Document: P1240R0
Revises: P0993R0
Date: 08-10-2018
Audience: EWG, SG7
Authors: Andrew Sutton (asutton@uakron.edu)
Faisal Vali (faisalv@yahoo.com)
Daveed Vandevoorde (daveed@edg.com)

Scalable Reflection in C++

Introduction

The first Reflection TS (based on P0194r5) 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 P099r0, *Value-based Reflection*. To support that reflection design, we have passed a number of extensions to the C++17 `constexpr` feature: immediate (i.e., "`constexpr!`") functions (P1073r1), `std::is_constant_evaluated()` (P0595r1), constexpr dynamic allocation (P0784r3), and expansion statements (P0589r0, P1306r0).

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

- Our preferred approach to the core language facilities, along with argumentation of why we think that design is desirable.
- 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.

A simple example

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

```
template<Enum T>
std::string to_string(T value) {  // Could also be marked constexpr
  for... (auto e : std::meta::members_of(reflexpr(T)) {
    if (unreflexpr(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 some 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 the type maintained by the compiler. The `members_of` function 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...`. This isn't true "iteration", however. The body of the statement repeated for each element of the vector so that the loop variable member 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 (unreflexpr(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 `unreflexpr` operator recovers the value of a reflected entity. 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 enshrines 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.

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:

```
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*

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 slightly different from the standard term *entity* (which, e.g., includes "values" but not "captures"). We call *alias* a namespace alias or a type alias.

Reflection type

What should the type of a reflection be? 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:

- 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.
- 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 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;
}
```

```
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.

As noted earlier, values of this type behave as handles to internal structures of the compiler that represent the operand. To reason about the kind of semantic information one can obtain through these reflection values, we categorize the values into four mutually exclusive 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.

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-entiy but does at least designate the property
              //  “prvalue of type int”.
```

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

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"
```

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.

An *invalid reflection* does not designate an entity, alias, or expression. This type of reflection is used to communicate "error situations".

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. These are described in sections below.

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`.

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). 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);
static_assert(reflexpr(g(1)) == reflexpr(g(1)); // May fail because g(1)
                                                // is an expression that is
                                                // 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 reification operators (e.g., `typename`, `unreflexpr`) or library facilities (`std::meta::entity`) described in the following sections.

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
```

It is unspecified whether reflections obtained from expressions that do not designate a declared entity compare equal. 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;
auto r1 = reflexpr(S);
struct S {};
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 predeclared:

```
namespace std::meta {
  constexpr! 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 {};
constexpr! 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 an integer 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 {};
constexpr! 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 {};
constexpr! 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();
}
```

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 operators to map reflection values back to source code constructs (the operand "*`reflection`*" below always stands for an 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.
- `unreflexpr(`*`reflection`*`)`
  If "*`reflection`*" designates a *constant expression*, this is an equivalent expression. Otherwise, if "*`reflection`*" designates a non-member-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 "`unreflexpr(`*`reflection`*`)`".

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

Examples:

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

namespace N { int f; }

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

struct S {
  constexpr! auto ri() { return reflexpr(S::i); };
private:
  int i:3; // Bit field.
} s;
int i1 = s.unreflexpr(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 list-generating variations of most of the reification constructs above. Let *`reflection_range`* be a *range* such that

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

would successively set `r` to a list 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)`
- `unreflexpr(reflection_range)...` expands to
  `unreflexpr(r1), ..., unreflexpr(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 list of namespaces can currently not appear anywhere in a C++ program.
- If any expansion produces an ill-formed reification construct, the whole construct is ill-formed but subject to SFINAE.
- This intentionally looks a lot like a pack expansion, but it is not exactly the same thing because the construct is not valid if the ellipsis does not immediately follow the reification syntax. For example:

  ```
  std::meta::info  types[] = {
                    reflexpr(int), reflexpr(double) };
  template<typename t, typename U> struct Y {};
  Y<typename(types)*...> yptrs;  // Error
  ```

  If the reification construct were able to produce a pack pattern, the latter would be equivalent to `Y<int*, double*>`. However, such a feature would be a considerable implementation burden. It is usually fairly straightforward to work around this limitation by using an additional template layer, or by using reflective functions that produce derived reflections. E.g., a call `make_pointers(types)` might produce a range of reflections similar to:

  ```
  { reflexpr(int*), reflexpr(double*) }
  ```

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

```
constexpr! 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 unreflexpr(params[0]);  // Okay
}
int g(int, char) {
  return unreflexpr(params[0]);  // Error: params[0] comes from function type
}                                // "int (int, int)" but this function has
                                 // type "int (int, char)".
```

When `unreflexpr` is applied to a reflection designating a constant expression, it is unspecified whether that constant expression is re-evaluated, or whether a memoized result value is produced. Note that this matter because this proposal includes constant operations that have effects that are context-dependent (e.g., produce different results depending on whether a type is complete or not). For example:

```
  struct S;
  constexpr r = reflexpr(std::meta::is_complete(reflexpr(S)));
  struct S {};
  constexpr bool b = unreflexpr(r); // Could be true or false.
```

Access checking

Reification constructs 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 reification constructs are not necessarily names. In fact, the only reification constructs that behave like names are the "`(. reflection .)`" and "`(. reflection_range .)...`" constructs. Those are therefore the only reification constructs subject to access checking. For example:

```
  class C {
    using Int = int;
```

```
public:
  static constexpr! auto r() { return reflexpr(Int); };
} c;
typename(C::r()) x;  // Okay: x has type int
C::(.C::r().) y;     // Error: Int is inaccessible.
```

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 {
  constexpr! 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 (as always, 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 {
  constexpr!
 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 {
  constexpr! auto current_source_file_name()->std::string {...};
  constexpr! auto current_source_line()->unsigned {...};
  constexpr! 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 currently do not rely on the previously-proposed `std::source_location` feature because is it does not use immediate functions, and it is not clear that it would do so when/if integrated into the working draft for the next standard.)

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 {
  constexpr! 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 also propose a predicate:

```
namespace std::meta {
  constexpr! 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 {
  constexpr! 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.

Expressions and entities

Consider:

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

As mentioned before, the reflection value `r` designates both the *expression* `i` and the *variable* `i`. However, the special function

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

applied to `r` produces a reflection designating just the *variable*.

More generally, `std::meta::entity` returns:

- its argument if its argument is an invalid reflection,
- or a reflection designating only a declared entity if its arguments designates a declared entity, or
- a reflection for an entity E if its argument is a reflection designating an alias of E, or
- an invalid reflection in all other cases.

Example:

```
void f();    // #1
int f(int);  // #2
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
```

To get the value of an entity, use the `unreflexpr` operator. Example:

```
unreflexpr(r)(42); // Calls f(42). No lookup of "f" is done.
(. r .)(42);   // Same as "f(42)", including lookup and overload resolution.
```

Template arguments

The proposed function

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

```
}
```

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 {
  constexpr! auto template_arguments_of(info reflection)
                                        ->std::vector<info> {...};
}
```

Conversely, the template producing a specialization can be obtained with

```
namespace std::meta {
  constexpr! 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.

A dual to the above operations is also proposed:

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

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 one of the reification primitives (`< `*`reflection`*` >`), but having both supports 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.
```

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 {
  constexpr! 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 `constexpr!` 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 {
  constexpr! 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 {
  constexpr! auto is_union(info reflection)->bool {...};

  constexpr! auto is_constructible(info reflection,
                                   std::span<info> arg_types)
                                                   ->bool {...};
  constexpr! 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 {
  constexpr! auto alignment_of(info type)->std::size_t {...};
  constexpr! auto rankinfo type)->int {...};
  constexpr! 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 (P0194r6) 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 {
  constexpr! 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 {
  constexpr! 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 {
  constexpr! auto is_scoped_enum(info entity)->bool {...};
}
```

and always evaluates to a constant.

We propose to replace `is_constexpr` by:

```
namespace std::meta {
  constexpr! 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 `constexpr!` 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 (`constexpr!`) functions and function templates are also identifiable:

```
namespace std::meta {
  constexpr! auto is_immediate(info entity)->bool {...};
}
```

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

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

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

`is_inline`:

```
namespace std::meta {
  constexpr! 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 {
  constexpr! auto is_deleted(info entity)->bool {...};
  constexpr! auto is_defaulted(info entity)->bool {...};
```

```
  constexpr! auto is_explicit(info entity)->bool {...};
  constexpr! auto is_override(info entity)->bool {...};
  constexpr! 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 {
  constexpr! auto is_class_member(info reflection)->bool {
    // Return true for class and class template members.
    ...
  };
  constexpr! auto is_local(info reflection)->bool {
    // Return true for local variables, local members.
    ...
  };
  constexpr! auto is_namespace(info entity)->bool {...};
  constexpr! auto is_template(info entity)->bool {...};
  constexpr! auto is_type(info entity)->bool {
    // Return true for types and type aliases.
    ...
  };
  constexpr! auto is_incomplete_type(info entity)->bool;
  constexpr! auto is_closure_type(info entity)->bool {...};
  constexpr! auto has_captures(info entity)->bool {...};
```

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

```
  constexpr!
  auto is_member_function_template(info entity)->bool {
    return is_function_template(entity) && is_class_member(entity);
  };
  constexpr!
  auto is_static_member_function(info entity)->bool {...};
  constexpr!
  auto is_static_member_function_template(info entity)->bool {...};
  constexpr!
  auto is_nonstatic_member_function(info entity)->bool {...};
  constexpr!
  auto is_nonstatic_member_function_template(info entity)
                                                    ->bool {...};
  constexpr! auto is_constructor(info entity)->bool {...};
  constexpr! auto is_constructor_template(info entity)->bool {...};
  constexpr! auto is_destructor(info entity)->bool {...};
  constexpr! auto is_destructor_template(info entity)->bool {...};
}
```

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 three predicates (with no equivalent in the TS) for reflections of expressions:

```
  constexpr! auto is_lvalue(info reflection)->bool;
  constexpr! auto is_xvalue(info reflection)->bool;
  constexpr! auto is_prvalue(info reflection)->bool;
  constexpr! auto is_glvalue(info reflection)->bool {
    return is_lvalue(reflection) || is_xvalue(reflection);
  }
  constexpr! 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 {
  constexpr! 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 {
  constexpr! 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 {
  constexpr! 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 {
  constexpr! auto source_line_of(info entity)->unsigned {...};
  constexpr! auto source_column_of(info entity)->unsigned {...};
  constexpr! 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 {
  constexpr! auto name_of(info entity)->std::string {...};
  constexpr! 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 .)`" reification construct. 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 {
```

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

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

```
namespace std::meta {
  constexpr! 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 {
  constexpr! 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 {
  constexpr! auto current_function()->info {...}
  constexpr! 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 {
  constexpr! auto byte_offset_of(info entity)->std::size_t {...};
  constexpr! auto bit_offset_of(info entity)->std::size_t {...};
  constexpr! auto byte_size_of(info entity)->std::size_t {...};
  constexpr! 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). (A precise specification of this requires a slight tightening of the C++ object model. All implementation already conform to the stricter model.)

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

```
namespace std::meta {
  constexpr! 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.f());
    // Reflection for type "S const".
```

Plural properties

We propose the following functions to retrieve subobject information:

```
namespace std::meta {
  constexpr! auto members_of(info class_type, auto ...filters)
                                          ->std::vector<info> {...};
  constexpr! auto bases_of(info class_type, auto ...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`).

For example:

```
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.
constexpr! 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.
```

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

```
namespace std::meta {
  constexpr!
  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 invalid reflection.

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

```
namespace std::meta {
  constexpr! 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 {
  constexpr! 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.

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 "`unreflexpr( ... )`" reification primitive:

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

Expansion statements

The facilities presented so far are quite powerful and allow us in principle to do all the things that are currently handled through pure template metaprogramming (TMP). However, one pattern that frequently occurs is the need to repeat some statements for a variety of entities described in the reflection domain.

Without an additional feature we can use template expansion techniques to achieve this, but it re-introduces some of the negatives of TMP that we're trying to avoid: poor readability and excessive usage of compiler resources. In particular, the repetition of statements requires recursively instantiated function templates. Since repetition is usually "local" to an operation, we would prefer not to generate new specializations (even with internal linkage) for each statement generated.

We therefore intend to adopt the *expansion-statement* construct proposed in P1306R0. We present a brief overview of that feature here, for completeness.

There are two basic forms for *expansion-statement*s and they are both similar to that of a range-based for loop:

```
for ... (element-declaration : expansion-initializer) statement
for constexpr (element-declaration : expansion-initializer) statement
```

In the first form (with the ellipsis), the *expansion-initializer* is an expression with one of the following properties:

- it contains unexpanded parameter packs, or
- it can appear as an initializer of a structured binding declaration.

In the second form (with constexpr), the expansion-initializer is an constant expression ([expr.const]) that can be used as the *range-for-initializer* in a range-based for loop.

In each case, the *expansion-initializer* denotes a compile-time sequence of *N* elements for which the body statement can be repeated, once for each element. Note that some expressions satisfy both properties (e.g., arrays can be destructured and iterated), hence the distinct forms.

An *expansion-statement* expands statically to the following pattern:

```
{ constexpr auto &&__range = expansion-initializer;
  constexpr auto __end = end-expr;

  constexpr auto __it_0 = begin-expr;
  <stop expansion if __it_0 == __end>
  {
    constexpr element-declaration = get-expr(__it_0)>;
    statement
  }

  constexpr auto __it_1 = next-expr(__it_0);
  <stop expansion if __it_1 == __end>
  {
    constexpr element-declaration = get-expr(__it_1)>;
    statement
  }

  // ... repeats N - 2 times
}
```

The meaning of placeholder expression *begin-expr*, *end-expr*, *get-expr*, and *next-expr* depend on the properties of the *expansion-initializer*.

For tuples the loop is computed over a compile-time index `I` that ranges from 0 to the size of the tuple.

- *begin-expr* is `0`
- *end-expr* is `std::tuple_size_v<decltype(__range)>`
- *next-expr*(`I`) is `I + 1` where `I` is the current tuple index
- *get-expr*(`I`) is `std::get<I>(__range)`

This is similarly defined for arrays of size *N*. Repetition is defined over a compile-time index `I` that ranges from 0 to the the extent of the array.

- *begin-expr* is `0`
- *end-expr* is `std::extent_v<decltype(__range)>`
- *next-expr*(`I`) is `I + 1` where `I` is the current tuple index
- *get-expr*(`I`) is `__range[I]`

For constexpr ranges, repetition is defined in terms of the a constexpr iterator `I`.

- *begin-expr* is that of the range-based for loop.
- *end-expr* is that of the range-based for loop.
- *next-expr*(`I`) is `std::next(I)`
- *get-expr*(`I`) is `*I`

(We omit the description for pack patterns and aggregate classes for brevity. They are included in P1306R0)

Note that the repeated declaration iterators is provided for exposition only. No such declaration is needed for tuples or arrays, but it *is* needed for range-based expansion.

Expansion-statements feature heavily in the examples in the following section.

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 #*).

```
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... (meta::info member : data_members)
      hash_append(h, t.unreflexpr(member));
  }
```

The algorithm is straightforward: recursively apply `hash_append` to each member for the class `T`. Within that call the expression `t.unreflexpr(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).

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<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.

```
using namespace meta = std::meta;

template<meta::info Class>
  requires meta::is_class(Class)
void create_table() {
  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... (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, 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.

Note that static 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 reflected value—a handle to an internal data structure—to runtime. In other words, for mixed runtime/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 simply 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 of reflections.

```
template<meta::info Type>
constexpr! const char* to_sql() {
  static_assert(false, “no translation to SQL”);
}
template<>
constexpr! const char* to_sql<reflexpr(int)>() {
  return “INTEGER”;
}
template<>
constexpr! 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>
constexpr! std::string create_table() {
  return create_table<reflexpr(T)>();
}
```

```
constexpr! std::string create_table() {
  std::ostringstream ss; // Assuming this should work
  ss << "CREATE TABLE " << meta::name_of(Class) << “(\n”;
  std::vector<meta::info> members = data_members.size();
  int num = 0;
  for (meta::info member : members) {
    ss << create_column(member);
    if (++num != members.size())
      ss << “,\n”;
  }
  ss << ");\n";
  return ss.str();
}

constexpr! std::string void create_column(meta::info member) {
  std::ostringstream ss;
  ss << meta::name_of(member) << " ";
  ss << to_sql(meta::type_of(member));
  return ss.str();
}

constexpr! 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 immediate function (`constexpr!`). 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 runtime and compile-time values in the implementation; everything just looks like runtime code.

Template argument list assignment

In the paper P0949R0, Peter Dimov proposes a facility to “assign” a list of template arguments for one template to another using the facility below:

`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:

```
using std::meta::info;
using std::vector;
constexpr! 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");
  }
}

constexpr! 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;
}
```

```
constexpr! 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`:

```
constexpr! 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:

```
// 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> vector<T> even_args(Ts ... p) {
  vector<T> result{};
  for constexpr (auto param: parameters_of(current_function())) {
    if (reflexpr(T) == type_of(param)) {
      result.push_back(unreflexpr(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 (P0194, or something like it).

Unfortunately, `get_all_data_members` 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.

```
using std::meta::info;
using std::vector;

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

```
// Convenience function to select nonvirtual bases and members.
constexpr! 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.
constexpr!
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... (auto member : get_members(reflexpr(T))) {
    f(p_obj->unreflexpr(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->unreflexpr(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`).

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. (We know of at least two ways to address the issue:

- 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 the “iterated expansion” feature.

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

// Convenience function to select nonvirtual bases and members.
constexpr! 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.
constexpr!
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->unreflexpr(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 that the first version, but it is nonetheless more efficient than a pure TMP-based solution.