Exploring the design space of metaprogramming and reflection

Document #: P0633R0
Date: 2017-03-20
Project: Programming Language C++
Audience: SG 7
Reply-to: Daveed Vandevoorde <daveed@edg.com>
Louis Dionne <ldionne.2@gmail.com>

1 Introduction

Metaprogramming is the craft of creating or modifying programs as the product of other programs. *Reflective metaprogramming* is metaprogramming that incorporates data from the program being modified. For example, one can imagine a metaprogram that automatically generates a hash function from a user-defined class type: Such a metaprogram would produce the hash function from meta-information reflected from the user-defined type.

The goal of this paper is to explore the design space for metaprogramming and reflection in C++. To this end, it is useful to identify the different dimensions of this design space. We believe those dimensions are:

1. Reflection: A way to represent significant aspects of the program as first class entities in the program.
2. Code synthesis: A way to generate new code and/or modify user-written code programmatically.
3. Control flow: A computationally complete mechanism to guide the reflection and code synthesis activities.

There are different ways to approach each of these dimensions, and hence there are multiple different *cuts* we may want to take in our design space. The rest of this paper explores that, with notes describing strengths and weaknesses.

2 The dimensions

We start by exploring all the approaches we can currently think of to implement each dimension described above. Here's a summarized view of the whole thing:

1. Reflection
   - Type syntax ([P0194])
   - Heterogeneous value syntax (each `reflexpr()` has a different type) ([P0590])

- Homogeneous value syntax (`reflexpr()` always returns `meta::info`) ([P0598])

2. Code synthesis
   - Raw string injection
   - Token sequence injection
   - Programmatic API
   - Metaclasses
3. Control flow
   - Classic template metaprogramming (type syntax)
   - Heterogeneous value metaprogramming (value syntax, but templates under the hood)
   - Constexpr programming

2.1 Reflection

This section gives an overview of the different proposed approaches for supporting reflection. We do not go into details here since the corresponding proposals already explain this.

2.1.1 Type syntax ([P0194])

The result of the reflection operator is meta information in the form of **a type that is unique for each entity that we reflect upon**. We are provided with `type_traits`-like operations to manipulate this meta information:

```
struct Foo { int x; float y; };
using MetaInfo = reflexpr(Foo);
using Members = get_data_members_t<MetaInfo>;
```

2.1.2 Heterogeneous value syntax ([P0590])

The result of the reflection operator is meta information in the form of **an object whose type is unique for each entity that we reflect upon**. We are provided with function templates that manipulate this meta information:

```
struct Foo { int x; float y; };
auto meta_info = reflexpr(Foo);
auto members = get_data_members(meta_info);
```

An important thing to notice here is that the reflection operator really returns objects of different types depending on what it is reflecting upon:

```
struct Foo { int x; float y; };
struct Bar { char w; long z; };
auto meta_info_Foo = reflexpr(Foo);
auto meta_info_Bar = reflexpr(Bar);
// The two meta_infos are objects of different types
```

2.1.3 Homogeneous value syntax ([P0598])

The result of the reflection operator is meta information in the form of **a constant expression whose type is always `meta::info`**, regardless of the entity we reflect upon. We are provided with normal constexpr functions that manipulate this meta information:

```
struct Foo { int x; float y; };
constexpr meta::info meta_info = reflexpr(Foo);
constexpr std::constexpr_vector<meta::info> members = get_data_members(meta_info);
```

Since the reflection operator returns objects of the same type regardless of the entity we reflect upon, traditional constexpr programming is applicable. Note that we postulate the existence of `constexpr_vector`, a constexpr variable length sequence.

2.2 Code synthesis

This section gives an overview of the different proposed approaches for supporting code synthesis. To showcase the three first approaches, which perform direct code synthesis, we solve the toy problem of programmatically generating a function that applies some function to each member of the following struct:

```
struct S {
  char x;
  int y;
  long z;
};
```

The reason why we use this toy problem as opposed to solving the same challenge in the general case is that solving it in the general case requires control flow, which we're trying to decouple from code synthesis.

2.2.1 Raw string injection

The idea is to allow arbitrary strings to be fed to the implementation:

```
template<typename Function>
void challenge(S const& s, Function f) {
  constexpr {
    meta::queue(meta::code("f(s.", NAME_OF_X, ");"));
    meta::queue(meta::code("f(s.", NAME_OF_Y, ");"));
    meta::queue(meta::code("f(s.", NAME_OF_Z, ");"));
  }
}
```

Here, `NAME_OF_X` & friends denote the name of the member of the struct obtained through a reflection API. Given this, the intent is that `meta::queue` queues up the string for consumption by the compiler when the constexpr evaluation completes (i.e., after the closing brace of `constexpr { ... }`). Hence, an instantiation of `challenge<F>` would expand to

```
template<>
void challenge<F>(S const& s, F f) {
  { f(s.x); }
  { f(s.y); }
  { f(s.z); }
}
```

Note that we presume the compiler would create a local scope into which each injected string is translated. This avoids "leakage" from one injection into another (and from injections into the surrounding context, which would raise tricky semantic questions).

Also, this example postulates a `constexpr { ... }` construct not seen before, which forces the compile-time evaluation of some statements. This conceivably be avoided by moving the body to a constexpr function (or lambda) and calling that function from a context that guarantees compile-time evaluation (e.g., a constexpr variable initializer), but expressing this directly seems highly desirable.

2.2.2 Token sequence injection

The idea is to allow "token sequences" to be fed to the implementation, with a mechanism to parameterize those token sequences:

```
template<typename Function>
void challenge(S const& s, Function f) {
  constexpr {
    -> { f(s.(.NAME_OF_X.)); }
    -> { f(s.(.NAME_OF_Y.)); }
    -> { f(s.(.NAME_OF_Z.)); }
  }
}
```

Here we assume that `-> { ... }` injects the `{ ... }` part of that construct after the closing brace of `constexpr { ... }`. Note that we introduced a notation `(. expression .)` to transform a reflected value back into an identifier. Other such notations would be required for a few other entities (e.g. `typename(expression)` to produce a type name). The expansion would be equivalent to the version using raw string injection.

2.2.3 Programmatic API

The idea is to allow programmatic construction of an intermediate representation of the generated code, not unlike what is done to the DOM of HTML web pages:

```
template<typename Function>
void challenge(S const& s, Function f) {
  constexpr {
    meta::queue(
      meta::expr_statement(meta::call_by_name("f",
        meta::select_member(reflexpr(s), NAME_OF_X))));
    meta::queue(
      meta::expr_statement(meta::call_by_name("f",
        meta::select_member(reflexpr(s), NAME_OF_Y))));
    meta::queue(
      meta::expr_statement(meta::call_by_name("f",
        meta::select_member(reflexpr(s), NAME_OF_Z))));
  }
}
```

2.2.4 Metaclasses

Herb Sutter and Andrew Sutton have been exploring a powerful notion of *metaclasses* (private communication). The idea here is to associate some metacode to class types using a shorthand notation and have that code run when the class definition is about to be completed. For example (using a different notation from what Herb and Andrew are exploring):

```
constexpr {
  meta::register_metaclass(
    "value",  // Metaclass name
    [](meta::info class_info) {  // constexpr lambda
      ...  // Check and/or modify the class represented by class_info.
           // E.g., diagnose that all members are value types too.
    });
}

class(value) X {
  int x, y;
};  // The constexpr function (or lambda) registered
```

The ability to "attach" metacode in such convenient ways should perhaps be considered to some degree when evaluating future directions for reflective metaprogramming.

2.2.5 Observations

Looking at these four approaches, we make the following observations:

- Only a metaprogramming API provides a natural path for a "code modification" system. For example, one could imagine an API to make a member function virtual after it has been declared.
- The API approach may require knowing details about the language, like expressions and other technical terms that would have to be reflected through the API. This may end up being verbose.
- The string injection approach could be more powerful than the token sequence approach, because strings are first-class values and could therefore be composed in more sophisticated ways. On the flip side, that opens the possibility of code that is hard to read and debug, because the construction of the string eventually injected into the program could be completely opaque.
- Although two implementations are known to already have string-injection facilities for other reasons, that may not be true of other implementations. The ability to replay a sequence of tokens, however, is present in all implementations since it is needed to correctly deal with member function bodies appearing in class definitions (they have to be stored and then replayed after the complete class definition has been seen).

2.3 Control flow

2.3.1 Classic template metaprogramming

This computation mechanism based on template specialization is well known, and we have extensive experience using it in libraries like [Boost.MPL]. It uses an arcane syntax, it is inflexible, and it does not scale properly.

2.3.2 Heterogeneous value metaprogramming

This is a newer way of metaprogramming made possible by type deduction facilities in C++14. This technique uses templates under the hood, just like the classic template metaprogramming approach, but those are hidden from the users by throwing a nice value-syntax API on top. This is used extensively in the [Boost.Hana] library. It works well, but it has some tricky corners and does not scale in terms of compilation times, just like classic template metaprogramming.

If we go down this road, Andrew Sutton's for-loop on tuples proposed in [P0589] and more extended "static" control flow could make things more palatable.

2.3.3 Constexpr programming

Given a way to interoperate between types and **homogeneous** constant expressions, we could use the full facilities available during constexpr programming for metaprogramming. If this can be figured out, it would provide the most natural way of metaprogramming, because constexpr programming is simply a subset of normal runtime programming.

3 Possible cuts in the design space

This section describes possible combinations of approaches from each dimension discussed above to assemble a full solution. We illustrate each solution by using an example inspired by Andrew Sutton that composes Howard Hinnant's proposed `hash_append` interface for members of a user-defined type.

3.1 Classic template metaprogramming with [P0194]

- Reflection: Type syntax
- Code synthesis: None needed for runtime, TODO for synthesizing types
- Control flow: Classic template metaprogramming

Here is an example of what can be achieved with this approach:

```
template<HashAlgorithm H, SimpleStruct S>
void hash_append(H& h, S const& s) {
  meta::for_each<meta::get_data_members_m<reflexpr(S)>>(
    [&](auto meta_mem) {
      using MetaMem = typename decltype(meta_mem)::type;
      hash_append(h, s.unreflexpr(MetaMem));
    }
  );
}
```

The advantage of this approach is that various parts of the solution are already available in the language; only a reflection system has to be added. The major disadvantages are:

- Template instantiation is a resource-intensive compiler activity. We have extensive experience showing it does not scale sufficiently for some heavier use cases ([Boost.MPL]).
- Template metaprogramming uses arcane notations and conventions when compared to plain C++ programming.

3.2 Heterogeneous value style with [P0590]

- Reflection: Heterogeneous value syntax
- Code synthesis: None needed for runtime, TODO for synthesizing types
- Control flow: Heterogeneous value metaprogramming

This approach is very similar to the one above, but we use objects instead of types, which allows us to stay closer to the usual syntax used in C++. Here's what can be achieved with the help of the Hana library (assuming heterogeneous value-based reflection):

```
template<HashAlgorithm H, SimpleStruct S>
void hash_append(H& h, S const& s) {
  hana::for_each(reflexpr(S).member_variables(), [&](MemberVariable v) {
    hash_append(h, s.*(v.pointer()));
  });
}
```

`hana::for_each` is an algorithm working on tuples that basically calls the provided lambda with each element of the tuple, in order. Note that `MemberVariable` is not a traditional type, but a concept name (allowing each expansion of the body to occur for different types). However, we could mitigate the notational burden by adopting a construct to expand code over the element of a tuple (and, potentially, tuple-like entities such as packs), as proposed in [P0590]. Note that this construct is useful independently of reflection. Here's how the example looks with this construct:

```
template<HashAlgorithm H, SimpleStruct S>
void hash_append(H& h, S const& s) {
  for... (MemberVariable v : reflexpr(S).member_variables()) {
    hash_append(h, s.*(v.pointer()));
  }
}
```

This is more readable, but it still involves a template-like expansion. The `for...` construct isn't a loop, but produces a repeated "instantiation" (or sorts) of its body for various values of `v`, just like `hana::for_each` does. This approach suffers from the following disadvantages:

- Template instantiation is a resource-intensive compiler activity. We have experience showing it does not scale sufficiently for some heavier use cases ([Boost.Hana]).
- Unless we introduce the ability to change the type of an object, the fact that we're manipulating heterogeneous objects means that we can't perform mutations. While this has some nice properties, it strays away from what we're used to in C++.

3.3 Homogeneous value syntax with [P0598]

- Reflection: Homogeneous value syntax
- Code synthesis: Can use anything
- Control flow: Constexpr programming

Another possibility is to allow metaprograms to be written in "plain C++". Running "plain C++" at compile time is already possible today through `constexpr` evaluation. Plain C++ operates primarily on values and therefore it makes sense to reflect program information into values rather than types. We could also imagine a handful of reflection types to reflect different entities (e.g., `meta::type` to reflect on types, `meta::decl` to reflect on declarations, `meta::expr` to reflect on expressions, etc..). This would also be a valid design choice, but we should be wary of keeping things like representing a template argument list (which may contain types, constants, and templates) easy to do.

That also means we'd want a variable length container type usable during constexpr evaluation (e.g., `constexpr_vector<T>` as described in [P0597]), which is a facility useful on its own (i.e., not just in the context of reflective metaprogramming).

A skeleton of the `hash_append` example might thus look like the following:

```
template<HashAlgorithm H, SimpleStruct S>
void hash_append(H& h, S const& s) {
  constexpr {
    for (meta::info member: meta::data_members(reflexpr(s))) {
      <generate a hash_append call for "member">
    }
  }
}
```

In this example, `reflexpr(s)` produces a value of a unique reflection type `meta::info` and `meta::data_members(reflexpr(s))` then produces a value of type `constexpr_vector<meta::info>` which can be iterated over. Note that those values can be discarded by the compiler when it is done evaluating the code. For the generation of the code in the body, we can use any of the direct code synthesis approaches. Raw string injection:

```
template<HashAlgorithm H, SimpleStruct S>
void hash_append(H& h, S const& s) {
  constexpr {
    for (meta::info member : meta::data_members(reflexpr(s))) {
      meta::queue(meta::code("hash_append(h, s.", meta::name(member), ");"));
    }
  }
}
```

Token sequence injection:

```
template<HashAlgorithm H, SimpleStruct S>
void hash_append(H &h, S const &s) {
  constexpr {
    for (meta::info member : meta::data_members(reflexpr(s))) {
      -> { hash_append(h, s.(.member.)); }
    }
  }
}
```

Programmatic API:

```
template<HashAlgorithm H, SimpleStruct S>
void hash_append(H& h, S const& s) {
  constexpr {
    for (meta::info member : meta::data_members(reflexpr(s))) {
      meta::queue(
        meta::expr_statement(
          meta::call_by_name(
            "hash_append",
            reflexpr(h),
            meta::select_member(reflexpr(s), member))));
      }
    }
}
```

4 Discussions and notes

4.1 About in-place program transformations

The ability to modify attributes of a declaration after the declaration has been completed is an attractive one, but it runs into many specific implementation difficulties. Most C++ front ends produce mostly-immutable representations of declarations (this is true, at least, for GCC, Clang, and EDG, but probably all others as well). For example, it may be very difficult for the implementation to allow turning a virtual function into a non-virtual function (however, the reverse transformation may well be less problematic).

As noted earlier, an API to perform such transformations is one of the few viable routes to such an ability. Because of the practical limitations on the sorts of transformations that can be handled by mainstream implementations it seems prudent to make such an API be very specific to the transformations that are found to be useful and feasible. For example, perhaps we could envision a `meta::make_member_virtual(...)` function, but leave out a hypothetical `meta::make_member_nonvirtual(...)`. Since actual synthesis of code is more cumbersome with the API approach, we can also consider the possibility of a compact code injection mechanism augmented (possibly at a later stage) with a limited program transformation API.

4.2 About injection points

When metacode synthesizes ("injects") code, that code must end up in some specific context and scope. When template instantiation is the synthesis mechanism, those contexts correspond to the points-of-instantiation for the templates being expanded:

1. Namespace scopes for ordinary function template instances
2. Class scopes for member function template instances
3. Somewhat more local scopes for generic lambda instances

For more direct approaches to code synthesis, we will want correspondingly direct ways of specifying where the injected code ends up. In our examples so far, the injected code always ended up in a local scope. However, it would not be hard to extend the kinds of notations introduced above to target other scopes. For example:

```
template<HashAlgorithm H, SimpleStruct S>
void generate_default_hash_function() {
  meta::info hash_type ...  // Compute some meta information.
  -> :: {  // Inject in file scope
    namespace MyLib {
      template<> typename(hash_type) hash(...) {
         ...
      }
    }
  }
}
```

or

```
template<SimpleStruct S>
void generate_helper() {
  // ...
  -> namespace {  // Inject in nearest namespace scope.
    void helper(...) { ... }
  }
}
```

or

```
template<SimpleStruct S>
void generate_print_raw_member() {
  // ... Compute some meta information.
  -> class {  // Inject in class scope.
    void print_raw() {
      generate_helper<...>();  // Cascading synthesis.
    }
  }
}
```

The last option, generating in class scopes, is interesting but also challenging when dealing with class templates. Consider the following example:

```
template<typename T> struct X {
  constexpr {
    generate_mem();  // Generates a member "mem" for X<T>.
  }
  void f() {
    mem();  // Error? ("mem" not found.)
  }
};
```

Here we added some metacode to insert a member `mem` in every `X<T>`. Such a member is likely to want to depend on the specific type `T` for which `X` is instantiated. That argues for running the metacode at instantiation time, but that means that in the generic version there is no member `mem`, which may limit the usefulness of the added member (e.g., another member cannot reference it). One possible approach to this issue is to distinguish code injected when the template definition is first encountered from code injected when it is instantiated:

```
template<typename T> struct X {
  constexpr {  // Injected in template definition
    -> class { void mem(); }
  }
  constexpr<> {  // Injected in template instantiation
    generate_mem();  // Generates a member "mem" for X<T>.
  }
  void f() {
    mem();  // Error? ("mem" not found.)
  }
};
```

4.3 About constexpr blocks

The `constexpr { ... }` block discussed above could be implemented as a constexpr declaration available at namespace scope, class scope, and block scope. Roughly speaking, the behavior within the constexpr declaration would match that of constexpr compound statements, which would allow for function-like behavior within the braces. In other words, this:

```
constexpr {
  ...
}
```

would be somewhat equivalent to this:

```
constexpr void __f() { ... }
__f();
```

As a quick example of what that allows, if we solve the problem of generating new declaration names, then implementing a naive tuple (e.g. without EBO) might be:

```
template<typename ...Args>
struct tuple {
  constexpr {
    for... (auto type : enumerate(Args))
      typename(value(type)) ("element_" + index(type));
      //                    ^^^^^^^^^^^^^^^^^^^^^^^^^^ declarator
  }
};
```

The template problem is still vexing. When a constexpr-declaration is written inside a template, when would you (or should you) evaluate `__f()`? Clearly, in the case above, only on instantiation. We could extend the notion of instantiation dependence to statements, so that the constexpr block is instantiated if and only if its compound-statement is not instantiation dependent. This part obviously needs to be fleshed out, but there's hope.

5 Acknowledgements

Thanks to Andrew Sutton for comments, and to everyone else in SG-7 for general discussion.

6 References

[P0597] Daveed Vandevoorde, *`std::constexpr_vector<T>`*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0597r0.html

[P0598] Daveed Vandevoorde, *Reflect through values instead of types*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0598r0.html

[P0590] Andrew Sutton & Herb Sutter, *A design for static reflection*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0590r0.pdf

[P0385] Matus Chochlík, Axel Naumann & David Sankel, *Static reflection: Rationale, design and evolution*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0385r2.pdf

[P0194] Matus Chochlík, Axel Naumann & David Sankel, *Static reflection*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0194r3.html

[Boost.Hana] Louis Dionne, *Boost.Hana, A modern metaprogramming library*
https://github.com/boostorg/hana

[Boost.MPL] Aleksey Gurtovoy & David Abrahams, *Boost.MPL*
http://www.boost.org/doc/libs/release/libs/mpl/doc/index.html

[P0589] Andrew Sutton, *Tuple-based for loops*
http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0589r0.pdf