Metaprogramming

Andrew Sutton (asutton@lock3software.com)
Document: P2237R0
Audience: SG7

1 Introduction

This is a big paper, in terms of both scale and ambition. This paper presents a direction for a comprehensive system of metaprogramming facilities in C++.

I think that we (the committee) are missing a big picture for metaprogramming, and that has hurt—and will continue to hurt—our ability to thoughtfully design and incorporate proposals into a coherent metaprogramming design. If you think of metaprogramming proposals as puzzle pieces, then we (the committee) have been trying to put these pieces together without a reference picture. That can't turn out well. We need that reference to know not only how the pieces fit together, but what they look like when combined.

The direction proposed in this paper is synthesized from a survey of various programming languages[1] and an extensive set of related C++ proposals. The reason for looking at other languages is to ensure that our terminology and ideas generally conform to common usage, and that we aren't inventing something entirely new. The purpose of surveying a large set of C++ proposals is to ensure that the reference picture is comprehensive and addresses many of the shared problems addressed the various proposals. Section XXX provides a rough roadmap from our current status to adoption.

The direction proposed in this paper does not start from a fresh slate; it stands firmly on top of the directions approved by SG7. In particular, the entire metaprogramming system builds on top of the ideas presented in P1240 [1]. However, there are some changes. In particular, some of the terminology has evolved as result of surveying other languages, and much of the syntax has definitely changed. I have also refined the semantics of several proposals, including those of P1240. This paper is explicit about what changes have been made and why.

I have two motivations for re-syntaxing proposals. First, I've come to the conclusion that the syntax of certain metaprogramming features should be sufficiently different from existing C++ notation, perhaps even rising to the level of "unfamiliar". Having looked at enough examples of static reflection [1] and source code injection [2], I've formed the opinion that those features blend in too well with the "normal code"; they hide in plain sight. That makes them hard (for me at least) to easily distinguish what parts of the code are data and which parts are control. Being able to visually differentiate those parts of a metaprogram should improve readability and maintainability. Avoiding features that encourage "write-only" metaprograms should be a priority for SG7. Second, this isn't a small language or library proposal, it's a system comprised of many language features and libraries. I want syntax and naming to be consistent across all these components. If not, we're going to end up designing Frankenstein's monster.

[1] Languages with similar compile-time programming features include Template Haskell, Julia, D, and Rust.

Drawing an initial reference picture with placeholder syntax is no longer satisfying to me. As a result, I've chosen notation that (I think) achieves my goals.

2 Background

Same-language metaprogramming has a long history in C++, arguably beginning in 1994 with a demonstration of how templates can be used to compute prime numbers at compile time [3]rapidly became a new programming tool generic libraries [4]. The ability to use template instantiation for computation enabled a wide range of new programming techniques and facilities, including formalization of type traits to support generic programming, and type-based optimization [5, 6]. However, template metaprograms are notoriously expert only, as they typically require a deep knowledge of the language to design and maintain them.

In 2007, N2235 formalized and extended constant folding, including the ability to evaluate simple (one-line) functions at compile-time (i.e., `constexpr`) [7]. Restrictions on what can be evaluated have been relaxed over the years, making more of the language available for constant evaluation [8]. Constant expressions make it significantly easier to write and maintain compile-time metaprograms.

As of 2020, we have a rich language for composing parts of programs at compile time. Constant expression evaluation provides a basis for working with static data and computing values, while template metaprogramming provides a facility for logical computation involving types (as opposed to values) and stamping out new functions and classes via instantiation.

As powerful as the language already is, these facilities still don't fully satisfy our requirements for metaprogramming.[2] In the mid-2010s, work began on a language extension for static reflection, ultimately resulting in a Technical Specification [9]. Although the technical underpinnings of that work have changed (reflecting values instead of types), the overall intent has remained the same. In 2016 and 2017, work on source code injection was also developed as a mechanism to support metaclasses [10, 2].

Many more proposals have been published since that work began. Based on surveyed papers, I see the following broad categories of use cases for extended metaprogramming support in C++.

- **Define structural algorithms.** The ability to inspect class structure lets us design and optimize generic algorithms based on class structure, making it easier to automatically provide common facilities such as serialization and hashing.
- **Reduce boilerplate.** Synthesizing and injecting source code can reduce the amount of code users need to write to interoperate with frameworks. For example, a library could provide tools to generate marking operations for a simple, opt-in garbage collector.
- **Simplify tool chains.** Comprehensive support for metaprogramming can eliminate the need for many build tools such as preprocessors. The ability to incorporate external specifications into a program and the ability to output data from our own code can obviate the need for specialized tools.
- **Improve user experience.** We can provide language and library support that make compile-time programming easier via better notation and e.g., compile-time tracing and assertions.

[2] If templates and constant expression evaluation did satisfy all our metaprogramming requirements, we (the committee) wouldn't have much need for SG7.

- **Evolve the language.** Metaprogramming may allow more features to be developed (or at least prototyped) as libraries rather than pure language features.

The features discussed throughout this paper address these use cases.

3 A system for metaprogramming

A *metaprogramming system* is a set of related language and library features that provide facilities for the definition, analysis, transformation, and use of compile-time data, including the controlled synthesis and injection of source code into the current translation unit.

The fact that metaprogramming features are related should not be overlooked or understated. We could define sets of unrelated or loosely related features for metaprogramming. However, doing so would almost certainly result in yet another Turing-complete sublanguage of C++ with a completely novel syntax. The macro language of Rust shows us how this might look if we took a similar approach in C++. Instead, we should prefer to build new features on top of existing or lower-level facilities.

This system is a layered system where each potential feature builds on the facilities provided by the layers below. While features on the interior of the system provide the low-level machinery for metaprogramming, features on the outside provide enhanced tools for library and framework developers as well as (I hope) an improved user experience.

The foundation of that system is constant expression evaluation and templates. Constant expressions allow the compile-time evaluation of functions to analyze and manipulate compile-time data, while templates support the controlled synthesis of new declarations and their injection into the current translation unit. Both are currently being extended to provide more extensive support for metaprogramming. Extensions to both constant expression evaluation and templates will directly support some features described in this document. These are discussed in Sections 4 and 5, respectively.

The core of the metaprogramming system is static reflection (Section 6), which provides the ability to inspect source code constructs and generate references, while source code injection (Section 7) allows various program constructs to be synthesized as data and then injected into the program. If constant expressions and templates define a basic computer for metaprogramming, then static reflection and source code injection are its assembly language. They define the primitives that allow for the programmatic construction of classes, functions, libraries, and frameworks.

On top of the metaprogramming core, we can provide many new features that simplify repetitive tasks or common programming idioms. These features are essentially syntactic sugar, albeit with sometimes complex rewrite rules. Examples include:

- Syntactic macros[3] to encapsulate idiomatic patterns
- Mixins for improved compositional design
- Metadata annotations as input to metaprogramming frameworks
- Function decorators to supplement or modify function behavior
- Metaclasses to supplement or modify class structure

[3] I would like to believe that I am the first person to describe macros as “syntactic sugar” instead of “core functionality”.

All these features are rooted in syntactic rewrites in terms of static reflection and source code injection.

The ability to create program elements incrementally supports more advanced forms of metaprogramming. In particular, this opens the door to defining functional transforms: metaprograms that generate new functions (or other declarations) based on the definition of their input (Section 8.4). Example use cases include:

- Automatically currying functions
- Automatic differentiation
- Other symbolic transformations and analyses

These use cases rely on deep inspection of function definitions (i.e., statements and expressions) to analyze the symbolic structure of functions, and to incrementally construct new functions or data based on those analyses. Continuing with the computer system analogy, this turns the metaprogramming system into a compiler.

Almost all computer systems need input and output. This is true for a metaprogramming system as well. There have been several proposals suggesting adding various forms of compile-time input and output.

- Compile-time tracing to improve compile-time debugging
- User-supplied errors and warnings to improve diagnostics
- Embedding data to insert raw data into programs
- Reading static configuration data at compile-time
- Generating supplemental compiler output for language bindings

Facilities for Compile-time I/O are discussed in Section 9.

Finally, the scope of these facilities is clearly not limited to compile-time programming. In particular, we want to allow runtime usage of our core facilities (Section 10) to enable use cases such as:

- Runtime inspection of objects
- Just-in time compilation

Runtime metaprogramming allows for more dynamic functionality in applications such as directly supporting bindings to dynamic languages like Python, and potentially self-optimizing programs via JIT compilation.

4 Constant expressions

Constant expressions (and their evaluation) provides the ability to write compile-time programs that analyze, transform, and use compile-time data. Importantly, they provide that facility using familiar syntax: functions, statements, and expressions.[4]

Constant expressions were initially tightly specified. `constexpr` functions were restricted to having bodies containing only a `return` statement and had reasonably strict limitations on the kinds of

[4] This contrasts with template metaprogramming, which specifies computation in terms of template instantiation.

expressions that could be used [7]. Over time, these restrictions have been relaxed, allowing nearly the entirety of C++ to be usable (or at least writable) in `constexpr` functions [8, 11].

This section describes some new features related to constant expression that impact metaprogramming.

4.1 Metaprograms

P0712 introduced the ability to write code that executes where it appears in the program [12]. Over time, this evolved in into a metaprogram declaration, or simply *metaprogram* [2].

```
consteval {
  for (int i = 0; i < 10; ++i)
    generate_some_code(i);
}
```

Presumably, `generate_some_code` is a `consteval` function that injects source code into the translation unit where this metaprogram appears. A metaprogram can be thought of as an unnamed `consteval` function that is called as the initializer of an unnamed variable.

Metaprogram declarations were initially designed as bootstrapping mechanism for executing functions which would synthesize and inject new source code into a translation unit. Over time, the need to positionally execute compile-time code been relaxed by the introduction of immediate functions [13], splicing (Section 6.3), and various forms of injection (Section 7).

4.2 Compile-time side effects

P0596 presents features that would allow for compile-time side effects in constant expressions, including compile-time output and mutable compile-time variables [14]. The problem is interesting because it requires evaluation that can modify the state of translation.

Runtime side effects are limited to a few different kinds: modifying an object, reading from a volatile object, and performing I/O are considered side effects. At compile-time we have different kinds of possible side effects:

- implicitly instantiating a template (Section 6),
- injecting source code (Section 7),
- modifying a compile-time only object (as in P0596),
- generating compiler diagnostics (as in P0596),
- or performing some other kind of compile-time I/O (Section 9).

P0992 describes a conceptual model for how translation and compile-time evaluation interact [15]. This is a useful tool for understanding how side-effects can be incorporated into that model. In an extremely literal interpretation of that model, we can interpret a request for constant expression evaluation as requiring the compiler to emit an entirely new program containing all the functions and variables needed to compute a constant expression. The compiler then compiles the generated program, to be executed on a robust implementation of the abstract machine, namely one that can detect all forms of undefined behavior and is presumably well-instrumented with debugging information. The resulting binary is then executed. Arguments to the constant expression can be passed via piped input (serialized as text) while output and errors can be piped back to the compiler (deserialized from text). The resulting value is then incorporated into the translation (e.g., as the size

of an array), or in the case of errors, transformed into an appropriate diagnostic. This model cleanly separates the semantic analysis subsystem of the compiler (e.g., name lookup, template instantiation, etc.) from the evaluation facilities.[5]

This model is well-suited for managing compile-time side effects as proposed in P0596. In particular, during (possibly tentative) evaluation, side effects can be managed transactionally, meaning they are requested during evaluation, but only committed (or aborted) after the evaluation completes.

In the P0992 model, side effects are simply transmitted back to the compiler (e.g., as a serialized sequence of effects) along with the return value or error. These effects can then be processed and applied to the translation unit, or they can be diagnosed and discarded based on the result of evaluation.

The case of mutable compile-time variables is interesting. A metaprogram may modify them several times during a single evaluation, but the final value is not available to the compiler until the evaluation completes.

```
consteval int counter = 42;
consteval {
  for (int i = 0; i < 5; ++i)
    ++counter; // normal effect
} // compile-time effect: counter = 47
int arr[counter] // counter == 47
```

In other words, modifying a compile-time variable during constant expression evaluation is a normal side effect during that evaluation. However, that it was modified at all is a compile-time side effect.

5 Templates

Templates provide a facility for injecting new declarations into source code, albeit in a somewhat constrained way. Each instantiated declaration names a specialization of a class template, function template, or variable template for some template arguments. We can't use this feature to insert e.g., members into a class, statements into a function, etc.

Variadic templates, or more precisely, parameter packs, provide another kind of injection facility. They allow the insertion of sequences of template and function arguments into contexts where such lists are allowed (e.g., template and function argument lists, base specifier lists, member initializer lists, etc.).

This section discusses current in-flight proposals to extend these features and how they relate to the various metaprogramming features described in Section 2.

5.1 Structured binding packs

There have been several proposals to extend parameter packs to work more generally by e.g., allowing indexing and declaring packs. The most comprehensive current proposal for extending

[5] This model was inspired by Clang's design, which architecturally separates constant expression evaluation from semantic analysis. It is practically impossible for the `constexpr` evaluator to perform semantic analysis without violating the physical architecture of the project.

parameter packs is P1858 [16]. Of special interest, is the ability to expand a type into pack, first introduced in P1061 [17]:

```
auto [...pack] = make_tuple(1, 2, 3);
```

This declares `pack` as a *structured binding pack*, which should be usable and expandable wherever a function argument pack can be used.

```
eat(pack...) // expands into function arguments
```

Although P1061 does not explicitly say so, we note here that structured binding packs are dependent in a slightly novel way. In particular, expressions involving structured binding packs are like type-dependent expressions in that some semantic analysis is deferred until the point of expansion. However, they are unlike type-dependent expressions in the sense that expansion of structured binding packs can occur at parse-time, as opposed to instantiation time as with type-dependent terms. That is, `pack...` is expanded immediately when `pack` is initialized to something concrete. If `pack...` is type-dependent, then no expansion occurs until template instantiation.

What we need to make this work is essentially something parallel to type dependence for these kinds of packs. For example, we might describe the variable pack as having *pack type*, which would be similar to having dependent type, except that its initializer is non-dependent. That is, we know how to expand `pack`. Similarly, the use of `pack` in `pack...` is *pack-dependent*, meaning that it refers to a declaration with pack type.

The ability to create structured argument packs plays an important role in splicing ranges of reflections (Section 6). We should be able to expand a range (as in concept) of reflections into references to the reflected constructs they designate (Section 6.3.8). That feature builds on the semantics described in P1061.

5.2 Expansion statements

The primary motivation for expansion statements was to simplify programming with heterogenous data types structures (i.e., tuples) [18]. They provide a control-like structure that allows the repeated synthesis of statements within a function. For example, we can easily print the elements of a tuple.

```
template<OutputStreamable... Args>
void print(const std::tuple<Args...>& tup) {
  template for (const auto& x : tup)
    std::cout << x << ‘ ‘;
  std::cout << ‘\n’;
}
```

The loop variable can also be (meaningfully) declared `constexpr`, making it a constant expression in the body of the loop.

We can also use expansion statements to easily optimize `std::visit` for variants by directly generating a switch statement instead of resorting to more complex template metaprogramming.

```
template <typename F, typename V>
decltype(auto) visit(F f, V const& v) {
  constexpr size_t n = variant_size_v<remove_cvref_t<V>>;
  switch (v.index()) {
```

```
    template for (constexpr int i : ints(0, n)) {
      case i:
        return invoke(f, std::get<i>(v));
    }
    default:
      unreachable();
    }
}
```

This same technique also allows us to encapsulate and generalize Duff's device [19] as a generic algorithm. The implementation is left as an exercise to the reader.

Although the original motivation for this feature no longer exists, expansion statements are still an extremely useful tool as fundamental building blocks of algorithms on heterogenous data types: they directly express traversal of e.g., tuples. The ability to make the loop variable `constexpr` is also directly usable by metaprograms that need to splice elements of a sequence into a function body (Section 6.3).

Expansion statements were approved by EWG for inclusion in C++20 very late in the standardization cycle and the clock ran out before the wording could be finished. The proposal has not yet been revived for C++23. The resurrection of the proposal needs to clarify the semantics of range traversal and should introduce support for `break` and `continue`. We should also extend structured bindings so they can decompose a `constexpr` range. This last feature requires adoption of P1481 [20].

5.3 Template function parameters

The ability to pass function arguments as constant expressions is particularly useful for certain abstractions such as providing [] operators for tuples. P1045 introduces the ability to declare `constexpr` function parameters which can be used for that purpose [21]. However, some members of the committee, myself included, prefer to use the `template` keyword over `constexpr` to introduce such parameters (hence the section name "template function parameters"). Otherwise, the idea is straightforward:

```
template<typename... Args>
class tuple
{
  auto& operator[](template int n) {
    return impl.get<n>(*this);
  }
};
```

Within this function, `n` is passed as a template argument instead of a normal function argument, meaning that it can affect both the signature and definition of the function. Here, both depend on the value provided.

Calling the function works as one might expect:

```
tuple<int, char, bool> tup;
tup[0] = 42;
tup[1] = 'a';
```

```
tup[2] = false;
```

The argument is provided in its usual position and substituted through the definition as a template argument.

This feature potentially helps resolve some of the tensions in the design of the reflection library. While this paper is based on the POSIX-like file-descriptor approach in P1240 [1], wrapping that with a strongly typed API [22] could improve usability and support common C++ design/programming techniques (e.g., function overloading). However, directly layering a strongly typed API on top of P1240 requires the ability to constrain overloads on the value of function arguments. P1733 [23] described a mechanism by which requires-clauses could be made to check the values of function arguments. This was later extended in P2049 [24] before both papers were sidelined to investigate whether template function parameters could solve the same problem. They mostly can.

This feature lets us define constructors with template function parameters and constrain their arguments.

```
struct class_info : type_info {
  class_info(template  info x) requires is_class(x)
    : type_info(x)
  { }

  // class-specific api
  // ...
};
```

This definition lets us diagnose errors in the initialization of reflections at the point of use.

```
meta::class_info ci = reflexpr(int); // error: reflexpr(int) does not
                                     // reflect a class
```

Ideally, this should also support the ability to define APIs in terms of overloaded functions.

```
consteval void print(type_info x);
consteval void print(class_info x);
consteval void print(namespace_info x);

// elsewhere
struct S { };
print(reflexpr(S)); // should call the class_info overload
```

Unfortunately, this doesn't quite work because overload resolution won't rank implicit conversion sequences between different classes, even when those classes can be partially ordered by inheritance. More work is needed, either on the design of the library or language features needed to make the library work.

This feature also relates to macros (Section 8.2), which introduces "reflection parameters," allowing arguments to be passed by reflection.

6 Static reflection

The overall direction, motivation, and use cases for static reflection are set by P0385 [25] and its preceding publications, starting with N3996 [26] and ultimately leading to the Reflection TS [9]. The design originates with Matúš Chochlík's work on his Mirror library [27], which uses types as handles to data describing C++ declarations. More generally, the term "mirror" describes an approach to reflection where program elements (e.g., declarations) are reified as existing first-class entities of the language [28].[6] In the Reflection TS, program elements are reified as types, called metaobjects, which makes them eligible to be used as data in template metaprograms.

However, the reification of source code constructs as types is not scalable; every metaobject becomes a permanent member of translation since types are "permanent"—they never go away. Extensive use of the facility will lead to large increases in the working set of a compiler and untenably high compile times.

Both P0953 [22] and P1240 [1] propose value-based approaches to static reflection. P0953 translates the design of the Reflection TS into an object-oriented style class hierarchy where reflections of program elements are represented as objects in that hierarchy. P1240 prefers to represent reflections as uniformly typed scalar values not unlike POSIX file descriptors. In both cases, the value of those objects are handles to the compiler's internal representation of the reflected program element, and a substantial library component provides queries for the various properties of reflections (e.g., types of declarations, names of variables, etc.).

Our preference is to base static reflection on P1240 as it provides a more efficient (faster compile-times) and flexible low-level facility for reflecting values. It should be possible to implement the more strongly typed design of P0953 on top of the lower-level facilities of P1240, possibly with the help of `constexpr` function parameters.

That said, some aspects of the design in P1240 should be revised.

The term "reification" as it is used in P1240 is unfortunately ambiguous. To reify something means to make something concrete, or in the context of programming languages to make something a "first class citizen". The reification operators in P1240 take reflection values and produce syntactic constructs. As I've come to see things, I think of the compile-time objects and values used by metaprograms as first-class citizens, not the syntax, so of course the term is misused. On the other hand, it is entirely reasonable to see the resulting syntax as the thing made concrete by a metaprogram, so the term makes sense. Given that, I think it is best to avoid the term.

The syntax of those operators has been criticized as being inconsistent with other names chosen by the language [29]. This seems like a good opportunity to address those issues.

What was called "reification" in P1240 is called "splicing" in this document. The term "splice" is used to describe the insertion of syntax in Haskell and Julia (with respect to macros), so there is at least some community use of the term to describe related operations. We also redesigned the syntax of these operators to consistently use the syntactic pattern `|x|` to denote the splice of a reflection into the program.

[6] In this approach the mirror is independent of the thing it reflects. Said otherwise, a mirror holds a reflection.

Finally, I intensely dislike the name `reflexpr`, and I am not the only one [30]. In an early draft of this paper, I had renamed the operator to `reify` because that fits my understanding of its behavior. It is worth noting that Template Haskell also includes a `reify` function that takes a `Name` and returns an `Info` value, so at least there is existing practice for the name. Unfortunately, some respondents disagreed (see the discussion above), so I've been left to consider alternatives.

For now, I will continue using `reflexpr`, but I would very much like something different. I suspect that I might prefer a syntactic operator over a name.

Here is a simple example, adapted from P1240, and updated with new syntax.

```
template<enumeral E>
const char* to_string(E val) {
  constexpr meta::info t = reflexpr(E);
  template for (constexpr meta::info x : meta::members_of(t)) {
    if (|x| == val)
      return std::display_name_of(x);
  }
  return "<unknown>";
}
```

As in P1240, `reflexpr` takes a name or other construct and yields *constant reflection*, a constant expression whose type is `meta::info`. The value of that expression is an opaque reference to the compiler's internal representation of the enumeration type. We assign that to the `constexpr` variable `t`. In general, reflections must either be constant expressions or only used during constant expression evaluation. Reflections should not leak into runtime code. The reflection operator and queryable properties are described in Section 6.2.

The `meta::members_of` function returns a range over `E`'s enumerators, each element of which is also reflected as a `meta::info` value. Here, we use an expansion statement because we need the value of `x` to be a constant expression within the body of the loop.

The expression `|x|` is a called a *splice*. It is replaced by an expression naming the entity designated by `x`, in this case the corresponding enumerator. The operand of a splice operator must be a constant reflection. This operator replaces the `idexpr` operator in P1240 (previously called `unreflexpr`). The reason for choosing a new notation is to find a uniform notation for inserting "snippets of code" into various program constructs. This concept is discussed in more detail in Section 6.3. A related feature, injection, is discussed in Section 7.

6.1 Examples

This section presents more advanced examples of reflection and splicing.

6.1.1 Structural copy

It's possible to use reflection to perform "structural" copies between record types. A structural copy copies the data members of one class into correspondingly-named members of a different class. A simple version of this algorithm is surprisingly easy to write:

```
template<class_type T, structural_subtype_of<T> U>
void structural_copy(const T& src, U& dst) {
```

```
  constexpr auto members = meta::data_members_of(reflexpr(src));
  template for (constexpr meta::info a : members) {
    constexpr meta::info b = meta::lookup(dst, meta::name_of(a));
    dst.|b| = src.|a|;
  }
}
```

The `structural_subtype_of` concept is satisfied when its second operand has at least the same data members (name and type) as its first data member. In this case, `U` must be a structural subtype of `T`. The implementation is left to the reader as an exercise.[7]

Structural copies and moves are especially useful in data access frameworks, where queries return small structs whose contents are copied (or moved) into business objects.

6.2 Reflection

A major component of static reflection is the ability to inspect the properties of source code constructs. This is done using the `reflexpr` operator and an extensive library of queries for the various properties of entities, expressions, declarations, etc. The `reflexpr` operator does exactly what it says: it takes a construct and yields an object that can be queried and manipulated programmatically.

When we say that the `reflexpr` operator takes a "name or other construct," we mean that it accepts one of the following as an operand:

- An *expression*
- A *type-id*
- A possibly qualified *template-name*
- A possibly qualified *namespace-name*
- The token `::`

The `meta::info` object produced by the `reflexpr` operator *reflects* the construct given as an operand. We also say that the object holds a reflection value, or more simply that it holds a reflection. There are two aspects of every reflection: its syntax and its semantics.

The syntactic aspect of a reflection allows queries about form of the reflected construct in source code. This allows for queries for a construct's source location, the declaration of a *type-name*, a type alias' definition, etc. Notably, library functions operating on the syntactic aspect of reflections support the programmatic traversal of the program's elements and structure.

The semantic aspect of a reflection includes analyzed properties of a construct and its corresponding entity, if any. This allows for queries about the kind of an entity, an expression's type and value category, the size of type, and the (external) name of an entity, etc. Library functions on the semantic aspect tend to deal directly with what the construct *means* rather than how it is written.

For example, consider the following:

```
using uint = unsigned int;
```

[7] This concept can be relaxed to require convertibility of source data members to destination data members.

```
constexpr meta::info t = reflexpr(uint);
```

The value of `t` reflects the *type-id* `uint`. We can query the following syntactic properties:

- `meta::location_of(t)` is line 2, column 38 (if supported).
- `meta::is_declared(t)` is true.
- `meta::declaration_of(t)` reflects the *alias-declaration* declaring `uint`.

The last operation lets us navigate to the declaration of `uint`. Suppose we add this to our code.

```
constexpr meta::info d = meta::declaration_of(t);
```

Now we can query the following:

- `meta::location_of(d)` is line 1, column 7 (required).[8]
- `meta::is_declaration(d)` is true.
- `meta::definition_of(d)` reflects the *defining-type-id* `unsigned int`.

Now we can add the following to our code to inspect the definition of the alias `uint`.

```
constexpr meta::info x = meta::definition_of(d);
```

If supported, we could query `x` for the location of the *type-id*, which may be interesting to some metaprograms. More likely, a metaprogram is interested in the semantic properties of that type. Those include:

- `meta::size_of(x)` is likely 4.
- `meta::align_of(x)` is likely 4.
- `meta::is_fundamental_type(x)` is true.
- `meta::is_integer_type(x)` is true.
- `meta::is_unsigned_type(x)` is true.

Obviously, we should not expect users to fully navigate the program's structure just to determine whether a type is an `unsigned int` or not. As a shortcut, these properties can be queried directly of the reflection `t` and yield the same results. In fact, all three reflections `t`, `d`, and `x` designate the same type: `unsigned int`. Every semantic library operation on these values yields the same results. In other words:

- `meta::is_unsigned_type(t)` is true.
- `meta::is_unsigned_type(d)` is true.
- `meta::is_unsigned_type(x)` is true.

The same is true for all the other semantic properties listed above. All reflections denoting the same entity are equivalent since all semantic properties are the same.

To be consistent, we also rely on this property to provide a definition of equality for reflections: two reflections are equal if and only if they designate the same entity. That is:

[8] This is the location of the declared identifier, not the start of the declaration. Implementations may choose different locations.

```
t == d && d == x
```

Note that splicing (Section 6.3) any of these reflections into a program will yield the same *type-id*.

```
|t| // unsigned int
|d| // unsigned int
|x| // unsigned int
```

These values are all equal because all three reflections designate the same entity, even though they appear at different places in the program.

Precisely what syntactic properties can be reflected is difficult to define because implementations represent syntax so differently. For example, many compilers do not generally record locations of *type-specifier*s or *type-id*s in their ASTs, preferring to directly link to a representation of an entity. Other compilers provide a greater degree of source code fidelity. Some compilers preserve each declaration in a translation unit, while others merge redeclarations. Finding an API that supports the greatest utility across a diverse set of implementation strategies is challenging. Minimally, we expect the following:

- An implementation must be able to navigate from a construct's use to its declaration (if any).
- An implementation must be able navigate to the definition of a declaration (if provided).
- Only one declaration of any entity is reachable during the traversal of a namespace.
- All semantic properties must be queryable.
- Source locations are only required for declarations.

This should guarantee that metaprograms within the same translation unit, executed by different compilers, have equivalent behaviors modulo the emission of compile-time diagnostics that use the source locations of non-declarative constructs.

6.2.1 Reflecting types

For the most part, the previous section tells us all we need to know about reflecting types. Note that not all compilers preserve the source locations of *type-id*s or the constructs that comprise them. The semantic properties of types generally correspond to type traits defined in the standard library.

Cv-qualified types and reference types are interesting because many libraries simply "see through" them. For example:

```
meta::size_of(reflexpr(const int&)) == meta::size_of(reflexpr(int))
```

This is consistent with the library definition of type traits—it also happens to make perfect sense.

Depending on the language's treatment of compile-time side effects (Section 4.2), some concepts (or type traits) may not be implementable via reflection queries. The concept `std::convertible` is a prime example of such a concept. The reason for this is that computing the result requires lookup, which can result in template instantiations (i.e., side effects). Thus far, compile-time side effects have been considered transactional: they are committed or aborted after an evaluation has finished. In order to implement `std::convertible` using reflection, instantiations would have to be observable *during* the computation, which implies that there is no clean separation between evaluation and translation [15].

6.2.2 Reflecting expressions

The `reflexpr` operator accepts an *expression* as an operand. The reason that P1240 allows the reflection of expressions is twofold. First, it generalizes the syntax of the Reflection TS, which requires call syntax to reflect overloaded names. Second, it aimed to be more consistent with the idea that we could reflect expressions that denote entities. While an expression specifies a computation, it might also denote a (compile-time) value. Because values are entities, we should be able to access the value of constant expressions.

There are few syntactic properties of expressions, namely their location and form. Whether the form of an expression can be queried is an open-ended design question. This essentially opens the door for tree-based traversal of expressions, which is a requirement for e.g., automatic currying and automatic differentiation (Section 8.7). This is also important for macros that operate on syntax (Section 8.1).

For any expression, we can query the following semantic properties:

- `meta::type_of(x)` yields the type of x.
- `meta::category_of(x)` yields the value category of x.
- `meta::is_declared(x)` is true if x denotes an entity.
- `meta::declaration_of(x)` yields the declaration of `x` if and only if `x` denotes an entity.

For reflections of core constant expressions, we can access their value by splicing (Section 6.3.3) or injecting (Section 7.2.4) them into the program.

Minimally, we must be able to reflect expressions that name enumerators, variables, functions, class members, and bit-fields. For example:

```
void f() { }
constexpr meta::info e = reflexpr(f);
```

Here, `e` reflects the *id-expression* `f`. We can use the library to query syntactic properties of the reflection.

- `meta::location_of(e)` is line 1, column 38.
- `meta::type_of(e)` reflects the type `void(*)()`.
- `meta::category_of(e)` is `lvalue`.
- `meta::is_declared(e)` is true.
- `meta::declaration_of(e)` reflects the *simple-declaration* declaring `f`.

The last operation gives us a way of navigating to the declaration of `f` from its use. Suppose we add this to our code to further inspect the declaration of the function `f`.

```
constexpr meta::info d = meta::declaration_of(e);
```

We can further query that declaration for its syntactic properties, return type, and potentially its definition.

- `meta::location_of(d)` is line 1, column 6.
- `meta::type_of(d)` reflects the function type `void()`.
- `meta::has_external_linkage(d)` is true.

- `meta::is_inline(d)` is false.
- `meta::is_constexpr(d)` is false.
- `meta::is_non_throwing(d)` is false.
- `meta::is_defined(d)` is true.

As with types, the reflections `e` and `d` compare equal because they both denote (or declare) the same entity. As a result, all the semantic queries above applied to `e` will yield the same results.

Whether we can reflect the definition of a function is an open question. To do so, we would also need to reflect statements.

6.2.2.1 Reflecting class members

A *qualified-id* used as the operand to `reflexpr` can select both static and non-static data members, including bitfields.

```
struct s {
  static int x;
  unsigned n : 12;
};
constexpr meta::info x = reflexpr(s::x);
constexpr meta::info n = reflexpr(s::n);
```

The syntactic and semantic properties of each should be obvious. For example:

- `meta::location_of(n)` is line 3, column 12.
- `meta::type_of(n)` reflects the type `unsigned int`.
- `meta::width_of(n)` is twelve.
- `meta::width_specifier_of(n)` reflects the *literal* 12.

Whether the last operation is supportable by all implementations is unknown.

6.2.2.2 Reflecting overloaded functions

When an *id-expression* denotes an overload set, we have two options:

- make the program ill-formed because we are not doing overload resolution, or
- explicitly support the ability to reflect on overloaded names.

The approach taken in the Reflection TS [9] and P1240 [1] makes the program ill-formed. However, supporting the reflection of overloaded names is not inherently a bad idea; it just lacks concrete use cases.

An alternative approach is to support the reflection of overloaded functions via call expressions. Here, the reflection operand is written as a call expression and the declaration reflected is the one selected by overload resolution [31]. This is the approach taken in the Reflection TS. For example:

```
void f(int n) { }
void f(double n) { }
constexpr meta::info e = reflexpr(f(0));
```

Here, `e` reflects the *postfix-expression* `f(0)`. Some syntactic properties of `e` are:

- `meta::is_declared(e)` is true
- `meta::declaration_of(e)` is a reflection of the *declaration* `f(int)`.

The entity denoted by the call expression is the function `f(int)`. Semantic properties of `e` are those of that function.

The fact that `reflexpr` accepts expressions as operands means that this naturally works for overloaded operators:[9]

```
struct s { };
s operator+(s, s);
constexpr meta::info refl = reflexpr(s{} + s{});
```

Here, the result of overload resolution is `operator+(s, s)`, which is the declaration referred to by expression.

6.2.3 Reflecting namespaces

A reflection of the global namespace can be acquired using `reflexpr(::)`. This allows a metaprogram to traverse the entire structure of a translation unit, if desired.

As with the Reflection TS, P1240 allows the reflection of namespaces. Unlike the reflection TS, we also permit the traversal of namespace members. The traversal visits exactly one declaration of each entity across all visible partitions of the namespace exactly once. Note that we can't directly reflect an unnamed namespace (they can't be named), but we can discover them by traversing the members of their enclosing namespace.

6.2.4 Reflecting concepts

Both the Reflection TS and P1240 are silent on the notion of reflecting concepts. Presumably, this is valid since a concept is a *template-declaration*. However, the library will need to be extended with capabilities to query properties of concepts. Of particular interest is whether we permit the reflection of a concept's normalized constraints. Combined with the ability to traverse expressions, this would allow a metaprogram to inspect the individual requirements of a concept. This is an open issue.

6.2.5 Reflecting modules

Both the Reflection TS and P1240 are silent on the concept of reflecting modules. However, rebasing both papers onto C++20 means that we must consider whether modules can be reflected, and if so, what properties can be derived. It may be desirable to support searching imported modules for exported names and their reachable semantic properties. However, no concrete use cases have been suggested. This is an open issue.

6.3 Splicing

We know that we can splice references (*id-expressions*) to variables, enumerators, and functions into a program using the splice operator `|x|`. However, we can also splice *type-ids*, *nested-name-specifiers*, and *qualified-namespace-specifiers*, all using the same notation.

---

[9] The Reflection TS only allowed syntax for function calls, including operator calls. In P1240, we generalized that to support a broader range of expression reflections, and the ability to use "natural" syntax to select amongst overloaded functions falls out of that generalization.

The splice operator takes a constant reflection as an operand and synthesizes syntax that refers to or denotes the kind of entity (or expression) being spliced. For example:

```
int x = 42;
constexpr meta::info d = reflexpr(x);
|meta::type_of(d)| y = |meta::initializer_of(d)|;
```

In this code:

- `|type_of(d)|` generates the *type-id* `int`.
- `|initializer_of(d)|` generates the *constant-expression* 42.

The result of compiling line 3 is this (hopefully unsurprisingly):

```
int y = 42;
```

Note that one thing splices cannot do is to synthesize new declarations. Splices are referential. The injection operator described in Section 7.2 is can be used to create new declarations.

The following sections discuss various properties of the splice operator.

6.3.1 Splicing equations

The splicing operators and the `reflexpr` operator have a very simple relationship. For any term `x` that can be used as an operand to the `reflexpr` operator, if the splice of that reflection is valid, then the following is true:

```
|reflexpr(x)| <~> x
```

I am using the `<~>` operator to denote the functional equivalence of the two terms. In other words, splicing a reflected construct is exactly as if that construct had been written by hand. For example:

```
|reflexpr(3 + 4)| <~> 3 + 4
|reflexpr(const int*)| <~> const int*
|reflexpr(std::pair)| <~> std::pair
|reflexpr(std)| <~> std
```

Obviously, not every splice is valid in the context where it appears. For example:

```
3 + |reflexpr(std)| // error: a namespace-name is not an expression
```

6.3.2 Splicing types

Splicing a reflection of a *type-id* essentially replaces the splice with a copy of the *type-id*. For example:

```
constexpr meta::info t = reflexpr(const int);
|t| x = 42; // x has type const int
```

A type can be spliced wherever a type can be used in C++, including base class specifiers, new expressions, names of destructors, etc.

6.3.3 Splicing expressions

Splicing a reflection of an expression essentially replaces the splice with a copy of the expression. For many kinds of expressions, especially constant expressions, this is straightforward. However, it is not possible to splice an expression reflection whose expression uses names that are not visible in the

scope where the splice appears. This means that, while it's possible to return "dangling reflections," it isn't possible to use them in a way where their names could be re-bound and acquire different meaning.

```
consteval meta::info f() {
  int x;
  return reflexpr(x);
}

void g(int x) {
  constexpr meta::info r1 = reflexpr(x);
  cout << |r1|; // prints the value of x
  cout << |f()|; // error: f::x is not visible here
};
```

This restriction primarily applies to local variables, function parameters, and template parameters. We can form valid references to most other names.

6.3.4 Splicing declarations

Splicing a declaration (even of an alias) inserts an *id-expression*, *type-id*, *template-name* or *namespace-name* that refers to that declaration. The inserted reference is not subject to additional analyses. That is, no additional lookup or access checking is performed on the expression.

```
int x = 42;
constexpr meta::info d = meta::declaration_of(reflexpr(x));
cout << |d|; // prints 42
```

The splice `|d|` inserts the *id-expression* `x`, which can only refer to the declaration on the first line.

6.3.5 Splicing members

Reflections of class members can be spliced into member access expressions:

```
struct S {
  int a;
};
constexpr meta::info r = reflexpr(S::a);

void f() {
  S x;
  cout << x.|r|; // prints the value of x.a
}
```

The ability to splice members is useful for e.g., computing hash values of simple structures:

```
template<hash_algorithm& H, trivial_class T>
void hash_append(H& hasher, const T& obj) {
  constexpr auto members = meta::data_members_of(reflexpr(T));
  template for (constexpr meta::info mem : members)
    hash_append(hasher, obj.|mem|);
}
```

Outside of a member access, the splice of a class member yields a pointer to that member.

```
void f() {
  S* p = new S;
  cout p->*|r|;
}
```

The hashing operation above can also be written using the pointer-to-member access operator.

6.3.6 Splicing qualified names

Splices can occur in *nested-name-specifier*s. This gives us the ability to treat scopes as first-class citizens.

```
namespace very::long_::name::space {
  struct X { };
  struct Y { };
};

contexpr meta::info ns = reflexpr(very::long_::name::space);
|ns|::X var;
```

The *nested-name-specifier* `|ns|::` splices a *nested-name-specifier* for its designated namespace. When the splice operand is non-dependent, it can designate a namespace (as shown here) or class type.

In the example above, we're using reflection to act as a kind of degenerate namespace alias, which isn't particularly useful. This feature allows *nested-name-specifier*s to be formed programmatically, allowing metaprograms to use names declared in the scope of another declaration.

```
template<typename T>
void f() {
  constexpr meta::info ns = meta::scope_of(reflexpr(T));
  typename |ns|::Y var;
  // do something with Y.
}
```

Because `ns` is value-dependent in this template, its splice is type-dependent. This means we need to use the usual notations for disambiguating dependent names terms in template definitions.

6.3.7 Splicing dependent reflections

A refection whose value depends on a template parameter is a *dependent reflection* (it is a value-dependent expression). A *dependent splice* is a splice of a dependent reflection (the splice is type-dependent). Just like members of dependent types (e.g., `T::X` where `T` is a template type parameter), dependent reflections introduce ambiguities in template definitions.

In general, the rules for disambiguating dependent splices are the same as those for normal template parameters. We need to write `typename` wherever we introduce a type, and we need to write `template` wherever a reflection designates a non-class template.[10] For example:

```
template<meta::info X>
```

[10] A splice of a class template is preceded by `typename`. The `<` is assumed to start a template argument list.

```
void f() {
  |X| * p;
}
```

Here, `|X| * p` is parsed as a *multiplicative-expression*. This is the same as if the first operand of * had been e.g., `T::X` where `T` is a type template parameter. To turn this into a declaration, we add `typename` before the dependent splice:

```
template<meta::info X>
void f() {
  typename |X| * p;
}
```

Similarly, if we have a dependent reflection that names a non-class template, we need to add the keyword `template` before it:

```
template<meta::info X>
void f() {
  template |X|<int>(“hello”);
}
```

Without the `template` keyword, the start of the template argument list would be interpreted as a less-than operator.

There are some interesting issues that arise using dependent splices. Consider the following:

```
int g(int);

template<meta::info X>
auto f() {
  return |X|(0);
}

f<reflexpr(g)>(); // OK
f<reflexpr(double)>(); // OK?
```

In the second call to `f`, we substitute a type into something that appears to be a call expression. However, we could consider the entire dependent pattern `|X|(0)` to be a placeholder for either a call or constructor, which is then resolved during instantiation. We don’t necessarily need to disambiguate the terms in this case because the entire pattern is always an *expression*.

This same issue exists with template arguments and *nested-name-specifier*s too.

```
template<typename T> void f();
template<int N> void f();
template<template<typename> class X> void f();

template<meta::info X>
void g() {
  f<|X|>();
```

```
}
```

It's not clear whether the splice of `X` should be considered a type, value, or template template parameter at parse time. However, it's also not clear that we need to specify its form. The argument is simply a dependent template argument splice, which is distinct from other kinds of template arguments. P1985 includes a semantically similar notion: a kind of template argument that potentially represents a type, value, or template [32].

The *nested-name-specifier* issue is similar but allows both types and namespaces. In fact, there is an example of this in the previous section:

```
constexpr meta::info ns = meta::scope_of(reflexpr(T));
typename |ns|::Y var;
```

The splice `|ns|` could insert the fully qualified name of either a class, enumeration, or namespace.

After much consideration, I think these uses are fine without additional keywords to disambiguate their meaning. This will require implementations to represent dependent splices a little differently than their less-specific counterparts. However, in these three cases (*postfix-expression*s, *template-argument*s, and *nested-name-specifier*s), splicing one kind of entity or another does not affect the general syntax of the string (i.e., a *postfix-expression* is still a *postfix-expression*).

6.3.8 Splicing packs

P1240 also includes the ability to splice sequences of elements from ranges of reflections by placing the ellipsis inside a reifier (i.e., splice) operation (e.g., `typename(...list)`). However, the design may not have fully baked. For example, it doesn't address the more complex pack expansion issues discussed in Section 5.1. I've redesigned this feature to build on top of that work and also remove the need for extra ellipses in cases where the expanded range is non-dependent.

For example, expanding enumerators into the initializer of an array is straightforward:

```
enum E { A, B, C };
constexpr auto enums = meta::enumerators_of(reflexpr(E));
int vals[] {|enums|...};
```

The splice `|enums|` is expanded in the initializer of `vals` to produce the comma-separated list of *id-expressions*: `A, B, C`.

We can also combine local argument packs with other expressions:

```
int vals[] {2 * |enums|...};
```

In this case, the expansion is over the *initializer-clause* `2 * |enums|`, which ultimately initializes vals with zero, two, and four.

Spliced ranges can also be folded:

```
(|enums| + ...)
```

Here, the `+` operator is folded over the reflections, computing the sum of enumerator values in the enumerator `E`.

We can also expand and splice packs of type reflections.

```
struct X : C1, C2, C3 { };
constexpr auto [...bases] = meta::bases_of(reflexpr(X));
std::tuple<|meta::type_of(bases)|...> tup;
```

Here, the expansion of `|meta::type_of(bases)|` produces the list of *type-id*s `C1`, `C2`, `C3`, so `tup` has type `std::tuple<C1, C2, C3>`.

The ability to splice packs of reflections significantly improves the metaprogramming capabilities of these features as much as the use of variadic templates has improved the ability to write generic code.

This functionality relies on the semantics of structured binding packs, proposed in P1061 [17] and elaborated on by this document (Section 5.1). In this case, the splice of a range (as in concept) of reflections introduces a *splice pack*. A splice pack of the form `|range|` is inherently pack-dependent and denotes an unexpanded pack. When expanded, the pattern is instantiated, replacing each reference to the splice pack with the i^{th} element of the splice's reflection range.

The problem gets more interesting when the splice operand is dependent. In this case, we allow an optional `...` to precede the operand, indicating that the splice is a pack:

```
void f(constexpr sequence_of<meta::info> auto args) {
  eat(2 * |...args|...)
}
```

The expression `...args` specifies its identifier as a pack. Note that because `args` is type-dependent, the splice is type-dependent rather than pack-dependent, meaning that expansion happens during instantiation, not parsing. P1858 uses the operator `[:]` to denote a dependent pack [16].

6.3.9 Splicing names

The ability to generate new names for declarations is an essential part of many (most?) useful metaprograms. This is done using the identifiersplice operator `|# str #|` which inserts strings into the source text to be interpreted as an *identifier*. Where that operator appears in the grammar determines the kind of token the operator yields.

The operator is spelled with two new tokens: `|#` and `#|`. The purpose of using the hash/pound symbol is to visually identify the phrase as a token, which is then subject to semantic analysis (lookup, redeclaration, etc.). Note that the token produced by the splice operator is not eligible for macro expansion since the analysis of its operand happened in an earlier phase of translation.

The operand of the token splice operator is a constant expression whose type is either:

- `const char (&)[N]`, the type of string literals
- `const char *`, a null-terminated string
- a `range` (as in the concept) of `char` values (e.g., `std::string` and `std::string_view`).

For example:

```
|# “foo” #| // produces the token foo
constexpr string blah = “blah”
constexpr int n = 42;
```

```
|# blah + to_string(42) #| // produces the token “blah42”
|# format(“get_{}”, 42) #| // produces the token “get_42”
```

Note that the last usage assumes that `std::format` will eventually be declared `constexpr`.

The identifier splice operator is used only to generate *unqualified-id*s. There are actually five ways to generate *unqualified-id*s using splicing.

- `|# str #|` generates an *identifier* `str`.
- `operator |# str #|` generates an *operator-function-id* whose *operator* is `str`.
- `operator |t|` generates a *conversion-function-id* for the reflected type `t`.
- `operator “” |# str #|` generates a *literal-operator-id* whose *identifier* is `str`.
- `~|t|` generates a destructor name for the reflected type `t`.

In the case of *operator-function-id*s, the str must be one of the overloadable operators (e.g., **“+=”**), “`()`”, or “`[]`”.

7 Source code injection

As noted in P0633, there are essentially three approaches to synthesizing and injecting new code into a program:

- Injecting strings
- Injecting tokens
- Injecting syntax[11]

Synthesizing new code from strings is straightforward, especially when the language/library has robust tools for compile-time string manipulation (e.g., concatenation, formatting, interpolation, etc.). Synthesizing new code from tokens is also straightforward, but we might need to invent new tools for combining and composing token sequences.

The injection of both strings and tokens are more or less equivalent; injecting strings just requires the extra step of tokenization. In both cases, the strings or tokens are syntactically and semantically unanalyzed until they are injected, at which point the synthesized code is syntactically and semantically analyzed.

Synthesizing programs from syntactic fragments is described in P1717 [2] and updated P2050 [33]. The central premise of this approach is that fragments of code to be injected should fully syntactically and semantically validated prior to its injection. Injecting source code fragments transforms the original syntax by performing a set of substitutions to (possibly) rebind identifiers at the point the fragment is injected. This is, in many ways, very close to how templates work.

Here’s a relatively simple example that uses code injection to generate properties: private data members with public accessors and mutators.

```
struct book {
  << property<string>(“author”);
  << property<string>(“title”);
```

[11] In P0633, the idea of injecting syntax is couched in terms of metaclasses. However, the underlying mechanism has evolved to become independent of metaclasses.

```
  // other book properties
};
```

The definition relies on *injection declarations* to insert members into the class. An injection declaration starts with `<<` and has a constant reflection as an operand. The reflected declaration returned by the `property` function is injected into the class as if it had been written by hand. For example, we should expect this class to have the following:

```
struct book {
private:
  std::string m_author;
public:
  std::string const& get_author() const {
    return m_author;
  }
  void set_author(std::string const& s) {
    m_author = s;
  }
  // ...
};
```

The `property` function is defined as:

```
template<typename T>
consteval meta::info property(string_view id) {
  string member_name = “m_” + id;
  string getter_name = “get_” + id;
  string setter_name = “set_” + id;
  return <class {
    private:
      T |# %{member_name} #|;
    public:
      T const& |# %{getter_name} #|() const {
        return |# %{member_name} #|;
      }
      void |# %{setter_name} #|(T const& x) {
        |# %{member_name} #| = x;
      }
  }>;
}
```

There is a lot of new syntax here. For starters, property is a `consteval` function that returns a reflection. More specifically, the function returns a *class fragment*, which looks like this:

```
<class { ... }>
```

The fragment is a class definition enclosed in angle brackets.[12] Here, we elide the members of that fragment. A class fragment contains members of classes to be injected into a program. This class fragment contains 5 members: two access specifiers, one data member, and two member functions.

The data member has this declaration:

```
T |# %{member_name} #|;
```

There are two new operators here (although both have been discussed in P1240 and P2050). The `|# ... #|` operator is the *identifier splice*. It takes its operand and generates an *unqualified-id* which will become the name of the data member (Section 6.3.9). The `%{...}` operator is the *unquote operator*,[13] which allows the values of local member variables of a metaprogram to be used in the specification of a fragment (Section 7.1.7). Ultimately, this declares a data member whose name is determined by the current value of `member_name`, which happens to be "author" in the first injection declaration and "title" in the second.

The getter and the setter member functions are similarly defined.

```
T const& |# %{getter_name} #|() const {
  return |# %{member_name} #|;
}
void |# %{setter_name} #| (T const& x) {
  |# %{member_name} #| = x;
}
```

Note that we do not need to unquote to refer to `T`. That's because `T` is not a local variable. In general, we only need to escape to capture values that can change between invocations of the metaprogram (i.e., parameters and locals). Because this is a `consteval` function, we cannot refer to non-const globals.

The following sections provide more context on different kinds of fragments and injection operators (there are several).

7.1 Source code fragments

A source code fragment is an *expression*. The type of that expression is `meta::info`, which means that it is a reflection. Specifically, that value reflects the syntax between the angle brackets: a class, namespace, compound statement, expression, etc.[14] A fragment value also contains the values of each unquote within the fragment. This is discussed in more detail in Sections 7.1.7 and 7.2.

For the most part, fragments are containers of other constructs. A class fragment does not really describe a class, it just describes members of other classes. Moreover, the different kinds of

[12] In some languages, notably Haskell and Julia, similar constructs are called *quotations*. We could have called this construct a *quoted class*, but the term "fragment" is a little more descriptive in this context.
[13] This could also be called fragment interpolation.
[14] The reason there are different kinds of fragments is because the body of each fragment is fully parsed according to its kind. For example, a class fragment contains class members, while a block fragment contains statements. Unless we were willing to define fragments as token soup, we could not have had just one kind of source code fragment.

constructs can only be injected into like contexts. That is, a class fragment can only be injected into a class.

The following sections elaborate on the various kinds of fragments and their uses.

7.1.1 Class fragments

A class fragment, which we've already seen, encapsulates a sequence of member declarations. The body of the class is parsed as a member-specification.

```
<struct {
  int x;
  int y;
}>;
```

Class fragments are just like classes in most senses; the use of the `struct` keyword makes members public by default. We could have used `class` instead to make them private. Additionally, just like normal classes, member function definitions, default member initializers, default arguments, etc. are all parsed in the complete class context. Class fragments can also be named:

```
<struct S { S* make() { ... } }>
```

This allows members within the fragment to reference their own type. Here, we have a fragment that injects a `make` function, which presumably allocates objects of the (eventual) type.

Class fragments can also have base classes:

```
<struct : virtual private X>
```

We allow the body of class fragment to be omitted if it would otherwise be empty. In this case, we have an unnamed class fragment that, when injected will add `X` as a virtual, private base of the receiving class.

7.1.2 Namespace fragments

A namespace fragment encapsulates a sequence of namespace members; functions, global variables, classes, and other namespaces. For the most part, namespace fragments are like class fragments, except that they contain namespace members (and are parsed as such).

```
<namespace {
  void f() { }
  void g() { }
}>
```

Injecting this namespace fragment will, unsurprisingly, produce two new functions, `f` and `g`.

7.1.3 Enumeration fragments

An enumeration fragment contains a sequence of enumerators:

```
<enum {
  X = 10, Y, Z,
}>
```

Injecting this fragment into an `enum` will add the enumerators `X`, `Y`, and `Z` with values 10, 11, and 12.

The generation of enumerators is commonplace in certain domains. For example, both Clang and GCC rely heavily on the preprocessor to define enumerations describing the various nodes in their respective abstract syntax trees.

7.1.4 Block fragments

A block fragment contains a sequence of statements:

```
<{ if (!is_constant_evaluated()) abort(); }>
```

When injected, the statement(s) of the block fragment are injected into the current block.

```
constexpr int f() {
  << <{ if (!is_constant_evaluated()) abort(); }>;
};
```

Note that names of variables declared in the outermost scope of the block fragment are visible after injection. Although this might seem like a curious design choice, it is consistent with other kinds of fragments.

7.1.5 Expression fragments

An expression fragment contains (somewhat ironically) an *initializer-list*. Here is an expression fragment containing a single *initializer-clause*:

```
<(42)>
```

Unlike the other kinds of fragments, expression fragments are not typically injected using an injection-declaration because they simply create *expression-statement*s. That is, this:

```
<< <(42)>;
```

Produces the uninteresting statement:

```
42;
```

Instead, expression fragments are typically inserted into a program via splicing. Also, if the expression fragment contains multiple *initializer*s, then the splice must be expanded.

```
constexpr meta::info inits = <(1, 2, 3)>;
vector<int> vec { |init|... };
```

The resulting vector is initialized with the values 1, 2, and 3.

Because an expression fragment contains an *initializer-list*, packs can be expanded in that context:

```
template<typename... Args>
consteval void f(Args... args) {
  constexpr auto frag = <(%{args}...)>;
  eat(|frag|...);
}
```

Note that the use of *args* must be unquoted in the expression fragment because it refers to a parameter.

7.1.6 Dependent names

There are many cases where a fragment depends on names declared at (or before) the point of injection. For example, a fragment used as an arithmetic mixin might depend on certain operators:

```
constexpr meta::info mixin = <struct T {
  T operator+(const T& b) {
    T tmp = *this;
    add(b);
    return tmp;
  };
}>;

struct adder {
  void add(adder& x) { ... }
  << mixin;
};
```

Although this seems reasonable, the program is ill-formed. Name lookup fails to find an appropriate declaration for `add`. Unfortunately, this kind of dependence is likely to be a critical component of fragments in large metaprogrammed frameworks. Exactly how this problem should be solved has been the subject of much debate and experimentation. I think there are essentially three ways to address this problem: explicitly provide a context argument, relying on two-phase name lookup, or explicitly declaring required names. We present them all here:

7.1.6.1 Context arguments

The simplest and most direct method for solving this problem is to pass a reflection of the injectee as an argument to the metaprogram:

```
constexpr auto mixin(meta::info cxt) {
  return <struct T {
    T operator+(const T& b) {
      T tmp = *this;
      typename |%{cxt}|::add(b);
      return tmp;
    };
  }>;
}

struct adder {
  void add(adder& x) { ... }
  << mixin(reflexpr(addr));
};
```

This is effectively equivalent to using CRTP.

Because it is not possible to reflect the injection context for block and expression fragments, reflections of local variables should be passed directly to the metaprogram composing them.

We considered providing a magical library function, say `meta::current_injection()` that returns a reflection of the current context. That would remove the need to pass context arguments in many but not all cases. However, there may be some specification and implementation difficulties providing such an operation.

7.1.6.2 Two-phase lookup

As an alternative to passing arguments, we could simply rely on two-phase lookup. Two-phase lookup is a natural approach to solving this problem that allows the use of dependent names without a prior declaration.

Source code fragments are roughly analogous to templates (injection is also like instantiation). A fragment is essentially a construct parameterized by its context.[15] Because source code fragments are dependent, two-phase lookup applies, which means we can write the fragment above like this:

```
 constexpr mixin = <struct T {
  T operator+(const T& b) {
    T tmp = *this;
    this->add(b);
    return tmp;
  };
}>;

struct adder {
  void add(adder& x) { ... }
  << mixin;
};
```

The call to `this->add()` is resolved during injection rather than during parsing. This approach allows fragments to implicitly depend on declarations at the injection site. In this sense it's somewhat analogous to (albeit in a kind of opposite way) name lookup in unconstrained templates.

Unfortunately, without extension this two-phase lookup "trick" is currently restricted to class templates. One suggestion would be double down on two-phase lookup and add `this::` as a new kind of *nested-name-specifier*, allowing two-phase lookup in non-class fragments. For example, an expression fragment could use this to implicitly refer to local variables:

```
constexpr add = <(this::a + this::b)>;

int f(int a, int b) {
  return |add|;
}
```

When injected, the `this::` essentially drops away so `a` and `b` are found using unqualified lookup. For classes and namespaces, the `this::` specifier would revert to qualified name lookup.

[15] One of the substitutions performed during injection is to replace the syntactic context of the fragment (class, namespace, block, etc.) with the context at the injection site.

7.1.6.3 Required declarations

A third alternative approach is to explicitly declare which names must be available at the point of injection.

```
constexpr mixin = <struct T {
  requires T(const T&);
  requires ~T();
  requires void add(const T&);
  T operator+(const T& b) {
    T tmp = *this;
    add(b);
    return tmp;
  };
}>;

struct adder {
  void add(adder& x) { ... }
  << mixin;
};
```

Here, the class fragment contains three required member declarations: a copy constructor, a destructor, and an add operation. This makes those names available for lookup in the remainder of the fragment.

When the fragment is injected, those names are looked up[16] and the found declarations are matched against the requirement. If any declaration is not found or matched, the program is ill-formed. During injection, references to the declarations are then replaced by references to those just matched.

This feature also allows for required type and template declarations:

```
requires typename T;
requires template<typename T> class X;
```

Curiously, this idea is similar to some aspects of C++0x concepts. We are essentially declaring the members of a concept's archetype for the purpose of checking a template (like) definition.

However, this begs the obvious question, "why not use C++20 concepts to constrain the injection context?" Unfortunately, concepts are absolutely the wrong tool for this job—for a rather unfortunate reason. The satisfaction of a concept's requirements is predicated on lookup. At the time a fragment is injected into a class, that class is incomplete, so all lookups will fail.

We experimented with an implementation of this feature but deprecated it in favor of the simpler facilities above. However, the idea remains interesting.

7.1.7 Unquote

The unquote operator allows the use of local variables and expressions involving local variables inside a fragment. For example:

---

[16] This requires a novel form of lookup since `class` or `enum` receiving the declaration would be incomplete.

```
consteval meta::info make_fragment(meta::info t) {
  return <struct {
    typename |%{t}| obj;
  }>;
}
```

The function returns a fragment that includes a splice involving the parameter `t`. Within the fragment, the unquote operator designates a placeholder for a constant expression. Because an unquoted expression is a placeholder, it is type-dependent, meaning that we need to write `typename` before the declaration. The value for that placeholder is computed during constant expression evaluation as part of the fragment value and is replaced by its corresponding value during injection.

The unquote operator can also include more complex expressions:

```
consteval meta::info make_fragment(meta::info t) {
  return <struct {
    typename |%{meta::add_pointer(t)}| ptr;
  }>;
}
```

The resulting fragment will inject a pointer to the type reflected by `t` into the program.

7.2 Injecting source code

There are three ways to inject source code:

- at the current point in a program,
- into an existing context, or
- at a site determined by the current constant evaluation.

The first two both use << to denote an injection. I like the << operator to denote injection because of its connotation for streaming. An injection declaration

```
<< frag;
```

should be thought of as streaming the contents of the `frag` into the source code at this point in the file. However, this is not the only injection operation. We also support the ability to inject code into existing declarations using << as an operator, which allows the injection of code into an existing class, namespace, or enumeration (Section 7.2.3).

The third kind of injection is a little different because the target isn't obvious from the context. We are essentially "sending" code to be injected somewhere else in the program as a side effect. As such, I like the -> notation that we use in P1717 (Section 7.2.4).

Injection (as a process) is similar to instantiation, except that a) there are no explicit template parameters or arguments, and b) the result does not produce complete specializations. Injection performs two sets of substitutions.

The first substitution replaces the name of the fragment with the name of the injection. This is easily seen with class fragments:

```
constexpr meta::info frag = <struct s { s* next; }>;
```

```
struct target {
   << frag;
};
```

When injected, the name `s` in the fragment is replaced by the name `target`, which produces this:

```
struct target {
  target* next;
};
```

The second of the substitutions involves unquoted operators in the fragment with the values computed during evaluation. The mechanism for doing this is a bit involved because we are mapping values computed during evaluation into placeholders identified at parse time. Consider:

```
consteval meta::info print(int n) {
  return <{
    cout << %{n};
  }>;
}

void f() {
  << print(0);
}
```

When the parser first encounters the unquote operator, it replaces it with a placeholder for a `constexpr` value. The unquoted expression is associated with the fragment expression. During evaluation, the fragment specifying the return value is evaluated. This evaluation packages up the current values of associated unquoted expressions as part of the fragment's value, effectively becoming a kind of closure for the fragment value.

The fragment value returned from `print` is then injected. During injection we substitute values in the closure for their corresponding locations within the fragment, producing the desired result.

The following sections describe different aspects of source code injection.

7.2.1 Injection declarations

We've already seen the injection declaration. Because these operations are syntactically declarations, they can appear at namespace scope, at class scope, and at block scope. This allows the injection of members into each of those contexts (from like fragments and reflections).

7.2.2 Injected enumerators

Enumeration fragments can be injected into `enum`s using an injection enumerator:

```
constexpr meta::info rbg = <enum { red, blue, green }>;
constexpr meta::info cmy = <enum { cyan, magenta, yellow }>;
enum class color {
  << rbg;
  << cmy;
};
```

Injection enumerators work just like injection declarations. Their operands are injected where the declaration appears in the program.

7.2.3 Injection operator

We can also inject fragments into other contexts using the << operator:

```
consteval r gen(meta::info& out) {
  out << <struct { ... }>;
}
```

Injecting into an existing context allows programs to be built incrementally. It allows library writers the ability to decompose complex compositions into smaller units.

However, we must be careful about injecting code into existing definitions. For example, we shouldn't be able to inject new virtual functions or data members into a class. That would be madness.[17] However, it might be possible to inject new non-virtual member functions and static data members and member functions. Injecting functions and variables into namespaces seems reasonably safe, however.

7.2.4 Injection statements

An injection statement allows code to be injected at the site of the current injection. This allows metaprograms to simply generate code and inject it without explicitly returning it. This can be useful when generating different subsets of a definition where you might not need (or want) to concatenate the results using the << operator. Here is a small metaprogram that injects links into a class:

```
consteval void gen_link(string_view id) {
  -> <struct s {
    s* |# %{id%} #| = nullptr;
  }>;
}

struct node {
  consteval {
    gen_link("next");
    gen_link("prev");
    gen_link("parent");
  }
};
```

Here, we use a metaprogram (consteval block) to invoke a sequence of link generators. The resulting class contains three pointers: `next`, `prev`, and `parent`.

7.2.5 Cloning declarations

Thus far, our injection functions have worked with various kinds of fragments. However, it is also possible to inject copies of existing entities. This ability features heavily in practically all examples

[17] Injecting new data members into a class after it was defined would change its layout, which would effectively break any use of the class prior to the injection. This is likely true for implicitly typed enumerations too.

of metaclasses where members are copied from a prototype class into a destination class (Section 8.4).

For example, here is a small metaprogram that copies all data members of a class into a fragment.

```
consteval meta::info copy_members(meta::info t) {
  meta::info frag = <struct { }>;
  for (meta::info m : meta::all_data_members_of(t))
    frag << m;
  return frag;
}
```

The injection of each member clones the existing member into the receiving context. In this case, the receiver is a fragment, but it could just as easily be another class. Many more complex metaclass examples involve cloning declarations.

This can be used to create structurally equivalent objects.

```
template<typename T>
struct structure_of {
  << copy_members(reflexpr(T));
}
```

However, cloning a member also copies its semantic properties such as access specifiers. In this case, any private members of `T` will remain private in `structure_of`. P1717 supports the ability to modify the specifiers of a declaration (including its name) as it is injected. To force all data members to be public, we would write this:

```
consteval meta::info copy_members(meta::info t) {
  meta::info frag = <struct { }>;
  for (meta::info m : meta::all_data_members_of(t)) {
    meta::make_public(m);
    frag << m;
  }
  return frag;
}
```

The `make_public` function does not modify the reflected entity; that would lead to madness. Instead, the operation modifies the local value of `m`, essentially setting a flag in its representation. When injected, the compiler consults which local modifications have been requested and applies them if possible.

There are only a handful of specifiers that can be changed when cloning a member.

- The access of a class member can be changed.
- A member can be made static.
- A member can be made virtual or pure virtual or marked override or final.
- A class can be marked final.
- A declaration can be made `constexpr`, `consteval`, or `constinit`.
- A declaration can be made inline.

- A declaration can be renamed.

Note that you can never *remove* a specifier. That is, we cannot clone a virtual member function as a non-virtual member.

7.2.6 Injecting parameters

Injecting a sequence of parameters into a function declaration has been a challenging problem since my earliest work on metaclasses. The problem is that there are really two aspects of injecting parameters: identifying the range of parameters to inject and then naming them within the body. Injecting parameters is relatively easy, referring to them has been difficult. Early experiments relied on splicing identifiers that could name the injected tokens, which works, but I've always felt that it would be better if we could do this without generating new identifiers. We should be able to refer directly to the parameters.

A promising approach is to declare injected parameters as a function parameter pack "initialized" by a list of parameters:

```
int f(auto... params << meta::parameters_of(some_fn)) {
  eat(params...)
}
```

Here, `args` is declared as an *injected parameter pack*. The "initializer" of that argument pack is a parameter reflection range. Within the body of the function, we can expand the pack in the usual way, including forwarding them as needed. We haven't found any use cases where we need to access specific parameters, but we could provide reflection facilities for inspecting certain kinds of packs, or we could adopt the pack indexing operators described in P1858 [16].

We can do the same with template parameters, except that we would need to adopt some kind of universal template parameter as described in P1985 [32].

```
template<template auto... params << meta::parameters_of(X)>
void f()
  some_other_template<params...>();
};
```

Note that we don't yet have a way of creating parameter fragments. It seems like that feature might be desirable, but there haven't been any concrete use cases.

The syntax for injecting parameters need not be limited to just this one context. We could conceivably allow this to be used to declare sequences of member variables and local variables as well.

7.2.7 Splicing fragments

In general, it is not possible to splice a fragment. A splice generates a reference to an existing entity, while a fragment is distinctly not an entity. The only exception is expression fragments, which specify complete (albethey dependent) computations.

8 Abstraction mechanisms

Static reflection and source code injection provide the low-level functionality needed to inspect a program's elements at compile-time and generate new elements based on its form and meaning.

Despite these features being relatively new in C++, we already know a number of ways these might be used to simplify various metaprogramming-related tasks.

The features discussed in this section are largely sourced from others' proposals, albeit with some elaboration and massaging to make those ideas fit my long-term vision for metaprogramming. Some of the features in this section are also new inventions (e.g., compile-time I/O in Section 9). For the most part, these features are largely speculative and would require significant work to move forward. However, I include them here because I think they have the potential to greatly improve the metaprogramming experience in C++.

8.1 Macros

We do not want more preprocessor macros. We want something that provides us with all the power of preprocessor macros with none (or at least very few of) the problems and pitfalls. P2040 provides an initial design on top of P1717 [34]. It considers the following example (adapted for the notation in this paper).

```
std::string expensive_computation();
int main() {
  enable_logging = false;
  log(reflexpr(expensive_computation()));
}
```

The `log` function is defined as a `consteval` function taking an expression reflection:

```
consteval void log(meta::info message) {
  -> <{
    if(enable_logging)
      std::cerr << |%{message}| << "\n";
  }>;
}
```

When executed, the expression reflected at the call site is injected into a block fragment that conditionally prints the result. This block fragment is then queued for injection into the current injection context.

The log function is effectively a *macro*. That is, it accepts an argument by name (in this case reflection) and then inserts that argument back into the program. Unlike preprocessor macros, this follows all the usual C++ rules: scoping, visibility, access…

Interestingly, P2040 doesn't actually propose anything beyond what is already present in this paper. The ability to reflect and splice expressions—even those involving local variables—is already accounted for. Note that in the log function, the splice is injected, meaning that it is actually applied in the `main` function and not where the splice appears lexically.

That said, it would be nice if we didn't need to explicitly reflect the argument. We should be able to declare functions in a way that accepts parameters by reflection.

```
consteval void log(reflexpr message) {
  << <{
    if(enable_logging)
```

```
        std::clog << |%{message}| << "\n";
    }>;
}
```

The semantics of this feature are closely related to both `constexpr` function parameters [35] in the sense that both of these are implicitly function templates. The definition is rewritten so that its reflected parameters are accepted as template arguments and (perhaps) not function arguments.

This also provides a basis for implementing parametric expressions [36]. In particular, a function taking a parametric expression is essentially a macro whose input is restricted to expressions of a particular type.

It is unlikely that macros will be able to be overloaded on parameter type due to parsing issues. Overloading on arity is likely to be possible. This restriction stems from the fact that arguments to macro must be parsed as reflection operands and not as expressions or template arguments.

```
consteval void print(reflexpr x) {
  cout << describe(x); // returns a string describing x
}
print(0);
print(int);
```

The compiler needs to know, at the point it starts parsing function arguments, that `print` is not a normal function. Again, this raises issues with dependent macros. We would need new syntax to explicitly denote that a call expression was a macro call or a normal call, which could be a simple as writing `reflexpr` before the start of the argument list.

```
print reflexpr(0)
```

It would also be nice if we could pattern-match against the syntax of the expression in the style of Rust. Rust provides a mini grammar for matching tree sub-expressions. However, because reflections are part of the regular language, we don't need to invent new syntax for matching; `if` statements can suffice. We do, however, need facilities for destructuring and traversing expressions as trees. Hopefully, the ongoing work on pattern matching [37] will provide more intuitive and convenient matching capabilities than lists of `if` statements.

8.2 Mixins

A mixin is a fragment of a class to be included into others. They often implement facets of structure of behavior that can be readily encapsulated. Traditionally mixins are supported using (multiple, sometimes private) inheritance and often via the Curiously Recurring Template Pattern (CRTP) and policy-based class design. These approaches work well for mixing interfaces, but the use of inheritance to express a form of composition can have some unintended consequences such as undesirable base class conversions, or worse, unmaintainable giant piles of template spaghetti.[18]

P1717 [2] directly and cleanly supports a style of mixins through injection declarations.[19] For example, a user-defined integer class should provide the usual arithmetic operators:

---

[18] Holmes, Odin. "C++ Mixins: Customization through Compile-time Composition". C++Now. 2018. Presentation.
[19] The style of mixins supported by P1717 is closely related to D's template mixin facility.

```
struct integer {
  // Mix in the usual arithmetic operators
  << arithmetic_operators;

  // Functions needed to implement the usual operators.
  void add(integer& x) { ... };
  void sub(integer& x) { ... };
};
```

The class definition simply injects a fragment that defines the required operations. Its definition is:

```
constexpr meta::info arithmetic_operators = <class T {
  T& operator+=(T const& x) {
    this->add (other); return *this;
  }
  T operator+(T const& x) const {
    T tmp = *this; return tmp += x;
  }
  T& operator-=(T const& x) {
    this->subtract(other); return *this;
  }
  T operator-(T const& x) const {
    T tmp = *this; return tmp -= x;
  }
}>;
```

The fragment's members are defined in terms of operations expected at the injection site (i.e., via two-phase lookup).

Defining mixins as global constants is a bit clunky. Most people expect mixins to be actual class definitions. We could add a new attribute to facilitate that expectation.

```
struct arithmetic_operators mixin {
  // same as above.
};
```

Of course, this is just syntactic sugar for the source code fragment above. Note that we could also extend this notion to support other kinds of mixins (namespace, enumeration, function, etc.). However, I'm not sure there are compelling use cases for those other kinds of mixins.

8.3 Attributes

User-defined attributes provide a mechanism for either associating compile-time data with declarations. The ideas in this section build on P1887 [37].

A *user-defined attribute* is an attribute that associates metadata (compile-time values) with a declaration. For example, we can explicitly annotate a test function with an attribute that describes the test.

```
[[+Catch::test_case("copy")]] void test_copy();
```

Here, `[[+Catch::test_case("copy")]]` constructs metadata for its function from a string literal describing the test case.[20] The attribute constructs a class value from the given arguments (i.e., `test_case` is a class). Metadata values should be accessible through the reflection API, thus allowing metaprograms to collect annotated classes, analyze their values, and generate code, likely a test suite in this example.

P1887 introduces a new standard attribute `decorator` that indicates that a class can be used as an attribute. The `test_case` attribute could be defined as follows:

```
namespace Catch {
  struct [[decorator]] test_case {
    constexpr test_case(const char* name) : name(name) { }
    const char* name;
  };
}
```

Whether we need the `[[decorator]]` attribute at all is an open question. Simply allowing any literal type to be used as annotation is not unreasonable.

8.4 Metaclasses

A *metaclass* is metaprogram that generates new code (usually the same class) from a class *prototype* [10, 2]. This can be used, for example, to generate common aspects of a class's structure based on the original specification as prototype.

For example, here is a declaration of pair using the `regular` metaclass.

```
template<typename T, typename U>
struct(regular) pair {
  T first;
  U second;
};
```

In this class, regular names a metaprogram that synthesizes declarations that (ostensibly) make pair a regular type: destructible, default constructible, copy constructible, and equality comparable.

The actual mechanism to make this work is a lexical trick; metaclasses are just syntactic sugar on top of the features described in Sections 6 and 7. To the compiler, the actual definition of `pair` looks like this:

```
namespace __hidden {
  template<typename T, typename U>
  struct pair {
    T first;
    T second;
  }
}
template<typename T, typename U>
```

[20] The leading + was deemed necessary to make them syntactically different than conventional attributes because user-defined attributes must be valid expressions.

```
struct pair {
  using prototype = __hidden::pair<T, U>;
  consteval {
    meta::info self = reflexpr(pair);
    regular(self, reflexpr(prototype));
  }
}
```

The original definition of `pair` is tucked away in an unnamable namespace and a new definition of `pair` is created. This new definition is largely defined by a metaprogram that simply calls the name of the class.

The `regular` definition "simply" pre-generates a number of common operations intended to model a regular type:

```
consteval void regular(meta::info& self, meta::info proto) {
  generate_default_constructable(self, proto);
  generate_movable(self, proto);
  generate_destructible(self, proto);
  generate_equality_comparable(self, proto);
}
```

I assume that each of the `generate_` functions inject a set of declarations into `self`.

8.5 Stereotypes

Metaclasses can be generalized into a broader set of metaprograms that rewrite declarations and definitions. A prior unpublished version of this paper characterized them variously as "decorators" and "transformers". However, Ville Voutilainen has suggested the name "stereotypes" from UML. In UML, a *stereotype* is a modeling element that allows designers to create new kinds of elements, specific to the designer's domain, based on existing one (e.g., classes and functions).

I somewhat cleverly suggested that we also borrow the stereotype syntax from UML. If we were to rephrase metaclasses using this notation, the `pair` class above would be:

```
template<typename T, typename U>
struct pair <<regular>> {
};
```

Here, I'm choosing to put stereotypes in the same position as attributes rather than immediately following the class key.

After some consideration, I think that stereotypes should not be restricted to generating definitions of predeclared classes like metaclasses do. Instead, the metaprogram invoked by the stereotype should be responsible for explicitly generating a set of program elements. Here is how the compiler should see the declaration of `pair`.

```
namespace __hidden {
  template<typename T, typename U>
  struct pair {
    T first;
```

```
    T second;
  }
}
consteval {
  regular(reflexpr(__hidden::pair));
}
```

The regular function is now responsible for generating the entire class. The definition of regular could be this:

```
consteval void regular(meta::info proto) {
  generate_class(proto);
}
```

I'm omitting the complex parts of the metaprogram for now because generating copies of template heads and function signatures is hard, and I'm not sure we've proposed enough features to do this, much less to do it in an expressive way.

On potentially interesting use of this feature is to use it with namespaces to generate closed, discriminated class hierarchies.[21]

```
namespace exprs <<variant>> {
  struct Expr { ... };
  struct Add : Expr { ... };
  struct Sub : Expr { ... };
  ...
};
```

These kinds of hierarchies typically require a fair amount of incidental metaprogramming: organizing and assigning integer values for derived classes, implementing testing functions for conversion, and automatically generating visitor functions. The `<<variant>>` stereotype would be responsible for synthesizing all of that code for us.

Another possible use of stereotypes is to improve the declaration of coroutines.

```
int count_lines(std::filesystem::path p) <<task>>;
```

The `<<task>>` stereotype can simply rewrite the signature to support the usual declaration where the "kind" of coroutine is encoded in the return type:

```
task<int> count_lines(std::filesystem::path p);
```

Even though this is a relatively simple and largely unnecessary transformation, it does clearly separate the "taskiness" of the function from its return type.

One final potential use of stereotypes would be to implement C#-style properties for class members. While properties don't make a good language feature for C++ on their own, they can be immensely

[21] A discriminated class hierarchy is one where the base class provides a discriminator. These are often used when virtual functions and dynamic dispatch are undesirable because of design or performance criteria. For example, Clang makes extensive use of this technique for its abstract syntax trees.

useful for metaprogramming frameworks that need to know about “properties” instead of data members and member functions.

```
class customer {
  string first_name [[*readwrite]];
  string last_name [[*readwrite]];
  string full_name [[*readonly(<(
    this::first_name + “ ” + this::last_name
  )>)]];
};
```

Here, `get` and `set` are metafunctions that generate code to control access to the `first_name` and `last_name` properties. We can also use fragments to define read-only members whose values are computed by other members of the class.[22]

That resulting output could be this.

```
class customer {
private:
  string first_name;
public:
  const string& get_first_name() const {
    return first_name;
  }
  void set_first_name(string const& value) const {
    first_name = str;
  }
private:
  string last_name;
public:
  const string& get_last_name() const {
    return last_name;
  }
  void set_last_name(string const& value) const {
    last_name = str;
  }
public:
  string get_full_name() const {
    return first_name + “ ” + last_name;
  }
};
```

[22] Recall that `this::` is tentatively prosed as a mechanism for deferring lookup in fragments until the point of injection. We could also have written a block fragment as the argument if the computation were more complex.

8.6 Analyzers

It's possible to use attributes to invoke non-modifying checks over declarations to check for naming consistency, the presence or absence of operations, etc. Note that such tools would require access to the compiler's diagnostic facilities in order to provide coherent diagnostics (Section 9.1).

I suspect that purely read-only analyzers don't exist outside of other framework metaprogramming tools such as decorators (Section **Error! Reference source not found.**), transformers (Section **Error! Reference source not found.**), and metaclasses (Section 8.4). In other words, this will fall out of the ability to diagnose style or consistency issues related to a transformation for a specific framework (e.g., Qt objects).

I don't believe that in-source analyzers can or should replace traditional static analysis tools. Because these kinds of analyzers are always invoked during compilation, programmers will want to balance compile-time concerns with the benefit provided by early diagnostics.

8.7 Programmatic synthesis

Thus far, all examples that synthesize and inject new code into a program use either templates or source code fragments. However, there are a wide range of applications that require more precise control over the synthesis of new code. Specifically, these languages need an API for building representations of new source code incrementally.

For example, the ability to derive new functions from existing functions requires both the ability to inspect the entirety of a function definition and the ability to build a new definition. We could use this feature, for example, to automatically curry functions.

```
int less(int a, int b);
auto f = curry(less);
```

The result of currying `f` is a nested sequence of lambdas:

```
auto f = [](int a) {
  return [a](int b) {
    return less(a, b);
  }
};

f(0)(1) // returns true
```

In this case, the `curry` metaprogram doesn't need to inspect the definition of its argument, only its signature. The construction of the initializer is not especially easy using fragments and might look something like this:

```
auto x0 = <{ return less(|# "a" #|, |# "a" #|) }>;
auto x1 = <{ return [|# "a" #|](int |# "b" #|) { << %{x0}; } }>;
auto x2 = <( [](int |# "a" #|) { << %{x1}; } )>;
auto f = |x2|;
```

The actual implementation of the `curry` metaprogram would need to be a recursive function that incrementally wraps lambdas around an underlying function call. I use identifier splices because it doesn't seem possible to declare lambda parameters before synthesizing the definition.

An API for synthesizing code could be significantly easier to read and write. After all, the construction of composite objects (like abstract syntax trees) is a well-understood problem. Moreover, such an API might be more efficient. The fragment-based definition requires a linear number of injections, while the library-based approach requires just one.

A more interesting use of programmatic code synthesis is automatic differentiation. This technique has become popular in industries employing machine learning as a method for rapidly implementing operations and their derivatives from a single specification. For example, suppose an application defines the following measure:

```
double f(double x, double y) {
  return x * y + sin(x);
}

auto dfx = autodiff(f, “x”);
```

Here the `autodiff` metaprogram takes a function and the parameter index for which we are computing the derivative. The metaprogram examines the definition of `f` and builds a computation graph that can be used to synthesize a definition for the derivate, which in this case would be `y + cos(x)`. Here, we can imagine derive returning a capture-free lambda expression returning the required result.

Both automatic currying and automatic differentiation require the ability to:

- traverse the structure of a function definition,
- programmatically build up a new function definition, and
- inject that definition into a new function.

This requires significant extensions to the reflection facilities and library (Section 6.2) so that the structure of statements and expressions are available to metaprograms. We will also have to define a new API for the programmatic construction of (essentially) abstract syntax trees. I’m not entirely sure what these APIs should look like. However, their specification will likely be challenging since every C++ compiler structures their internal representations differently. A project like IPR may help us find a path forward [38, 39].

9 Compile-time I/O

The ability to read from and write to external files during compilation raises some very interesting prospects for application designers. This section explores the different kinds of input and output streams that could be available for metaprograms.

9.1 User-defined diagnostics

The idea of supporting custom diagnostics is not new. The `static_assert` facility was originally designed to require a diagnostic, although that was relaxed later. P1267 suggests new attributes that would allow diagnostics on constrained overloads [40]. Those attributes could also be extended to concept definitions themselves. But these diagnostics are attached to language facilities. Metaprogramming enables programmable, compile-time static analysis, which would allow diagnostics to be constructed and issued programmatically. For example, suppose we have decorator/metaclass that injects allocator fields into target class.

```
template<typename T, typename U>
struct pair <<allocator_aware>> {
  // members
  allocator* my_alloc;
};
```

Here, the class author may have mistakenly added an extra allocator field, which isn't needed for all specializations of the template. When instantiating e.g., `pair<int, char>`, the allocator becomes (likely) unused. The `allocator_aware` metaprogram could issue a compiler warning for that case.

```
 warning: unused allocator my_alloc
```

The code that generates that error could look like this:

```
meta::warning(meta::location_of(decl))
  << "unused allocator in " << meta::name_of(decl)
```

Here, the `warning` function returns a stream pinned to the location of the declaration.

User-defined diagnostics should not be usable in non-`constexpr` code. Even in `constexpr` functions, we would have to ensure the diagnostic does not "leak into runtime". This is also true for compile-time tracing (Section 9.2).

Interacting with a compiler's diagnostic system may be more involved than simply providing strings for output. For example, modern compilers highlight various names in their output and show the context where a diagnostic is sourced. Moreover, there are typically flags used to selectively configure which warnings are in force (or not). It may be desirable to *declare* diagnostics in a way that allows the compiler to integrate them into its own frameworks.

9.2 Compile-time tracing

P0596 proposes `constexpr_report` [41, 14], which supports the printing of information to a console. The console is (likely to be) a distinct output stream from the diagnostic stream, and should not require e.g., source locations. This is intended for arbitrary user output to assist in debugging.

An updated interface to `constexpr` tracing should likely be based on more robust text-processing facilities instead of a set of overloads for specific values. It might also be nice to provide a streaming interface to the console:

```
contexpr int add(int a, int b) {
  meta::console << "add: " <<  a << ' ' << b;
  return a + b;
}
```

Here, console is a kind of `ostream` like `cout` or `cerr`.[23] Whether we would need different versions of the console to accept wide characters is an open question.

The original proposals ensured that `constexpr` tracing did not affect the behavior of the abstract machine, meaning the statement has no effect on runtime behavior. With a streaming interface,

[23] We shouldn't use `cout` or `cerr` because the compiler may already use those streams for different purposes (e.g., outputting compiled code and diagnostics).

implementations might have to work a little harder to suppress runtime code generation for such statements.

P0596 also includes `constexpr_assert`. I think the design of that feature should be covered in the contracts proposals and not here.

9.3 Compile-time file I/O

The ability to read from and write to external files at compile-time is game changing. It allows a program's definition to depend on external data in a way that isn't really possible in modern C++. You can approximate those dependencies with clever uses of the preprocessor and `constexpr` variables, but those approaches generally require an external program to format the source data (into a C++-compatible format).

One especially powerful use of compile-time file I/O is GUI programming. Some modern UI frameworks decouple the UI specification from its behavioral specification (sometimes called code behind). In Windows, for example, XAML files specify the structural and styling elements of a view, while the behavior of the UI is implemented in code.

Instead of having external tools generate partial classes or other complex bindings, we could use compile-time I/O to directly import the XAML specification and then use programmable attributes and other metaprograms to synthesize the structural elements of their corresponding classes, and apply defaults, styles, and generate event handlers at compile time (or at least set up the code so that it executes when needed).

The same technique could also be used to help manage data types in an object-relational mapping. In that case, a translation unit could import a schema, and metaprograms could either generate the entire data model in one shot or populate pre-defined classes with their corresponding fields and any respective default values.

However, there are three big issues related to compile-time file I/O: security, tooling, and data races.

Security issues arise when the language allows source code to open any file the compiler has permission to read. If this were allowed, a rogue library could quietly embed your private SSH key into the program and then potentially send it across a wire at runtime. Fortunately, SG7 and EWG have jointly decided that the compile-time file I/O should be restricted, so we don't need to revisit that design issue.

The second issue is tooling. If we allow programs to access composed (concatenated) strings as file names, then it becomes potentially impossible for a build system to track dependencies between a static import and the translation unit that depends on it. Ideally, it should be easy to identify the complete set of dependencies for any module.

Finally, for programs that write files at compile-time, race conditions become an issue. Because translation units can be parallel, it's conceivable for two metaprograms to write to the same output file at the same time. However, some frameworks might choose to allow parallel writes to a synchronized external resource.

In short, we need to provide limited access to a set of non-code resources for which the build system can track dependencies. I think we can look at modules as providing a solution for these problems.

C++ modules are physical artifacts associated with a module name. The mapping between module names and their corresponding binary and source files is effectively maintained by the build system. This mapping essentially becomes a set of inputs to the compiler provided by the build system.

9.3.1 External resources

We should be able to import external, non-source code resources just like modules. In this approach, a *resource* is an external data source available during compilation for compile-time reads and writes. Each resource is identified by a *resource name*, which is translated, in an implementation-defined way, to the underlying data source.

```
// in my/app.cpp
export module my.app;
import std.meta;
import readable my.app.version -> std::meta::resource version;
```

This module unit imports two things: a `std.meta` module and a resource named `my.app.version`. The `readable` identifier indicates that the resource is intended as an input to the translation unit. This can also be `writable` or `mutable`. The imported resource version is associated with a variable declaration with internal linkage whose type is `meta::resource`. The variable provides user access to the underlying file while its type determines how the file can be accessed. Note that `std.meta` must be imported in order to declare the resource variable.

Just like `meta::info`, the `meta::resource` type is an implementation-defined scalar with many of the same properties (e.g., can't be reinterpreted). Resource values are handles to files opened by the compiler on behalf of the translation unit.

We can also import resources using header names.

```
// in my/app.cpp
export module my.app;
import std.meta;
import readable "my/app/icon.png" -> std::meta::resource icon;
```

There are two operations defined for `meta::resource` values: `meta::read` and `meta::write`.[24] These operations support low-level I/O operations on external data. These functions have the following declarations.

```
namespace meta {
  constexpr int read(resource rc, byte* buf, int n);
  constexpr int write(resource rc, const byte* buf, int n);
}
```

They are essentially analogous to the POSIX read and write functions but work on compile-time resources. Obviously, we would prefer not to work with such low-level resources. The following section describes more advanced ways of working with compile-time resources.

[24] We could conceivably allow resources to be closed or queried, but I'm not aware of any strong motivating use cases.

9.3.2 Reading files

Instead of importing a resource and then (painstakingly) pulling data using successive `read`s, we can import a resource as a formatted input stream.

```
export module my.app;
import std.meta;
import readable my.app.version -> std::meta::istream version;
```

An imported `meta::istream` is essentially a `consteval` variable as described by P0596 [14] that is defined over a `meta::resource` value. It can be modified (read from) during constant expression evaluation but is otherwise constant between evaluations.

Input streams make it easier to read data into programs:

```
namespace app {
  static consteval int read_version() {
    int v;
    version >> v;
    return v;
  }
  export int get_version() {
    return read_version();
  }
}
```

Note that the contents of a resource do not become a part of the translation unit. This is purely a way of getting data into a translation unit. This is also true for classes with resource constructors (Section 9.3.5) and resource adaptors (Section 9.3.6).

9.3.3 Writing files

If we're going to provide the ability to read from external resources, we should also provide the ability to *write* to external resources. To write to a resource, we import it as a *writable resource* by declaring it `writable` and binding the resource to a variable that supports output operations such as `meta::ostream`.

```
export module my.app;
import writable my.app.toc -> meta::ostream os;

// content of translation unit
// ...

consteval {
  // Write out the name of all the classes
  for (meta::info x : get_all_classes())
    os << meta::name_of(x) << '\n';
}
```

The `my.app.toc` resource name is mapped to a file to which the contents of the translation unit are written. The class `meta::ostream` is the class for formatted output. Just like meta::istream, it is defined over an underlying `meta::resource` value.

The ability to write to external resources directly supports the generation of data or language bindings based on the definitions of C++ classes and functions. For example, GCC could replace its use `gengtype` (an application-specific C/C++ preprocessor) to generate mark and sweep operations for garbage collecting AST nodes. With this feature, each translation unit would simply dump its own collection of GC routes and collection functions to be linked into the program later.

The build system (by way of the module mapping) is ultimately responsible for determining the disposition for opening mutable files. In this case, it seems reasonable to truncate the file each time it is opened for writing. We could also choose to append to the file, although it isn't clear what the immediate applications of appending are.

In general, translation units with writable resources introduce the potential for race conditions in the build. However, maintaining a unique file/resource mapping gets around the need to synchronize output.

9.3.4 Modifying files

A *mutable resource* allows both reads and writes. The ability to both read from and write to external files supports a wide range of metaprogramming applications. The simplest kind of mutable resource is `meta::stream`. This allows the metaprograms of a translation unit to both write to and read from that resource.

For example, here is a small metaprogram that increments a build number each time the translation unit is compiled.

```
// in my/app.cpp
export module my.app;
import extern mutable my.app.build -> meta::stream data;

consteval int build() {
  int value;
  data >> value;
  ++value;
  data << value;
  data.close();
  return value;
}

constexpr int build_number = build();
```

The `meta::stream` class is the type used for mutable resources. It supports both formatted input and output. The `build` function returns the current build value, increments it, and writes it back to the resource. We explicitly close the file to prevent subsequent increments after initial update. The function is executed when the `build_number` initializer is evaluated.

We don't have to work directly with the underlying stream. We can abstract over it the same way we abstract over input streams for constant resources. For example, we can encapsulate the build counter in a class.

```
// in my/app/meta.cpp
export module my.app.meta;
export struct build_counter {
  constexpr build_counter(meta::stream& s)
    : stream(s)
  {
    stream >> value;
  }

  constexpr build_counter& operator++() {
    stream << ++value;
    stream.close();
    return *this;
  }

  meta::fstream& stream;
  int value;
};

// in my/app.cpp
export module my.app;
import my.app.meta;
import mutable extern my.app.build -> build_counter build;

constexpr int build_number = ++build;
```

As noted earlier, the ability to write to files at compile-time introduces the potential for race conditions in parallel builds. For simple text files, allowing multiple readers and writers is surely a path to bedlam. However, providing a resource that can synchronize access provides for some truly interesting applications. For example, we can define a counter that uniquely distributes identifiers across multiple translation units.

9.3.5 Resource constructors

When the resource is intended to represent a constant value, we shouldn't need to write the code that extracts those values. Ideally, we should be able to bind the resource to a variable that automatically initializes itself with the contents of the associated file. For example, importing a version string should be as simple writing this:

```
// in my/app.cpp
export module my.app;
import std.core;
import readable my.app.version -> std::string version;
```

Any class that has a *resource constructor* can be bound to an external resource. A resource constructor is a `consteval` constructor that takes a `meta::resource` or an lvalue reference to one of stream classes above. For example, the resource constructor for `std::string` is this:

```
class string {
  consteval string(meta::istream& is) : string() {
    is >> *this;
  }
};
```

Having defined the resource, we can now export the parsed value or use it internally.

```
namespace my_app {
  export std::string version = ::version;
}
```

Resources cannot be exported from a module. However, the name of the resource is globally available so any modules that want to load the resource are available to do so.

Although we tend to think of resources as files, this isn't necessarily true. A resource could be anything that can be read from or written to by the compiler. Files are obvious. Resources could also be popened files. Resource could even be open sessions with running programs).

9.3.6 Resource adaptors

In many cases, we might want the resource variable to be structured (i.e., typed) according to the contents of the resource. For example:

```
// in my/app/view.cpp
export module my.app.view;
import microsoft.ui.xaml;
import readable my.app.view.xaml -> xaml::meta::document doc;
```

I should be able to use doc to navigate through the contents of the XAML document like so:

```
cout << doc.navbar.menu_item[0].name;
```

Here, `xaml::document` is a *resource adaptor*, a class template with the following definition.

```
template<meta::istream& In>
class document {
  consteval {
    // Read XML from In and inject data members
    // corresponding to the various elements found
    // in the XAML file.
  }
};
```

Now, we can refer to elements in the XAML document directly, including e.g., as operands to metaclasses.

```
struct(xaml::view(doc.view)) view {
  // application-specific code
```

```
}
```

The `xaml::meta::view` function is a metaprogram that populates my view class with data members corresponding to the elements in the original XAML file.

The ability to import data into a translation unit removes the need for additional tooling from the build. The fact that annotations and metaprograms are used to explicitly generate content from imported data means that the user has as much control over what gets generated as the framework would allow.

9.3.7 Embedding data

External resources can be used to embed binary data into programs [42]. P1045 introduces features for embedding binary data in programs. The `std::embed` function reads a file at compile time and returns a span of bytes over a binary object, which is compiled as part of the translation unit. To manage build system dependencies, the `#depend` directive is used to enumerate loaded files. For example:

```
#depend “my/app/icons/home.png”
#depend “my/app/icons/person.png”

// ...
constexpr auto home = std::embed(“my/app/icons/home.png”);
constexpr auto person = std::embed(“my/app/icons/person.png”);
```

The variables home and person are spans over the data embedded into the translation unit by `std::embed`. We can use the module system to simplify the embedding of data into a program.

```
export module my.app;
import extern my.app.icons.home -> meta::data home;
import extern my.app.icons.person -> meta::data person;
```

The `meta::data` type is essentially a wrapper around a span of bytes that holds the content of its underlying resource. This allows the same kinds of uses as intended by P1045. This type is also a library type that needs to be implemented with compiler magic. In particular, the compiler must actually embed the resource into the translation unit, so the data is available at link-time.

Unfortunately, implementing this feature through the module system would make it unavailable in code that has not (or cannot) migrate to use to modules. The proposed design for `std::embed` and `#depend` would work reasonably well in those cases.

9.4 Shared resources

This feature does not directly account for resources shared by multiple translation units in parallel builds. While this isn’t an issue for readable resources, it is a significant concern for writable and mutable resources. Synchronizing access between multiple translation requires one of two solutions.

- The file could be locked during the translation of a unit that imports it. This is undesirable for one major reason: it potentially serializes a parallel build. It may be possible to design resources that perform fine-grained locking (e.g., only around writes), although it’s not clear how portable that solution is.

- The resource could be an executable program that allows multiple simultaneous readers and writers (e.g., a database client). Note that reading and writing to such a client requires a non-trivial degree of interaction between the metaprogram running in the translation unit and the client servicing multiple connections. However, complex interactions like this could provide for some truly powerful metaprogramming capabilities. This could, for example, be used to automatically generate the entire data layer of an application directly from a running database instance.

The ability to use interactive clients as external resources will probably impose new requirements on tooling vendors, especially build systems. However, the addition of modules to C++20 has already required tool vendors to start rethinking how C++ builds should be structured. Hopefully, this feature will shoehorn onto the new build features needed to support C++20.

10 Runtime facilities

The last section of this paper deals with runtime metaprogramming, which could potentially be an equally large, independent paper. However, I discuss it here because I would very much like runtime metaprogramming facilities to relate to our compile time metaprogramming facilities.

Runtime metaprogramming has direct parallels for the two main features of compile-time metaprogramming: introspection and injection. Runtime introspection entails the ability to access and query properties of objects at runtime, especially their types. Injection entails the synthesis of new code (at runtime) and the injection of that code into the running binary.

At the time of writing, I don't have a clear picture what language support for runtime metaprogramming should look like. But, because of the parallels to compile-time metaprogramming, I suspect the libraries should be reasonably similar. It might be jarring for users to switch between such closely related features with completely different interfaces. The following sections discuss use cases for the different aspects of runtime metaprogramming.

10.1 Dynamic reflection

I think there are essentially three things I'd want from the dynamic type of an object.

- The ability to access the static type of an object
- The ability to compare and hash types
- The ability to inspect properties of types (data members, member functions)
- The ability to create objects from a reflected type

This is reasonably straightforward to implement. In fact, I've started building a library called *nemesis* that does exactly this [43]. It provides a simple function template `reflect`, which, when instantiated, produces an object describing the type of its argument. At the time of writing, that object also contains "descriptors" for each member of the class it describes.

This minimal set of features has some useful applications.

- Dynamic objects are similar to `std::any` except that they provide operations that directly support data member access and member function invocation. The ability to overload `operator.` would provide a better interface for those operations.
- Customizable polymorphism would allow class hierarchies to customize the storage and layout of their vtables. Here, compile-time metaprogramming can be used to annotate the

root of a hierarchy to automate the construction of and layout of vtables, which would naturally store the instantiated static type information. This can be used to support access to the dynamic type of an object and checked conversions.

- Partially open multimethods can be implemented on top of the ability to access the static and dynamic types of objects and their associated type hierarchies. Essentially, we could statically construct a dispatch table from its parameter types, using the dynamic types of objects as keys for lookups. The complete hierarchy must be known to implement the transformation, which imposes some requirements on the software's design. Making this work with DLLs is also a problem.
- Type factories for plugins become nearly trivial to implement. We simply need a method of discovering which types are provided by a DLL, and then use our library to invoke constructors to allocate objects.

There are almost certainly more applications that can be built from these simple examples.

10.2 Dynamic code injection

Dynamic injection includes the abilities to synthesize new code and incorporate it into a running executable. As with compile-time injection, there are two basic approaches to synthesizing new code: use pre-written patterns and instantiate them, or programmatically construct new code as e.g., abstract syntax trees (as in Section 8.7).

P1609 proposes a version of the first approach, which is essentially dynamic template instantiation [44]. The feature is reasonably straightforward. An interface is provided for building template arguments as runtime values. These arguments can be supplied to a dynamic template instantiation operator which instantiates a function template with those arguments. The resulting instantiation is made available to the host environment as a function pointer.

P1609 is thus far limited to the injection of new functions. It seems reasonable to also allow the instantiation of new classes.[25] Those could return runtime class information objects, which can be used to create objects of the dynamically injected type.

A more general approach would allow for the programmatic construction of entire functions via a tree-building library. An application could construct a tree representing a C++ function, ship that off to the compiler for translation and load the resulting code back into the function. Note that P1609 already requires a subset of this functionality: the template arguments for dynamic instantiation are constructed programmatically.

11 Implementation experience

There is a significant amount of implementation experience for many of the features discussed sections 5, 6, 7. In particular, much of the static reflection and source code injection work has been implemented Clang by myself and Watt Childers, including basic support for metaclasses.[26] However, we have not yet started work on more general abstraction facilities (e.g., macros) or compile-time file I/O. I understand that these features are concurrently being prototyped in EDG's frontend.

---

[25] I'm not entirely sure what use cases there are for dynamically creating and loading types. Presumably, there are use cases because other languages (notably C#) support the ability to do this.

[26] https://github.com/lock3/meta

12 Roadmap

Standardizing language support for metaprogram will require a significant effort by many people. I'd like to see static reflection land in C+23. For that to happen, I think three things have to happen:

- Adopt P1858, generalized pack expansions. The core language model for introducing new, nondependent packs is needed for static reflection.
- Update and adopt P1306. Expansion statements make certain aspects
- Update P1240 and send it to a combined EWG/LEWG session for adoption, and plan for concrete review in their respective committees.

Despite being short list, this is still a very tall order.

The rest of the functionality discussed in this paper can be targeted toward C++26 or later. I will update this roadmap in future versions of this paper to record progress made and future goals.

13 Conclusions

This is a big paper that covers a lot of ground. Much of this paper—especially Sections 3 to 7—describes features that have already been proposed in other papers, although with somewhat different syntax and possibly different semantics. Section 8 (Abstraction Mechanisms) also pulls in ideas from previous proposals, but ensures that they are firmly rooted in a the metaprogramming system described in Section 3. Section 9 (Compile-time I/O) is largely new, especially the ideas around external resources. These ideas needed to be fleshed out so they wouldn't be left behind. I also think that the ability to directly incorporate external data into a program is arguably one of the most powerful features presented in this paper. More work is needed on runtime metaprogramming, but I am strongly of the opinion that that work cannot advance independently of compile-time metaprogramming.

The metaprogramming system in this paper combines many different features from different sources and ideas into what I think is a coherent design. However, making this design a reality will take a lot of time and work. That said, we now have at least one complete reference picture to help us (the committee) proceed.

14 Acknowledgements

Thanks Wyatt Childers who has been the sounding board for most of the ideas presented in this paper. Tyler Sutton also helped sketch the design for compile-time file I/O. Additional thanks to Barry Revzin, Daveed Vandevoorde, Faisal Vali, and Roland Bock. Thanks to Jeff Chapman, Sam Goodrick, and Laura Bonko for additional edits and comments.

15 References

Note that the bibliography generator does not produce document numbers for WG21 papers. That will be fixed in a future revision of the paper.

[1] A. Sutton and D. V. Faisal Vali, "Scalable Reflection in C++".

[2] A. Sutton and W. Childers, "Compile-time Metaprogramming in C++," 2019.

[3] D. Vandevoorde, N. M. Josuttis and D. Gregor, C++ Templates: The Complete Guide, 2 ed., Addison-Wesley Professional, 2017, p. 832.

[4] T. Veldhuizen, "Using C++ Template Metaprograms," *C++ Report,* vol. 7, no. 4, pp. 36-43, May 1995.

[5] D. Abrahams and A. Gurtovoy, C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley Professional, 2004, p. 390 .

[6] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley Professional, 2001, p. 360 .

[7] G. Dos Reis, B. Stroustrup and J. Mauer, "Generalized Constant Expressions," 2007.

[8] R. Smith, "Relaxing Constraints on constexpr Functions," 2013.

[9] D. Sankel, "C++ Extensions for Reflection," 2020.

[10] H. Sutter, "Metaclasses: Generative C++," 2018.

[11] L. Dionne, R. Smith and D. Vandevoorde, "More constexpr containers," 2019.

[12] A. Sutton and H. Sutter, "Implementing Language Support for Compile-time Metaprogramming," 2017.

[13] R. Smith, A. Sutton and D. Vandevoorde, "Immediate Functions," 2018.

[14] D. Vandevoorde, "Side Effects in Constant Evaluation: Output and consteval Variables," 2019.

[15] A. Sutton, "Translation and evaluation: A Mental Model for Compile-time Programming," 2018.

[16] B. Revzin, "Generalized Pack Declaration and Usage," 2020.

[17] B. Revzin, "Structured Bindings Can Cntroduce a Pack," 2019.

[18] A. Sutton, S. Goodrick and D. Vandevoorde, "Expansion Statements," 2019.

[19] Wikipedia, "Duff's Device," [Online]. Available: https://en.wikipedia.org/wiki/Duff%27s_device. [Accessed 09 08 2020].

[20] N. Lesser, "Constexpr Structured Bindings," 2019.

[21] D. Stone, constexpr Function Parameters, 2019.

[22] M. Chochlík, A. Naumann and D. Sankel, "constexpr reflexpr," 2019.

[23] D. Sankel and D. Vandevoorde, "User-friendly and Evolution-friendly Reflection: A Compromise," 2019.

[24] A. Sutton and W. Childers, "Constraint Refinement for Special-cased Functions," 2020.

[25] M. Chochlík, A. Naumann and D. Sankel, "Static Reflection: Rationale, Design, and Evolution," 2017.

[26] M. Chochlík, "Static Reflection," 2014.

[27] M. Chochlík, "Mirror Reflection Utilities," 08 07 2020. [Online].

[28] G. Bracha and D. Ungar, "Mirrors: Design Principles for Meta-level Facilities of Object-oriented Programming Languages," in *Object-oriented Programming Languages, Systems, and Applications (OOPSLA)*, Vancouver, Canada, 2004.

[29] M. Naydenov, "Reflection Naming: Reification," 2020.

[30] M. Naydenov, "Reflection Naming: Fix reflexpr," 2020.

[31] M. Chochlík, A. Naumann and D. Sankel, "Function Reflection," 2018.

[32] M. Pusz, G. Ažman, B. Gustafsson and C. MacLean, "Univesal Template Parameters," 2020.

[33] A. Sutton and W. Childers, "Tweaks to the Design of Source Code Fragments," 2020.

[34] C. Jabot, "Reflection-based Lazy Evaluation," 2020.

[35] D. Stone, "constexpr Function Parameters," 2019.

[36] J. Rice, "Parameteric Expressions," 2018.

[37] C. Jabot, "Strongly-typed Reflection on Attributes," 2020.

[38] G. Dos Reis and B. Stroustrup, "A Principled, Complete, and Efficient Representation of C++," *Mathematics of Computer Science,* vol. 5, pp. 335-356, 2011.

[39] G. Dos Reis, "IPR," [Online]. Available: https://github.com/GabrielDosReis/ipr. [Accessed 07 2020].

[40] H. Dusíková and B. A. Lelbach, "Custom Constraint Diagnostics," 2018.

[41] D. Vandevoorde, std::constexpr_trace and std::constexpr_assert, 2017.

[42] J. Meneide, "std::embed".

[43] A. Sutton, "nemisis," 2020. [Online]. Available: https://gitlab.com/lock3/nemesis. [Accessed Jul 2020].

[44] H. Finkel, "C++ Should Support Just-in-Time Compilation," 2020.

[45] B. Stroustrup and A. Sutton, "A Concept Design for the STL," 2013.

[46] A. Sutton and H. Sutter, "A Design for Static Reflection," 2017.

[47] S. Murzin, M. Park, D. Sankel and D. Sarginson, "Pattern Matching," 2020.

[48] B. Stroustrup, Design and Evolution of C++, Addison-Wesley Professional;, 1994, p. 480.

[49] J. Meneide, "Module Resource Dependency Propagation".

[50] G. Dos Reis, J. D. Garcia, J. Lakos, A. Merideth, N. Meyers and B. Stroustrup, "Support for Contract-based Programming in C++," 2018.

[51] J. Coe and R. Orr, "Extension methods for C++," 2015.