Document: P1240R1
Revises: P1240R0
Date: 10-03-2019
Audience: SG7
Authors: Wyatt Childers (wchilders@lock3software.com)
Andrew Sutton (asutton@uakron.edu)
Faisal Vali (faisalv@yahoo.com)
Daveed Vandevoorde (daveed@edg.com)

Scalable Reflection in C++

Revision history

R0	Initial revision introducing scalar reflection model, reifiers, extensive API, and many examples.
R1	Added reflexpr(... xyz). Revised reifier syntax somewhat. Report on implementations. Introduce `<meta>` header. Reorganized presentation slightly.

Introduction

The first Reflection TS (based on N4766) exposes reflection information as types (to simplify integration with template metaprogramming techniques). However, SG7 agreed some time ago that the future of reflective constructs in C++ should be value-based (see also P0425r0). Specifically, the compile-time computations required for reflective metaprogramming should make use of constexpr evaluation, which, unlike template metaprogramming, allows for ephemeral intermediate results (i.e., they don't persist throughout the compilation process) and for mutable values. This approach was most recently described in P0993r0, *Value-based Reflection*. To support that reflection design, we have passed a number of "constexpr" extensions in C++20: `consteval` functions (P1073), `std::is_constant_evaluated()` (P0595), and constexpr dynamic allocation (P0784). We have also proposed *expansion statements* (P1306), which are more broadly useful but especially convenient for reflective metaprogramming: That feature was approved by the evolution working group for C++20, but did not make it to a WG21 vote for lack of time completing the Core wording review. We hope (and count on) expansion statements getting added to the working draft for C++23 in the relatively near future.

That in itself still leaves plenty of design options for the reflection interface itself. What follows is an extensive document describing:

- The representation and properties of "reflections" (with argumentation for our specific design and considerations of alternatives).
- Mechanisms for *reification*: Turning reflections into ordinary C++ source constructs (again, with design discussions).
- A brief discussion about templates and their instances.
- Principles to translate existing standard template metaprogramming facilities to the reflection domain.
- Principles to translate the Reflection TS facilities to the value-based reflection domain.
- Some examples to argue that proposals to add additional template metaprogramming facilities are unneeded because the underlying functionality is better handled in the reflection domain.
- An appendix listing the meta-functions being worked on one ongoing implementation.

A simple example

The following function uses static reflection facilities presented in this paper to compute the string representation of an enumerator value.

```
#include <meta>

template<Enum T>
std::string to_string(T value) {  // Could also be marked constexpr
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {
    if (exprid(e) == value) {
      return std::meta::name_of(e);
    }
  }
  return "<unnamed>";
}
```

In broad strokes, the function does the following:

1. Gets the sequence enumerators from the enumeration type `T`,
2. Iterates over those enumerators, searching for the first that matches `value`,
3. Returns the name of that iterator.

Each of these operations relies on a feature included in this proposal. In particular, getting the sequence of iterators requires that we first get a queryable representation of the enumeration type `T`. This is done using the `reflexpr` operator; it returns a *reflection*: a handle to an internal representation of type `T` maintained by the compiler. The `members_of` function (declared in a newly proposed standard header `<meta>`) returns a compile-time `std::vector` containing reflections of each enumerator in the enum.

To iterate over the vector we use an *expansion-statement*, spelled `for constexpr`. This isn't true "iteration", however. The body of the statement is repeated for each element of the vector so that the loop variable (`e` above) is initialized to `*(vec.begin() + 0)`, `*(vec.begin() + 1)`, …,

`*(vec.begin() + n - 1)` in each successive repetition. The loop variable is also implicitly declared `constexpr` within each repeated body. In other words, each repetition is equivalent to:

```
{
  constexpr std::meta::info e = *(vec.begin() + I);
  if (exprid(e) == value)
    return std::meta::name_of(e);
}
```

where `I` represents the i^{th} repetition of the loop's body.

Within the expansion body, the `exprid` construct recovers the value of a reflected entity. We call this recovery process *reification* and the operators — like `exprid` — that enable it *reifiers*. This can be compared with the parameter `value` to determine if they are the same. Finally, the `name_of` function returns a compile-time string containing the identifier of the matched enumerator. If none of the enumerators matched (possible, e.g., when bit-ORing together enumerator values), we return a string `"<unnamed>"` (which won't collide with a valid identifier).

This is called *static reflection* because all of the operations used to query types and enumerators are computed at compile time (i.e., statically). There is no additional runtime meta-information that must be generated with such facilities, which reinforces the zero-overhead principle that is so fundamental to C++. There is no runtime representation of the enumeration type and its enumerators. Only information that is ODR-used is present in the final program.

Implementation status

Two implementations of this proposal are under way.

The first and most mature/complete is a fork of Clang by Lock3 Software (by, among others, Andrew and Wyatt, authors of this paper). It includes a large portion of the capabilities presented here, albeit not always with the exact syntax or interfaces proposed.

The second is based on the EDG front end (by Faisal and Daveed) and is in very early stages, which just a few cases of the `reflexpr` operator, a few reifiers, and a few meta-library interfaces implemented.

Reflections

The `reflexpr` operator

The first Reflection TS introduced the `reflexpr` operator to obtain reflection values encoded as types. Ironically, the spelling is more appropriate for the value-based reflection since the corresponding operation is indeed an "expression" (i.e., a construct that produces a value; in the TS it produces a type). In any case, all discussions so far have agreed that reusing that token spelling (which took quite some

bikeshedding effort during the first design round) is desirable. In other words, value-based reflection will allow us to write:

```
constexpr std::meta::info reflection = reflexpr(name_or_expr);
```

The value of `reflection` (i.e. the result of a call to reflexpr) is a compile-time value that *designates* some view of the program by the implementation (specifically, the compiler front end). I.e., it can be thought of as a handle to an internal structure of the compiler. In the rest of this proposal we refer to the result of `reflexpr` as a *reflection* or a *reflection value*.

Note that `reflexpr` is the "gateway" into the reflected world, but it is not the only source of *reflections* (or *reflection values*): We will further introduce a variety of functions that derive reflections from other reflections (e.g., we'll present a function that returns reflections for the members of a class given a reflection for that class). Whatever the source of a reflection, we say that it *designates* language concepts such as entities or value categories. As will be shown later, a reflection can *designate* multiple notions. For example, `reflexpr(f(x))` designates the called function `f` (if indeed that is what is called) and the type and value category of the call result.

The operand of `reflexpr` must be one of the following:

- a *type-id*, including possibly a *simple-type-specifier* that designates a *template-name*
- a possibly qualified *namespace-name*
- the scope-qualifier token "`::`" (designating the global namespace)
- an *expression*[1]

In the case where the `name_or_expr` is an expression, it is unevaluated but *potentially constant evaluated*. That implies that given "`struct S { int x; };`", the expression "`reflexpr(S::x)`" is permissible in this context. We will elaborate the available reflected semantics later in this paper. Note also that since `reflexpr(name_or_expr)` is an expression, `reflexpr(reflexpr(name_or_expr))` is valid (generally producing a distinct reflection).

In this paper we call *declared entity* any of the following: a namespace (but not a namespace alias), a function or member function (that includes implicit special members, but not inherited constructors), a function or template parameter, a variable, a type (but not a type alias), a data member, a base class, a capture, or a template (including an alias template, but not a deduction guide template). Note that this is slightly different from the standard term *entity* (which, e.g., includes "values" but not "captures"[2]). We call *alias* a namespace alias or a type alias.

[1] Including, potentially, a comma expression.

[2] This paper does not currently deal with structured bindings because their exact nature in the standard is still somewhat in flux at the time of this writing. Once they're clarified, we intend to revisit their status as a "declared entity".

Reflection type

What should the type of a reflection be? We propose it to be a new scalar type[3], distinct from all other scalar types, that supports — aside from reading, assigning, and copying — only the scalar operations of equality/in-equality and contextual conversion to `bool`. In addition we propose specific *reifiers* (that transform a reflection value into a type or a name, see below) and library functions that can operate on constexpr reflections and constexpr sequences of reflections and generate new reflection-values as needed. All other operations on reflection values are then composed from these aforementioned operations. We present our rationale below for this design choice.

It is tempting to organize reflection values as class type values using a hierarchy of class types that try to model the language constructs. For example, one could imagine a base class `Reflection`, from which we might derive a class `ReflectedDeclaration`, itself the base class of `ReflectedFunction` and `ReflectedVariable`.

We do not believe this is the best approach, however, for at least the following reasons:

- Although the relationship between major language concepts is relatively stable, we do occasionally make fundamental changes to our vocabulary (e.g., during the C++11 cycle we changed the definition of "variable"). Such a vocabulary change is more disruptive to a class hierarchy design than it is to certain other kinds of interfaces (we are thinking of function-based interfaces here).
- Class hierarchy values aren't friendly to value-based programming because of slicing; instead, it works better with "reference" programming, which is particularly expensive for constexpr evaluation.
- Class types are not easily used as nontype template arguments, particularly when we want to restrict effects to compile time (the recently added support for nontype template arguments (P0732R2) causes run-time variables to be synthesized, but doing so would be meaningless for compile-time reflection values). As it turns out, instantiating templates over reflection values is an important idiom when it comes to reification.
- Implementations of constexpr evaluation usually handle non-pointer scalar values significantly more efficiently than class values.

Regarding this last point, the following compile-time test:

```
constexpr int f() {
  int i = 0;
  for (int k = 0; k<10000; ++k) {
    i += k;
  }
  return i/10000;
}
```

[3] We could define it with `using info = decltype(reflexpr(void))`.

```
template<int N> struct S {
  static constexpr int sm = S<N-1>::sm+f();
};
template<> struct S<0> {
  static constexpr int sm = 0;
};
constexpr int r = S<200>::sm;
```

compiles in about 0.6 seconds on a compact laptop (2016 MacBook m7), but wrapping the integers as follows:

```
struct Int { int v; };
constexpr int f() {
  Int i = {0};
  for (Int k = {0}; k.v<10000; ++k.v) {
    i.v += k.v;
  }
  return i.v/10000;
}
template<int N> struct S {
  static constexpr int sm = S<N-1>::sm+f();
};
template<> struct S<0> {
  static constexpr int sm = 0;
};
constexpr int r = S<200>::sm;
```

doubles the compile time to 1.2 seconds. Adding a derived-class layer would further increase the time. Another increase would result from attempting to access the classes through references (as would be tempting with a class hierarchy) because address computations require some work to guard against undefined behavior.

Because of these various considerations, we therefore propose that the type of a reflection is an unspecified scalar type, distinct from all other scalar types, whose definition is:

```
namespace std::meta {
  using info = decltype(reflexpr(void));
}
```

Namespace `std::meta` is an associated namespace of `std::meta::info` for the purposes of argument-dependent lookup (ADL): That makes the use of various other facilities in that namespace considerably more convenient. (In this sense, `std::meta::info` is similar to an enumeration type.)

By requiring the type to be scalar, we avoid implementation overheads associated with the compile-time evaluation of class objects, indirection, and inheritance. By making the type unspecified but distinct, we avoid accidental conversions to other scalar types, and we gain the ability to define core language rules that deal specifically with these values. Moreover, no special header is required before using the `reflexpr` operator.

Reflection categories

As noted earlier, reflection values behave as handles to internal structures of the compiler. To reason about the kind of semantic information one can obtain through these reflection values, we categorize the values into four groups:

- Declared-entity reflections
- Alias reflections
- Expression reflections
- Invalid reflections

Note, declared-entity-reflections *only* designate the declared-entity; alias-reflections always designate a declared-entity in addition to providing the name of the alias; and, expression-reflections might or might not designate a declared-entity (e.g., an *id-expression* might designate a variable), but always designate properties of the expression. *Invalid reflections* will be discussed in more detail later, but they represent various kinds of failures when creating reflection using means other than the `reflexpr` operator.

For the most part, reflections of *names* designate the declared entity those names denote: variables, functions, types, namespaces, templates, etc. For example:

```
reflexpr(const int) // designates the type const int
reflexpr(std) // designates the namespace std
reflexpr(std::pair) // designates the template pair
int* f(int);
reflexpr(decltype(f(3))) //  designates the type int*
reflexpr(std::pair<int, int>) // designates the specialization
```

Reflections of *expressions* designate a limited set of characteristics of those expressions, including at least their type and value category. For example:

```
reflexpr(1) // designates at least the property “prvalue of type int”.
```

(Further on we will present functions to examine and/or reify the designated notions.)

If an expression also names a declared entity (via a possibly-parenthesized *expression-id*), then it also designates that entity. For example:

```
int x;
```

```
reflexpr((x)) // designates the declared-entity 'x' (variable) as well as its value category
reflexpr(x+1) // does not designate a declared-entity but does at least designate the property
              // "prvalue of type int".

reflexpr(std::cout) // designates the object named by std::cout as well as the
                    // type and value category (lvalue) of the expression.
```

If an expression is a *constant expression* it also designates that constant value:

```
reflexpr(0) // designates the value zero and the property "prvalue of type int"
reflexpr(nullptr) // designates the null pointer value and the property "prvalue
                  // of type decltype(nullptr)
reflexpr(std::errc::bad_message) // designates the enumerator, its constant value,
                                 // and the property "prvalue of type
                                 // std::errc"
```

If an expression represents a call at its top level, it also designates the function being called:

```
reflexpr(printf("Hello, ")) // designates printf and the property "prvalue
                            // of type int"
reflexpr(std::cout << "World!") // designates the applicable operator<<
                                // and "lvalue of type std::ostream"
constexpr int f(int p) { return p; };
reflexpr(f(42)) // designates f, the value 42, and "prvalue of type int"
reflexpr(f(42)+1) // designates the value 43 and "prvalue of type int";
                  // does not designate f because the call is not "top level"
```

Now consider:

```
constexpr int const i = 42;
constexpr auto r = reflexpr(i);
```

As mentioned before, reflections can be categorized into four groups: declared-entity, alias, expression, or invalid. In this example, the reflection value `r` is an "expression reflection" and thus designates both the *expression* `i` (i.e. you can obtain information about properties of the expression such as its lvalueness) and the *variable* `i`. However, sometimes it is useful to obtain a reflection that designates only the entity (and not the expression). For example, we might want to query the type of the *variable* `i` (`int const`) instead of the *expression* `i` (`int`). It also can be useful when comparing if two reflections refer to the same entity, as we will show later. We therefore provide the special function

```
namespace std::meta {
  consteval auto entity(info reflection)->info {...};
}
```

applied to `r` produces a reflection designating just the *variable* (i.e., a "declared-entity reflection").

More generally, `std::meta::entity` extracts the declared-entity from its argument by returning:

- its argument – if its argument is a declared-entity reflection or an invalid reflection,
- a declared-entity reflection designating an entity E – if the argument is an alias or expression reflection that also designates E, or
- an invalid reflection in all other cases (e.g., `entity(reflexpr(42))` is an invalid reflection).

When the `reflexpr` operand is the name of an *alias* (type or namespace) the reflection designates the aliased entity indirectly (i.e., properties of the alias can be queried directly). For example:

```
using T0 = int;
using T1 = const T0;
constexpr meta::info ref = reflexpr(T1);
```

Here, `ref` designates both `T1` (directly) and the type `const int` (indirectly). This allows users to work both with the alias and its meaning.

In a more abstract sense, reflections designate semantic notions (names, types, value categories, etc.) rather than syntax (tokens that comprise an expression and the relation of those tokens to others). This principle helps guide decisions about the design of language and library support for reflection.

The queryable properties of these reflections are determined by the kind of "thing" they reflect. Details are provided below.

Reflecting ranges and packs

It turns out to be useful to be able to lift a *range* of constant values into a range of reflections for expressions denoting constant values. We therefore propose a variation of the `reflexpr` operator that does exactly that. E.g.:

```
constexpr std::vector<int>  v{ 1, 2, 3 };
constexpr std::vector<std::meta::info>
                            rv = reflexpr(... v);
```

Such a lifted vector can then be reified into a template argument context:

```
std::integer_sequence<int, valueof(... rv)> is123;
  // same as std::integer_sequence<int, 1, 2, 3>
```

(Reification, including the `valueof(... range)` refier, is described later on.)

This feature uses the same mechanism as *expansion statements* (see P1306). Consequently, this version of the `reflexpr` operator works not only with range values, but also with values that can be deconstructed using structured bindings. For example:

```
constexpr struct S { int i = 2; float f = 1.0; } s;
constexpr std::vector<std::meta::info> = reflexpr(... s);
  // similar to "… = { reflexpr(s.i), reflexpr(s.f) };"
```

Equality and equivalence

Reflections can be compared using `==` and `!=` operators. If two reflections designate declared entities or aliases of such entities but do not designate expression properties of an expression that is not an *expression-id*, the reflections compare equal if the entities are identical (i.e., the comparison "looks through" aliases). Any reflection also (obviously) compares equal to itself and to copies of itself. Otherwise, if two reflection both do not designate declared entities or they designate expression properties of an expression that is not an *expression-id*, their equality is unspecified. Otherwise, they compare unequal. For example:

```
typedef int I1;
typedef int I2;
static_assert(reflexpr(I1) == reflexpr(I2));

float f = 3.0;
static_assert(reflexpr(f) == reflexpr((f));
static_assert(reflexpr(f) == reflexpr(::f);

void g(int);
constexpr auto r = reflexpr(g(1)), s = r;
static_assert(r == s);  // Obviously.
static_assert(reflexpr(g(1)) == reflexpr(g(1)); // May fail because g(1)
                                                // is an expression that is
                                                // not an expression-id.
```

In the last case, users can more precisely specify whether they intend to compare entities or computed values (if possible) using the refiers (e.g., `typename`, `exprid`) or library facilities like `std::meta::entity` described above. For example:

```
void f();    // #1
int f(int);  // #2
constexpr auto r = std::meta::entity(reflexpr(f(42)));
                                                // Reflection for function #2.
```

```
static_assert(r == reflexpr(f(42)); // May or may not fail
static_assert(r == entity(reflexpr(f(0))));  // Always succeeds
```

Also note that:

```
static_assert(reflexpr(I1) == reflexpr(int));
```

The same principle applies to namespace aliases:

```
namespace N {};
namespace N1 = N;
namespace N2 = N;
static_assert(reflexpr(N1) == reflexpr(N2));
static_assert(reflexpr(N1) == reflexpr(N));
```

For reflections obtained from operands that involve template parameters, the result depends on the template arguments used for substitution.

```
template<typename T, typename U> struct Fun {
  static_assert(reflexpr(T) == reflexpr(U));
};
Fun<int, int> whee1;  // Ok
Fun<int, char> whee2;  // error: static assertion failed
```

We already mentioned that it is unspecified whether reflections obtained from expressions that do not designate a declared entity compare equal. That also applies to expressions that just consist of a literal. For example:

```
static_assert(reflexpr(1) == reflexpr(1)); // May or may not fail.
```

Note that the properties associated with a declared entity may change over various contexts, but that does not change the reflection. For example:

```
struct S;
constexpr auto r1 = reflexpr(S);
struct S {};
constexpr auto r2 = reflexpr(S);
static_assert(r1 == r2);
```

However, queries against the reflection value (e.g., to obtain a list of class members) may change as a consequence of the changes in the underlying entity.

An additional comparison function is proposed:

```
namespace std::meta {
  consteval auto same_reflections(info, info)->bool { ... };
}
```

If either `x` or `y` designate an alias (type or namespace) `same_reflections(x, y)` returns `true` if `x` and `y` designate the same alias and `false` otherwise. Otherwise (i.e., if neither `x` nor `y` designate an alias), `same_reflections(x, y)` returns `x == y`. In other words, `same_reflections(x, y)` is like the equality operator except that it doesn't "look through" aliases. For example:

```
using std::meta::same_reflections;
static_assert(!same_reflections(reflexpr(N1), reflexpr(N2)));
static_assert(!same_reflections(reflexpr(reflexpr(N1)),
                                reflexpr(reflexpr(N1))));
static_assert(!same_reflections(reflexpr(reflexpr(N1)),
                                reflexpr(reflexpr(N2))));
```

To compare the values of reflected objects, references, functions, or types, the reflection can first be reified using one of the operators described below.

A Note About Linkage

Although in most respects we propose that `std::meta::info` is an ordinary scalar type, we also give it one "magical" property with respect to linkage.

Before explaining this property, consider again what a reflection value represents in practice: It is a handle to internal structures the compiler builds up for the current translation unit. So for code like:

```
struct S {};
consteval auto f() {
  return reflexpr(S);
}
```

the compiler will construct an internal representation for struct S and when it encounters "`reflexpr(S)`" it will update a two-way map between the internal representation of S and a small structure underlying the `std::meta::info` value returned by `reflexpr(S)`.

Now consider:

```
// Header t.hpp:
struct S {};
template<std::meta::info reflection> struct X {};

// File t1.cpp:
#include "t.hpp"
enum E {};
consteval auto d() {
  return reflexpr(E);
}
X<reflexpr(S)> g() {
  return X<reflexpr(S)>{};
}

// File t2.cpp:
#include "t.hpp"
extern X<reflexpr(S)> g();
int main() {
  g();
}
```

The files t1.cpp and t2.cpp are compiled separately. The contexts in which the "`reflexpr(S)`" construct is encountered are therefore different and it is not practical to ensure that the *underlying* values ("bits") of the `std::meta::info` results are identical. However, it is *very* desirable that the types `X<reflect(S)>` are the same types in both translation units and that the above example *not* produce an ODR violation.

We therefore specify "by fiat" that:

- `reinterpret_cast` to or from `std::meta::info` is ill-formed
- accessing the byte representation of `std::meta:info` lvalues produces unspecified (possibly inconsistent) values
- `std::meta::info` values `A1` and `A2` produce equivalent template arguments if `std::meta::same_reflections(A1, A2)` produces `true`.

Thus the following variation of the previous example is also valid:

```
// File t1.cpp:
```

```
enum E {};
consteval auto d() {
  return reflexpr(reflexpr(E));
}
X<reflexpr(reflexpr(S))> g() {
  return X<reflexpr(reflexpr(S))>{};
}

// File t2.cpp:
extern X<reflexpr(reflexpr(S))> g();
int main() {
  g();
}
```

(In practice, this means that reflection values are mangled symbolically, according to what the reflection value actually designates.)

Invalid reflections

In what follows we are going to propose a large collection of standard reflection operations, some of which generate new reflection values. Sometimes, the application of some of these operations will be meaningless. E.g., consider:

```
namespace std::meta {
  consteval auto add_const(info)->info {...};
}
```

which is meant to take a reflection of a type and add a type qualifier on top. However, what happens with something like:

```
constexpr auto r = add_const(reflexpr(std));
```

which suggests the meaningless operation of adding a `const` qualifier to namespace `std`? Our answer is that an implementation will not immediately trigger an error in that case, but instead create a reflection value that represents an error. Any attempt to reify such a reflection is ill-formed (but subject to SFINAE).

It is useful for user code to also be able to produce invalid reflections. To that end, we propose the following function:

```
namespace std::meta {
```

```
  consteval
    auto invalid_reflection(std::string_view message,
                            std::string_view src_file_name =
                                      current_source_file_name(),
                            unsigned line = current_source_line(),
                            unsigned column =
                                          current_source_column())
          ->info {...};
}
```

which constructs a reflection that triggers a diagnostic if reified outside a SFINAE context (ideally, with the given message and source position information).

Note that the functions

```
namespace std::meta {
  consteval auto current_source_file_name()->std::string {...};
  consteval auto current_source_line()->unsigned {...};
  consteval auto current_source_column()->unsigned {...};
}
```

produce source location for the first call in the chain of immediate function calls leading up to a call to these functions. (We do not rely on the previously-proposed `std::source_location` feature because is it does not use immediate functions for the needed functionality.)

Invalid reflections can also be used to generate compiler diagnostics during constexpr evaluation using the `diagnose_error` function. This can be a valuable debugging aid for authors of metaprogramming libraries, and when used effectively, should improve the usability of those libraries.

```
namespace std::meta {
  consteval void diagnose_error(info invalid_refl) {...};
}
```

This function causes the compiler to emit an error diagnostic (formally: it makes the program ill-formed if it is invoked outside a deduction/SFINAE context), hopefully with the message and location provided by the argument.

Finally, we propose a predicate:

```
namespace std::meta {
  consteval auto is_invalid(info)->bool {...};
}
```

that can be used to test for, and, e.g., filter out invalid reflective operations. We also provide a convenience overload of this function:

```
namespace std::meta {
  consteval auto is_invalid(std::vector<info>)->bool {...};
}
```

which returns `true` if any element of the given vector is an invalid reflection. This is particularly useful because some important reflection facilities return vectors of reflection values that callers are likely to want to check for invalid entries.

Initialization of reflections

Objects of reflection type are zero-initialized to the invalid reflection.

Conversions on reflections

A prvalue of reflection type can be contextually converted to a prvalue of type `bool`. An invalid reflection converts to `false`; all other reflections convert to `true`.

Hashing reflections

We propose that the `std::hash` template be specialized for `std::meta::info`. We also propose that the resulting hash value be consistent across translation units.

Reification

In the context of this paper, “reification” (from the Latin “res”, meaning “thing”) refers to the process of turning a “reflection value” back into a “program source thing”. We propose a few primitive *reifiers* to map reflection values back to source code constructs (the operand “*`reflection`*” below always stands for a constant expression of type `std::meta::info`):

- `typename(`*`reflection`*`)`
  A *simple-type-specifier* corresponding to the type designated by “*`reflection`*”. Ill-formed if “*`reflection`*” doesn't designate a type or type alias.
- `namespace(`*`reflection`*`)`
  A *namespace-name* corresponding to the namespace designated by “*`reflection`*”. Ill-formed if “`reflection`” doesn't designate a namespace.
- `template(`*`reflection`*`)`
  A *template-name* corresponding to the template designated by “*`reflection`*”. Ill-formed if “*`reflection`*” doesn't designate a template.
- `valueof(`*`reflection`*`)`
  If “*`reflection`*” designates a *constant expression*, this is an expression producing the same

value (including value category). Otherwise, ill-formed.

- `exprid(`*`reflection`*`)`
  If "*`reflection`*" designates a function, parameter or variable, data member, or an enumerator, this is equivalent to an *id-expression* referring to the designated entity (without lookup, access control, or overload resolution: the entity is already identified). Otherwise, this is ill-formed.
- `[: `*`reflection`*` :]`
  If "*`reflection`*" designates an alias, a named declared entity, this is an *identifier* referring to that alias or entity. Otherwise, ill-formed.
- `[< `*`reflection`*` >]`
  Valid only as a template argument. Same as "`typename(`*`reflection`*`)`" if that is well-formed. Otherwise, same as "`template(`*`reflection`*`)`" if that is well-formed. Otherwise, same as "`valueof(`*`reflection`*`)`" if that is well-formed. Otherwise, same as "`exprid(`*`reflection`*`)`".

(All the "ill-formed" cases above are subject to SFINAE if they result from substitution during deduction.)

Note that it is not practical to just have one reifier (say, "`unreflexpr`") because a C++ parser must be able to classify the construct (type name, template name, etc.) a priori. E.g., consider:

```
template<std::meta::info refl> void f() {
  unreflexpr(refl) *p;  // (1)
}
```

If `unreflexpr` could produce a type or an expression, a compiler couldn't decide whether (1) is the declaration of a pointer or a multiplication.

Technically, we could merge `valueof` and `idexpr` into a single reifier, and an earlier version of our proposal did so (using `unreflexpr`). However, a reflection like "`reflexpr(f(42))`" can designate both a constant value and a declared-entity (the called function): With that design `unreflexpr` reified the constant value and a special function (`std::meta::entity`, presented later) was used to strip the constant value and leave only the declared entity. In the end, however, we found this error-prone and too opaque: The ability to express the intent unambiguously is worth the extra keyword.

For the bracketed reifiers, the delimiters each consist of two tokens. In other words, "`[ :`" is not a single token (and thus "`[ : `*`reflection`*` : ]`" is a valid alternative formatting). Otherwise, ambiguities would ensue with, e.g., the global scope qualifier ("`::`"). Other options were considered. In particular, parentheses-based alternatives were first considered (e.g., "`(. `*`reflection`*` .)`" to produce identifiers). However, parentheses are useful to disambiguate expression constructs and we didn't want to obstruct that design space in the future. For example, suppose we introduced "`(:`" and "`:)`" as delimiters, then that would harm the future possibility of a prefix unary operator that starts with a colon, since parenthesizing an expression that starts with that character would create the "`(:`" delimiter instead.

Here are some examples illustrating various uses of these reifiers:

```
typename(reflexpr(int)) i = valueof(reflexpr(42));
     // Same as "int i = 42;".

constexpr int J = 42;
idexpr(reflexpr(i)) = valueof(reflexpr(J));
    // Same as "i = 42;".
idexpr(reflexpr(i)) = idexpr(reflexpr(J));
    // Same as "i = J;" (which is indistinguishable from the former line).


idexpr(reflexpr(i)) = idexpr(reflexpr(i));
    // Same as "i = i;".

idexpr(reflexpr(i)) = valueof(reflexpr([]{ return J; }()));
    // Same as "i = 42;".

idexpr(reflexpr(i)) = valueof(reflexpr([]{ int i; return J; }()));
    // Error: The lambda call is not a constant expression.

idexpr(reflexpr(i)) = idexpr(reflexpr([]{ int i; return J; }()));
    // Error: The idexpr on the right reifies as the member function call operator,
    // which cannot be assigned to an int variable!

namespace N { int f; }

void [: reflexpr(N::f) :](int);
     // Same as "void f(int);".

struct S {
  consteval auto ri() { return reflexpr(S::i); };
private:
  int i:3; // Bit field.
} s;
int i1 = s.idexpr(s.ri()); // Okay: Refers to S::i without needing name
                           //       lookup at this point.
int i2 = s.[:s.ri():]; // Error: Same as "s.i", which is an access violation.
```

Furthermore, we propose sequence-generating variations of most of the reifiers above. Let *`reflection_range`* be a *constant range* such that

```
for (std::meta::info r : reflection_range) ...
```

would successively set `r` to a sequence of values `r1`, `r2`, `r3`, ... `rN`.
Then:

- `typename(... reflection_range)` expands to
  `typename(r1), …, typename(rN)`
- `template(... reflection_range)` expands to
  `template(r1), …, template(rN)`
- `valueof(... reflection_range)` expands to
  `valueof(r1), …, valueof(rN)`
- `idexpr(... reflection_range)` expands to
  `idexpr(r1), …, idexpr(rN)`
- `[: ... reflection_range :]` expands to
  `[: r1 :], …, [: rN :]`
- `[< ... reflection_range >]` expands to
  `[< r1 >], …, [< rN >]`

Examples:

```
std::meta::info  t_args[] = { reflexpr(int), reflexpr(42) };
template<typename T, T> struct X {};
X<[<... t_args>]> x;  // Same as "X<int, 42> x;".
template<typename, typename> struct Y {};
Y<[<... t_args>]> y;  // Error: same as "Y<int, 42> y;".
```

Some observations:

- Empty ranges and singleton ranges expand as expected.
- There is no "`namespace(... reflection_range)`" construct because a sequence of namespaces can currently not appear anywhere in a C++ program.
- If any expansion produces an ill-formed reification, the whole construct is ill-formed but subject to SFINAE.
- Originally, we planned for the ellipsis to follow the reifying construct (e.g., "`typename(rf)...`"), which was more reminiscent of pack expansion. However, it turns out that it was so much like pack expansion that it introduced an ambiguity: If the operand of the reifier is a pack pattern, we don't know whether to expand the range or the pack pattern (each option has reasonable use cases). Using the ellipsis as a prefix resolves the ambiguity. We also considered replacing the ellipsis by a contextual keyword `seq`, but in the end preferred the ellipsis aesthetic.

Reifying a function-local alias or declared entity outside its potential scope is ill-formed. For example:

```
consteval auto refl_int_alias() {
  typedef int Int;
  return reflexpr(Int);
```

```
  }
  typename(refl_int_alias()) x;  // Error: Cannot reify local alias here.
```

Similarly, a parameter obtained from a function type `F` can be reified as an expression only within the potential scope of the corresponding argument of a function of the same type. For example (`parameters_of` will be described later on):

```
using F = int (int, int);
auto params = std::meta::parameters_of(reflexpr(F));
int f(int, int) {
  return idexpr(params[0]);  // Okay
}
int g(int, char) {
  return idexpr(params[0]);  // Error: params[0] comes from function type
}                            // "int (int, int)" but this function has
                             // type "int (int, char)".
```

Reifying identifiers

Because the `[: reflection :]` reifier resolves to an identifier, it can be used to denote objects, references, functions, user-defined types, templates, etc. That means the grammar productions where an identifier reifier can appear must be updated. For example:

unqualified-id:
...
`[:` *reflection* `:]`

If reflection is value-dependent, then the template keyword is needed to indicate that the following `<` begins a template argument list. Hence:

template-id:
`template`$_{opt}$ `[:` *reflection* `:]` `<` *template-argument-list*$_{opt}$ `>`

We expect that we will want the capability of generating identifiers from "constant string values". We have not yet concluded a design of such a capability, but something along the lines of the following should be possible:

```
consteval char const* make_temp_name() { ... }
constexpr char const* name = make_temp_name();
int [: reflexpr(name) :] = 42;
```

A delicate issue here is how to describe a “cosntant string value”.

Access checking

Reifiers provide an alternative way to refer to declarations and therefore we must decide whether they are subject to access control. Access control ordinarily applies to *names*, but reifier constructs are not necessarily names. In fact, the only reifier constructs that behave like names are the “`[: reflection :]`” and “`[: ... reflection_range :]`” constructs. Those are therefore the only reifier constructs subject to access checking. For example:

```
class C {
  using Int = int;
public:
  static consteval auto r() { return reflexpr(Int); };
} c;
typename(C::r()) x;  // Okay: x has type int
C::[:C::r():] y;     // Error: Int is inaccessible.
```

Templates and reflection

Reflection most works “after instantiation”. For example, in:

```
template<typename T> void f(T p) {
  constexpr auto r = reflexpr(T);
}
```

the expression `reflexpr(T)` is always a *dependent* expression that doesn’t produce an actual value until `f` is instantiated. I.e., this does not provide a mechanism to get a handle on a reflection for the template parameter `T`. However, that doesn’t mean that we don’t propose *any* facilities to reflect templated entities.

Template arguments

We propose a function

```
namespace std::meta {
  consteval
  auto has_template_arguments(info reflection)->bool {...};
}
```

that returns `true` if and only if the given reflection corresponds to a template specialization (in the standard sense: implicit specializations are included).

The actual template arguments can be obtained through

```
namespace std::meta {
  consteval auto template_arguments_of(info reflection)
                                        ->std::vector<info> {...};
}
```

Conversely, the template producing a specialization can be obtained with

```
namespace std::meta {
  consteval auto template_of(info reflection)->info {...};
}
```

Note that the resulting reflection value (like that for reflecting a template directly) represents that template as completely known at any point it is examined (including not only the primary template definition, but also partial and full specializations). If the given reflection is not that of a specialization, an invalid reflection is returned.

Template substitution

We also propose a facility that is the "dual notion" of the previous functions:

```
namespace std::meta {
  consteval auto substitute(info templ, std::span<info> args)
                                                    ->info {...};
}
```

This produces a declared-entity reflection for an instance given a reflection for a template and a span of reflections for specific arguments. A substitution error in the immediate context of the substitution produces an invalid reflection (this is akin to SFINAE). A substitution error outside that immediate context renders the program ill-formed. An incomplete substitution (where not all parameters are substituted by nondependent arguments) also produces an invalid reflection. Note, this functionality can also be expressed using the reifier of the form `[< reflection >]`, but having both improves readability depending on the context.

Example:

```
using namespace std::meta;
template<typename ... Ts> struct X {};
template<> struct X<int, int> {};
constexpr info type = reflexpr(X<int, int, float>);
```

```
constexpr info templ = template_of(type);
constexpr vector<info> args = template_arguments_of(type);
constexpr info new_type =
              substitute(templ, span<info>(args).subspan(0, 2));
typename(new_type) xii;  // Type X<int, int>, which selects the specialization.
                         // There is no mechanism to instantiate a primary template
                         // definition that is superseded by an explicit/partial
                         // specialization.
```

Another example illustrates how substitutions can produce non-SFINAE errors:

```
template<typename T> struct A {
  T::type I;
};
template<typename T, T::type N> struct Y {};
constexpr info ASpec = reflexpr(A<int>); // No instantiation yet.
constexpr info new_type2 =
  substitute(reflexpr(Y), std::vector<info>{ ASpec, reflexpr(5)});
    // Error: Substitution of Y<A<int>, 5> requires A<int> to be instantiated
    //      outside the immediate context of the substitution.
```

Template parameters

Although applying `reflexpr` to dependent constructs doesn't produce an actual value until instantiation/substitution, reflections of template parameters can be obtained from the reflection of a template:

```
namespace std::meta {
  consteval auto parameters_of(info reflection)
                                          ->std::vector<info> {...};
}
```

Given the reflection of a template, this returns a vector containing a reflection for each template parameter. Each of these reflections can be a type, a constant, or a template. However, not all operations applicable to types/constants/templates are applicable these reflections. For example, it is not possible to apply the `std::meta::substitute` operation on the reflection of template template parameter (but it *is* possible to apply the `std::meta::parameters_of` to such a reflection).

Transcribing the standard library's [meta] section

The standard library [meta] section (in clause [utilities]) provides a large number of utilities to examine and construct types. We propose that all those utilities be given a counterpart in the value-based

reflection world, with needed declarations made available through a new standard header `<meta>`.

For example, consider the type transformation trait

```
std::make_signed<T>
```

which produces a result through its member type

```
std::make_signed<T>::type
```

We propose to have a `std::meta` counterpart as follows:

```
namespace std::meta {
  consteval auto make_signed(info reflection)->info {...};
}
```

This is expected to be implemented using an intrinsic in the compiler (although that is not a requirement). For a reflection value `r` corresponding to a type `T` such that

```
std::make_signed<T>::type
```

is valid, using the new function as `make_signed(r)` is equivalent to:

```
reflexpr(std::make_signed<typename(r)>::type)
```

(except for not actually instantiating templates in a quality implementation). For a reflection value for which the above transformation would not be valid (e.g., `reflexpr(void)`), however, the function returns an invalid reflection.

Most templates specified in [meta.trans] can be transcribed in a similar way, but a few take additional nontype template parameters. Their transcription is also straightforward however. We illustrate this with the `std::enable_if` template whose `consteval` counterpart can be implemented efficiently without intrinsics. The already-standard template-based interface is usually implemented as follows:

```
namespace std {
  template<bool, typename T = void> struct enable_if {};
  template<typename T> struct enable_if<true, T> {
    using type = T;
  };
```

```
}
```

The reflection counterpart is then (including a hypothetical implementation):

```
namespace std::meta {
  consteval auto enable_if(bool cond,
                           info type = reflexpr(void))->info {
    if (cond) {
      return type;
    } else {
      return invalid_reflection(
               "enable_if condition false", current_source_name(),
               current_source_line(), current_source_column());
    }
  };
}
```

(We encourage programmers to prefer *requires-clause*s over `enable_if` for constraining templates.)

The type traits predicates described in [meta.unary] and [meta.rel] are just as easily mapped to the value-based reflection world. For example, the three templates

```
namespace std {
  template<typename T> struct is_union;
  template<typename T, typename ... Args> struct is_constructible;
  template<typename B, typename D> struct is_base_of;
}
```

have counterparts as follows:

```
namespace std::meta {
  consteval auto is_union(info reflection)->bool {...};
  consteval auto is_constructible(info reflection,
                                  std::span<info> arg_types)
                                                  ->bool {...};
  consteval auto is_base_of(info base_type,
                            info derived_type)->bool {...};
}
```

The other cases follow the same patterns.

The three templates in [meta.unary.prop.query]:

```
namespace std {
  template<typename T> struct alignment_of;
  template<typename T> struct rank;
  template<typename T, unsigned I = 0> struct extent;
}
```

are slightly irregular, but the corresponding functions can still be intuited:

```
namespace std::meta {
  consteval auto alignment_of(info type)->std::size_t {...};
  consteval auto rankinfo type)->int {...};
  consteval auto extent(info type, unsigned dim = 0)->int {...};
}
```

The helper templates in [meta.help] and [meta.logical] are not needed for value-based reflection since their counterparts are core language features (like the integer types and the logical operators).

Adapting the Reflection TS' [reflect] section

The Reflection TS (N4818) introduces a large number of template metafunctions. This proposal steals many of those features and adapts them to the value-based reflection world. However, we make some changes to better align the semantics with the constraints of the language definition and the flexibility of our value-based approach.

Predicates

Let's start with the predicates. For example, `is_public` gets a counterpart as follows:

```
namespace std::meta {
  consteval auto is_public(info base_or_mem)->bool {...};
}
```

That function fails to evaluate to a constant (a SFINAEable error) if `base_or_mem` does not designate a base class or a class member (that constraint corresponds to the concepts requirements imposed for the class template `is_public` proposed in the Reflection TS). `is_protected`, `is_private`, `is_accessible` (which checks a member is accessible from the context of invocation), `is_virtual`, and `is_final` are handled in the same way. For example:

```
struct S { int x; };
constexpr bool t = std::meta::is_public(reflexpr(S::x));
                 // = true;
constexpr bool e = std::meta::is_public(reflexpr(S));
                 // Error: reflexpr(S) is not a base or member.
```

The `is_unnamed` metafunction is transcribed similarly:

```
namespace std::meta {
  consteval auto is_unnamed(info entity)->bool {...};
}
```

but this time the function only evaluates to a constant if the given reflection represents a namespace, a data member, a function, a template, a variable, a type, or an enumerator.

`is_scoped_enum` becomes

```
namespace std::meta {
  cosnteval auto is_scoped_enum(info entity)->bool {...};
}
```

and always evaluates to a constant.

We propose to replace `is_constexpr` by:

```
namespace std::meta {
  consteval auto is_declared_constexpr(info entity)->bool {...};
}
```

which is a constant value if entity designates a variable, a function, a static data member, or a template for these. (We propose the alternative name to distinguish the entities that are declared with the `constexpr` or `consteval` specifier from entities that are effectively `constexpr` (e.g., a function template may be declared `constexpr` and its instances would produce `true` values with this predicate; however, the instances may not actually be `constexpr` functions; conversely, lambda call operators and special member functions may be `constexpr` functions without being declared `constexpr`). Immediate (`consteval`) functions and function templates are also identifiable:

```
namespace std::meta {
  consteval auto is_consteval(info entity)->bool {...};
}
```

Instead of `is_static` (for variables) we propose:

```
namespace std::meta {
  consteval
  auto has_static_storage_duration(info entity)->bool {...};
}
```

because "`is_static`" suggests a query about a storage class specifier rather than a storage duration.

```
namespace std::meta {
  consteval auto is_inline(info entity)->bool {...};
}
```

produces a constant value for reflections of variables, functions, variable/function templates, and namespaces.

A number of function properties produce a constant value for reflections of functions only:

```
namespace std::meta {
  consteval auto is_deleted(info entity)->bool {...};
  consteval auto is_defaulted(info entity)->bool {...};
  consteval auto is_explicit(info entity)->bool {...};
  consteval auto is_override(info entity)->bool {...};
  consteval auto is_pure_virtual(info entity)->bool {...};
}
```

The following predicates always produce a constant value given a reflection. They produce a `false` value for invalid reflections, and otherwise return true if the predicate applies to the reflected entity:

```
namespace std::meta {
  consteval auto is_class_member(info reflection)->bool {
    // Return true for class and class template members.
    ...
  };
  consteval auto is_local(info reflection)->bool {
    // Return true for local variables, local members.
    ...
  };
  consteval auto is_namespace(info entity)->bool {...};
  consteval auto is_template(info entity)->bool {...};
  consteval auto is_type(info entity)->bool {
    // Return true for types and type aliases.
    ...
  };

// namespace std::meta cont'd...
  consteval auto is_incomplete_type(info entity)->bool;
  consteval auto is_closure_type(info entity)->bool {...};
  consteval auto has_captures(info entity)->bool {...};
  consteval auto has_default_ref_capture(info entity)->bool {
```

```
    // Return true even there is no effective capture (i.e., it's syntactical only).
    ...
  };
  consteval auto has_default_copy_capture(info entity)->bool {
    // Return true even there is no effective capture (i.e., it's syntactical only).
    ...
  };
  consteval auto is_simple_capture(info entity)->bool {...};
  consteval auto is_ref_capture(info entity)->bool {...};
  consteval auto is_copy_capture(info entity)->bool {...};
  consteval auto is_explicit_capture(info entity)->bool {...};
  consteval auto is_init_capture(info entity)->bool {...};
  consteval auto is_function_parameter(info entity)->bool {...};
  consteval auto is_template_parameter(info entity)->bool {...};
  consteval auto is_class_template(info entity)->bool {...};
  consteval auto is_alias(info reflection)->bool {...};
  consteval auto is_alias_template(info reflection)->bool {...};
  consteval auto is_enumerator(info entity)->bool {...};
  consteval auto is_variable(info entity)->bool {...};
  consteval auto is_variable_template(info entity)->bool {...};
  consteval auto is_static_data_member(info entity)->bool {
    return is_variable(entity) && is_class_member(entity);
  };
  consteval auto is_nonstatic_data_member(info entity)->bool {
    // Return true for nonstatic data members, which includes bit fields.
    ...
  };
  consteval auto is_bit_field(info reflection)->bool {
    // Return true for bit fields, but also for expressions that are bit field selections.
    ...
  };
  consteval auto is_base_class(info entity)->bool {...};
  consteval auto is_direct_base_class(info entity)->bool {...};
  consteval auto is_virtual_base_class(info entity)->bool {
    return is_base_class(entity) && is_virtual(entity);
  }
  consteval auto is_function(info entity)->bool {...};
  consteval auto is_function_template(info entity)->bool {...};
// namespace std::meta cont'd...
  consteval auto is_member_function(info entity)->bool {
    return is_function(entity) && is_class_member(entity);
  };
  consteval
```

```
  auto is_member_function_template(info entity)->bool {
    return is_function_template(entity) && is_class_member(entity);
  };
  consteval
  auto is_static_member_function(info entity)->bool {...};
  consteval
  auto is_static_member_function_template(info entity)->bool {...};
  consteval
  auto is_nonstatic_member_function(info entity)->bool {...};
  consteval
  auto is_nonstatic_member_function_template(info entity)
                                                    ->bool {...};
  consteval auto is_constructor(info entity)->bool {...};
  consteval auto is_constructor_template(info entity)->bool {...};
  consteval auto is_destructor(info entity)->bool {...};
  consteval auto is_destructor_template(info entity)->bool {...};
}  // namespace std::meta
```

Note that `is_bit_field` above is more general than what the TS proposed since it applies not only to the reflection of data members but also to expressions, because “bitfieldness” is a significant property of an expression. Similarly, we add the following five predicates (with no equivalent in the TS) for reflections of expressions:

```
  consteval auto is_lvalue(info reflection)->bool;
  consteval auto is_xvalue(info reflection)->bool;
  consteval auto is_prvalue(info reflection)->bool;
  consteval auto is_glvalue(info reflection)->bool {
    return is_lvalue(reflection) || is_xvalue(reflection);
  }
  consteval auto is_rvalue(info reflection)->bool {
    return is_pralue(reflection) || is_xvalue(reflection);
  }
```

The following predicate produces a constant value given the reflection of a function type or closure type, or an alias thereof:

```
namespace std::meta {
  consteval auto has_ellipsis(info entity)->bool {...};
}
```

The following predicate produces a constant value given the reflection of a

function type or an alias thereof:

```
namespace std::meta {
  consteval auto is_member_function_type(info entity)->bool {...};
}
```

Given the reflection of a function or template parameter, `std::meta:has_default` returns whether it has an associated default argument:

```
namespace std::meta {
  consteval auto has_default(info entity)->bool {...};
}
```

Singular properties

The following functions can be used to identify a source location of declared entities:

```
namespace std::meta {
  consteval auto source_line_of(info entity)->unsigned {...};
  consteval auto source_column_of(info entity)->unsigned {...};
  consteval auto source_file_name_of(info entity)
                                          ->std::string {...};
}
```

Although these produce a constant result for any reflection value, the returned value is unspecified if the reflection is not that of a declared entity (or alias).

The name of declared entities can be accessed through the following:

```
namespace std::meta {
  consteval auto name_of(info entity)->std::string {...};
  consteval auto display_name_of(info entity)->std::string {...};
}
```

For named declared entities/aliases, `name_of` returns a constant string containing the same identifier as that produced by the "`[: info :]`" reifier. For any other operand, it produces a constant empty string.

The `display_name_of` function produces an unspecified constant non-empty string for any reflection (implementations are encouraged to produce a string that is helpful in identifying the reflected item).

Aliases can be "looked through" using the aforementioned function `entity`:

```
namespace std::meta {
```

```
  consteval auto entity(info reflection)->info {...};
}
```

A reflection for the type associated with an entity or expression can be retrieved with

```
namespace std::meta {
  consteval auto type_of(info reflection)->info {...};
}
```

If reflection describes an entity (not an expression) that is not a variable, base class, data member, function, or enumerator, this function returns an invalid reflection.

A “parent” entity can be identified with

```
namespace std::meta {
  consteval auto parent_of(info reflection)->info {...};
}
```

For members of classes or namespaces this returns a reflection of the innermost class or namespace. For a base class, this returns the class type from which the base class was obtained (only direct and virtual base classes can be reflected). For function-local entities that are not class members, `parent_of` returns the a reflection of the enclosing function. For reflections that do not designate an alias or a declared entity, `parent_of` returns an invalid reflection.

The innermost enclosing function and class can also be queried:

```
namespace std::meta {
  consteval auto current_function()->info {...}
  consteval auto current_class_type()->info {...}
}
```

That is particularly useful to deal more efficiently with parameter packs (an example will be presented later on). Note that when invoked from an immediate function in a context that does not require a constant-expression, these functions return the result as if invoked from the calling function.

Given the reflection of a base or nonstatic data member of a class (but not a class template), layout information can be retrieved with the following functions:

```
namespace std::meta {
  consteval auto byte_offset_of(info entity)->std::size_t {...};
  consteval auto bit_offset_of(info entity)->std::size_t {...};
  consteval auto byte_size_of(info entity)->std::size_t {...};
  consteval auto bit_size_of(info entity)->std::size_t {...};
```

```
}
```

For reflections that do not designate a base or a nonstatic data member, this does not successfully produce a constant value. `byte_offset_of` returns the byte offset of the given base or nonstatic data member (within the parent class). For bit-fields, the offset of the first byte containing the bit field is returned; the bit offset of the first bit (counting from the least significant bit) within that byte is produced by `bit_offset_of` (for non-bit-fields, that function returns zero). `byte_size_of` produces the allocated size of the associated subobject, except that is does not produce a constant value for bit fields (for base classes, the result may be less than `sizeof` applied to the base class type). `bit_size_of` produces the allocated size of the associated bit field subobject, and does not produce a constant value for non-bit-field reflections. (A precise specification of this requires a slight tightening of the C++ object model. All implementations already conform to the stricter model.)

The following facilities permit examining parameter types and the "`this`" binding type:

```
namespace std::meta {
  consteval auto this_ref_type(info func_type)->info {...};
}
```

For a member function type `this_ref_type` returns the reflection of the parent class associated with the member type, with any member function *cv-qualifiers* and *ref-qualifiers* added on top. For example:

```
struct S {
  int f() volatile &&;
  int g() const;
} s;
constexpr auto r = this_ref_type(reflexpr(s.f());
    // Reflection for type "S volatile &&".
constexpr auto r = this_ref_type(reflexpr(s.g());
    // Reflection for type "S const".
```

Plural properties

We propose the following function templates to retrieve subobject information:

```
namespace std::meta {
  template<typename ...Fs>
  consteval auto members_of(info class_type, Fs ...filters)
                                          ->std::vector<info> {...};
  template<typename ...Fs>
  consteval auto bases_of(info class_type, Fs ...filters)
                                          ->std::vector<info> {...};
}
```

If called with an argument for `class_type` that is the reflection of a non-class type or a capturing closure type (or an alias/cv-qualified version thereof), these facilities return a vector containing a single invalid reflection.

Otherwise, if no `filters` argument is passed to it, `members_of` returns an "unfiltered" vector of reflections for the following kinds of direct members of a class type (represented by `class_type`): nonstatic and static data members and member functions, member types (enumeration and class types) and member aliases, and member templates other than deduction guides. Generated members are included, but inherited constructors, injected-class-names, and unnamed bit fields are not (the standard doesn't consider those members either). Nonstatic data members appear in declaration order (but not necessarily consecutively).

If any "filters" are passed, they are applied as predicates to the unfiltered vector, and members for which a predicate produces `false` are left out. Predicates are applied left-to-right with short-circuit semantics (i.e., later predicates are not applied if an earlier predicate produced `false`).

Similarly, without `filter` arguments, invoking `bases_of` returns a vector of reflections for the direct bases of the given (non-capturing-closure) class type. Predicates can be added to narrow down the bases of interest.

The following example illustrates some uses of `members_of`:

```
struct S {
  double x;
  int y;
  void f();
};
constexpr auto class_type = reflexpr(S);
constexpr auto s_members = members_of(class_type);
static_assert(s_members.size() == 7);
```

```
  // x, y, f(), the destructor, and generated constructors.
constexpr auto s_data_members =
              members_of(class_type, is_nonstatic_data_member);
static_assert(s_data_members.size() == 2);
  // x and y.
consteval auto has_integral_type(std::meta::info  reflection) {
  return std::meta::is_integral(std::meta::type_of(reflection));
};

constexpr auto s_imembers =
              members_of(class_type, is_nonstatic_data_member,
                                     has_integral_type);
static_assert(s_imembers.size() == 1);
  // Just y.
constexpr auto s_nested_types =
                   members_of(class_type, is_type);
static_assert(s_members.size() == 0);
  // S has no nested types.
```

The enumerators of an enumeration type can be inspected using `enumerators_of`:

```
namespace std::meta {
  consteval
  auto enumerators_of(info enum_type)->std::vector<info> {...};
}
```

If the argument passed for `enum_type` is not a reflection for an enumeration type, this returns a vector containing one element which is an invalid reflection.

The parameters of a function type or the parameters of a template can be inspected using:

```
namespace std::meta {
  consteval auto parameters_of(info reflection)
                                        ->std::vector<info> {...};
}
```

If the argument passed for `reflection` is not a reflection for a function, a member function, a function type, a closure type, or a template, this returns a vector containing one invalid reflection. Otherwise, the vector contains an entry for each ordinary parameter: No entry is made for the "`this`" parameter or for an ellipsis parameter.

A function is also available to introspect lambda captures associated with a closure type:

```
namespace std::meta {
  consteval auto captures_of(info closure_type)
                                          ->std::vector<info> {...};
}
```

If the argument passed for `func_type` is not a reflection for a function or closure type, this returns a vector containing one invalid reflection.

Of note here is that we are *not* proposing a function to retrieve members of namespaces: Due to their"open scope" nature, we believe that capability is somewhat meaningless. There is however no known technical reason preventing us from doing so.

We are also *not* proposing functions to expose the structure of templates. In particular, there is no mechanism to retrieve the members of a class template or the function parameters of a function template.

Anonymous unions

Consider:

```
struct S {
  bool flag;
  union {
    int x;
    float f;
  };
};
constexpr auto dmembers =
               members_of(reflexpr(S), is_nonstatic_data_member);
static_assert(dmembers.size() == 2);
```

The vector `dmembers` here will contain two reflections: One for `flag` and one for an unnamed data member of the unnamed union type. Conversely, `parent_of(reflexpr(S::x))` produces a reflection for that unnamed union type rather than for `S`. Despite its lack of a declared name (which means `name_of` returns an empty string constant), the unnamed data member can be referred to with the "`idexpr( ... )`" reifier:

```
constexpr S s = { false, { .x = 42 } };
static_assert(name_of(dmembers[1]) == "");  // Okay.
static_assert(s.idexpr( dmembers[1] ).x == 42);  // Okay.
static_assert(
  remove_reference(
     reflexpr(decltype(s.idexpr( dmembers[1] ))))) ==
                          parent_of(reflexpr(S::f)));  // Okay.
```

Let's take that last line apart.

In the left-hand side of the equality test `dmembers[1]` is a reflection of the unnamed data member for the anonymous union. Therefore, `s.reflexpr( dmembers[1] )` is an lvalue designating the anonymous union subobject of `s`, and thus `decltype` applied to that produces a reference to the anonymous union type. The `reflexpr` operator returns the reflection designating that type and `remove_reference` finally returns a reflection designating the underlying union type.

In the right-hand side, `reflexpr(S::f)` is a reflection designating the member `S::f`, which is actually a member `S::<unnamed-union-type>::f`. Therefore, `parent_of` also produces a reflection designating the underlying union type of the anonymous union, and the assertion succeeds.

Metaprogramming Examples

We believe that the facilities presented here permit the kind of computation previously performed with C++ template metaprogramming and that they are preferable over TMP because they scale better. We therefore suggest that no broad set of TMP facilities should be further added to the language.

Examples in this section are drawn from a variety of sources, including P0385R0 by Matúš Chochlík and Axel Naumann and P0949R0 by Peter Dimov.

Hashing

We can also use the approach above to synthesize an overload of `hash_append` (proposed by Howard Hinnant *et al.* in N3980, *Types Don't Know #*).

```
#include <meta>
using namespace meta = std::meta;
template<HashAlgorithm H, StandardLayoutType T>
  bool hash_append(H& algo, const T& t) {
    constexpr auto data_members =
        members_of(reflexpr(T), meta::is_nonstatic_data_member);
    for constexpr (meta::info member : data_members)
      hash_append(algo, t.idexpr(member));
  }
```

The algorithm is straightforward: recursively apply `hash_append` to each member for the class `T`. Within that call the expression `t.idexpr(member)` yields a *postfix-expression* for the designated member in the class object. The resolution of that postfix-expression does not require name lookup or access control (unlike by-name mechanisms), and it works even for bit fields (unlike mechanisms based on pointer-to-member values).

Note that this uses *expansion statements* as proposed by P1306 (and that is typical of practical uses of reflection). An expansion statement requires a compile-time range, which is why `data_members` must be a `constexpr` variable.

Schema generation

We can use this same pattern to generate SQL schemas from C++ classes. The implementation here mixes runtime SQL generation with static reflection, in order to demonstrate the interaction between these two features.

The entry point for the facility is a function template that takes a (standard layout) type parameter writes the corresponding SQL `CREATE TABLE` statement.

```
template<StandardLayoutType T>
void create_table() {
  return create_table_from_reflection<reflexpr(T)>();
}
```

This function simply delegates to a function parameterized by its reflection. Because reflection is expected to be an “advanced” feature, it is probably advisable to hide it from user-facing interfaces. The SQL generating function template is shown below.

```
#include <meta>
using namespace meta = std::meta;

template<meta::info Class>
  requires meta::is_class(Class)
void create_table_from_reflection() {
  std::cout << "CREATE TABLE " << meta::name_of(Class) << “(\n”;
  constexpr auto members =
      meta::members_of(Class, is_non_static_data_member);
  int size = members.size(), num = 0;
```

```
  for constexpr (meta::info member : members) {
    create_column<member>();
    if (++num != size)
      std::cout << ",\n";
  }
  std::cout << ");\n";
}
```

This function emits a `CREATE TABLE` statement for the name of the class, and "iterates" over the class's data members — again using an *expansion statement* (P1306) — emitting column definitions for each (see below for `create_column`). We maintain the member count so that we can correctly insert commas into the output after each column.

Reflection facilities can only be used at compile time. Because this function mixes runtime code (`std::cout`) with static reflection (`meta::info`), we need to ensure that reflections do not "mix" with the runtime systems. We cannot, with this approach to generating SQL, pass the reflected class as a function argument, as that would leak the reflection — handle to an internal data structure that is only meaningful during translation — to run time. In other words, for mixed run-time/reflective algorithms reflection values must be passed as template arguments. We explore an alternative design of this algorithm in the following section.

Creating a column is straightforward: We serialize the member's name and translate its C++ type into SQL.

```
template<meta::info Member>
  requires meta::is_non_static_data_member(Member)
void create_column() {
  std::cout << meta::name_of(Member) << " ";
  std::cout << to_sql(meta::type_of(Member));
}
```

Finally, we need a facility to translate C++ types to SQL types. Here, we use a series of explicit specializations over reflections, with the generic case (i.e., primary template) triggering an instantiation error if used.

```
template<meta::info Type>
consteval const char* to_sql() {
  static_assert(false, "no translation to SQL");
}

template<>
consteval const char* to_sql<reflexpr(int)>() {
  return "INTEGER";
```

```
}

template<>
consteval const char* to_sql<reflexpr(float)>() {
  return “FLOAT”;
}
// etc.
```

Schema generation (take two)

The approach above mixes runtime SQL generation with static reflection: We call a function to print the schema to `std::cout`. An alternative approach is to synthesize the schema as a compile-time string, and then print the result later. The entirety of that function is shown below:

```
template<StandardLayoutType T>
consteval std::string create_table() {
  return create_table_from_reflection<reflexpr(T)>();
}
```

```
#include <meta>
template<meta::info Class>
  requires meta::is_class(Class)
consteval std::string create_table() {
  std::string result(“”); // Assuming this should work
  result += "CREATE TABLE " + meta::name_of(Class) + “(\n”;
  std::vector<meta::info> members = data_members.size();
  int num = 0;
  for (meta::info member : members) {
    result += create_column(member);
    if (++num != members.size())
      result += “,\n”;
```

```
  }
  result += ");\n";
  return result;
}

consteval std::string void create_column(meta::info member) {
  std::string result("");
  result += meta::name_of(member) + " ";
  result += to_sql(meta::type_of(member));
  return result;
}

consteval const char* to_sql(meta::info type) {
  static std::unordered_map<meta::info, const char*> types {
    {reflexpr(int), "INTEGER"},
    {reflexpr(float), "FLOAT"},
    // etc.
  };
  [[assert: meta::is_type(type)]];
  [[assert: types.count(type) != 0]];
  return types.find(type)->second;
}
```

There are significant differences between this and the earlier example. In essence, this implementation looks like a normal program except that each function is an `consteval` function In other words, because the entire facility is expected to run at compile time, we don't have to maintain a clear separation between the run-time and compile-time values in the implementation; everything just looks like run-time code.

Template argument list assignment

In P0949R0, Peter Dimov proposes a facility to "assign" a list of template arguments for one template to another using:

```
mp_assign<ClassTmpl1<A1, A2, ...>, ClassTmpl2<B1, B2, ...>>
```

This is an alias for `ClassTmpl1<B1, B2, ...>`. The template arguments A*n* and B*n* are all type arguments. If the arguments of `mp_assign` are not of those forms, a substitution failure occurs.

Using the features proposed in this paper, we can implement this facility as follows:

```
#include <meta>
using std::meta::info;
```

```
using std::vector;
consteval info class_template_of(info inst) {
  using namespace std::meta;
  info tmpl = template_of(inst);
  if (is_class_template(inst) || is_invalid_reflection(tmpl) {
    return tmpl;
  } else {
    return invalid_reflection("Not a class template instance");
  }
}

consteval vector<info> template_type_arguments_of(info  inst) {
  using namespace std::meta;
  auto args = template_arguments_of(inst);
  for (auto arg: args) {
    if (is_invalid(arg) && args.size() == 1) {
      // template_arguments_of was invalid: Propagate the error.
      return args;
    } else if (!is_type(arg)) {
      // Not a type argument.
      return vector<info>{
              invalid_reflection("Not all arguments are types")};
    }
  }
  return args;
}
```

```
consteval info rf_assign(info inst1, info inst2) {
  using namespace std::meta;
  info tmpl1 = class_template_of(inst1);
  info tmpl2 = class_template_of(inst2);
  if (is_invalid(tmpl2)) return tmpl2;
  auto args1 = template_type_arguments_of(inst1);
  if (args1.size() == 1 && is_invalid(args1[0])) return args1;
  auto args2 = template_type_arguments_of(inst1);
  return substitute_template(tmpl1, args2);
}
```

If needed, `mp_assign` could be expressed in terms of `rf_assign`:

```
template<typename T1, typename T2>
  using mp_assign =
         typename(rf_assign(reflexpr(T1), reflexpr(T2)));
```

Note that in our implementation of `rf_assign`, much of the code is dedicated to implementing the constraints of `mp_assign`. However, those constraints exist only because of two TMP limitations:

1) parameter packs cannot model mixed-kind template argument lists, and
2) template template parameters cannot accept function/variable templates.

In the reflection world we can easily lift those constraints, which produces the following simplified-yet-more-powerful implementation of `rf_assign`:

```
consteval info rf_assign(info inst1, info inst2) {
  using namespace std::meta;
  return substitute(template_of(inst1),
                    template_arguments_of(inst2));
}
```

Dealing more efficiently with parameter packs

Currently, parameter packs are generally dealt with through recursive template instantiation (i.e., a form of TMP, with all its disadvantages). With the set of features presented here, many interesting applications of packs can be expressed more directly and using fewer compilation resources. Here is a simple example:

```
#include <meta>
// Function taking an arbitrary number of arguments and returning a vector containing copies of the
// arguments that have the given type T.
template<typename T, typename ... Ts>
std::vector<T> select_values_of_type(Ts ... p) {
  std::vector<T> result{};
  for constexpr (auto param: parameters_of(current_function())) {
    if (reflexpr(T) == type_of(param)) {
      result.push_back(idexpr(params[i]));
    }
  }
  return result;
}
```

Applying functions to all members

P0949r0 presents a TMP metafunction `get_all_data_members` aimed at collecting reflection information for all the data members of a class (not just the direct ones) using the facilities of the first reflection TS (N4818).

Unfortunately, `get_all_data_members` as presented in P0949r0 has a number of problems:

- It doesn't correctly use the TMP-based reflection API to access base classes (it looks like it treats a base class as a base class *type*). Fixing that is nontrivial.
- Its logic ignores virtual bases.
- The TMP-based reflection API doesn't deal well with bit fields (it relies on pointer-to-member constants, which cannot point to bit fields).

To address those shortcomings, we present a different interface with similar capabilities:

```
template<typename T, typename F>
void apply_to_all_data_members(T &&r_obj, F &&f);
  // Invoke f(r_obj.x) for every accessible data member of r_obj, including
  // those in base classes (and possibly hidden by more-derived member declarations).
```

With the facilities we have proposed in this paper, this can be implemented as follows.

```
#include <meta>
using std::meta::info;
using std::vector;
```

```
// Convenience function to retrieve accessible nonstatic data members of a given class:
consteval auto get_members(info  classinfo) {
  return members_of(classinfo, is_nonstatic_data_member,
                               is_accessible);
};

// Convenience function to select nonvirtual bases and members.
consteval auto is_not_virtual(info base_or_mem) {
  return !is_virtual(base_or_mem);
};

// Utility to get the reflection information for the types of base classes (rather than the base
// classes themselves) of a given class.
consteval
auto get_base_types(info classtype, bool virtual_bases) {
  auto result = bases_of(classtype,
                         is_accessible,
                         virtual_bases ? is_virtual
                                       : is_not_virtual);
  // Replace each base reflection by the reflection of its type.
  for (auto &info : result) {
    info = type_of(info);
  }
  return result;
};

template<typename T, typename F>
void apply_to_data_members(T *p_obj, F &f) {
  for constexpr (auto member : get_members(reflexpr(T))) {
    f(p_obj->idexpr(member));
  }
}
```

```
template<typename T, typename F>
void apply_to_base_data_members(T *p_obj, F &f,
                                bool virtual_bases,
                                bool skip_direct_members) {
  // Recursively traverse (depth-first) either the nonvirtual or virtual base classes (depending
  // on the virtual_bases flag). We do this by collecting the base class types and casting
  // the pointer one level up.
  auto type = reflexpr(T);
  for ... (auto basetype : get_base_types(type, virtual_bases)) {
    apply_to_base_data_members<T, F>(
      static_cast<typename(basetype)*>(p_obj), f,
      virtual_bases, /*skip_direct_members=*/false);
  }
  if (!skip_direct_members) {
    //  Now that the base classes have been traversed, handle the data members at this level.
    for ... (auto member : get_members(type)) {
      f(p_obj->idexpr(member));
    }
  }
}

template<typename T, typename F>
void apply_to_all_data_members(T const &&r_obj, F &&f) {
  T const *p_obj = std::addressof(r_obj);
  apply_to_base_data_members<T, F>(
    p_obj, f, /*virtual_bases=*/true,
              /*skip_direct_members=*/true);
  apply_to_base_data_members<T, F>(
    p_obj, f, /*virtual_bases=*/false,
              /*skip_direct_members=*/false);
}
```

This implementation reads like ordinary C++ code. Every invocation instantiates three function templates, independently of how complex type `T` is (though the amount of code in each instantiation does depend on `T` because of the expansion statements).

This implementation still has a weakness, however: The notion of "accessibility" of bases and members is determined from the context of the implementation, not that of the call to `apply_to_all_data_members`. (The same limitation is imposed by the first Reflection TS.) We do not at this time propose to resolve that issue but we know of at least two ways to address it:

- more powerful code injection primitives, or
- introduce a reflection for "context".

The second option would be less efficient since that context would have to be passed along as a template

argument, which would cause each invocation of `apply_to_all_data_members` to have a distinct instantiation.

Alternatively, here is an implementation that doesn't use *expansion statements*. Instead, it relies on *fold-expressions*.

```
// Convenience function to retrieve accessible nonstatic data members of a given class:
consteval auto get_members(info  classinfo) {
  return members_of(classinfo, is_nonstatic_data_member,
                               is_accessible);
};

// Convenience function to select nonvirtual bases and members.
consteval auto is_not_virtual(info base_or_mem) {
  return !is_virtual(base_or_mem);
};

// Utility to get the reflection information for the types of base classes (rather than the base
// classes themselves) of a given class.
consteval
auto get_base_types(info classtype, bool virtual_bases) {
  auto result = bases_of(classtype,
                         is_accessible,
                         virtual_bases ? is_virtual
                                       : is_not_virtual);
  // Replace each base reflection by the reflection of its type.
  for (auto &info : result) {
    info = type_of(info);
  }
  return result;
};

template<typename T, typename F, std::meta::info ... members>
void apply_to_data_members(T *p_obj, F &f) {
  (void)(f(p_obj->idexpr(members)), ...);  // Fold-expression.
}
```

```
template<typename T, typename F, std::meta::info ... classtypes>
void apply_to_base_data_members(T *p_obj, F &f,
                                bool virtual_bases,
                                bool skip_direct_members) {
  using namespace std::meta;
  // Use a fold-expression to recurse through given bases if needed.
  (apply_to_base_data_members<
     T, F, [<...get_base_types(classtypes, virtual_bases)>]
   >(static_cast<typename(typeof(bases))*>(p_obj), f,
     virtual_bases,
     /*skip_direct_members=*/false), ...);
  if (!skip_direct_members) {
    // Use another fold-expression to handle the data members of each specified class type.
    (apply_to_data_members<
       T, F, [<...get_members(classtypes)>]
     >(p_obj, f), ...);
  }
}

template<typename T, typename F>
void apply_to_all_data_members(T const &&r_obj, F &&f) {
  T const *p_obj = std::addressof(r_obj);
  apply_to_base_data_members<T, F, reflexpr(T)>(
    p_obj, f, /*virtual_bases=*/true,
              /*skip_direct_members=*/true);
  apply_to_base_data_members<T, F, reflexpr(T)>(
    p_obj, f, /*virtual_bases=*/false,
              /*skip_direct_members=*/false);
}
```

Clearly this is far less readable than the first version. It also involves more instantiations than the first version, but it is nonetheless more efficient than a pure TMP-based solution.

Appendix: Meta-library synopsis

This appendix briefly lists declarations for all the intrinsic meta-functions being worked on in the Lock3 Software implementation. As mentioned, these declarations are eventually meant to be brought into a program by including the standard header `<meta>`. In its current form the list differs slightly from the discussions in this paper because of implementation realities. We expect to harmonize the two over time.

Parameters to queries are named to represent the subset of `meta::info` values accepted by each function:

- `reflection` – accepts any value.

- `invalid` – accepts only an invalid reflection.
- `function` – accepts any value that designates a function.
- `variable` – accepts any value that designates a variable.
- `bitfield` – accepts any value that designates a bitfield.
- `type` – accepts any value that designates a type.
- `xxx_type` – accepts any value that designates a type of kind `xxx` (e.g., `enum_type`)
- `templ` – accepts any value that designates a template.
- `special` – accepts any value that designates a template specialization.
- `entity` – accepts any value that designates an entity.
- `parameter` – accepts any value that designates a function or template parameter.
- `expression` – accepts any value that designates an expression.
- `argument` – accepts any value that designates a function or template argument.
- `base` – accepts any value that designates a base class specifier.
- `mem` – accepts any value that designates a class member.
- `mem_function` – accepts any value that designates a member function.
- `spec_mem_function` – accepts any value that designates a special member function.
- `base_or_mem` – accepts any value that designates a base class specifier or a class member.

Some operations are polymorphic and accept combinations. For example, the parameter of `is_public` is `base_or_mem`. Operations accepting sequences are pluralized (e.g., `reflections`), meaning that the restriction applies to all elements of the sequence. Also, note that function accepting a `declarator` will accept a function, variable, or bitfield.

Although all the declarations are being considered and worked on, some are not implemented or not tested. The list below highlights those declarations as follows:

Highlights for interfaces still requiring work	
`Red`	Not implemented, because of not-yet-resolved limitations
`Orange`	Not implemented, because of fixable limitations
`Yellow`	Not *yet* implemented
`Green`	Implemented, but not tested

The synopsis of the `<experimental/meta>` header is:

```
namespace std::meta {
  // Reflection type
  using info = decltype(reflexpr(void));

  // Classification
  consteval bool is_invalid(info reflection);

  // Scope
  consteval bool is_local(info reflection);
  consteval bool is_class_member(info reflection);

  // Variables
  consteval bool is_variable(info reflection);
  consteval bool has_static_storage_duration(info variable);
  consteval bool has_thread_local_storage_duration(
      info variable);
  consteval bool has_automatic_storage_duration(
      info variable);

  // Functions
  consteval bool is_function(info reflection);
  consteval bool is_nothrow(info function);
  consteval bool has_ellipsis(info function);

  // Classes
  template<typename ...Args>
  consteval std::vector<info> members_of(
                                info class_type, Args ...filters);
  template<typename ...Args>
  consteval std::vector<info> bases_of(
                                info class_type, Args ...filters);

  // Classes
  consteval bool is_class(info reflection);
  consteval bool is_union(info reflection);
  consteval bool has_virtual_destructor(info class_type);
  consteval bool is_declared_class(info class_type);
  consteval bool is_declared_struct(info class_type);



  // Data Members
```

```
consteval bool is_data_member(info reflection);
consteval bool is_static_data_member(info reflection);
consteval bool is_nonstatic_data_member(info reflection);
consteval bool is_bit_field(info reflection);
consteval bool is_mutable(info reflection);

// Member Functions
consteval bool is_member_function(info reflection);
consteval bool is_static_member_function(info reflection);
consteval bool is_nonstatic_member_function(info reflection);
consteval bool is_normal(info mem_function);
consteval bool is_conversion(info mem_function);
consteval bool is_override(info mem_function);
consteval bool is_override_specified(info mem_function);
consteval bool is_deleted(info mem_function);
consteval bool is_virtual(info mem_function);
consteval bool is_pure_virtual(info mem_function);

// Special Member Functions
consteval bool is_constructor(info reflection);
consteval bool is_default_constructor(info reflection);
consteval bool is_copy_constructor(info reflection);
consteval bool is_move_constructor(info reflection);
consteval bool is_copy_assignment_operator(info reflection);
consteval bool is_move_assignment_operator(info reflection);
consteval bool is_copy(info mem_function);
consteval bool is_move(info mem_function);
consteval bool is_destructor(info reflection);
consteval bool is_defaulted(info spec_mem_function);
consteval bool is_explicit(info spec_mem_function);

// Access
consteval bool has_access(info reflection);
consteval bool is_public(info base_or_mem);
consteval bool is_protected(info base_or_mem);
consteval bool is_private(info base_or_mem);
consteval bool has_default_access(info base_or_mem);

// Linkage
consteval bool has_linkage(info reflection);
consteval bool is_externally_linked(info reflection);
consteval bool is_internally_linked(info reflection);
```

```
  // General purpose
  consteval bool is_extern_specified(info reflection);
  consteval bool is_inline(info reflection);
  consteval bool is_inline_specified(info reflection);
  consteval bool is_constexpr(info reflection);
  consteval bool is_consteval(info reflection);
  consteval bool is_final(info reflection);
  consteval bool is_defined(info reflection);
  consteval bool is_complete(info reflection);

  // Namespaces
  consteval bool is_namespace(info reflection);

  // Aliases
  consteval bool is_alias(info reflection);
  consteval bool is_namespace_alias(info reflection);
  consteval bool is_type_alias(info reflection);
  consteval bool is_alias_template(info reflection);

  // Enums
  consteval bool is_enum(info reflection);
  consteval bool is_unscoped_enum(info reflection);
  consteval bool is_scoped_enum(info reflection);
  consteval std::vector<info> enumerators_of(info enum_type);

  // Enumerator
  consteval bool is_enumerator(info reflection);

  // Templates
  consteval bool is_template(info reflection);
  consteval bool is_class_template(info reflection);
  consteval bool is_function_template(info reflection);
  consteval bool is_variable_template(info reflection);
  consteval bool is_member_function_template(info reflection);
  consteval bool is_static_member_function_template(reflection);
  consteval bool is_nonstatic_member_function_template(
                                                 info reflection);
  consteval bool is_constructor_template(info reflection);
  consteval bool is_destructor_template(info reflection);
  consteval bool is_concept(info reflection);


  // Specializations
```

```
  consteval bool is_specialization(info reflection);
  consteval bool is_partial_specialization(info reflection);
  consteval bool is_explicit_specialization(info reflection);
  consteval bool is_implicit_instantiation(info reflection);
  consteval bool is_explicit_instantiation(info reflection);

  consteval info template_of(info special);
  consteval bool has_template_arguments(info reflection);
  consteval std::vector<info> template_arguments_of(info special);
  consteval info substitute(info templ, std::span<info> args);

  // Base classes
  consteval bool is_base_class(info reflection);
  consteval bool is_direct_base_class(info reflection);
  consteval bool is_virtual_base_class(info reflection);


  // Parameters
  consteval bool is_function_parameter(info reflection);
  consteval bool is_template_parameter(info reflection);
  consteval bool is_type_template_parameter(info reflection);
  consteval bool is_nontype_template_parameter(info reflection);
  consteval bool is_template_template_parameter(info reflection);
  consteval bool has_default_argument(info parameter);

  consteval std::vector<info> parameters_of(
                                          info function_or_templ);

  // Types
  consteval bool is_type(info reflection);
  consteval bool is_fundamental_type(info type);
  consteval bool has_fundamental_type(info reflection);
  consteval bool is_arithmetic_type(info type);
  consteval bool has_arithmetic_type(info reflection);
  consteval bool is_scalar_type(info type);
  consteval bool has_scalar_type(info reflection);
  consteval bool is_object_type(info type);
  consteval bool has_object_type(info reflection);
  consteval bool is_compound_type(info type);
  consteval bool has_compound_type(info reflection);
  consteval bool is_function_type(info type);
  consteval bool has_function_type(info reflection);
  consteval bool is_class_type(info type);
```

```
consteval bool has_class_type(info reflection);
consteval bool is_union_type(info type);
consteval bool has_union_type(info reflection);
consteval bool is_enum_type(info type);
consteval bool has_enum_type(info type);
consteval bool is_unscoped_enum_type(info type);
consteval bool has_unscoped_enum_type(info reflection);
consteval bool is_scoped_enum_type(info type);
consteval bool has_scoped_enum_type(info reflection);
consteval bool is_void_type(info type);
consteval bool has_void_type(info reflection);
consteval bool is_null_pointer_type(info type);
consteval bool has_null_pointer_type(info reflection);
consteval bool is_integral_type(info type);
consteval bool has_integral_type(info reflection);
consteval bool is_floating_point_type(info type);
consteval bool has_floating_point_type(info reflection);
consteval bool is_array_type(info type);
consteval bool has_array_type(info reflection);
consteval bool is_pointer_type(info type);
consteval bool has_pointer_type(info reflection);
consteval bool is_reference_type(info type);
consteval bool has_reference_type(info reflection);
consteval bool is_lvalue_reference_type(info type);
consteval bool has_lvalue_reference_type(info reflection);
consteval bool is_rvalue_reference_type(info type);
consteval bool has_rvalue_reference_type(info reflection);
consteval bool is_member_pointer_type(info type);
consteval bool has_member_pointer_type(info reflection);
consteval bool is_member_object_pointer_type(info type);
consteval bool has_member_object_pointer_type(info reflection);
consteval bool is_member_function_pointer_type(info type);
consteval bool has_member_function_pointer_type(
                                              info reflection);
consteval bool is_closure_type(info type);
consteval bool has_closure_type(info reflection);

// Type properties
consteval bool is_incomplete_type(info type);
consteval bool has_incomplete_type(info reflection);
consteval bool is_const_type(info type);
consteval bool has_const_type(info reflection);
consteval bool is_volatile_type(info type);
```

```
  consteval bool has_volatile_type(info reflection);
  consteval bool is_trivial_type(info type);
  consteval bool has_trivial_type(info reflection);
  consteval bool is_trivially_copyable_type(info type);
  consteval bool has_trivially_copyable_type(info reflection);
  consteval bool is_standard_layout_type(info type);
  consteval bool has_standard_layout_type(info reflection);
  consteval bool is_pod_type(info type);
  consteval bool has_pod_type(info reflection);
  consteval bool is_literal_type(info type);
  consteval bool has_literal_type(info reflection);
  consteval bool is_empty_type(info type);
  consteval bool has_empty_type(info reflection);
  consteval bool is_polymorphic_type(info type);
  consteval bool has_polymorphic_type(info reflection);
  consteval bool is_abstract_type(info type);
  consteval bool has_abstract_type(info reflection);
  consteval bool is_final_type(info type);
  consteval bool has_final_type(info reflection);
  consteval bool is_aggregate_type(info type);
  consteval bool has_aggregate_type(info reflection);
  consteval bool is_signed_type(info type);
  consteval bool has_signed_type(info reflection);
  consteval bool is_unsigned_type(info type);
  consteval bool has_unsigned_type(info reflection);
  consteval bool has_unique_object_representations(info type);
  consteval bool has_type_with_unique_object_representations(
                                                  info reflection);

  consteval std::size_t size_of(info reflection);
  consteval std::size_t byte_size_of(info reflection);
  consteval std::size_t bit_size_of(info reflection);
  consteval std::size_t byte_offset_of(info reflection);
  consteval std::size_t bit_offset_of(info reflection);
  consteval std::size_t alignment_of(info reflection);
  consteval std::size_t rank(info reflection);
  consteval std::size_t extent(info reflection);

  // Type operations
  consteval bool is_constructible(
                      info reflection, std::span<info> arguments);
  consteval bool is_trivially_constructible(
                       info reflection, std:span<info> arguments);
```

```
  consteval bool is_nothrow_constructible(
                      info reflection, std::span<info> arguments);
  consteval bool is_default_constructible_type(info type);
  consteval bool has_default_constructible_type(info reflection);
  consteval bool is_trivially_default_constructible_type(
                                                        info type);
  consteval bool has_trivially_default_constructible_type(
                                                   info reflection);
  consteval bool is_nothrow_default_constructible_type(info type);
  consteval bool has_nothrow_default_constructible_type(
                                                   info reflection);
  consteval bool is_copy_constructible_type(info type);
  consteval bool has_copy_constructible_type(info reflection);
  consteval bool is_trivially_copy_constructible_type(info type);
  consteval bool has_trivially_copy_constructible_type(
                                                   info reflection);
  consteval bool is_nothrow_copy_constructible_type(info type);
  consteval bool has_nothrow_copy_constructible_type(
                                                   info reflection);
  consteval bool is_move_constructible_type(info type);
  consteval bool has_move_constructible_type(info reflection);
  consteval bool is_trivially_move_constructible_type(info type);
  consteval bool has_trivially_move_constructible_type(
                                                   info reflection);
  consteval bool is_nothrow_move_constructible_type(info type);
  consteval bool has_nothrow_move_constructible_type(
                                                   info reflection);
  consteval bool is_assignable_type(info type,
                                    info assigned_type);
  consteval bool is_trivially_assignable_type(info type,
                                              info assigned_type);
  consteval bool is_nothrow_assignable_type(info type,
                                            info assigned_type);
  consteval bool is_copy_assignable_type(info type);
  consteval bool has_copy_assignable_type(info reflection);
  consteval bool is_trivially_copy_assignable_type(info type);
  consteval bool has_trivially_copy_assignable_type(
                                                   info reflection);
  consteval bool is_nothrow_copy_assignable_type(info type);
  consteval bool has_nothrow_copy_assignable_type(
                                                   info reflection);
  consteval bool is_move_assignable_type(info type);
  consteval bool has_move_assignable_type(info reflection);
```

```
  consteval bool is_trivially_move_assignable_type(info type);
  consteval bool has_trivially_move_assignable_type(
                                                  info reflection);
  consteval bool is_nothrow_move_assignable_type(info type);
  consteval bool has_nothrow_move_assignable_type(
                                                  info reflection);
  consteval bool is_destructible_type(info type);
  consteval bool has_destructible_type(info reflection);
  consteval bool is_trivially_destructible_type(info type);
  consteval bool has_trivially_destructible_type(info reflection);
  consteval bool is_nothrow_destructible_type(info type);
  consteval bool has_nothrow_destructible_type(info reflection);

  consteval bool is_swappable(info reflection);
  consteval bool is_nothrow_swappable(info reflection);
  consteval bool is_swappable_with(
                             info reflection1, info reflection2);
  consteval bool is_nothrow_swappable_with(
                            info reflection1, info reflection2);

  // Captures
  consteval std::vector<info> captures_of(info reflection);

  consteval bool has_default_ref_capture(info reflection);
  consteval bool has_default_copy_capture(info reflection);

  consteval bool is_capture(info reflection);
  consteval bool is_simple_capture(info reflection);
  consteval bool is_ref_capture(info reflection);
  consteval bool is_copy_capture(info reflection);
  consteval bool is_explicit_capture(info reflection);
  consteval bool is_init_capture(info reflection);
  consteval bool has_captures(info reflection);

  // Type relations
  consteval bool is_same(info reflection1, info reflection2);
  consteval bool is_base_of(info base_type, info derived_type);
  consteval bool is_convertible(info from_type, info to_type);
  consteval bool is_nothrow_convertible(
                                     info from_type, info to_type);


  // Invocation
```

```
  consteval bool is_invocable(
                        info function, std::span<info> arguments);
  consteval bool is_nothrow_invocable(
                        info function, std::span<info> arguments);
  consteval bool is_invocable_r(
            info function, std::span<info> arguments, info result);
  consteval bool is_nothrow_invocable_r(
            info function, std::span<info> arguments, info result);

  // Type transformation
  consteval info remove_const(info type);
  consteval info remove_volatile(info type);
  consteval info remove_cv(info type);
  consteval info add_const(info type);
  consteval info add_volatile(info type);
  consteval info add_cv(info type);
  consteval info remove_reference(info type);
  consteval info add_lvalue_reference(info type);
  consteval info add_rvalue_reference(info type);
  consteval info remove_pointer(info type);
  consteval info add_pointer(info type);
  consteval info remove_cvref(info type);
  consteval info decay(info type);
  consteval info make_signed(info type);
  consteval info make_unsigned(info type);

  consteval info aligned_storage(
                      std::size_t length,
                      std::size_t align = /* default-alignment */);
  consteval info aligned_union(
                        std::size_t length, std::span<info> types);

  consteval info enable_if(bool cond, info type = reflexpr(void));

  // Associated types
  consteval info this_ref_type_of(info mem_function);
  consteval info common_type(const std::vector<info>& types);
  consteval info underlying_type_of(info reflection);
  consteval info invoke_result(
                        info function, std::span<info> arguments);
  consteval info type_of(info reflection);
  consteval info return_type_of(info function);
```

```
  // Associated reflections
  consteval bool is_entity(info reflection);
  consteval info entity_of(info reflection);
  consteval info parent_of(info reflection);
  consteval info definition_of(info reflection);

  // Names
  consteval bool is_named(info reflection);
  consteval std::string name_of(info named);
  consteval std::string display_name_of(info named);

  // Expressions
  consteval bool is_lvalue(info reflection);
  consteval bool is_xvalue(info reflection);
  consteval bool is_prvalue(info reflection);
  consteval bool is_glvalue(info reflection);
  consteval bool is_rvalue(info reflection);
}
```