Metaprogramming by design, not by accident

Document #:	P0425R0
Date:	2017-06-18
Project:	Programming Language C++
Audience:	SG 7

Reply-to: Louis Dionne <ldionne.2@gmail.com>

1 Introduction

In [P0633], we introduced a clear distinction between three aspects of compile-time programming that are mostly orthogonal: reflection, control flow and code synthesis. We then explored the design space of compile-time programming by presenting several different designs for each aspect. This paper picks one particular design for reflection and control flow, namely constexpr programming and homogeneous value-based reflection, and presents a path towards that. This paper punts on the issue of code synthesis for the moment; while it is clearly an important feature, we feel like it is a bigger chunk which can be addressed separately. It is also the case that solving reflection and control flow on their own will yield tremendous benefits, regardless of whether or exactly how code synthesis is solved, so we want to start making progress on that front.

It should be noted that this paper is a reorganization of various SG 7 papers into a coherent vision more than a novel idea. However, the author thinks this may help focus the discussion in SG 7.

2 Motivation

Following [P0633] and discussion with various people in SG 7, we feel like there is a clear preference for homogeneous `constexpr`-based metaprogramming over the other alternatives. Classic template-based metaprogramming is very difficult to use, does not scale well and is basically equivalent to inventing a new language within C++. [Boost.Hana]-style metaprogramming is more usable, but it also does not scale in terms of compilation times, and it still has some usability concerns anyway.

On the other hand, `constexpr`-based type-level metaprogramming is closest to runtime C++ and could yield the best compile-time performance. If this can be made to work, this would be the best and most consistent approach to do metaprogramming in C++. The goal of this paper is to lay down what's required to make this possible.

3 Proposed path

The cut in the design space that we're exploring in this paper is constexpr programming and homogeneous value-based reflection. This is in essence very similar to [P0598] and [P0597] taken together. Note that the following path is meant to be concrete enough to be actionable and discussed with compiler implementers, but vague enough to allow for different implementation strategies if one fails:

1. Allow support for variable-size containers in constant expressions
2. Provide a `constexpr` representation for C++ types (e.g. `std::meta::type`)
3. Provide a way to convert from this representation back to a C++ type
4. Allow interoperation between `constexpr` containers and parameter packs

3.1 Variable-sized containers in constant expressions

This is of utmost importance if we want to implement the equivalent of a type list (e.g. `mpl::vector`) using `constexpr`. We have a few options to get there:

1. Provide a compiler-backed magic type that represents a contiguous sequence of objects usable inside constant expressions. This is `constexpr_vector` proposed in [P0597].
2. Provide a `constexpr`-friendly allocator, as proposed in P0639R0 (should be published in the pre-Toronto 2017 mailing).
3. Add support for *new[]-expressions* in constant expressions.

We believe that solving this problem is worth doing on its own, even without the other steps proposed in this paper, since that will make `constexpr` much more powerful and will allow complex applications to be built on top of it without jumping through many hoops (see for example the JSON parser in [Constexpr all the things!]). Note that options (2) and (3) are particularly nice, since they basically mean we could use `std::vector` (or any other standard container!) within constant expressions, which is clearly the most intuitive and consistent way of expressing compile-time sequences, since it does not differ from runtime programming.

In addition, it would be immensely useful to have the ability to use placement-new inside constant expressions, as this solves a bunch of problems that come with the approaches documented above. For example, with placement new, we could implement `constexpr` variants, which not only makes sense but also solves several concerns for the reflection proposal (the reflected members of a type could be a sequence of variants of different reflected entities). However, it seems like that would add a lot of complexity to implementations.

3.2 Constexpr representation for C++ types

Just like we have runtime type information, we could have compile-time type information through a type like `std::meta::type`. This is similar to `std::metainfo` from [P0598], but we probably need

something to represent types and not just arbitrary reflected entities, since that is too general for most use cases. The specific API should be left to the reflection proposal, but we suggest having a way to replicate at least the functionality present in `<type_traits>` one way or another. This paper will use `REFLEXPR(T)` to denote the `std::meta::type` representing `T`, while happily punting on the actual syntax.

3.3 Conversion from a `std::meta::type` to a C++ type

Given a `constexpr std::meta::type` (which could be the result of a compile-time computation), we need a way to get back a C++ type so that we can use the result of the computation to influence our program (e.g. declare variables or instantiate templates). This is `typename` from [P0598]. Whether this operator could only apply to `std::meta::type`s or to reflected entities in general is beyond the scope of this proposal, but we need *something*. This paper will use `TYPENAME(t)` to denote the C++ type associated to a `constexpr std::meta::info t`, while happily punting on the actual syntax.

3.4 Interoperation between `constexpr` containers and parameter packs

Since most compile-time computations will yield sequences of types, we believe it would be useful to have a way of extracting the contents of such a sequence in a way that makes it easy to influence the program. Our current de-facto compile-time sequence is parameter packs, so we suggest translating constexpr sequences to that representation. One possible solution would be to add support for expanding `constexpr` sequences using `...`:

```
constexpr std::vector<std::meta::type> types = {
  REFLEXPR(Foo), REFLEXPR(Bar), REFLEXPR(Baz)
};
f(types...); // calls f(types[0], types[1], types[2])
f(g(types)...); // as usual; calls f(g(types[0]), g(types[1]), g(types[2]))
```

The exact customization point that we would use to provide this functionality is to be determined. We could use `std::get`, but we would have to be careful not to clash with structured bindings. We may be able to use some iterator-based protocol instead, a bit like what the range-based `for` loop does.

4 Examples

This section contains a few examples of how type-level programming would look like with all the steps in this proposal completed. It is highly desirable to come up with more examples to validate that this design will help us solve the problems that we currently solve with template metaprogramming.

4.1 Sorting types by alignment

Here, we show how to sort types by their alignment. In conjunction with additional functionality to track which types are at which index, this could be used to do things like structure packing:

```
constexpr std::vector<std::meta::type>
sort_by_alignment(std::vector<std::meta::type> types) {
  std::sort(v.begin(), v.end(), [](std::meta::type t, std::meta::type u) {
    return t.alignment() < u.alignment();
  });
  return v;
}

constexpr std::vector<std::meta::type> types{
  REFLEXPR(Foo), REFLEXPR(Bar), REFLEXPR(Baz)
};
constexpr std::vector<std::meta::type> sorted = sort_by_alignment(types);
std::tuple<TYPENAME(sorted)...> tuple;
```

As we can see, the nice thing about this approach is that `sort_by_alignment` itself is just a `constexpr` function, which means that we can write it using normal C++. In fact, it even becomes possible to use the `constexpr`-friendly parts of the standard library, which is far from being the case for any other approach ([Boost.Hana]-like or [Boost.MPL]-like).

We use `std::meta::type::alignment()` without introduction, assuming the API of `std::meta::type` would provide something to get its alignment.

4.2 Getting the common tuple of a set of tuples

Given a set of tuples, we can find the type of the tuple to which all tuples can be converted. This is the tuple of the common types for all 0th elements, all 1st elements, 2nd elements, etc... Something like this is implemented in the [range-v3] library:

```
// Assuming something like this:
namespace std::meta {
  constexpr std::meta::type common_type(std::initializer_list<std::meta::type> types) {
    return REFLEXPR(std::common_type_t<TYPENAME(types)...>);
  }
}

template <typename ...Tuples>
constexpr auto common_template_arguments(std::vector<std::meta::type> tuples) {
  // - zip_with is part of the Ranges TS
  // - `.template_arguments()` could return a vector of the template arguments
  //   for a reflected class template instantiation
  std::vector<std::meta::type> common_args = std::zip_with(std::meta::common_type,
                                                           tuples.template_args()...);
```

```
  return common_args;
}

template <typename ...Tuple>
using common_tuple_t = std::tuple<
  TYPENAME(common_template_arguments(REFLEXPR(Tuple)...))...
>;
```

Note that this example exposes the potential for ambiguity when using `...` to expand nested sequences. For example, should `tuples.template_args()...` expand to

```
tuples[0].template_args(), tuple[1].template_args(), ...
```

or to

```
tuples.template_args()[0], tuples.template_args()[1], ...
```

and hence be ill-formed because `std::vector` has no `template_args` member function? This probably means that piggy-backing on `...` for constexpr sequence expansion is a bad idea, and we should think of something else to provide equivalent functionality.

5 Limitations of this proposal

Since this proposal does not include code synthesis/generation, causing runtime side effects is not convenient with this proposal alone. For example, it is not clear how one would write a generic JSON serializer using just the functionality included in this proposal, since that would require bridging between iteration on a `constexpr` sequence of reflected members and the runtime world (e.g. to write the data to a stream). That being said, pure type-level computations without runtime side effects are a thing, and this is what we're trying to solve here (while leaving the door fully open to adding code synthesis on top), as presented in [P0633].

6 Going forward

If the general direction presented in this paper seems pleasing to the Committee, we can start solving the subproblems presented here (and in fact some people have already started). We should also generate more examples of how this will be used in the real world, perhaps by taking existing template metaprogramming applications and translating them to this proposal.

7 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

[P0633] Daveed Vandevoorde & Louis Dionne, *Exploring the design space of metaprogramming and reflection*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0633r0.pdf

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

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

[Constexpr all the things!] Bean Deane and Jason Turner
https://youtu.be/HMB9oXFobJc

[range-v3] Eric Niebler, *Range v3*
https://github.com/ericniebler/range-v3