
```

This operation would involve the creation of two metaobjects:

1. the ***Meta-ObjectSequence*** and
2. the ***Meta-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 ***Meta-ObjectSequence*** into a typelist on demand, which can be implemented trivially with the `meta::unpack_sequence` template:

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


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

10.12 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 = reflexpr(my_ns::my_class);

std::cout << meta::get_base_name_v<mo> << std::endl;
std::cout << meta::get_base_name_v<meta::get_scope_m<mo>> << std::endl;
meta::for_each<meta::get_data_members_m<mo>>(my_func);
```

do something like one of the following:

A)
```
meta::object mo = reflexpr(my_ns::my_class);

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

B)
```
meta::object mo = reflexpr(my_ns::my_class);

std::cout << mo.get_base_name() << std::endl;
```

[46] zero-th

```
std::cout << mo.get_scope().get_base_name() << std::endl;
mo.get_data_members().for_each(my_func)
meta::for_each(meta::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[47].

Having said that, there is nothing preventing us to implement a façade with similar syntax as either of the above[48], 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], [2] 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[49] can and will be heterogeneous. 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.

10.13 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].

10.14 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 [8]?*

A: Some of these use cases are discussed in the sections 5.3, 5.4 and 5.9 of this paper.

[47] Yes we are aware of the Boost.Hana library and the metaprogramming paradigm which it brings.
[48] and even others
[49] namespaces, native types, structured types, enumerations, functions, etc.

11 Acknowledgments

Thanks to Ricardo Fabiano de Andrade for suggesting the name `unreflexpr` for the reverse reflection operator and suggested some of its use cases. Thanks to Roland Bock for suggesting an important feature, that can be used together with identifier-generation – variadic composition and its use cases. Also thanks to Klaim - Joël Lamotte for reading the draft of this paper and providing valuable feedback and comments.

References

[1] Getting started: Building and running clang,. `http://clang.llvm.org/get_started.html`.

[2] Chochlík, M. Mirror c++ reflection library documentation (previous version). `http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/`, 2012.

[3] Chochlík, M. Github.com: Implementation of the reflexpr 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] Price, M. A compile-time string library template with udl operator templates. Tech. Rep. N4236, 2014.

[8] Snyder, J., and Carruth, C. Call for compile-time reflection proposals. Tech. Rep. N3814, 2013.

[9] Tomazos, A. Compile-time string: std::string_literal. Tech. Rep. N4121, 2014.

Appendix

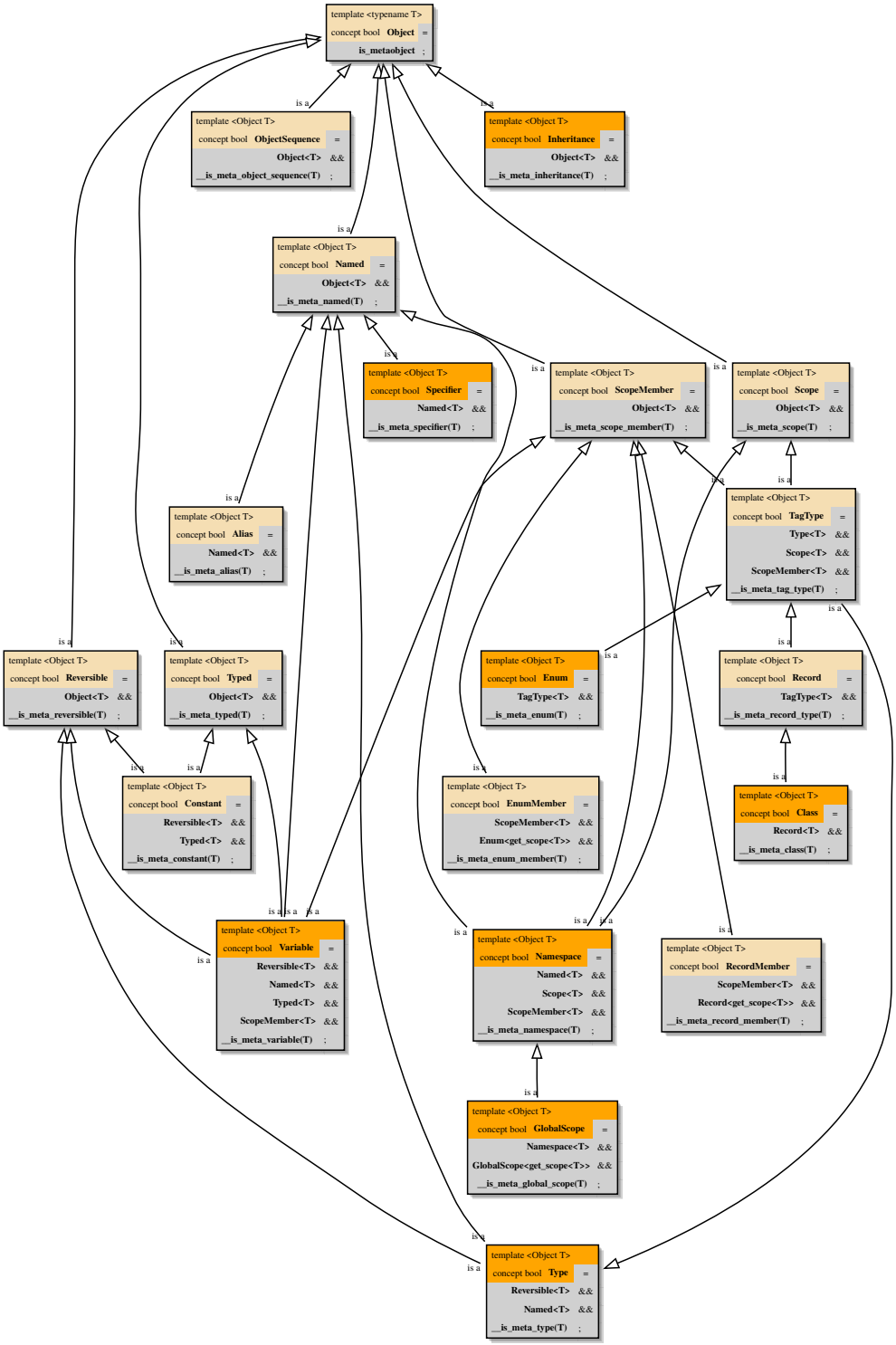

Figure 1: Metaobject concept hierarchy