The syntactic gap: string-injection metaprogramming capabilities, and how the semantic alternative can match them

Document #: ?????

Date: 2021-03-26

Audience: SG7

Reply-to: David Rector davrec@gmail.com

Abstract

P0712/P2237 define "metaprograms" (`consteval { }`), within which code generation via "injection" (`<<`) of semantically structured "fragments" is permitted, with the benefit of typical function-scope semantics: control flow statements, local variables, etc.

Metaprograms are allowed in class, namespace, and block scopes. But this is a subset of all required code generation contexts which require function-scope semantics, demonstrated by the fact that a cruder "string-injection" feature, of the kind offered via D string mixins, would permit metaprogramming that is presently impossible with P2237 facilities - though not without its own drawbacks.

As remedy, this paper proposes:

1. Allowing metaprograms in comma-separated contexts as well: enumerator lists, (template) parameter lists, (template) argument lists, (constructor) initializer lists, and base specifier lists; and

2. Adding corresponding fragment kinds to allow injection in these contexts.

In addition, a generalized fragment syntax is proposed based on the reflection operator syntax `^`, to handle the new fragment kinds.

These changes should extend the capabilities of P2237's "semantic" code generation facilities to nearly match those of a syntactic metaprogramming approach, while avoiding its drawbacks.

Contents

2. Under P2237, control flow is either not possible or not efficient in many desirable injection contexts.

3. Syntactic injection provides for arbitrary injection with control flow, but with costs.

4. Proposal: `consteval {}` must be permitted in additional contexts, and new fragment kinds introduced.

5. Proposal: new fragment syntax

6. `Sum<T,U>` example

7. Alternative: replace `consteval {}` with `<<` or `...[::]...`

1. Syntactic vs. semantic metaprogramming

P2237 [1] summarizes metaprogramming facilities under advanced consideration, among them the "metaprogram" `consteval { }` (originally introduced in P0712 as a "constexpr block" [2]) and the injection operator `<<`.

What kinds of things should follow `<<`? The answer distinguishes what might be called "semantic" vs. "syntactic" metaprogramming approaches.

Purely "syntactic" metaprogramming allows injection of only one kind of entity: a string. The user constructs, manipulates, and injects strings containing ordinary code syntax at compile time; the compiler parses the resulting strings as dependencies are sufficiently resolved. Such an approach is used by D string mixins 3, but is also possible in C++ 4. An example of possible C++ syntax:

```
template<struct T>
class Interface {
public:
  consteval {
    template for (auto data : meta::data_members_of(^T))
      << format("virtual {} {}() const = 0;",
                meta::name_of(meta::type_of(data)),
                "get_" + meta::name_of(data));
  }
};
```

By contrast, "semantic" metaprogramming can be characterized by injection of non-string entities ("fragments"), with string manipulation and injection restricted to producing identifiers to be used within those entities.
An example of the P2237 syntax:

```
template<struct T>
class Interface {
public:
  consteval {
    template for (auto data : meta::data_members_of(^T)) {
      string getter_name = "get_" + meta::name_of(data);
      using type = decltype([:data:]);
      << <class {
           virtual type |#%{getter_name}#|() const = 0;
           // (Note: in subsequent examples we will omit the
           // "unquote operator" %{}, discussed in [1],
           // for simplicity.)
         }>;
    }
  }
};
```

The two approaches produce the same instantiated declarations for each argument T. Which is better?

The benefit of a syntactic approach is that its injection capabilities are limitless: users can generate any entity whose syntax they can represent via compile time string manipulations, typically involving many string-returning reflection queries.

The downsides of a syntactic approach have been discussed extensively on the SG7 mailing list, and include such matters as poor syntax highlighting, a tendency to produce "write-only" code, and difficult debugging [5, 6]. These problems might be summed up by this analogy: semantic metaprogramming is to syntactic metaprogramming as templates are to macros.

A purely semantic approach is thus preferable, *so long as* it is as capable as a syntactic approach. To the extent it is not as capable, the choice is less clear.

The mechanisms summarized in P2237 are not yet as capable as a syntactic approach, but can be.

2. Under P2237, control flow is either not possible or not efficient in many desirable injection contexts.

Section 3 of P0712 describes the goal of metaprograms ("constexpr blocks" then) and the fragment injection facilities therein as to provide "the ability to execute constexpr in (nearly) any context, and the ability to specify what code gets injected" [2].

However, metaprograms are presently only permitted in class, namespace, and block scopes, which is a subset of all desirable injection contexts. Injection *with control flow* is not possible – or at least not efficient – into expression lists,

enumerator lists, and parameter lists, and is certainly not possible in still other contexts for which the necessary fragment types do not yet exist.

Syntax examples and notes from relevant sections of P2237 are reproduced below for clarity:

P2237 Section 7.1.5

> ... Expression fragments are typically inserted into a program via splicing. Also, if the expression fragment contains multiple initializers, then the splice must be expanded.
>
> ```
> constexpr meta::info inits = <(1, 2, 3)>;
> vector<int> vec{ ...[:inits:]... }; // (adjusted to use P2320 [7] syntax)
> ```
>
> The resulting vector is initialized with the values 1, 2, and 3.

P2237 Section 7.2.2

> Enumeration fragments can be injected into enums using an injection enumerator:
> `constexpr meta::info rbg = <enum { red, blue, green }>;`
> `constexpr meta::info cmy = <enum { cyan, magenta, yellow }>; enum class color { << rbg; << cmy; };`

P2237 Section 7.2.6

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

In each of these cases, `consteval {}` is not presently supported, and without it there is no apparent and efficient to add control flow over injection mechanisms.

To be sure, a combination of P2237 facilities *does* potentially allow for an *inefficient* mechanism to inject code with control flow into the above contexts; this possibility, and its fatal inefficiency, is discussed in section 7.

3. Syntactic injection provides for arbitrary injection with control flow, but with costs.

In a syntactic metaprogramming approach, arbitrary injection with arbitrary control flow and other function scope semantics is possible and straightforward, but not without drawbacks:

3.1: Syntactic injection, with control flow, of expressions and enumerators

```
consteval void maybe_comma(bool &first) {
  if (first)
    first = false;
  else
    << ",";
}

// Expression injection:
constexpr meta::info inits = <(1, 2, 3)>;
consteval {
  << "vector<int> vec{";
  bool first = true;
  template for (auto init : inits)
    if (cond(init)) {
      maybe_comma(first);
      << meta::to_string(init);
    }
  maybe_comma(first);
  << "4, 5, 6 };";
}

...
// Enumerator injection:
consteval {
  << "enum class color {"
  bool first = true;
  auto inject_members = [](meta::info d) {
    template for (meta::members_of(d)) {
      maybe_comma(first);
      << meta::name_of(d);
    }
  };
  if (cond)
    inject_members(rgb);
  else
    inject_members(cmy);
  << "};"
}
```

This IO-style is perfectly capable of performing any injection task, but is difficult to read, unpleasant to write (see only the ubiquitous `maybe_comma(first)` idiom), and forces much of one's code into string literals, resulting in assorted difficulties already mentioned.

Fortunately, it is straightforward to extend the P2237 features to allow it to perform the above metaprograms efficiently, and thereby allow the semantic approach to (nearly) subsume the capabilities of the syntactic approach.

4. Proposal: `consteval {}` should be permitted in "comma-separated" contexts, and new fragment kinds introduced.

`consteval {}` is presently allowed in class, namespace, and block scopes; that is, contexts in which declarations or statements are introduced followed by semicolons. It should additionally be allowed in:
a) the expression, enumerator, and function injection contexts referenced above, and
b) other "comma-separated" contexts which require new fragment kinds as well.

To state the goal more generally: in order to match syntactic metaprogramming capabilities, it seems *any* context in which a variable number of entities can be supplied (declarations, statements, expressions, intializers, template arguments, etc.) should be considered an "injection context", with an associated fragment kind and the ability to accept one or more `consteval { }` metaprograms amidst its entities.

4.1 Semantic injection with control flow in various contexts

```
constexpr meta::info inits = <(1, 2, 3)>;
vector<int> vec{
  consteval {
    template for (auto init : inits) {
      if (somecond(init))
        << init;
    }
  },
  4, 5, 6
};
...
enum class color {
  consteval {
    if (cond)
      << rbg;
    else
      << cmy;
  },
  grue
};
...
int f(int i,
```

```
      consteval {
        template for (auto p : meta::parameters_of(some_fn))
          if (cond(p))
            << p;
      },
      int j)
{
  eat(i,
      consteval {
        template for (auto p : meta::parameters_of(some_fn))
          if (cond(p))
            << <( [:p:] ) >;
      },
      j);
}

template<consteval { /*temp param fragment injections*/ },
         typename T>
class SomeTemplate;

SomeTemplate<consteval { /*temp arg fragment injections*/ }, T>;

Foo(const T& t, const U& u)
  : bar(u.bar),
    consteval { /*ctor init fragment injections*/ },
    other(t.other)
{}

class B : consteval { /* base spec. fragment injections */ },
          private Foo
{};
```

5. Proposal: new fragment syntax

There are five fragment kinds presented in P2237 section 7:

- namespace fragments: `<namespace {}>`
- class fragments: `<class {}>` / `<struct {}>`
- enum fragments: `<enum {}>`
- statement fragments: `<{}>`

- expression fragments `<()>`

This syntax is elegant as is.

However, the previous section has introduced additional needs for:

- parameter fragments
- template parameter fragments
- template argument fragments
- initializer fragments
- constructor initializer fragments
- base specifier fragments

and perhaps still others.

It is difficult to conceive of any elegant symbolic extensions to the P2237 syntax that would incorporate these new fragment types.

Instead, a new fragment syntax is proposed which can handle an arbitrary quanitity of fragment kinds. It differs slightly for

1. Class, namespace, and enum fragment kinds vs.
2. All others.

5.1 Proposed syntax for defining *non*-class/namespace/enum fragments

```
auto f = ^<T>{ };
```

where `T` is a non-dependent identifier indicating the fragment kind.

Statement and expression fragment kinds would be changed as follows:

```
^<frag::stmt>{ [:m:] = 42; }  // was <{ [:m:] = 42; }>
^<frag::expr>{ 3, [:m:] + 4 } // was <( 3, [:m:] + 4 )>
```

Six new fragment kinds would be introduced:

```
^<frag::parm>{ int i, int j }
^<frag::tparm>{ typename T, int N }
^<fram::targ>{ [:Trefl:], [:Nrefl:] }
^<frag::init>{ 3, .[:m:] = 4 }
^<frag::cinit>{ [:m:](3), [:n:](4) }
^<frag::base>{ public Foo, private virtual [:Barrefl:] }
```

The identifier `T` in `^<T>{}` could be interpreted either of two ways:

1. As a built-in enumerator value. The type of the expression would remain `meta::info`.

2. As a built-in fragment type, each a subclass of `meta::info`. The type of the expression would then be `T`.

Option 2 seems preferable, but option 1 is also reasonable, should implementation concerns demand all reflections and fragments be of `meta::info` type and absolutely no more.

5.2 Proposed syntax for defining class, enum, and namespace fragments

Class fragments cannot be handled in the same way as the above fragments, since they need to allow for base specification (`<class : Base {}>`).

Enum and namespace fragments do not have this problem, but it would be advisable to handle them similarly to class fragments, for uniformity.

This presents an opportunity to link fragment specification syntax to reflection syntax, by expressing class, namespace, and enum fragments as reflections of anonymous entities of those kinds:

```
auto f1 = ^class : private Base { }; // was <class : private Base{ }>
auto f2 = ^namespace { };            // was <namespace { }>
auto f3 = ^enum { };                 // was <enum { }>
```

As before, the type of each expression could either remain `meta::info`, or become these subclasses thereof: `frag::cls`, `frag::ns`, and `frag::en` respectively.

P2320 addresses possible language conflicts of `^` operator in the context of reflection (e.g. by investigating the potential for conflict with block literals, which also use `^`).[7] The same conclusion it draws – that there is no apparent conflict – applies equally to this case.

In sum, this dual fragment syntax is easy to extend to arbitrary new fragment kinds as future needs are identified, and links fragment syntax to reflection `^`

syntax, which is suitable because reflections and fragments are precisely the two `meta::info`-typed expressions able to take non-`meta::info` operands. That is, these are the two "entry points" to higher level "meta" space, and the syntax should naturally imply this commonality.

6. `Sum<T,U>` example

The `public` content below demonstrates some of the proposed syntax.
(Note again that usage of the "unquote operator" `%{}` [1] is omitted in this syntax, as well as in previous fragment examples, for simplicity.)

```
/// A sum of arbitrary objects of types T and U.
///
/// Real-world use: objects of type Sum<T,U> can be returned
/// by a general operator+(const T&, const U &)
/// definition, enable_if'd for all Ts and Us for which
/// declval<T>() + declval<U>() is not otherwise supported.
///
/// For now we assume T and U are plain structs.  Each
/// instantiation Sum<T,U> is also a plain struct.
///
template<plain_struct T, plain_struct U>
class Sum {

  // --- Boilerplate compile-time data calculation --- //
  using comptime_data_t =
          tuple<set<string>,
                set<string>,
                set<string>,
                map<string, meta::info>>;
  static constexpr comptime_data_t comptime_data = [](){
    comptime_data_t data;
    auto &Tset  = std::get<0>(data);
    auto &Uset  = std::get<1>(data);
    auto &TUset = std::get<2>(data);
    auto &types = std::get<3>(data);

    template for (tfield : meta::fields_of(^T)) {
      string name = meta::name_of(tfield);
      Tset.insert(name);
      TUset.insert(name);
      types[name] = meta::type_of(tfield);
    }
    template for (ufield : meta::fields_of(^U)) {
      string name = meta::name_of(ufield);
```

```
      Uset.insert(name);
      bool inTtoo = TUset.insert(name).second;
      if (inTtoo) {
        meta::info &type = types[name];
        type = ^decltype(
            std::declval<[:type:]/*tfield type*/>() +
            std::declval<decltype([:ufield:])>() );
      } else
        types[name] = meta::type_of(ufield);
    }
    return data;
  }();
  static constexpr auto &Tset  = std::get<0>(comptime_data);
  static constexpr auto &Uset  = std::get<1>(comptime_data);
  static constexpr auto &TUset = std::get<2>(comptime_data);
  static constexpr auto &types = std::get<3>(comptime_data);

  // --- Class fragment injection --- //
public:
  consteval {
    for (auto name : TUset)
      << ^struct { [:types[name]:] |#name#|; };
  }

private:
  static consteval void
  inject_cinits_from_tu(meta::info trefl, meta::info urefl) {
    for (auto name : TUset) {
      if (Tset.contains(name)) {
        if (Uset.contains(name))
          << ^<frag::cinit>{ |#name#|([:trefl:].|#name#| +
                                      [:urefl:].|#name#|) };
        else
          << ^<frag::cinit>{ |#name#|([:trefl:].|#name#|) };
      } else
        << ^<frag::cinit>{ |#name#|([:urefl:].|#name#|) };
    }
  }

public:
  Sum(const T &t, const U &u)
    : consteval { inject_cinits_from_tu(^t, ^u); }
  {}

  // --- Parameter and constructor-initializer injection --- //
private:
```

```
  static consteval void
  inject_fields_as_parms() {
    for (auto name : TUset)
      << ^<frag::parm>{ [:types[name]:] |#name#| };
  }
  static consteval void
  inject_fields_as_cinits() {
    for (auto name : TUset)
      << ^<frag::cinit>{ |#name#|(|#name#|) };
  }
public:
  Sum(consteval { inject_fields_as_parms() })
    : consteval { inject_fields_as_cinits() }
  {}

  // --- Initializer fragment injection --- //
  operator T() const {
    return T{
      consteval {
        for (auto name : Tset)
          << ^<frag::init>{ .|#name#| = |#name#| };
      }
    };
  }
  operator U() const {
    return U{
      consteval {
        for (auto name : Uset)
          << ^<frag::init>{ .|#name#| = |#name#| };
      }
    };
  }
};
```

7. Alternative: replace `consteval {}` with `<<` or `...[::]...`

The new fragment types discussed above seem to be necessary to further the goal of matching syntactic metaprogramming capabilities.

And, it is difficult to conceive of any good alternatives to the generalized fragment syntax presented here, given how many fragment kinds need to be handled.

However, there is a reasonable alternative to *how* fragments should be injected, based on certain unusual injection facilities already present in P2237.

Section 7.2.3 of P2237 describes the injection operator `<<` as able to inject into an alternative context; e.g. one can write `<< frag` to inject `frag` into the

current context, or `dst << frag` to inject `frag` into `dst` (which must be another `meta::info` object).

This permits an intriguing possibility: `consteval {}` can technically be done away with altogether, at least for injection purposes, by either:

1. Making use of the splice operator, with pack expansion, e.g. `...[:frag:]...`, to expand a fragment into each of the contexts above, or

2. Allowing direct use of the injection operator in its place in these contexts: `<< frag`.

Indeed, the syntax examples presented in section 2 here already demonstrate both mechanisms in action: in P2237's section 7.1.5 pack expansion of an expression fragment is permitted, while in 7.2.2 an "injection enumerator" allows injection outside `consteval {}`, while section 7.2.6 proposes a similar external usage of `<<` for injection of parameters.

Here is how the relevant parts of the `Sum<T,U>` example might be written using pack expansion syntax in all cases in place of `consteval {}`:

```
public:
  // --- Class fragment injection --- //
  ...[:
    []() {
      auto res = ^struct {};
      for (auto name : TUset)
        res << ^struct { [:types[name]:] |#name#|; };
      return res;
    }()
    :]...;

  // --- Constructor initializer injection --- //
private:
  static consteval meta::info
  build_cinits_from_tu(meta::info trefl, meta::info urefl) {
    auto res = ^<frag::cinit>{};
    for (auto name : TUset) {
      if (Tset.contains(name)) {
        if (Uset.contains(name))
          res << ^<frag::cinit>{ |#name#|([:trefl:].|#name#| +
                                          [:urefl:].|#name#|) };
        else
          res << ^<frag::cinit>{ |#name#|([:trefl:].|#name#|) };
      } else
```

```
        res << ^<frag::cinit>{ |#name#|([:urefl:].|#name#|) };
    }
    return res;
  }
public:
  Sum(const T &t, const U &u)
    : ...[:build_cinits_from_tu(^t, ^u):]...
  {}

  // --- Parameter and constructor-initializer injection --- //
private:
  static consteval auto build_fields_as_parms() {
    //...
  }
  static consteval auto build_cinits_from_params() {
    //...
  }
public:
  Sum( ...[:build_fields_as_parms():]... )
    : ...[:build_cinits_from_params():]...
  {}

  // Initializer injection
private:
  static consteval auto
  build_inits_for_set(const decltype(Tset) &nameset) {
    //...
  }

public:
  T operator T() {
    return T{ ...[:build_inits_for_set(Tset):]... };
  }
  U operator U() {
    return U{ ...[:build_inits_for_set(Uset):]... };
  }
};
```

This alternative offers the exact same capabilities to `consteval {}`, and the pack expansion syntax might well be considered to have better aesthetics. Which is better?

The `consteval {}` version seems to win handily on build time-efficiency grounds, as the `...[:buildfrag():]...`-based alternative will require copying the contents of the result of `buildfrag()` into the ultimate destination during evaluation, absent compiler heroics (which will also count against efficiency, and which there-

fore rarely show up for constant evaluation tasks – see only the rationale for the `meta::info` type given in P1240, which notes that even subobject lookup is not optimized during constant evaluation [8]).

Therefore, procedures for permitting function scope semantics during injection that are based on injecting into a temporary fragment within a consteval function, then injecting the result of a call to that function into the ultimate destination context (via pack expansion or an injection entity such as the injection enumerator), will not suffice: we really do need `consteval {}` directly in the destination contexts for efficient injection with control flow and other function scope semantics.

Should the aesthetics or readability of `consteval {}` in these contexts be problematic, a potential solution is to change metaprogram syntax to make it sufficiently "unfamiliar," which is precisely the motivation behind the unusual `[::]` and `|##|` syntax in P2237 and P2320. This could look something like:

```
Sum( [[{ inject_fields_as_parms() }]] )
  : [[{ inject_fields_as_cinits() }]]
{}
```

On balance, `consteval {}` seems to stand out sufficiently on its own in each of the contexts considered above.

References

[1]: Sutton, A. "Metaprogramming." P2237, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2237r0.pdf, 10/2020.

[2]: Sutton, A., Sutter, H. "Implementing language support for compile-time metaprogramming." P0712, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0712r0.pdf, 06/2017.

[7]: Sutton, A., Childers, W., Vandevoorde, D. "The Syntax of Static Reflection." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2320r0.pdf, 02/2021.

[8] Childers, W., Sutton, A., Vali, F., Vandevoorde, D. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1240r1.pdf, 10/2018.