Document P0712R0
Date 2017-06-18
Audience SG-7
Reply-to Andrew Sutton <asutton@uakron.edu>
Herb Sutter <hsutter@microsoft.com>

Implementing language support for compile-time metaprogramming

Table of Contents

1 Introduction

This paper describes our experiences over the past year experimenting with and implementing metaclasses for C++: a facility for defining new class-type abstractions (e.g., `interface`). That work, described in P0707R0, requires compile-time evaluation, static reflection, and programmable code synthesis or injection. In other words, to implement metaclasses, we had to implement everything else. This also means that we had to design a number of new features from scratch, consider their impact, and in several cases, throw them away and start over. This is not a proposal—those will come later.

This paper is presented semi-chronologically and in a bottom-up style. We worked towards a definition of metaclasses, not from them.

Here is a brief summary of our conclusions from this work:

1. Current approaches (P0385 and P0590) to static reflection are inherently flawed. The one-to-one mapping of reflection to class consumes a lot more resources in the compiler than is desirable. For efficient computation involving metaprogramming, static reflection must be as cheap as possible.
2. Vendors should informally agree on a common set of compiler intrinsics to support reflection.
3. Source code injections with tokens does not solve our metaprogramming problems and has serious name binding issues.
4. Source code injection is potentially transformative. Our work barely scratches the surface of this feature; we suspect it will be a rich source of discussion and future experiments and proposals.
5. Metaclasses are an abstraction mechanism based on source code injection, but there are some sticky issues in how they should be applied to create new classes.

2 Static reflection

The ultimate goal of metaclasses is to modify class definitions based on the contents of an original class definition. This is not possible to do without some ability to reflect on—to access the compile time information about—members of the original class definition. Even as a standalone feature, reflection is a significant new feature for C++. We experimented with two different approaches.

2.1 Reflecting objects

Although there had been an existing proposal for static reflection by Matus and Axel. (P0385R0 and P0194R1), our initial design called for a totally different approach. P0385R0 was rooted in template metaprogramming; the reflection operator returns a class type whose static members describe properties of the reflected entity. Our proposal called for a reflection operator that returned objects, not types. This section discusses our first approach to static reflection and the issues encountered.

For example, testing if a name is a function can be done like this:

```
$x.is_function()
```

The expression `$x` yields some class-type object and its member `is_function()` would return true if `x` is indeed a function of some kind. In our current compiler, we also accept `reflexpr(x)` as an alternative spelling of `$x`.

The `$` operator is designed to accept an *id-expression*, a *type-id*, or a *namespace-name* as an operand. This allows reflection to be applied to any named entity in the language. In our initial implementation, the expression `$x` expands to the expression `cppx::meta::reflection{p}` where `p` is the integer

representation of an internal AST node pointer for x. In other words, we simply replaced the original expression with an expression to construct an object whose data member(s) refer to the reflected entity.

The reflection class is fairly straightforward, but its interaction with the compiler is a little more interesting. Here is a partial definition of the `reflection` class and its `is_function` member.

```
struct reflection {
  std::intptr_t node; // The AST node pointer

  constexpr bool is_function() const {
    return __reflect_is_function(node);
  }
};
```

The member function is `constexpr` because we expect to evaluate it at compile time; deferring evaluation until runtime would require us to emit AST information in order to compute the query. The definition invokes an intrinsic whose value is true of false, based on the value held in `node`. The design of the intrinsic interface is discussed in Section 2.5.

Throughout the remainder of this document, we refer to any expression whose type is the result of the reflection operator as a *reflection*. In a section below, the type of the reflection operator depends on the entity reflected. This term applies in both contexts. Certain constructs, especially in later sections, discuss declaration reflections (a reflection of a variable, function, or class), type reflections (a reflection of a type name), and namespace reflections (a reflection of a namespace).

The approach seems very straightforward, until you write a simple little test program:

```
int main() {
  auto x = $main;
  std::cout << x.is_function() << '\n';
}
```

This didn't work out quite the way we would have liked: the compiler crashed when trying to generate code for the `is_function` member function. This is because we didn't implement code generation behavior for the `__reflect_is_function` intrinsic (oops).

On the other hand, you can't actually implement it. The problem you run into is that, at the time we try to generate code for that expression, there is no AST information to evaluate! We just have a `this` parameter. To make this work, we have two options:

1. Emit AST information as part of an object reachable from `this`, which allows reflection queries to be evaluated at runtime.
2. Add a new annotation to the language that guarantees a function is evaluated at compile time.

Adopting #1 has serious issues. In particular, we would be implementing dynamic reflection. Our goal of manipulating class definitions at compile time relies on our ability statically compute properties of declarations. Therefore, we adopted solution #2.

2.1.1 Immediate functions

To make this work, we added a new declaration specifier immediate, which can be used with (or without) constexpr to ensure compile-time evaluation of function calls. Here's the new definition of the `reflection` class.

```
struct reflection {
  std::intptr_t node; // The AST node pointer
```

```
  immediate bool is_function() const {
    return __reflect_is_function(node);
  }
};
```

The `immediate` function specifier alone implies `constexpr`. We can also write function this way:

```
immediate constexpr bool is_function() const {
  return __reflect_is_function(node);
}
```

They have the same meaning. However, now our simple example fails to compile.

```
int main() {
  auto x = $main;
  std::cout << x.is_function() << '\n'; // error: x is not constant
}
```

This is better. If the arguments to `is_function` are not constant (as here), then the expression is not a constant expression, and the program is ill-formed. Note that declaring `x constexpr` will make the program well defined.

2.1.2 Immediate types

One of the more peculiar aspects of this extension is it allows the `reflection` type to be used as a function parameter:

```
void f(reflection x);
```

That might seem reasonable until you realize that `f`'s definition might live in a different translation unit, which implies that AST data is being passed between functions at runtime. That is unlikely to yield a meaningful program.

We didn't implement this, or even discuss it much, but it seems imminently useful introduce a new kind of literal type: an *immediate type*, which can only be used as a parameter of an `constexpr` or possibly `immediate` function. However, we did not pursue that design at the time.

2.1.3 Compile-time string constants

Returning strings from immediate functions has proven to be difficult, and is in fact, one reason we moved to a different approach. One of the simplest metaprograms that somebody can write (and will) is to simply print the name of a type. Like this:

```
std::cout << $x.name() << '\n';
```

Unfortunately, implementing that turned out to be somewhat tricky. Here is a potential implementation of the `name` member function.

```
struct reflection {
  immediate const char* name() {
    return __reflect_name(node);
  }
};
```

So far so good… But what happens when you evaluate `__reflect_name`? It needs to return a compile-time lvalue that points to a character array containing the characters of the name. The question that needs to be answered is: where is that array stored?

We did not have a good answer to this question at the time. However, Richard Smith suggested that we could add a new string literal to a cache in the AST context (Clang's "global" repository of AST information) that could be emitted as string constants in the resulting object file. This approach has a nice property that only names requested would be added to the object file. We don't need to proactively dump every identifier in the translation unit into the object file. That would be madness.

2.1.4 Conclusions

Ultimately, we moved away from this approach. There were some deep questions about the library's design. In particular, if $ has a single type, how does a metaprogram determine what set of operations are valid for a reflection. For example, the set of queries on a variable are different than those on a class.

We eventually decided to work in a direction that was more in line with the proposal P0385 proposal, although retaining the object-like syntax.

2.2 Reflecting class objects

The approach we ultimately adopted for static reflection is to cause the reflection operator to generate reflection objects whose type is determined by the kind of entity reflected. In particular, the type of each reflection is a class template specialization whose template argument is the encoded AST node pointer.

For example:

```
void foo(int n) {
  int x;
  auto r1 = $int; // r1 has type meta::fundamental_type<X>
  auto r2 = $foo; // r2 has type meta::function<X>
  auto r3 = $n; // r3 has type meta::parameter<X>
  auto r4 = $x; // r4 has type meta::variable<X>
}
```

In contrast to the approach above, we simply moved the node pointer from reflection's state to its type. For example, here is a possible implementation of the `meta::variable` class.

```
template<std::intptr_t X>
struct variable {
  static constexpr const char* get_name() {
    return __reflect_name(X);
  }
  static constexpr auto get_type() {
    return __reflect_type(X);
  }
  // ...
};
```

Each reflect property is a static `constexpr` member function that evaluates some compiler intrinsic. When called the member function is instantiated, and the intrinsic is replaced by an expression that represents the computed value. In the case of get_type, the return type is deduced from the type of the intrinsic. That function could return `fundamental_type`, `class_type`, etc.

Note that all questions about compile-time evaluation fall away in this model. Because the intrinsics expand to expressions during instantiation, they implicitly compute their values at compile time. This means, as shown in the example above, that reflections can be treated as normal objects; they don't need to be `constxpr` variables. The net result of this approach is that it makes metaprogramming look and

feel just like normal programs, which we felt was a Good Thing. In retrospect, however, this doesn't quite live up to expectations.

This approach is equivalent to that described in P0385. The primary differences are a) how the node pointer is encoded, and b) the syntax used to access properties. Otherwise, the core of both approaches can be considered equivalent. This is well-documented in P0590.

2.2.1 Heterogeneous collections

One of the biggest problems we encountered in this approach is the representation of collections. The original metaclass proposals require us to iterate over the members of a class in order to compute properties and manipulate definitions. However, because each member of a class may have a different kind and therefore reflection, the set of members defines a heterogeneous container. Programming with heterogeneous has not typically been for the faint of heart, although there are few libraries that make it much easier (e.g., Louis Dionne's Boost.Hana).

Rather than computing a `std::tuple` directly from the members of a class, we simply exposed enough compiler information to implement overloads of get and a specialization of `tuple_size`. In particular, the we rely on a pair of intrinsics:

- `__reflect_num_members(X)` expands to the number of members in the AST node X.
- `__reflect_member(X, I)` expands to the Ith member of X.

These are used to define a simple tuple-like class that can be used with e.g., Boost.Hana. However, the programming model with heterogeneous containers can still be non-intuitive and can require some template metaprogramming.

2.2.2 Filtered tuples

The biggest pain point of this approach is filtering or selecting a subset of members. For example, most metaprograms don't want to look at all members of a class; they may want to examine just the member functions or member variables. We implemented this as a wrapper around our simple compile-time tuple class.

Unfortunately, the implementation has quadratic complexity when iterating elements of a tuple, which is not particularly good. In fact, it is particularly awful. It is actually possible to observe (visually) the slowdown caused by the algorithm's quadratic behavior when iterating over even relatively small lists (e.g., 20 members or so).

There may be better approaches to implementing a filtered tuple, or there may simply be a bug in the current implementation. However, creating a `std::tuple` have been a better choice for the implementation since that can be done in linear time.

2.2.3 Expansion statements

In order to simplify the programming model, we designed a new language feature that would allow us to "iterate" over the elements of a tuple. For example, we can print the names of each member like this:

```
for... (auto m : $C.members())
  std::cout << m.name();
```

The "loop" expands to a set statements that would print the name of each member in turn. That idea is presented in P0589, although `for` loop does not include the ellipsis.

While this feature does make it easier to implement certain algorithms on containers, it is incomplete. In particular, it was determined that we need additional mechanisms better control substitution in the body of templates.

2.2.4 Conclusions

In general, this approach yields a reasonable programming style for metaprogramming. Improvements could probably be found for working with heterogeneous collections and their subsets.

That said, C++ cannot continue with either this approach to static reflection or the equivalent approach described in P0385. Both approaches require the introduction of a new class for each unique reflection. Our approach does this by instantiating a template, the P0385 approach does this by internally stamping out new classes.

The problem with these approach is that classes are expensive data structures inside the compiler; they require a significantly larger amount of memory than almost any other object within the compiler. Consider that each class has a set of members that is never empty (e.g., injected class name). Building classes is also not a free operation. The compiler must select and select and/or generate special member functions which can involve overload resolution. Finally, classes never go away. Once synthesized, they must be maintained for the duration of the translation.

These are the reasons that compiling translation units involving significant template metaprograms is slow. Static reflection and other elements of this work have the potential to replace most (many? all?) uses of template metaprogramming. We have an opportunity to significantly improve compile-time performance, but we cannot do that using these approaches.

2.3 Deflection

The reflection operator lets us get information about an expression or entity. However, we also want to go the other way: from a reflection to an entity. How you do this depends on the kind of entity.

For reflections of (static) variables and functions, you may want a pointer to that object. For such objects, that associated value can be accessed by writing `$x.pointer()`. For reflections of enumerators, you might want the value, which can be accessed by writing `$e.value()`. That could likely be extended for any constexpr variable.

In order to interoperate with other parts of the language (e.g., generating a type name), we need additional facilities. We designed 4 and implemented 3. The `typename` and `namespace` specifier were motivated by discussions with Daveed Vandevoorde.

2.3.1 The `declname` id and `hasname` operators

The `declname` id is a new kind of *unqualified-id* that transforms its operands into an *id-expression*.

unqualified-id:
    identifier
    operator-function-id
    conversion-function-id
    literal-operator-id
    ~ class-name
    ~ decltype-specifier
    template-id
    declname-id

declname-id:
    `declname` (*id-component-seq*)

id-component-seq:
    id-component-seq id-component

id-component

id-component:
constant-expression

The `declname` operator takes a sequence of constant expressions, evaluates them, transforms them into strings, and concatenates them. The components of an id can be:

- string literals,
- integers, and
- a declaration reflection.

When the id-component is a declaration reflection, its unqualified name is appended to the id.

There is no operator separating the operands of a sequence. Adjacency is interpreted as concatenation. Note that each name component can involve arbitrarily complex computations.

When used as an expression, the *id-expression* names the object that it refers to. This can be used, for example, to call a function through its reflection.

```
void foo() { ... }
void foo_bar() { ... }
void g() {
  auto x = $foo;
  return declname(x "_bar")();
}
```

In the return statement, `declname(x "_bar")` yields a reference to the function `foo_bar`, which is then called.

When used within a declaration, the `declname` operator generates an *id-expression* that becomes a *declarator-id*. This can be used to generate declarations with new names. For example:

```
void declname($foo "_" 2)(int n);
```

This generates a function declaration with the name `foo_2`.

Note that this is closely related to the `idreflexpr` operator in P0385R1 and `$identifier` operator in P0385R2. The operator name `declname` is largely a placeholder for a better name. The name `idexpr` is a reasonable alternative.

The `hasname` operator is closely related to the `declname` operator, and allows a programmer to determine if a declaration has a particular name. It takes two operands: an *id-expression* referring to a declaration and *unqualified-id* representing the name being tested for.

postfix-expression:
...
`hasname` (*id-expression* , *unqualified-id*)

The first operand refers to a declaration, and the second is an unresolved-id (lookup is not performed). The expression is true if the referenced declaration's unqualified name matches the *unqualified-id*.

The `hasname` operator is largely a workaround for the lack of support for compile-time strings and string manipulation. We would prefer something more elegant.

2.3.2 The typename specifier

The `typename` specifier is a *simple-type-specifier* used to generate a type from a reflection. It has the syntax:

simple-type-specifier:
...
typename-specifier

typename-specifier:
`hasname` (*id-expression* , *unqualified-id*)

It accepts a type reflection and is the type reflected by that expression. For example:

```
struct S { };
typename($S)* s; // equivalent to 'S* s';
```

The `typename` specifier should also be available as part of a *nested-name-specifier*, although we have not yet specified this.

An alternative approach was to define a nested type member within the reflection. That would have allowed this:

```
using X = typename decltype($int)::type;
```

We chose to extend the language for obvious reasons.

2.3.3 The namespace specifier

The namespace specifier is a form of qualified-id used to generate a scope as part of another name. In other words, it acts as a computed nested-name-specifier. It has the syntax:

qualified-id:
nested-name-specifier `template`$_{opt}$ *unqualified-id*
`namespace` (*constant-expression*) `template`$_{opt}$ *unqualified-id*

The namespace specifier is replaced by the qualified id of the reflected namespace. For example:

```
auto estd = $std::experimental;
using foo = namespace(estd)::optional<int>;
```

This is not yet implemented.

2.4 Library support

Static reflection is not just a language feature; there is a non-trivial library component as well. In particular, we define a set of classes that are instantiated by the reflection operator. The library consists of the set of class templates instantiated by the reflection operator, based on the kind of entity reflected.

Entity	*Reflection type*
Variable	`meta::variable<X>`
Member variable	`meta::member_variable<X>`
Function	`meta::function<X>`
Constructor	`meta::constructor<X>`
Destructor	`meta::destructor<X>`

Member function	`meta::member_function<X>`
Conversion operator	`meta::conversion<X>`
Function parameter	`meta::parameter<X>`
Enumerator	`meta::enumerator<X>`
Class type	`meta::class_type<X>`
Union type	`meta::union_type<X>`
Enum type	`meta::enum_type<X>`
Fundamental type	`meta::fundamental_type<X>`
Qualified type	`meta::qualified_type<X>`
Namespace	`meta::ns<X>`
Translation unit	`meta::tu<X>`

Note that this list only represents the current set of reflections. It is likely that this list will grow.

The properties of reflected entity depend on its declaration, its definition, the language, and compiler options. In general, there are three kinds of information that can be requested of any reflection

- *Specifiers* are flags and values that indicate how a declaration was written.
- *Attributes* correspond to the written C++ attributes of a declaration.
- *Traits* are the computed from specifiers, attributes, and language rules.

The implementation does not currently support queries for specifiers or attributes. Only queries for traits are supported. We will eventually want to add support for written specifiers and attributes.

Reflection classes have no non-static data members. All properties are defined as static member functions and variables. The properties of a reflection class depend on the entity they reflect. These can be grouped into concepts, defined by the table below.

Concept	*Members*

`NamedEntity`	`const char* name()`
	`const char* qualified_name()`
	`ScopeEntity declaration_context()`
	`ScopeEntity lexical_context()`
	`linkage_t linkage()`
	`access_t access()`

`ScopeEntity`	`Tuple members()`
`Type`	`NamedEntity` `typename type;`
`UserDefinedType`	`Type, ScopedEnity`

`MemberType`	`bool is_complete()`
	`Tuple member_variables()`
	`Tuple member_functions()`

	Tuple constructors() Destructor destrutors()
ClassType	MemberType bool is_polymorphic() bool is_abstract() bool is_final() bool is_empty()
UnionType	
EnumType	UserDefinedType bool is_complete() bool is_scoped()
TypedEntity	auto type()
Variable	NamedEntity, TypedEntity storage_t storage() bool is_inline() bool is_constexpr() T* pointer()
MemberVariable	NamedEntity, TypedEntity bool is_mutable() T C::* pointer()
Function	NamedEntity, TypedEntity bool is_constexpr() bool is_noexcept() bool is_defined() bool is_inline() bool is_deleted() Tuple parameters() T(*)(...) pointer()
Method	NamedEntity, TypedEntity bool is_noexcept() bool is_defined() bool is_inline() bool is_deleted() Tuple parameters() T (C::*)(...) pointer()
PolymorphicMethod	MemberFunction bool is_virtual() bool is_pure_virtual() bool is_final() bool is_override()
Constructor	MemberFunction bool is_constexpr() bool is_explicit()

	`bool is_defaulted()` `bool is_trivial()`

`Destructor`	`PolymorphicMemberFunction`
	`bool is_defaulted()`
	`bool is_trivial()`

`MemberFunction`	`PolymorphicMethod` `bool is_constexpr()`
`ConversionFunction`	`MemberFunction` `bool is_explicit()`
`Parameter`	`NamedEntity, TypedEntity`
`Enumerator`	`NamedEntity, TypedEntity` `T value()`

Again, this is incomplete. As the proposals evolve, we will determine what properties are available in reflected source code.

2.5 Intrinsic API

The reflection library communicates with the compiler through a set of compiler intrinsics that compute the values of queries (e.g., `__reflect_name`). We went through several iterations of design for this API.

In each iteration, however, the first operand of these traits is always an integer constant containing an encoded AST node pointer. During evaluation, that argument is converted back into an AST node, and the corresponding query evaluated. That's the easy part.

The harder part is designing a balanced API so that:

- it is not an expansive new collection of intrinsics
- it does not create synchronization challenges between the compiler and library, and
- multiple vendors would be able to provide those facilities.

Naturally, it also needs to be complete.

The first approach was to simply create new intrinsics for each query. This is certainly the easiest design, but it does result in a lot of new intrinsics. It's not clear if that's good or bad.

A second design tried to minimize the interface by taking a second integer parameter that determined what query to run. For example, you could request the name of a node by writing:

```
__reflect(node, get_name)
```

The `get_name` expression is an enumeration value that tells the compiler to return the name. This approach does not work with the reflection model described above, because the type of expression depends on the value of the 2nd argument. Moreover, it requires the library and the compiler to agree on the mapping of "query selectors" to values, which we found to be burdensome in practice.

The current design is a closer to the first. We have a small set of intrinsics that return computed values for nodes. When instantiated, these are replaced by an expression representing the computed value.

Intrinsic	Description
`__reflect_name(x)`	Expands to a string literal containing the unqualified name of an entity

`__reflect_qualified_name(x)`	Expands to a string literal containing the qualified name of an entity.
`__reflect_traits(x)`	Expands to an integer literal that encodes the computed traits of an entity. The encoding depends on the kind of entity, but includes linkage, storage, accesss, whether defined, whether virtual, etc.
`__reflect_num_members(x)`	Expands to an integer literal determining the number of members in a declaration (class, namespace, etc.).
`__reflect_member(x, i)`	Expands to a reflection of the i^{th} member of a declaration. The type of the reflection is determined by the kind of the member.
`__reflect_num_parameters(x)`	Expands to an integer literal that determines the number of parameters of a function.
`__reflect_parameter(x, i)`	Expands to a reflection of the *i*th parameter.

3 Source code injection

One of the other pillars of the metaclass proposal is the ability to generate or inject code. We addressed this issue by creating a general-purpose facility for source code injection, which takes on two components: the ability to execute `constexpr` in (nearly) any context, and the ability to specify what code gets injected.

The goal (eventually) was to identify and design small features that could more easily support the notion of metaclasses. The two features discussed in this section define the core feature set needed for metaprogramming in general, not just metaclasses.

3.1 Constexpr blocks

A constexpr block allows the execution of compile-time code in namespace, class, and block scope. Here is an example of a namespace-scoped `constexpr` block.

```
constexpr {
  for... (auto x : $X.members())
    // Do something with x
}
```

The constexpr keyword is followed by a *compound-statement*. Those statements are executed at when the closing brace is reached.

Internally, this is parsed as if it were a function:

```
constexpr void __unnamed_fn() {
  for... (auto x : $X.members())
    // Do something with x
}
```

That function is immediately evaluated as if initializing a `constexpr` variable, like this:

```
constexpr int __unnamed_var = (__unnamed_fn(), 0);
```

Class-scoped `constexpr` blocks have similar semantics, except that the synthesized functions and variables are also static.

```
struct Foo {
  constexpr {
    for... (auto x : $X.members())
      // Do something with x
  }
}
```

Again the block is executed at the closing brace of the *compound-statement*.

Block scope `constexpr` are a little different; they are internally modeled as lambda functions with no capture. For example, this code

```
void f() {
  constexpr {
    /* constexpr block body */
  }
}
```

Is approximately equivalent to this:

```
void f() {
  auto __lambda = []() constexpr { /* constexpr block body */ };
  constexpr int __var = (__lambda(), 0);
}
```

We had considered whether a non-empty capture list would be useful – and it might. However, it could only capture `constexpr` declarations. That still might be useful. We currently have no mechanism for controlling capture in a block-scoped `constexpr` block.

On the surface, this might not make much sense. Constant expressions can't have side effects, so it seems like there would be little value in supporting a feature that evaluates side-effect-free `void` functions. That changes when we start injecting source code.

3.2 Injection statements

An injection statement defines a *fragment* of code at some point in a program, called the *injection site*. It has the following syntax:

injection-statement:
-> `namespace` *identifier*$_{opt}$ { *declaration-seq* }
-> `class` *identifier*$_{opt}$ { *member-specification* }
-> `do` *compound-statement*
-> { *expression* }

Each alternative defines a fragment of something that can be injected. A *namespace fragment* is a sequence of namespace-scoped declarations to be injected. A *class fragment* is a sequence of class-scoped declarations to be injected. A *block fragment* is a sequence of statements to be injected. A *statement fragment* is a single expression to be injected.

These fragments are parsed and analyzed. The keyword is needed to tell the compiler how to parse the contents within the block. Namespace and class injections support an optional identifier, which can be used within the fragment for self-references. More on that later.

Injection statements can appear in a `constexpr` block. For example:

```
namespace N {
  constexpr {
    -> namespace { int f() { return 0; } }
  }
} // namespace N
```

This block contains a namespace fragment injection. When the block executes, the injection statement is queued as a kind of side effect of evaluation. Otherwise, the statement has no observable behavior. After execution completes (assuming no errors are encountered) queued injections are applied. The injection site is immediately after the `constexpr` block where they were queued.

Currently, source code injections are applied only at the end of a `constexpr` block; a program that produces injections from the evaluation of any other constant expression is ill-formed. However, this is not currently enforced by the compiler.

Source code injection transforms the fragment of code by substituting its original context with that of the `constexpr` block. In this case, no such substitutions are needed; a new version of the function f is injected into the namespace N. The resulting program is:

```
namespace N {
  int f() { return 0; }
} // namespace N
```

Consider another example:

```
constexpr void make_links() {
  -> class C {
    C* next;
    C* prev;
  }
}
struct list {
  constexpr { make_links(); }
};
```

Here, we have a class fragment injection within a namespace-scoped function. This is fine. In this case, we've provided a name for the fragment because we need to refer to the type of the enclosing class.

The `list` class contains a `constexpr` block that calls `make_links`. When that executes, the class fragment will be queued, and later applied at the closing brace of the `constexpr` block (the injection site). When the injection is applied, we transform the injected members, substituting `C` for `list`. The resulting class is:

```
struct list {
  list* next;
  list* prev;
};
```

Statement injection works similarly. Note that statement injections are parsed as compound-statements. The entire block is injected, creating a new scope. This is necessary to prevent nested declarations from leaking into the call site.

Here is a slightly more involved example that applies a polymorphic function object (with call operator overloads) to an object in a class hierarchy rooted at `expr`.

```
template<typename F>
decltype(auto) apply(expr* e, F fn) {
  switch (e->get_node_kind()) {
    constexpr {
      for (auto x : $expr::kind) {
        -> do {
          case x.value():
            return fn(static_cast<expr_type_t<$x.value>*>(e));
        }
      } // for
    } // constexpr
  } // switch
}
```

The body of the switch statement is a `constexpr` block. When executed, that block will inject a case statement for each enumerator in the hierarchy's discriminator type, `expr::kind`. Each statement invokes the function call operator on the node, statically cast to its most derived type.

This does assume an external facility, `expr_type_t`, which defines the mapping of enumerators to types. Note that this could also be generated (elsewhere) by injecting the type trait specializations that define the mapping.

When instantiated, the resulting specialization is:

```
struct print_fn {
  void operator()(expr* e) { } // Don't actually print
};

void apply<print_fn>(expr* e, print_fn fn) {
  switch (e->get_node_kind()) {
    case expr::bool_literal_kind:
      return fn(static_cast<bool_literal*>(e));
    case expr::int_literal_kind:
      return fn(static_cast<int_literal*>(e));
    ...
  }
}
```

Note that statements are embedded in block statements.

The complexity of the original example can be improved by factoring parts of the code generation into separate functions. For example:

```
template<typename F, Enumerator K>
constexpr void make_apply_case(expr* e, F fn, K kind) {
  -> do {
    case kind.value():
      return fn(static_cast<expr_type_t<kind.value()>*>(e));
  }
}
```

```
template<typename F>
constexpr void make_apply_cases(expr* e, F fn) {
  for... (auto x : $expr::kind.enumerators())
    make_apply_case(e, fn, x)
}

template<typename F>
decltype(auto) apply(expr* e, F fn) {
  switch (e->get_node_kind()) {
    constexpr { make_apply_cases(); }
  }
}
```

We have not implemented expression injection yet and are still exploring its design.

The implementation requires injections to match the context at the injection site. For example, you cannot inject statements into a namespace:

```
constexpr {
  -> do { while (false) ; }
} // error: applying statement-fragment in a namespace
```

There was some discussion in Kona that would allow more broadly scoped injections to "float" to an injection site in an enclosing context. At the time of writing, we determined to restrict injection to work only on matching contexts.

This construct gives a rich set of tools for programmatically generating source code. It is also ripe for extension and experiments. In particular, this feature is made dramatically more powerful if we allow injections to be named to capture or names from their enclosing contexts. We have just begun exploring those ideas.

3.3 Unparsed injection

Our initial attempt at source code injection was to treat the content in the braces as unparsed tokens. When the injection was applied, we would then parse those tokens based on the current context at the injection site. This proved to be enormously problematic, but there were really two language issues that make this undesirable:

It doesn't allow injected fragments to refer to (and depend on) declarations in an enclosing local scope. For example:

```
struct S {
  int a, b, c;
  constexpr {
    for... (auto x : $S.member_variables())
      -> { void declname("get_" $x.name())() const; }
  }
};
```

If the injection statement is simply a list of tokens, then the expression `$x` is not meaningful at the injection site. It would be as if we were trying to parse this class definition:

```
struct S {
  int a
  void declname("get_" $x.name())() const;
```

```
  void declname("get_" $x.name())() const;
  void declname("get_" $x.name())() const;
};
```

Clearly, this is not what we want. Instead, we need the `declname()` operator to be applied at the time we parse the fragment. In other words, we need this to emit:

```
struct S {
  int a
  void get_a() const;
  void get_b() const;
  void get_c() const;
};
```

We tried to fix in the context of metaclasses by trying to define a token replacement for certain constructs, but this turned out to add other problems. In particular, this replacement allows names to be unintentionally captured if the happen to share a name with nominated replacement.

In the face of these issues, we moved to the parsed fragments described above.

4 Compiler interaction

Certain features of the metaclass facility require more interaction with the compiler. We need better support for generating compile-time diagnostics and debugging metaprograms and metaclasses. Our work here is very immature; much more can be done in this space.

4.1 Diagnostics

Our metaclass design requires that we be able to emit user-defined error diagnostics. We implemented a minimal interface for emitting these diagnostics. These are packaged in a `compiler` object. For example, emitting a diagnostic works like this:

```
compiler.error(loc, "some error message");
```

Unsurprisingly, the compiler emits "some error message" at the indicated location. At least, it would if source code locations were actually implemented. The current implementation omits the location parameter.

Under the hood, this function invokes a compiler intrinsic that accepts the given arguments

```
struct compiler_type {
  static constexpr void
  error(source_location loc, const char* msg) {
    __compiler_error(loc, msg);
  }
} compiler;
```

The semantics of `__compiler_error` are interesting. First, it is syntactically allowed within a constant expression. When executed, the program aborts. This means that any compile-time evaluation that reaches this expression will stop evaluating. At that point, the implementation is required to emit a diagnostic containing the given message.

We also provide a simple assert-like wrapper:

```
compiler.require(loc, cond, msg);
```

This calls `error(loc, msg)` if `cond` is false.

4.2 Debugging

The ability to inject arbitrary code allows source code to be constructed programmatically. Programmers need some mechanism to debug the output of these programs. We added a small new feature that allows the compiler to emit the generated code. This is also invoked through the compiler object.

```
struct S { };

constexpr {
  compiler.debug($S);
}
```

When executed, the compiler pretty prints the reflected declaration to standard error.

The implementation is similar to that of compiler error.

```
struct compiler_type {
  template<Reflection T>
  static constexpr void debug(T refl) {
    __compiler_debug(refl);
  }
```

It invokes an intrinsic, passing a reflection object as an argument. Behaviorally, the expression has no effect.

5 Metaclasses

Having established all of this infrastructure, we can now easily describe metaclasses. Here is one of our motivating examples:

```
$class interface {
  virtual ~interface() = default;

  constexpr {
    for... (auto x : $interface.member_functions()) {
      x.make_pure_virtual();
      x.make_public();
    }
    compiler.require($interface.member_variables().empty(),
                     “an interface cannot have member variables”);
  }
}
```

We represent interfaces as a named class injection. The interface metaclass is a fragment to be injected into its subscribing class. But, it does have some other properties.

A metaclass is applied to a class declaration or definition by using its name instead of the traditional class key (`class` or `struct`). For example:

```
interface IFoo {
  void foo();
  void bar();
};
```

We call the written definition of IFoo the *prototype*. The metaclass modifies the contents and semantics a prototype to produce a new class. In particular, the compiler disables all default generation for the prototype, and using the metaclass to apply its own user-defined defaults by source code injection.

The metaclass is injected into the prototype just before the closing brace of the class definition. This means that its rules apply to all elements of the class. The resulting class is:

```
class IFoo {
  public: virtual void foo() = 0;
  public: virtual void bar() = 0;
  virtual ~IFoo() = default;
};
```

When a metaclass is injected, each declaration within the metaclass is injected into the prototype, including any `constexpr` blocks in the order of declaration, using the transformation process described above. When a `constexpr` block is injected, it is evaluated as if it had been written in place. The mechanics are straightforward and easy to describe.

5.1 Modifying declarations

There a small number of operations that can modify members of a class (e.g., `make_public`). These are implemented as compiler intrinsics. For example:

```
template<std::intptr_t X>
struct member_function {
  // ...
  static constexpr void make_public() {
    __make_public(X);
  }
};
```

These expressions are queued just like source code injections. They represent a request to change a declaration. The current implementation simply modifies the internal state of the node, and updates any extra information maintained by the class.

That said, modifying declarations is… tricky. Changing an access specifier is nearly trivial. Making a function virtual affects properties of the entire class: it becomes polymorphic or abstract and its virtual table is created or changed. Fortunately, that wasn't too hard in Clang.

Making a member variable static is very challenging, because Clang represents static and non-static member variables as different AST nodes. Making a member static would entail building replacing the old node with a new one. This seems particularly brittle and would require the specification to encode special rules for certain modifications (e.g., you can't make something static if you've already referred to it).

We also have serious concerns about implementability in other compilers.

This points to the idea that current approach is not, perhaps, the best solution.

5.2 Metaclass application

As noted, our implementation modifies a subscribing class in place; it does not produce a new class. This is largely due to the fact that we chose to model metaclasses as injections: that's just how they work. However, as mentioned above, modifying individual declarations may have serious technical restrictions.

We had originally considered an approach where we instantiate an entirely new class from the original, applying rules as needed. But the *name* of that class is somewhat problematic. It has been suggested to

name the prototype some internal name like `__blah_blah_blah`, and then to create the final class with the name given by prototype.

Unfortunately, this doesn't work well when self-references are used. For example:

```
interface IFoo {
  IFoo* get_foo();
};
```

We want to parse prototype as a normal class. After all, metaclasses affect class semantics, not their syntax. Ostensibly, resolved type names would point to the wrong class type: `get_foo` should return a pointer to the prototype, not the final type. Maybe that isn't a big deal because we're eventually going to replace that with a pointer to the final type during injection.

One alternative is to create define the prototype in a hidden and inaccessible namespace, and then to synthesize the final class at the original point of definition.

```
namespace look_away {
  interface IFoo { ... }; // prototype definition
}
// synthesize final IFoo here
```

This approach is actually very similar to how our compiler parses metaclasses. The "class part" is defined within the context of a "metaclass".

We had also considered synthesizing the final class on top of the original class: that is, replace the internal representation of the node with the new. That would preserve any external references to the class (i.e., uses of a forward declaration). However, this feels a little too sneaky.

The other issue to consider is, when creating a final class separate from the prototype is how modifications are ordered and when their effects are visible. For example:

```
$class mc {
  constexpr {
    for... (auto x: $mc.member_variables())
     x.make_static(); // Assuming this works
  }
  constexpr {
    compiler.require($mc.member_variables().empty(),
                     "too many member variables!");
  }
  int y;
}
mc some_class {
  int x;
}; // error?
```

In our current model, this is well-formed. The first `constexpr` block makes the member `x` static. That effect is visible after the before the execution of the second `constexpr` block. When that executes, there are no static members, so compilation succeeds. Finally, the non-static member variable `y` is injected.

We had considered other formulations of metaclass application semantics. One thought was to transfer all members first, then apply `constexpr` blocks. The opposite order was also discussed. However, re-ordering the application of injections makes it difficult for programmers to determine the ultimate

"shape" of the final class. This also makes metaclass rules different from the injection mechanism described above, which we find to be undesirable.

An interesting outcome of moving from in-place modification to transformation is that modifications (e.g., `make_virtual`) could only be used within metaclasses. Other injections do not "own" their contexts. This seems like a reasonable design choice.

6 Acknowledgements

This work would not have been possible without discussion with and feedback from WG21 committee members, in particular Daveed Vandevoorde. His metacode proposal has greatly influenced many of the features discussed here. The paper P0633R0 By Daveed Vandevoorde and Louis Dionne provides a good context for many of the features discussed in this paper. Chandler Carruth, and Richard Smith have also contributed ideas, limitations, and discussion that influenced the direction of this work. Special thanks to Jennifer Yao contributed significantly to the implementation and surfaced a number of technical issues related to our design.

This material is based upon work supported by the National Science Foundation under Grant No. 1422655.