Document: **P1717R0**
Date: **2019-06-17**
Audience: **SG7**
Authors: **Andrew Sutton**
(asutton@lock3software.com)
Wyatt Childers
(wchilders@lock3software.com)

Compile-time Metaprogramming in C++

Introduction

This paper proposes several new facilities for compile-time metaprogramming which build upon the design for value-based, static reflection presented in P1240R0. This paper expands significantly on earlier work presented in P0712R0. Specifically, the proposed features include facilities for generating or injecting statements and declarations.

Static reflection (P1240R0) provides (mostly) read-only facilities for inspecting source code. There are only a few ways reflections can be used to generate new source code contracts: reifying identifiers and types, and instantiating templates. The metaprogramming facilities presented in this paper provide significantly enhanced capabilities, including the ability to generate new statements, declarations, enumerators, parameters, etc. These capabilities are needed to support software frameworks with significant generative requirements (e.g., Qt, WinRT, etc.).

There are essentially two ways we can generate new code that can be inserted into a program: programmatically or declaratively.

A programmatic approach entails the construction abstract syntax trees as constexpr values, and then translate into C++ constructs to actually generate the code. This approach has some problems in C++. First and foremost, C++ is a big language. The number of tree nodes may be oppressively large. Also, it's not at all clear what the right level of abstraction of the tree should be. For example, are users expected to explicitly build trees with the right conversions or should that be inferred by the compiler?

A declarative approach would allow the user to specify code-to-be-generated using patterns (much like templates!). While this approach lets us avoid a heavyweight library for building abstract syntax trees, it poses a number of difficult questions. In particular, how much semantic analysis is done on the source code in the fragment? Can patterns depend on local variables or parameters? Can patterns depend on declarations at the point where they are inserted?

Ultimately, we adopted a declarative approach for specifying code-to-generated. It provides a more direct method of specifying what code should be generated, and allows that to be done in a natural way: using the syntax that you want to generate. Note that this approach does not preclude adding programmatic code generation later.

There are three central features introduced by this paper:

- *metaprograms* allow the execution of code where it appears in a translation,
- *source code fragments* are partial specification of various constructs, and
- *source code injection* allows the insertion of fragments at particular points in the program.

These features are designed to build on and extend the features of P1240 (Scalable Static Reflection) and P1306R1 (Expansion Statements).

A First Example

Before enumerating specific features and their details, we present a basic example that demonstrates how metaprograms, source code fragments, and source code injection can be used to implement a `tuple` class template. The entire definition is shown below:

```
template<typename... Types>
class tuple {
  consteval {
    int counter = 0;
    for... (meta::info type : reflexpr(Types)) {
      auto fragment = __fragment struct {
        typename(type) unqualid("element_", counter);
      };
      -> fragment;
      ++counter;
    }
  }
};

tuple<bool, char, int> tup;
```

The `tuple` class is defined by its metaprogram. A metaprogram is a sequence of statements, enclosed in `consteval { ... }`, that are executed where they appear in the translation unit. Metaprograms appearing in templates are executed during instantiation, not when the template is parsed. Metaprograms are evaluated after parsing the closing brace.

The metaprogram iterates over the types in the template argument pack `Types` using an expansion statement (P1306R1). Each expansion of the pack constructs an unnamed class fragment, `__fragment struct { ... }`, which encapsulates members to be injected into the `tuple` class. A source code fragment is a kind of literal that contains a pattern of code that can be injected into source code. This expression produces a value that can be injected later. The syntax of source code fragments is a placeholder for syntax to be determined later.

The class fragment contains a single data member declaration. The type of the declaration is given as `typename(type)`; this is the proposed reification operator described in P1240. Given a reflection (of a type), it produces a *type-id*.

The `unqualid` operator is another reification operator; this one introduces a new *unqualified-id*. It takes a sequence of arguments, which are concatenated to form the new identifier. Here, identifiers have the form “elementN” where N is determined by the value of counter.

Note that the `operands` for the typename and `unqualid` reifiers require constant expressions. However, the variables `type` and `counter` are not declared as such. References to local variables within a fragment are actually placeholders for eventual values supplied during evaluation. We discuss this in greater detail in sections below.

The statement `-> fragment;` is an *injection statement*. This registers a request with the compiler to inject the given fragment in the context where the metaprogram was evaluated. The expression of this statement is not required to be a constant expression.

In this example the metaprogram is not executed until the class template is instantiated. This happens when declaring the variable `tup`. When the metaprogram executes, it will enqueue a sequence of requests to inject fragments. Each request contains a fragment value, which is comprised of a reflection of the class fragment (a static component), and a mapping of placeholders to their corresponding values.

Processing an injection request is similar to template instantiation. Each member of the class fragment (not the class itself) is synthesized into translation unit while performing a number of substitutions (described below).

The final definition of `tuple<bool, char, int>` will be:

```
template<typename... Types>
class tuple {
  bool element0;
  char element1;
  int element2;
};
```

Obviously, this is not a complete definition of tuple. To fully define the class we would also need to inject constructors and member accessors.

Source Code Fragments

A source code fragment is an expression that specifies a “snippet of code" that is intended to be injected, transforming it into executable code. There are several kinds of fragments: class fragments, namespace, fragments, enum fragments, and block fragments. The kind of the fragment determines what kinds of code can be injected.

The type of the expression is a *fragment type*. It contains, among other things, a reflection of the fragment. We explain the structure and contents of fragment values below.

Class fragments

A class fragment specifies a sequence of class members that can be injected into class scope. Class fragments are written like this:

```
__fragment class {
  // member declarations
};
```

When injected, the members of the fragment (not the fragment itself) are inserted into the class being defined. For example:

```
constexpr auto f = __fragment class {
  int x;
  int f() { return x; }
};
struct S {
  consteval { -> f; }
};
```

After injection, `S` will have members `S::x` and `S::f` (which returns `S::x`).

The fragment can also be written using `struct` instead of `class`. The choice between `class` and `struct` controls only the default access level of its elements.

```
__fragment class {
  int x; // x is private
};
__fragment struct {
  int y; // y is public
};
```

A class fragment can also be named, which allows its nested members to refer to that fragment.

```
constexpr auto links = __fragment class links {
  links* next;
  links* prev;
};
```

When injected, references to the type of class fragment is replaced with the class into which the fragment is injected. For example:

```
class node {
  consteval { -> links; }
};
```

After injection node will be equivalent to the following:

```
class node {
  node* next;
  node* prev;
};
```

As shown above, member functions of class fragments can refer to member variables of class fragments. As with normal classes, member function definitions and default member initializers are parsed only at the close of the class (fragment).

Namespace Fragments

A namespace fragment defines a sequence of members that can be injected at namespace scope. Namespace fragments can be written like this:

```
__fragment namespace {
  // declarations
};
```

When injected, the nested declarations in the fragment are injected into the namespace in which the injection was initiated. For example:

```
constexpr fns = __fragment namespace {
  void f() { }
  void g() { return f(); }
};
```

```
// In global scope
consteval -> fns;
```

The injection declaration will copy the functions `f` and `g` into the global namespace.

Unlike class fragments (or perhaps more like normal namespaces), definitions are parsed where they appear inside a namespace fragment. The usual name lookup rules apply.

Like class fragments, namespace fragments can also be written with a name, which allows self-references.

```
__fragment namespace a {
  // namespace members
};
```

This is allowed for consistency with class fragments more than utility.

Enum Fragment

An enum fragment specifies a list of enumerators that can be injected into an enum.

```
__fragment enum {
  // enumerators
};
```

For example:

```
constexpr auto f = __fragment enum {
  A, B, C = 42
};

enum E {
  consteval { -> f; }
};
```

After evaluation, `E` will have the enumerators `A`, `B`, and `C`.

As with class and namespace fragments, a name can also be given even though it's use within the fragment would serve little to no purpose.

Block Fragments

A block fragment specifies a sequence of statements that can be injected at block scope. Block fragments can be written like this:

```
__fragment {
  // statements
};
```

When injected the statements inside of the fragment are injected into the compound statement, in the position in which the injection was initiated. For example:

```
constexpr auto f = __fragment {
  int a = 4;
  int b = 2;
  return a + b;
};

int g() {
  consteval { -> f };
}
```

The injection declaration will copy the statements from the fragment `f` into `g`, resulting in g being equivalently to the following:

```
int g() {
  int a = 4;
  int b = 2;
  return a + b;
}
```

Statement fragments can introduce local variables, that are used outside of the fragment. For example:

```
constexpr auto f = __fragment {
  int a = 4;
  int b = 2;
};

int g() {
  consteval -> f;
  return a + b;
}
```

This will result in logic equivalent to the first example, i.e.:

```
int g() {
  int a = 4;
  int b = 2;
  return a + b;
}
```

Parameterizing Fragments

It's useful to define fragments that depend on user-supplied values (i.e., fragments parameterized by some set of variables). Our design allows for implicit parameterization by capturing local variables. An example of this can be seen in our initial example:

```
consteval {
  int counter = 0; // #1
  for... (meta::info type : reflexpr(Types)) {
    auto fragment = __fragment struct {
      typename(type) unqualid("element_", counter); // #2
    };
    -> fragment;
    ++counter;
  }
}
```

The variable declared in #1 is used within the class fragment at #2.

In actuality, the use of `counter` at #2 is not a direct use of the variable declared at #1; that name refers to an implicitly declared placeholder that will eventually be replaced by the value of counter (#1) when the fragment is created during constant expression evaluation. A placeholder is simply a constexpr variable declared within the fragment for the purpose of type checking and analysis, and later for source code injection.

The placeholder also becomes part of the type and value of the fragment expression. The type of the class fragment above is essentially defined by the following class:

```
struct fragment_closure {
  constexpr fragment_closure(meta::info x, int c)
    : reflection(x), counter(c)
  { }
  meta::info reflection; // reflects the class of the fragment
```

```
    int counter;            // the value of the placeholder
  };
```

When the fragment is evaluated, we create an object of this class, which reflects the underlying class of the fragment and stores the current value of the local counter variable. These captured values are then used during source code injection to replace the constexpr placeholders initially created when parsing the fragment.

Non-local dependencies

While writing fragments, you sometimes want to depend on a declaration which will be found in the context being injected into. Rather than leave references to such dependencies unresolved, we introduce a new feature that explicitly requires their declaration within the fragment.

Required Declarators

Required declarators allow you to find declarations that have yet to be declared, and are specified with the syntax:

```
requires decl-specifier-seq declarator ;
```

For example:

```
constexpr int foo(int n) { return n + 1; }

consteval -> __fragment namespace {
  requires constexpr int foo(int n);
  int bar(int n) { return 2 * foo(n); }
};
```

Within the definition of bar, the identifier foo will be resolved to the immediately preceding requirement. When the fragment is injected, two things happen. First, the required declaration is processed for injection. Instead of injecting the declaration, we lookup the name of the required declaration to find a matching declaration in the injection context, including declaration specifiers. There is an exact match, so we establish (another) substitution that replaces required declarations with those found by lookup during injection.

Argument dependent lookup is not used to resolve required declarations.

To further demonstrate the resolution rules for required declarations, consider:

```
int foo(int n) { return n + 1; }
```

```
consteval -> __fragment namespace {
  requires constexpr int foo(int n); // error: no matching declaration
  int bar(int n) { return 2 * foo(n); }
};
```

In this case, the required declaration will not be resolved successfully because the declaration specifiers do not match. In particular, the fragment expects to find and use a constexpr function, but this is not provided in the injection context.

Finally, consider:

```
int foo();

consteval -> __fragment namespace {
  requires int& foo(); // error: no matching declaration
  foo() = 42;
};
```

Here, the required declaration will not be resolved because the return type of the declarator does not match the return type of the found declaration.

Obviously, there are many relaxations of strict matching that could be made. Allowing resolution to select `constexpr` functions when none are expected is totally reasonable. Covariant return types might also be admissible. We have left the rules simple for now.

It's important to note that a required declaration can only ever resolve to one declaration. Functions in C++ however, can be overloaded, resulting in multiple candidates. Required declarators use argument dependent lookup, to resolve these candidates and select a single function.

Consider one more example:

```
void f() {} // #1
void f(int a) {} // #2

consteval -> __fragment namespace {
  requires void f(); // matches #1 only
  f();
};
```

The resolution rules select the best match if the required function is overloaded. If there is no best match, resolution will fail. If we want to require both, we must require both explicitly, as follows:

```
void f() {}
void f(int a) {}

consteval -> __fragment namespace {
  requires void f();
  requires void f(int a);
};
```

As with overload resolution, required declaration resolution will instantiate default arguments.

```
int f(int a = 0, int b = 0) { }

consteval -> __fragment namespace {
  requires int f(); // ok
};
```

That said, uses of the function within the fragment must conform to the signature given. For example:

```
int f(int a = 0, int b = 0) { }

consteval -> __fragment namespace {
  requires int f();
  int result = f(1); // error: no matching function
};
```

Note that the required declaration of `f` hides the original declaration of `f`.

Required Types

Required types are specified with the syntax:

```
requires typename identifier;
```

Where identifier is the identifier for a type-name to be resolved when this declaration is injected. Prior to injection, inside the fragment preceding the required type declaration, identifier is a dependent type. At the point of injection the type-name is looked up. If a corresponding type is found, it is then substituted for all uses of the required type.

As a simple example, consider:

```
constexpr auto frag = __fragment {
  requires typename Foo;
  Foo x;
  return x.val;
};

struct Foo {
  int val = 10;
};

int return_num() {
  consteval -> frag;
}
```

Required types function much like template type parameters. They denote the existence of a type, but not its properties.

Required Templates

It should also be possible to require templates. This feature has not yet been fully specified, although we believe it will be analogous to template template parameter declarations.

Source Code Injection

In this section, we describe the various mechanisms for injecting source code and their semantics.

Injection statements

An injection statement can appear with a metaprogram or any constexpr function called by a metaprogram. They look like this:

```
consteval {
  -> expression;
}
```

The `->` operator is followed by an expression whose value is either a fragment or a reflection (discussed below). When executed, the statement enqueues a request to inject the given value at the completion of the currently executing metaprogram. When the metaprogram completes, it should have a (possibly empty) queue of injection requests. These requests are injected in the order that they were generated.

Injection (of a fragment) is closely related to template instantiation, except that we are not substituting template arguments. We are simply generating new versions of the declarations, statements, enumerators, etc. at the point where the metaprogram appears in the program. There are two sets of substitutions:

- *Context substitution* replaces the name of class, namespace, etc. of a fragment with that of the context in which injection occurs.

- *Placeholders substitution* replaces placeholders of captured local variables with their corresponding values computed during constant expression evaluation.

For example:

```
struct S {
  consteval {
    int n = 42;
    -> __fragment struct X {
      X(X const&) = default;
      int number = n;
    };
  }
};
```

The final definition of S will be:

```
struct S {
  S(S const&) = default; // context substitution
  int number = 42; // placeholder substitution
};
```

Note that a program is ill-formed if a constant expression injects fragments during constant expression evaluation. For example:

```
constexpr int f() {
  -> __fragment struct { int x; };
  return 0;
}
int array[f()]; // error: cannot inject class members here
```

Injection Declarations

An injection declaration can also be used to inject code and is written as:

```
consteval -> constant-expression;
```

The expression of an injection declaration has the same requirements as that of an injection statement, except that it must be a constant expression. The syntax above is equivalent to:

```
consteval {
  -> constant-expression;
}
```

The injection happens at the point the declaration appears in the translation unit.

Declaration Cloning

Sometimes, we don't want to inject a fragment, but rather a copy of an existing declaration. This is particularly true for metaclasses (described below), where methods are frequently copied from a prototype (a kind of fragment) into the output class. For example, we might want to copy members of this class into a different class:

```
struct S {
  int x;
  int f() { return x; }
}
```

We can inject `S::x` and `S::f` like so:

```
struct my_class {
  consteval -> reflexpr(S::x);
  consteval -> reflexpr(S::f);
}
```

The resulting body of `my_class` will be identical to that of `S`. Note that if we had not injected S::x first, the injection of `S::f` would fail since its definition refers to `S::x`.

The expression of injection statements and declarations can be either a fragment (as above) or simply a reflection of a declaration. When the expression reflects a declaration, that declaration is injected as if it had been written in a fragment.

Sometimes, we also want to modify the access of the declaration. For example, we may want to make `S::x` private as it is injected.

```
struct my_class {
  consteval {
    meta::info x reflexpr(S::x);
    meta::make_private(x);
    -> x;
  }
  consteval -> reflexpr(S::f);
}
```

We support setting the access of a declaration to public, private, and protected.

We might also want to add properties to a declaration. For example, we may want to make `S::f` pure virtual as it is injected.

```
struct my_class {
  consteval -> reflexpr(S::x);
  consteval {
    meta::info f = reflexpr(S::f);
    meta::make_pure_virtual(f);
    -> f;
  }
}
```

The call `make_pure_virtual` modifies the reflection `f` so that, when injected, the function is made pure virtual. Every reflection value contains a set of bits designating which properties will be added to a declaration when injected. Note that you cannot remove properties, only add. We currently support the addition of the following properties:

- virtual and pure virtual
- constexpr
- static and thread local

We also allow the generation of new names for the declaration.

```
struct my_class {
  consteval -> S::x;
  consteval {
    meta::info f = reflexpr(S::f);
    meta::make_constexpr(f);

    char const* name = meta::get_name(f);
    meta::set_new_name(f, __concatenate(name, “_c”));

    -> f;
  }
}
```

The name of a declaration is modified by the `set_new_name` function. The `__concatenate` intrinsic is a workaround for lacking constexpr strings. It produces a new string literal from its inputs. The final definition of my_class is:

```
struct my_class {
  int x;
  constexpr int f_c() { return x; }
};
```

Nested Injection

With more complex metaprograms, we often need to nest injections. Suppose our application architecture is a metaprogramming based component system. Components can be defined from sets of mixins, implemented using class fragments.

Suppose our initial codebase looks something like this:

```
class entity_a {
  consteval -> position_mixin;
  consteval -> acceleration_mixin;
  consteval -> inventory_mixin;

  void adjust_velocity(...);
  // ...
};

class entity_b {
  consteval -> position_mixin;
  consteval -> acceleration_mixn;
  consteval -> stopwatch_mixin;

  void adjust_velocity(...);
  // ...
};
```

There's an inherent notion of "movement" represented by the injection of mixins for position and acceleration. We can further reduce code duplication by creating a new fragment composed of the others:

```
constexpr moveable_mixin = __fragment class {
  consteval -> position_mixin;
  consteval -> acceleration_mixin;

  void adjust_velocity(...);
};
```

We can adjust our previous entities accordingly:

```
class entity_a {
    consteval -> moveable_mixin;
    consteval -> inventory_mixin;
    // ...
  };

  class entity_b {
    consteval -> moveable_mixin;
    consteval -> stopwatch_mixin;
    // ...
  };
```

The use of the `moveable_mixin` triggers recursive source code injections. The outermost injection is the `moveable_mixin` itself. This causes the injection of the nested injection declarations in that fragment, which can also trigger additional (inner) injections.

Parameter Injection

Sometimes it becomes useful to be able to clone parameters from one function into another. For example, if we wanted to create a metafunction for integration into a simple timing system, we can do that using parameter injection.

Let's say our existing code looks like this:

```
  class bar {
    void foo(int a, int b, int c) {
      ...
    }
  };
```

We want to rework this code to integrate with our timings system:

```
  class bar {
    void foo(int a, int b, int c) {
      begin_timing("bar::foo");
      ...
      end_timing("bar::foo");
    }
  };
```

We can write a meta function, and make bar a metaclass (discussed below), so that we can change this transformation from a change that's potentially an impractical number of lines, to a single line change. The metaclass (actually just a constexpr function) could be defined as follows:

```
constexpr void timed(meta::info source) {
  for (meta::info member : meta::range(source)) {
    if (meta::is_member_function(member)) {
      make_impl(member);
      make_wrapper(member);
    }
  }
}
```

The `make_impl` function essentially renames the source method and clones its definition.

```
constexpr void make_impl(meta::info member) {
  char const* name =
    __conctatenate(meta::name_of(member), “impl”);
  meta::make_private(member);
  meta::set_new_name(member, name);
  -> member;
}
```

The `make_wrapper` function is more complex, as it requires the synthesis of an entirely new function definition.

```
constexpr make_wrapper(meta::info member) {
  char const* name = meta::name_of(member);
  char const* qname = meta::qualified_name_of(member);
  char const* impl = __conctatenate(name, “impl”);
  meta::range params(member);
  -> __fragment struct {
    void unqualid(name)(-> params) {
      begin_timing(qname);
      unqualid(impl)(unqualid(... params));
      end_timing(qname);
    }
  };
}
```

The class fragment in the function declares a single member function, highlighted here for convenience.

```
void unqualid(name)(-> params) {
  begin_timing(qname);
  unqualid(impl)(unqualid(... params));
  end_timing(qname);
}
```

The name of the function is generated from the name of the original member. The parameters, indicated by `-> params`, are injected from the original list of parameters of that function. In general, any sequence of reflections of parameter declarations can be injected into a function declarator. Moreover, there may be multiple injections at various positions in the parameter list.

The function sandwiches a call to the previously cloned implementation with timing functions. The call to the underlying function is comprised of two uses of `unqualid`. The first names the implementation function. Note that we can't use the `idexpr` reifier because we can't reflect the underlying implementation. It hasn't been injected yet. The second use of `unqualid` expands generates a sequence of identifiers for each of the parameters in `params`.

When injected, the resulting functions will resemble the code we had previously written by hand.

Unlike parameter packs, injected parameters do not participate in template argument deduction or ordering rules for overload resolution. Injecting parameters produces concrete parameters.

Injection into Alternative Contexts

The injection statement, by default, will inject into the current context of the metaprogram. However, in certain circumstances it is desirable to inject into other contexts. For example, our initial tuple implementation would need this feature to inject `get` overloads into the class's namespace.

Injection into the Parent Namespace

Injection into the parent namespace can be achieved by adding the namespace keyword, before the fragment or reflection being injected.

As an example, let's say we wanted to add a factory function to the enclosing namespace of our class, generating logic similar to the following:

```
class factory_instance {
  int value;

public:
  factory_instance(int v) : value(v) { }

  int get_value() { return value; }
  void set_value(int v) { this->value = v; }
};

factory_instance create_factory_instance() {
  return factory_instance(10);
}
```

To inject into the parent namespace, you must specify an injection context, for example:

```
-> namespace expression;
```

The `namespace` keyword causes the injection request to "float" into the namespace in which the class is written. We can use this feature as follows:

```
class factory_instance {
  int value;

public:
  factory_instance(int v) : value(v) { }

  int get_value() { return value; }
  void set_value(int v) { this->value = v; }

  consteval -> namespace __fragment namespace {
    requires typename factory_instance;
    factory_instance create_factory_instance() {
      return factory_instance(10);
    }
  }
};
```

Note the use of the required type in the namespace fragment. This is needed because the parsing of `create_factory_instance` is not deferred until the end of the class: factory_instance is incomplete at this point in the program. Declaring the required type allows us to defer semantic analysis until injection.

Injecting into a Specified Namespace

In some cases, it may be desirable to inject into a specific namespace. As an example, let's say, you're creating a geometry library, and you want all areas function to live under the same namespace, but be defined with the class. As an example consider:

```
namespace {
  struct square {
    unsigned width, height;
  };

  namespace area {
    unsigned calculate(square const& s) {
      return s.width * s.height;
    }
  }
}
```

To inject into a specific namespace, you must specify an injection context, for example:

-> namespace (*namespace-name*) *expression*;

This comes together as follows:

```
namespace shape {
  namespace area { }

  struct square {
    unsigned width, height;

    consteval -> namespace(area) __fragment namespace {
      requires typename square;

      unsigned calculate(square const& s) {
        return s.width * s.height;
      }
    }
  };
}
```

Metaclasses

A metaclass uses reflection and source code injection to define special kinds of classes. The original motivation of metaclasses aims to reduce the core language rules of “class-like” declarations (classes, structs, unions, interfaces, namespaces, etc.) by making them library features. For example, if we want to define an interface, we would write this:

```
class(interface) comparable
{
  bool equal_to(comparable const& x);
  bool less_than(comparable const& x);
};
```

A metaclass is specified as part of the *class-key*, in parentheses following `struct` or `class`. The name in parentheses designates a `constesxpr` function that transforms the contents of class definition. The metaclass definition above is roughly equivalent to this:

```
class comparable_prototype
{
  bool equal_to(comparable const& x);
  bool less_than(comparable const& x);
};

class comparable {
  using prototype = comparable_prototype;
  constexpr -> interface(reflexpr(prototype));
};
```

The semantics of interface are defined only by that function. For completeness, that function is defined as:

```
consteval void interface(meta::info source) {
  for (meta::info mem : meta::range(source)) {
    meta::compiler_require(
      !meta::is_data_member(mem),
      "interfaces may not contain data");
    meta::compiler_require(
      !meta::is_copy(mem) && !meta::is_move(mem),
      "interfaces may not copy or move; consider "
      "a virtual clone() instead");

    if (meta::has_default_access(mem))
      meta::make_public(mem);

    meta::compiler_require(
      meta::is_public(mem),
      "interface functions must be public");

    meta::make_pure_virtual(mem);

    -> mem;
  }

  -> __fragment struct X { virtual ~X() noexcept = default; };
};
```

This function does a few things, first, it iterates over the prototype designated by the parameter `source`. Inside of the loop, it then checks some basic requirements of an interface, ensuring that there are no data members in the prototype, and that there are no copy or move constructors, or assignment operators.

The interface metafunction then begins to set modifiers on the member to transform the member when it is injected. First, it checks to see if the member has default access. This is specifically defined to mean that the member's access level has been inherited from the prototype class, and was not set explicitly by an access specifier.

If the member has default access, its access level is then explicitly set to public. Then, we check to make sure that the member is now public. This in turn, effectively makes a compiler error for any member which is declared following a non-public access specifier, inside of the prototype. Finally, the metafunction makes the member pure virtual, and then queues it for injection.

After iterating over existing members, the metafunction then finishes the interface class by injecting a virtual destructor.

Type Transformations

Type transformers provide a facility similar to metaclasses except that they can be applied to already-existing classes. Type transformations have the following syntax:

```
using class identifier as id-expression
  ( nested-namespace-specifier_opt class-name ) ;

using class identifier as id-expression
  ( nested-namespace-specifier_opt class-name , expression-list ) ;
```

The *id-expression* names a constexpr function: the transformation function. The first operand to that function is the name of the class to transform. Additional arguments to function can be provided after the class name. This is equivalent to writing the following:

```
class identifier {
  using prototype = class-name;
  constexpr -> id-expression(reflexpr(prototype), expression-list);
};
```

In other words, we are generating a new class whose members are computed from the initial input.

For example, if we want to build an interface from an existing class, we can apply the interface metaprogram without modifying the initial source.

```
using class IShape as interface(shape);
```

This will generate a new abstract base class using `shape` as an initial specification.

Conclusion and Future Work

This paper presents a large number of new (and sometimes complex) features that support compile-time generative programming in C++. The syntax is still unformalized, and there are a number of design decisions that may need to be reconsidered as the language evolves. For example, a future version of static reflection may be allowed to directly modify the translation context, which is something that we have generally tried to avoid. With that change, we could change source code injection so that their results would be immediately available rather than queued for completion. That said, we believe this design is a good first step towards standardized language support for compile-time metaprogramming.