Reflection and metaprogramming

Users guide

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:

- 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.
- Vendors should informally agree on a common set of compiler intrinsics to support reflection.
- Source code injections with tokens does not solve our metaprogramming problems and has serious name binding issues.
- 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.
- 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

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

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

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

The definition of equal can be defined by a simple algorithm:

```
template<typename T>
bool equal(const T& a, const T& b) {
```

```
  for... (auto member : reflexpr(T).member_variables()) {
    auto m1 = member.pointer();
    auto m2 = member.pointer();
    if (a.*m1 != a.*m2)
      return false;
  }
  return true.
}
```

There are a number of features to unpack. First, the `reflexpr` operator is the reflection operator. It applies to names and produces a value that can be queried for properties of the corresponding entity. Here, `reflexpr(T)` yields a class type object that for which we can query its member variables via `reflexpr(T).member_variables()`. This yields a *tuple* of objects that describe the members of the class.

Normally for loops don't iterate over tuples. However, that is not a normal for loop. This is an *expansion* loop; instead of iterating over the range of elements, the loop expands, so the body is instantiated for each element of the tuple (giving the illusion of iteration).

Within the loop `member` is a reflection of a member variable. The `pointer` function yields a pointer to the member function, which can be applied to the class object to access the actual value, thus letting us compare objects for equality.

2.1 The reflection operator

There reflection operator has four forms:

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

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

The result of the expression is prvalue of unspecified class type, that satisfies a concept described in the library section below. Each concept requires a set of operations that allow users to query for particular properties.

The value can be used with the projection operators (described below) and the meta library to query the semantic properties of reflected entities.

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

2.2 Reflection values

As noted, the reflection operator to generates class 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 std::meta::fundamental_type<X>
  auto r2 = $foo; // r2 has type std::meta::function<X>
  auto r3 = $n; // r3 has type std::meta::parameter<X>
  auto r4 = $x; // r4 has type std::meta::variable<X>
}
```

The AST node point is stored in the template argument X. Note that this fact—and the name of the class—is an implementation detail. The type returned by the implementation matches a concept, which are described in the following sections.

For exposition, however, here is a possible implementation of the `meta::variable` class.

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

Each reflected 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 `constexpr` 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.

2.3 Heterogeneous collections

One of the biggest problems we encountered in this approach is the representation of collections. Some reflected properties are sequences (e.g., the members of a class). 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., Boost.Hana).

Examples like e.g., `member_variables()`, above, return a tuple-like object that can be used to traverse the members of a class. The for expansion provides an improved interface for such traversals.

2.4 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.

2.5 Reification

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.

2.5.1 The typename operator

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

```
typename(reflection)
```

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

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

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

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

2.5.2 The `idexpr` operator

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

```
idexpr-id:
  idexpr ( id-component-seq )
id-component-seq:
  id-component-seq , id-component
id-component:
  identifier
  string-literal
  integer-literal
  character-literal
```

The `idexpr` 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.

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 idexpr(x “_bar”)();
}
```

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

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

```
void idexpr($foo “_” 2)(int n);
```

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

2.6 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.

Traits are the computed from specifiers, attributes, and language rules. The classes yielded by the reflection operator are implementation-defined, except that they conform to one or more of the concepts listed in the table below.
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.

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	ClassType
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() void make_static() void make_thread_local();
MemberVariable	NamedEntity, TypedEntity bool is_mutable() T C::* pointer() void make_public() void make_protected() void make_private() void make_static()
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() void make_public() void make_protected() void make_private()

PolymorphicMethod	MemberFunction
	bool is_virtual()
	bool is_pure_virtual()
	bool is_final()
	bool is_override()
	void make_virtual();
	void make_pure_virtual();

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

3 Source code injection

Reflection and reification alone is insufficient to support metaprogramming. We need the ability to inject constructs that are “bigger” than simply types, values, and ids. The following sections describe the features needed to support these more interesting applications.

3.1 Metaprograms

A metaprogram is a sequence of statements executed where it appears in the program *at compile time*. It is a block statement preceded by the `constexpr` keyword. 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 interpreted 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);
}
```

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; they are not simply sequences of tokens. 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.

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.