Value-based Reflection

Document Number: P0993r0
Audience: SG7
Date: Apr 2, 2018
Andrew Sutton <asutton@uakron.edu>
Herb Sutter <hsutter@microsoft.com>

1 Introduction

This paper describes an implementation of value-based reflection for C++ in the Clang compiler, including issues related to the design of the feature and its accompanying library. This is not a motivation paper. Readers are assumed to be (somewhat) familiar with concepts discussed in the various of preceding publications and proposals.

Static reflection is a programming facility that exposes read-only data about entities in a translation unit compile-time values. Static reflection does not require support for runtime compilation since reflection values can be used with existing generative facilities (i.e., templates) or additional generative facilities (i.e., metaprogramming) to produce new code.

In contrast, *dynamic reflection* provides information for navigating source-code data structures at runtime. Language supporting dynamic reflection also tend to make additional facilities available for generating and JIT-compiling new code. Dynamic reflection and code generation are not in the scope of this work.

2 Related work

The proposals by Matúš Chochlík, Axel Naumann, David Sankel, the most recent of which is P0194r5, support reflection by way of template metaprogramming. These proposals are currently going through LWG and CWG review.

The authors of this paper have also proposed a version of static reflection that was, in many ways equivalent to that of Chochlík, Naumann, Sankel (P0590r0). However, this approach supported metaprogramming using `constexpr` evaluation.

A more recent proposal by the same authors propose a variation of reflection that supports reflection through constexpr programming (P0953r0). This proposal changes the meaning of the reflexpr operator so that it is a constant expression that yields a pointer to an object that reflects some construct in the translation unit. For example:

```
template <typename T>
T min(const T& a, const T& b) {
  constexpr reflect::Type const * metaT = reflexpr(T);
  log() << "min<"
        << metaT->get_display_name()
        << ">(" << a << ", " << b << ") = ";
  T result = a < b ? a : b;
  log() << result << std::endl;
  return result;
}
```

The most significant part of the proposal is shown in gray. The object is a base class in a hierarchy of entities that describe the elements of the program. There is one potential problem with this approach. Each reflection would implicitly generate a namespace scope `constexpr` object, probably with internal linkage. Presumably, we don't want these objects to be exchanged between translation units.

This paper modifies our previous work by reflecting values directly in objects rather than in the type of that object. Unlike P0953r0, we the value returned by `reflexpr` is opaque; it is effectively an `int`. Many more details and discussion of our approach follow.

3 Reflection and generative algorithms

Static reflection enables the ability to write algorithms that operate on the semantic structure of entities in a program. We can, for example, use static reflection to define an operator equal to compare the values of classes. That is, we want to do this:

```
struct S {
  int a, b, c;
};

int main() {
  S s1 { 0, 0, 0 };
  S s2 { 0, 0, 1 };
  assert(equal(s1, s1));
  assert(!equal(s1, s2));
}
```

The definition of equal can be defined by two functions. The first, is the "driver", which sets up the reflective algorithm:

```
template<typename T>
bool equal(const T& a, const T& b) {
  constexpr meta::object type = reflexpr(T);
  constexpr meta::object first = meta::first_member(type);
  return compare<first>(a, b);
}
```

This calls a second function, compare, which will recursively compare the members of a and b. The `reflexpr` operator yields a token describing the (eventual) class substituted for T. The type of this value is `meta::object`. The `front` function yields a reflection of the member of the class, whatever that may be.[1]

The `compare` function is non-trivial; it is show below.

```
template<meta::object X, typename T>
bool compare(const T& a, const T& b) {
  if constexpr (!meta::is_null(X)) {
    if constexpr (meta::is_data_member(X)) {
      auto p = valueof(X);
      if (a.*p != b.*p)
        return false;
    }
```

[1] It's probably going to be the injected class name, which I argue is something that should *never* be reflected.

```
      return compare<next(X)>(a, b);
    }
    return true;
  }
```

Note that the algorithms is not constexpr and that the compile-time values of the reflection are passed via template arguments.

We use `constexpr if` to selectively instantiate branches of the function: whenever the reflection X is non-null (i.e., does not correspond to an entity), we can continue recursing through the list. Otherwise, we have reached the end of the list of members, so the objects must be equal.

If X is non-null and also a reflection of a data member, then we can compare the corresponding values of that data member in the objects a and b. The `valueof` operator yields a pointer-to-member value for the reflected data member. This can be used to extract and compare the actual values in a and b.

An implementation using concepts is potentially more clearly specified, although less compact.

```
  // Base case
  template<meta::Null X, typename T>
  bool compare(const T& a, const T& b) {
    return true;
  }

  // Compares data members
  template<meta::DataMember X, typename T>
  bool compare(const T& a, const T& b) {
    auto p = valueof(X);
    if (a.*p != b.*p)
      return false;
    return compare<next(X)>(a, b);
  }

  // Skips non-data members
  template<meta::Object X, typename T>
  bool compare(const T& a, const T& b) {
    return compare<next(X)>(a, b);
  }
```

In this example, the concepts `Object`, `Null` and `DataMember` can be implemented using `meta::` reflection predicates. For example:

```
  template<typename T>
  constexpr bool Null = meta::is_null(T);
```

In both implementatinos above, the reflection value *must* be passed as a template argument due to the use of the `valueof` operator. This operator takes a constant expression operand (a reflection) and yields a value whose type is determined by the entity reflected. Here, in each recursion through the function, `valueof(X)` will yield a pointer to a differently typed data member.

Note that the algorithm cannot be written iteratively:

```
  for (auto m : meta::members(reflexpr(T))) {
    if (meta::is_data_member(m)) {
      auto ptr = valueof(m); // error: m not a constant expression
    }
  }
```

We further note that the earlier proposed expansion-based for loop [p0589r0] will also fail to satisfy the requirement because the loop variable cannot be declared `constexpr` (although that may be fixed in a future version).

This algorithm is *generative*. That is, the composition of the algorithm, not just its computation, depends on the values of reflected entities. In such cases, meta objects must be passed as template arguments. This distinction between normal and generative programming can make value-based reflection difficult to use; it requires implementers to understand when and how to context switch between the different processes of translation and evaluation.

4 Non-generative algorithms

Not all reflective algorithms are generative (although we suspect that most are). Here is a simple example that counts (inaccurately) the number of entities nested within another.

```
constexpr int count(meta::object x) {
  int n = 0;
  for (auto k : x) {
    // See through templates...
    if (meta::is_template(k))
      k = meta::templated_declaration(k);
    // Count recursively
    if (meta::has_members(k))
      n += count(k);
  }
  return 1 + n;
}
```

And it's accompanying use:

```
constexpr int n = count(reflexpr(std));
std::cout << n << '\n';
```

Note that the variable n must be `constexpr` in order to force compile-time evaluation of the `count` function.

In order eliminate the additional declaration, we introduce a new feature: immediate functions. An *immediate function* is one whose result is required to be computed at its call site. This is an extended form of `constexpr`, which allows a function to be evaluated in particular contexts (e.g., as an initializer of a `constexpr` variable).

We could, for example, modify the definition of count like so:

```
immediate int count(meta::object x) {
  // Same as above
}
```

The immediate specifier ensures that the function will be evaluated where it is used. That lets us use the function more simply:

```
std::cout << count(reflexpr(std)) << '\n';
```

Note that, within the definition of count, the `meta::` functions must also be immediate functions. We discuss issues around immediate functions in much more detail in Section

5 Syntax and semantics

This section provides a high-level overview of the changes in the standard necessary to implement this feature.

5.1 Lexical conventions

This proposal adds the following keywords:

```
reflexpr   valueof   idexpr  immediate
```

5.2 New operators

We add a number of new operators to the language. In particular, we add a single reflection operator (takes a name denoting an entity and yields a `meta::object` value), and a number of projection operators (introduces a value, unqualified-id, type-specifier, or nested-name-specifier whose meaning is determined by the value of a `meta::object`).

5.2.1 The reflection operator

There reflection operator has four forms:

```
reflexpr(id-expression)
reflexpr(type-id)
reflexpr(namespace-name)
reflexpr(template-name)
```

In each case the operand of `reflexpr` is a name of some kind. That is, it denotes an entity or set of functions, as determined by the rules of name lookup.

The result of the expression is prvalue `meta::object`. The value can be used with the projection operators and the meta library to query the semantic properties of reflected entities.

Any expression whose type is `meta::object` is called a *reflection*. A reflection is a constant expression and can be used anywhere a constant expression of type `meta::object` is expected (e.g., as a template argument). Throughout, we use reflection as a grammar term to denote a constant expression whose type is `meta::object`.

5.2.2 The idexpr operator

The `idexpr` operator is a variadic projection operator whose form is:

```
idexpr(arg1, arg2, ..., argN)
```

Each argument is a constant expression that is ether a reflection, has integral type, or has type `const char*`. The operator forms an *unqualified-id* whose spelling is the concatenation of the operands as follows:

- If the argument reflects a named declaration, then the identifier of the declaration.
- If the argument has integral type, then the textual representation of the operands value in decimal notation.
- If the argument has type const char*, then the sequence of characters in the null-terminated character array.

- Otherwise, the program is ill-formed.

The *unqualified-id* is then subject to lookup, if used in an expression, or it can be used in a declaration (in which case it is also looked up). For example:

```
void idexpr("foo_", 3)() { } // defines foo_3
void f() {
  idexpr("foo_", 3)(); // OK: calls foo_3
  idexpr("foo_", 2)(); // error: no such function foo_2
}
```

If any operands of `idexpr` are dependent, an *unqualified-id* cannot be formed; and the *id-expression* persists in its syntactic form. This is effectively what compilers do when any type or expression is dependent: preserve the syntax until instantiation.

Two `idexpr` ids are the same if they would generate the same *unqualified-id*. In the dependent case, this implies that the operands must be the same.

Note: If an implementation uses (a form of) name lookup during template instantiation to find the templated declaration corresponding to an instantiation, this feature will cause problems. Clang does this when instantiating initializers of member variables in a class template instantiation. One solution is to use the index of the member. Another more robust approach is to separately instantiate all dependent `idexpr` names to determine if a match can be found during lookup.

5.2.3 The typename operator

The `typename` operator can be used to form a *type-id* from a reflection. It has the syntax:

```
typename(reflection)
```

The reflection must reflect a type. This can appear wherever a type-id is allowed or as a *type-specifier* in a declaration. In the latter case, the same rules apply for using *typedef-name*s in as a type-specifier. Its use is straightforward:

```
template<meta::object T>
auto f() {
  typename(T) x;
  return x;
}
```

When instantiated, the value of T is used to form a type-id.

```
f<reflexpr(int)>(); // has type int
f<reflexpr(std::string)>(); // has type std::string
f<reflexpr>(f)>(); // error: reflects a template, not a type
```

5.2.4 The valueof operator

The `valueof` operator yields the constant value of a reflected variable with static storage, function, pointer to member, enumerator, or non-type template parameter. It has the form:

```
valueof(reflection)
```

The reflection must reflect one of the entities above. The type and value of the expression depend on the entity:

- For a reflected variable or static data member, the meaning of the expression is the same as if referring to the reference to the object declared by the variable.
- For a reflected function or static member function, the expression is the same as if taking the address of that function.
- For a reflected enumerator, the expression is the same as if referring directly to that enumerator.
- For a reflected non-static class member, the expression is the same as taking the address of that class member.
- For a non-type template parameter, the expression is the same as if referring to that non-type template parameter.

For example:

```
// at global scope
int x = 42;
int f() { return x; }
enum E { A };
struct S { int x; }
valueof(reflexpr(x)) // has type int&, value of 42
valueof(reflexpr(f)) // has type int (*)(), value of &f
valueof(reflexpr(A)) // has type E, value of A (i.e., 0)
valueof(reflexpr(S::f)) // has type int S::*, value of &S::f
```

5.2.5 The namespace operator

The namespace operator can be used to form a nested-name-specifier referring to a reflection of a namespace or class.[2] It has the syntax:

```
namespace(reflection)
```

The reflection must be that of a namespace, class, or scoped enum. Note that you could presumably also use the `typename` operator for the same effect.

5.2.6 A general purpose projection operator

It may be possible to collapse all of these different operators, except `idexpr`, into a single projection operator. This has been previously called postfix-$ in Herb's metaclass proposal PXXX. That particular spelling does not work, because it is effectively unparseable in the contexts where projections may appear.

However, an alternative spelling, say `deflexr(e)`, may work, if the entity reflected by the expression e has an unambiguous interpretation in the context where the operator appears.

One benefit of using named operators is that it allows overloading. For example:

```
static int x;
typename(meta::type(reflexpr(x)) y; // OK: y has type int
typename(reflexpr(x)) z; // OK: z has the same type as x (int)
```

This isn't possible using a single projection operator, especially in the last case:

```
deflexpr(x) z; // error: ambiguous parse
```

[2] This has not been implemented.

Specifically, it is not clear whether the `deflexpr(x)` phrase is intended to be used as an expression (i.e., as if using `valueof`), or a *type-specifier* in a declaration.

My preference is for specifically named operators.

5.2.7 Immediate functions

In order to provide library support, we provide a new specifier (and semantics) for function declarations.

The immediate specifier is a new *function-specifier*. When present, the declared function implicitly declared `constexpr`. A declaration of an immediate function must be a definition.

NOTE: This section is intentionally incomplete See Section 7.2 for discussion of design issues around immediate functions.

6 Library support

There is a large support library required for static reflection. This library must define:

- A type to represent the value of a reflected entity or other construct (possibly with language support (e.g., `std::nullptr_t`).
- A set of algorithms and data types need to query reflected entities.

This paper uses the namespace `cppx::meta` to enclose these definitions.

6.1 The meta::object type

The `meta::object type` is the result type of *reflection-expression*s and other library operations that return metaobjects. It is declared as:

```
using object = /* implementation-defined */;
```

This type must:

1. be a regular (default constructible, copyable), literal type,
2. be usable as the type of a non-type template parameter,
3. contextually convertible to bool
4. be able to encode references to internal compiler data structures,
5. only not allow non-default values to be created, except by the implementation.

The first two requirements are readily satisfied by any integral type.

The 3rd imposes a minimum size for the type. This is likely no smaller than the size of a pointer in the compiler's host architecture. Thus far, we have been able to encode all reflectable constructs as either 64-bit AST node pointers, or keys into internal tables, using the low-order bits in the address to differentiate between different kinds of data structures.

Note that every `meta::object` is either a reflection (a handle to an internal data structure) or the *null reflection* value. The latter is used throughout the library to indicate ends of lists and absent optional values.

The 4th requirement prohibits programmers from misinterpreting arbitrary integer values as reflection values. The only valid user construction is the null reflection. Unfortunately, this property is not enforceable for integral types; we would need to introduce a special type to accommodate these semantics.

Further, it is desirable if this type is only usable in `constexpr` functions, since the passing of its values at runtime will produce undefined behaviors. We discuss this idea in more detail in Section XXX.

My implementation defines `meta::object` like this:

```
enum class object : std::uintptr_t { };
```

This generally supports the 4th requirement, except that a user can create of this type using a `static_cast`. Note that this assumes that the size of `std::uintptr_t` are the same for both the host and target architecture. Language support for this type would be needed to guarantee size requirements.

6.2 Library functions

Functions in the meta library query semantic properties of the reflected entity or construct. Here is a brief synopsis of the main functions in the library:

```
enum object_kind {
  null_reflection,
  // ... one label for each kind of entity or construct ...
  namespace_reflection,
  class_reflection, // etc.
};

// Classifiers
immediate object_kind kind(object);
immediate bool is_null(object);
immediate bool is_namespace(object);
immediate bool is_variable(object);
immediate bool is_function(object);
immediate bool is_parameter(object);
immediate bool is_class(object);
immediate bool is_union(object);
immediate bool is_data_member(object);
immediate bool is_bitfield(object);
immediate bool is_member_function(object);
immediate bool is_constructor(object);
immediate bool is_destrutor(object);
immediate bool is_conversion(object);
immediate bool is_enum(object);
immediate bool is_enumerator(object);
immediate bool is_template(object);
immediate bool is_type(object);

// Context traversal
immediate object context(object);
immediate bool has_members(object);
immediate object first_member(object);
immediate object next_member(object);
immediate bool has_type(object);
immediate object type(object);
immediate object templated_declaration(object);
```

```
  // Members
  class member_iterator { ... }
  class member_range;
  immediate member_range members(object);

  // Semantic properties
  immediate bool is_named(object);
  Immediate bool const char* name(object);

  immediate bool is_extern(object);
  immediate bool is_static(object);
  immediate bool is_thread_local(object);
  immediate bool is_mutable(object);
  immediate bool is_virtual(object);
  immediate bool is_pure_virtual(object);
  immediate bool is_explicit(object);
  immediate bool is_inline(object);
  immediate bool is_friend(object);
  immediate bool is_final(object);
  immediate bool is_override(object);

  immediate bool is_complete(object);
  immediate bool is_defined(object);
  immediate bool is_defaulted(object);
  immediate bool is_deleted(object);

  immediate bool is_public(object);
  immediate bool is_private(object);
  immediate bool is_protected(object);
```

Omissions are to be expected; this is just an initial set of operations that have been somewhat obvious or necessary for examples thus far. For example, there is, as of yet, no method of traversing base class specifiers or elements of an overload set.[3]

The semantic properties of entities are queried based on the set of declaration-specifiers and other qualifiers appearing in the various declarations. This is intentional. We (as programmers) more commonly think about whether or not a function is static than we do that it has internal or external linkage. Or similarly that a variable is a static, not that it has static storage. In other words, the specifier is a reasonable surrogate for the semantic property.

The `context`, `first_member` and `next_member` functions give an interface for walking up and down a tree. These functions will return the null reflection when there is no enclosing context (i.e., the global namespace), first member, or next member.

[3] Although this did appear in a previous reflection proposal.

The reflections returned by `first_member` and `next_member` are those corresponding to the first declaration introducing an entity and not subsequent re-declarations or synonyms.[4]

```
namespace N {
  void f();
  struct S { };
  void f() { }
  using M::g;
  using namespace N2;
}

constexpr void walk() {
  meta::object obj = meta::first_member(reflexpr(N));
  while (obj) {
    // visit(obj)
    obj = meta::next_member(obj);
  }
}
```

The elements visited, in order are:

1. `void f()`, where with the definition being visible.
2. `struct S { }`.

Neither the *using-declaration* nor the *using-directive* are included in the traversal. Those should could be accessed by other functions such as `using_names` and `using_namespaces`.

The `member_iterator` and `member_range` classes build on the `first_member` and `next_member` functions in an obvious way. These facilities define a forward range over the members of a namespace or class, parameters of a function, enumerators, etc.

6.3 Implementation

The library facilities are implemented with the help of intrinsics. For example `meta::kind` is implemented like this:

```
immediate object_kind kind(object x) {
  return __reflect_index(x):
}
```

The `__reflect_index` intrinsic is somewhat like an intrinsic type trait except that its value is computed during `constexpr` evaluation and not translation (i.e., it is not a projection operator). The library currently requires 5 or 6 intrinsics, each returning various properties of a reflected entity.

7 Design issues

The following sections describe open questions and issues related to this design.

[4] This is a change in semantics from the current implementation. I currently yield the set of declarations, but I have found this to be somewhat troublesome in practice. I should never, for example, be forced to think about the injected class name. It also raises questions. Should access-specifier-declarations be reflected? Static assertions? Is a friend really a declaration?

7.1 Reflecting entities vs. declarations

This proposal specifically emphasizes the reflection of properties related to *entities* (and some associated constructs) and not their syntactic elements: declarations. There are a few reasons for this. First, to do otherwise, would mean that the reflection operator (described in more detail below), when given a name, would be required to produce a list of declarations that introduce the name. For example:

```
static void f();
void f() { }
constexpr meta::object fn = reflexpr(f);
```

How does a user determine if `f` is `static`? If the reflection operator reflects entities, then the answer is straightforward: the first declaration determined the linkage of the name.

If the reflection operator yields a list of declarations, then one of two things must happen. Either:

- Users would be required to traverse the list of declarations to determine the cumulative properties of the entity or entities found;
- The reflection support library would be required to implement the function above, or it could expose an interface allowing direct query of entity corresponding to the declset.

I believe that forcing users to do a) would ultimately be a disaster. This example is fairly simple and the solution obvious. However, taken in the extent, we would effectively require compile-time implementations of many of the things that the already does for us, like determine the meaning of a name. Consider:

```
using T = int;
using U = const T;
U x = 0;
constexpr meta::object var = reflexpr(x);
constexpr meta::object type = meta::type(var); // #1
```

What should the variable `type` reflect at #1? I would be very surprised if the answer is not `const int`.

If the answer is that it reflects something else, say, a `typedef` type, then the user will have to work harder to find the semantic properties of that type. This entails either walking through a list of "desugarings" or finding a function that does the right thing. Moreover, it is not clear if that interpretation can be supported by all of today's conforming implementation.

Users are not, and should not be expected to be, compilers.[5] In either case, there is additional compile-time overhead needed to compute the result. I strongly suggest that reflection reflect the semantic properties of declarations rather than their syntax.

It should be possible to extend the semantics of the proposal to include the ability to find declarations or specific uses of entities (e.g., give me all declarations of `f` or the aliased structure of U).

Note also that static reflection is limited to entities introduced into a single translation unit. Reflections of entities cannot be passed (as values) between separately compiled translation units. Note, however, that because of the ODR, all properties of the same entity, queried in multiple translation units must be the same.

[5] Although the converse seems to be a popular opinion.

7.2 The runtime divide

For many uses of reflection, we run into problems ensuring that reflective (but not necessarily generative) algorithms are evaluated at compile-time. For example, consider this use of the `constexpr count` function from the introductory sections.

```
std::cout << count(reflexpr(std)) << '\n';
```

This usage of `count` is interesting. It seems like it should be okay, but It is almost certainly incorrect. We discuss the reasons for this from the perspective of our implementation experience.

First, we note that `count(reflexpr(std))` may not be evaluated at compile time. There is nothing that prevents an implementation from evaluating a call to `constexpr` function if all the operands are constant. In this is the case, then the program would be well-formed and correct.

If, however, the function is not evaluated during translation, this may result in a compiler error, although not a C++ diagnostic. The call to count constitutes an odr-use of the function, which will cause the compiler to emit the definition, along with definitions of the various `meta::` functions used within the body of `count`.

Almost all `meta::` functions are implemented using compiler intrinsics (more below). These intrinsics traffic between `meta::object` values and program entities. In other words, they *must* be evaluated at compile-time. During lowering, those values are absent, and so the compiler cannot generate operations corresponding to those intrinsics.

This is not a solved problem in our implementation. The next three sections describe possible solutions to this problem.

7.2.1 Immediate application

We partially solve the problem by introducing a new form of `constexpr` specifier that requires the function to be evaluated at the call site. Using this, we can write `count` like so:

```
immediate int count(meta::object x) { ... }
```

When the analysis of a call expression to an immediate is finished, then that expression is evaluated as a constant expression. If evaluation fails, the program is ill-formed.

That satisfies this usage scenario:

```
std::cout << count(meta::object x) << '\n';
```

It is also worth noting that the examples given of `std::file()` and `std::line()` in Bjarne's paper, PXXX, also fall into this category.

```
Logger("log entry from file {1} line {2}", std::file(), std::line());
```

The use of `std::file()` and `std::line()` would need to be evaluated at the call site in order to produce constants to be substituted into the log entry later. Their implementation must be by compiler intrinsic, which returns a constant representing the current *point of translation* (the point in a program where translation is required to evaluate a constant expression).

This feature raises other questions:

Can one immediate function call another immediate function? Yes, but you cannot immediately evaluate the callee at the call site if the operands refer to parameters of calling immediate function. For example:

```
immediate int f(int n) { return f(n - 1); }
```

At the point you analyze the recursive call, the expression n - 1 cannot be evaluated. In other words, there is some criteria where by which immediate evaluation is actually deferred.

Can a constexpr function call an immediate function? This is a variation on the theme above. The answer is yes, but likely only when the call can be evaluated at the call site:

```
immediate int f(int n) { n + 1; }
constexpr int g(int n) {
constexpr int x = f(42); // OK: 42 is a constant
constexpr int y = f(n); // error: cannot evaluate immediately
```

The reason that the call to `f(n)` would be an error is that g could be odr-used in a non-`constexpr` context, requiring its definition to be lowered to IR. The value of n is not a constant in that context.

Should `constexpr` evaluation contexts really be self-selecting? In other words, is it sane to let a function determine when it is evaluated rather than leaving that to the developer? The answer is probably yes, as long as the function has no side effects. Today, constant expressions do not have side effects. But that may not always be the case.

For example, metaprogramming and `constexpr` debugging both introduce novel kinds of side effects for constant expressions. In our metaprogramming implementation, we cache requests to generate source code during compile-time evaluation. These requests are interpreted by translation process to modify the program after the completion of a metapgram. Similarly, our (admittedly very simple) constexpr debugging scheme requests the emission of log messages, which are subsequently processed by the translation into compiler diagnostics. As a side note, these effects are never visible during evaluation.

An immediate function that has side effects can generate those effects at unexpected times. For example:

```
immediate void print(int n) {
  meta::compiler.print(n);
}

immediate int f(int n) {
  for (int i = 0; i < n; ++i)
    print(42);
}
```

The translation of this program fragment will print the number 42 exactly once.

Can you overload immediate and constexpr functions? We don't know, but it's probably not a good idea.

Is immediate part of the type system? We haven't formulated rules that explain what happens when you try to take the address of an immediate function. This could allow immediate functions and their evaluation to leak across translation unit boundaries. This would not be good.

7.2.2 A constexpr operator

As an alternative (or complement?) to we could consider adding a new constexpr operator that evaluates any expression where it appears.

```
std::cout << constexpr(count(reflexpr(std))) << '\n';
```

The `constexpr` operator would evaluates the `count` function at this point in the translation, yielding the value. This is exactly what an immediate function would do implicitly.

This operator may be useful in other contexts outside of reflection. Other committee members have expressed interest in something similar.

7.2.3 Metatypes

Neither immediate functions nor a `constexpr` operator solve the problem originally alluded to: it is possible to lower function calls involving `meta::objects`. We would prefer for this to not happen.

We think we could use the type system to achieve this effect. Specifically, we want something like the following rule:

> *A potentially evaluated expression that contains a use of a function whose type is a metatype, other than within a constant expression, is ill-formed.*

A metatype is a type whose objects can only be creating during the evaluation of a constant expression. We can define it inductively:

- The type `meta::object` is a metatpye.
- A pointer to an object of metatype is a metatype.
- A reference to an object of metatype is a metatype.
- An array of objects of metatype is a metatype.
- A function whose return type is a metatype or whose parameter-type-list contains a metatype is a metatype.
- A class is a metatype if it is a literal type, and
  - Any of its base class specifiers denote metatypes
  - Any of its non-static data members have metatypes
- And probably more cases…

Note that this is orthogonal to the features needed to add new `constexpr` evaluation contexts.

7.3 The meta::object type

There are some implementation restrictions on this value. Primarily, it must be usable as a template argument. Second, do the extent possible, it must be impossible for users to create non-default initialized objects of this type. Only the implementation is permitted to do so.

The current implementation makes this a scoped `enum` whose underlying type is `std::uintptr_t`.

I believe it would be better to add a new fundamental type that captures the desired semantics. In particular, we *really* want to enforce the requirement that users may not construct objects of this type except:

a) by default construction,
b) as the result of a copy,
c) as a result of `reflexpr`, or

d) as a result of a `meta::` library function.

Doing otherwise would almost certainly produce and bit pattern that does not correspond to internal compiler object. That will lead to internal compiler errors and very possibly serious security problems.

Note that programmers must also be prevented from using `static_cast` or `reinterpret_cast` to refer to objects of `meta::object` type.

8 A compile-time evaluation model for static reflection

In our note on `constexpr` evaluation, we imagine a hypothetical compiler that implements `constexpr` evaluation by emitting a subset of the program being translated and producing an executable program. The implementation (translation) is free to instrument that program to provide additional information that can be used to e.g., generate good diagnostics.

To implement static reflection in this model, we can augment the generated program-to-be-executed by serializing the abstract semantics graph as global data. All `meta::object` values are simply indexes into the nodes of the ASG. All applications of intrinsics evaluate the reachable semantic properties of those entities. This effectively restates static-reflection as dynamic reflection.[6]

Note that projection operators are applied during translation, although they depend on the evaluation of their operands. These operands are never "leaked" into the evaluation.

[6] Defining this as a library facility would actually give us dynamic reflection capabilities. This need not be hypothetical.