User-friendly and Evolution-friendly Reflection: A Compromise

Document #: P1733R0
Date: 2019-06-15
Project: Programming Language C++
SG7 Reflection
Reply-to: David Sankel
<dsankel@bloomberg.net>
Daveed Vandevoorde
<daveed@edg.com>

Contents

1 Abstract

The two primary `constexpr`-based reflection proposals for C++ are at odds with each other. "constexpr reflexpr" ([P0953R2]) focuses on a simple and familiar API, but could potentially make evolution of the language difficult. "Scalable Reflection in C++" ([P1240R0]) solves this issue, but some of us consider it to have a significant cost in terms of usability. This paper is an attempt to bridge the two approaches by introducing a generally useful feature, parameter constraints, and providing guidelines for a reflection API that is both user-friendly and has a straightforward evolutionary path as the language changes.

2 Introduction

The Reflection TS ([N4766]) adds, for the first time, reflection facilities in C++. It is based on the traditional template metaprogramming paradigm with the addition of concepts.

Over time it became clear that an approach based on `constexpr` would be superior in both compile-time efficiency and approachability by non-experts. Two independent efforts were made to design such a system culminating in "constexpr reflexpr" ([P0953R2]) and "Scalable Reflection in C++" ([P1240R0]). While the fundamental machinery in these two proposals is similar, the user-level APIs have significant differences.

At the time of this writing, the committee has not converged on either paper. In this document we will briefly summarize the chief differences and outline a new design, based on a new parameter constraints feature, that has the potential to bridge the gap between the two proposals.

2.1 Typeful Reflection and its Drawbacks

"constexpr reflexpr" ([P0953R2]) is typeful in that values returned by the `reflexpr` operator have types corresponding to the kind of syntax being reflected. See the following example:

```
template <typename T>
void dump() {
    constexpr reflect::Record metaT = reflexpr(T);
    std::cout << "name: " << metaT.get_display_name() << std::endl;
    std::cout << "members:" << std::endl;
    for(const RecordMember member : metaT.get_public_data_members())
        std::cout
            << "  " << member.get_type().get_display_name()
            << " " << member.get_name() << std::endl;
}
```

Each of these types have corresponding operations and together form a type hierarchy. For example, all `reflect::Class` objects are also `reflect::Type` objects.

The [P0953R2] authors claim that this style of API reflects good practice and will be familiar to most C++ developers. They also point out that overload resolution works well and, for many simple use cases, "advanced" features such as non-type template parameters and concepts aren't necessary at all.

On the other hand, the [P1240R0] authors point out,

> Although the relationship between major language concepts is relatively stable, we do occasionally make fundamental changes to our vocabulary (e.g., during the C++11 cycle we changed the definition of "variable"). Such a vocabulary change is more disruptive to a class hierarchy design than it is to certain other kinds of interfaces (we are thinking of function-based interfaces here)

They also claim that a type-based hierarchy will have a significant negative performance impact on code which uses reflection. A full critique of this approach can be found in [ReflexprRebuttle].

2.2 Monotype Reflection and its Drawbacks

"Scalable Reflection in C++" ([P1240R0]) takes a different approach. They provide the following example.

```
template<Enum T>
std::string to_string(T value) { // Could also be marked constexpr
  for... (auto e : std::meta::members_of(reflexpr(T)) {
    if (unreflexpr(e) == value) {
      return std::meta::name_of(e);
    }
  }
  return "<unnamed>";
}
```

In this approach values returned by the `reflexpr` operator all have the same type, `meta::info`. Instead of constraining functions, such as `members_of`, based on parameter types, sentinel `meta::info` objects are used to represent errors.

```
namespace std::meta {
  constexpr! auto members_of(info class_type, auto ...filters)
                        ->std::vector<info> {...};
}
```

> If called with an argument for `class_type` that is the reflection of a non-class type or a capturing closure type (or an alias/cv-qualified version thereof), these facilities return a vector containing a single invalid reflection.

By not tying reflection facilities to a type hierarchy, performance is significantly improved and the language is better able to evolve.

This approach has been critiqued by the authors of [P0953R2] for not following Object Oriented principles and for having a style foreign to most programmers.

> Object-oriented design is simple to reason about and easy to write. It fits naturally into C++ and its focus on values, type-safety, and conceptual abstractions.

They also claim that the compile-time performance benefits of the approach will shrink as compiler technology improves. For a full critique, see [P1477R0].

2.3 Moving Forward

Is there a way past the impasse? A subset of the authors of [P0953R2] and [P1240R0] got together for several brainstorm sessions to determine what a compromise solution, if any, would look like. The end result was a hybrid between the two approaches utilizing a newly proposed language feature, parameter constraints.

3 Parameter Constraints

Parameter constraints are an extension of concepts such that parameters can be utilized in `requires` clauses. Consider the following overload set for the familiar power function:

```
double pow( double base, int iexp );
double pow( double base, int iexp ) requires (iexp == 2); // proposed
```

The second declaration is identical to the first except it has an additional constraint that the second argument is `2`. Its implementation could be heavily optimized given this additional information.

Overload resolution in C++ happens at compile time, *not* run time, so how could this ever work? Consider the call to `pow` in the following function.

```
void f(double in) {
  in += 5.0;
  double d = pow(in, 2);
  // ...
}
```

Here the compiler knows *at compile time* that the second argument to `pow` is `2` so it can theoretically make use of the overload with the parameter constraint. In what other cases does the compiler know at compile time the value of a parameter?

As it turns out, we already have standardese for such an argument (or generally an expression) in C++: *constant expression*.

In short, this concepts extension will allow for parameter identifiers to appear in requires clauses and during overload resolution:

— if the argument is a *constant expression* it is evaluated as part of evaluation of the requires clause, and
— if the argument is *not* a *constant expression* the entire overload is discarded.

3.1 Relation to clang's `enable_if` `__attribute__`

As it turns out there already exists similar functionality implemented as an experimental clang extension (See [ClangAttributes]). The above example can be written as follows using this extension:

```
double pow( double base, int iexp );
double pow( double base, int iexp ) __attribute__((enable_if(iexp == 2, "")));
```

The `enable_if` clang attribute, when combined with its `unavailable` attribute, can be used to effectively check some precondition violations at compile time. See the following example taken from clang's documentation:

```
int isdigit(int c);
int isdigit(int c)
  __attribute__((enable_if(c <= -1 || c > 255, "chosen when 'c' is out of range")))
  __attribute__((unavailable("'c' must have the value of an unsigned char or EOF")));

void foo(char c) {
  isdigit(c);
  isdigit(10);
  isdigit(-10);  // results in a compile-time error.
}
```

Our proposed concepts extension can solve this problem as well, albeit without the diagnostic, by making use of `= delete`.

```
int isdigit(int c);
int isdigit(int c) requires(c <= -1 || c > 255) = delete;
```

4 User-friendly and Evolution-friendly Reflection

Parameter constraints are a neat feature, but what do they have to do with reflection?

[P0953R2]'s user-centric API calls for a type hierarchy representing various elements of C++'s abstract syntax tree. This tree could change significantly over time with new revisions of the language. Because of this, `reflexpr` expressions should not result in values whose type are tightly bound to this hierarchy. Instead, these values should be *convertible* to values within the hierarchy.

```
constexpr
meta::cpp20::type t = reflexpr(int); // reflexpr(int) produces a meta::info
                                     // object which is converted to a
                                     // meta::cpp20::type object.
```

Conversions to the `meta::cpp20` hierarchy can be made cleanly and without templates using parameter constraints in conversion constructors.

```
namespace meta::cpp20 {
struct type {
  consteval type(meta::info i) requires(meta::is_type(i));
  //...
};
struct class_ {
  consteval class_(meta::info i) requires(meta::is_class(i));
  //...
};
}
```

However, to provide users with seamless interaction with overloading, the following needs to be supported somehow.

```
void print(meta::cpp20::namespace t); // #1
void print(meta::cpp20::type v);      // #2
void print(meta::cpp20::class_ c);    // #3
//...

namespace foo { /*...*/ }
class bar{};

void f() {
  print(reflexpr(foo)); // Matches #1
  print(reflexpr(int)); // Matches #2
  print(reflexpr(bar)); // Desire to match #3, but ambiguous between #2 and #3.
}
```

This can be solved, however, by making conversions from `meta::info` objects only to the bottom-most-leaves (or logically "most derived" classes) of the type hierarchy.

```
namespace meta::cpp20 {
struct type {
  consteval type(meta::info i) requires( meta::is_type(i)
                                      && !meta::is_class(i)
                                      && !meta::is_union(i)
                                      && !meta::is_enum(i) );
  //...
};
struct class_ {
  consteval class_(meta::info i) requires( meta::is_class(i) );
  //...
};
}
```

Now `print(reflexpr(bar))` will unambiguously select the desired overload.

4.1 Upcasting and Downcasting

Once in the type hierarchy, casting upward can be implemented in the usual way.

```
namespace meta::cpp20 {
struct class_ {
  consteval class_(meta::info i) requires( meta::is_class(i) );

  consteval operator type();
  //...
};
}
```

```
class bar{};
void g() {
    constexpr meta::cpp20::class_ c = reflexpr(bar);
    meta::cpp20::type t = c;
}
```

A downcast function template can be provided to go in the reverse direction.

```
namespace meta::cpp20 {
    consteval class_ make_class_from_info(meta::info i);
        // Fails with an exception if !meta::is_class(i)

    template<typename T>
    consteval T downcast(meta::cpp20::type t) {
        if constexpr (std::is_same_v<T, meta::cpp20::class_ ) {
            return make_class_from_info(t.info());
        } else //...
    }
}
```

```
class bar{};
void h() {
    constexpr meta::cpp20::type t = reflexpr(bar);
    constexpr auto c = meta::cpp20::downcast<meta::cpp20::class_>(t); // OK
}
```

For programmer convenience, we can additionally provide a `most_derived` function which will take in a `meta::cpp20::object` (the most base class in the hierarchy) and return an instance of the most derived type for that object.

```
namespace meta::cpp20 {
  consteval std::span<type_> get_member_types(class_ c) const;
}

struct baz {
  enum E { /*...*/ };
  class Buz{ /*...*/ };
  using Biz = int;
};

void print(meta::cpp20::enum_);  // #1
void print(meta::cpp20::class_); // #2
void print(meta::cpp20::type);   // #3

void f() {
  constexpr meta::cpp20::class_ metaBaz = reflexpr(baz);
  for...(constexpr meta::cpp20::type member_ : get_member_types(metaBaz)) {
    print( meta::cpp20::most_derived(member_) ); // Calls #1, #2, and then #3
  }
}
```

most_derived can be implemented using parameter constraints:

```
namespace meta::cpp20 {

template<bool dummy=true>
consteval type most_derived(object o) requires(  meta::is_type(o.info())
                                             && !meta::is_class(o.info())
                                             && !meta::is_union(o.info())
                                             && !meta::is_enum(o.info()))

template<bool dummy=true>
consteval enum_ most_derived(object o) requires( meta::is_enum(o.info()))

//...
}
```

4.2 Multiple User-defined Conversions

It likely wouldn't be unusual to have a have a declaration like,

```
consteval X reflective_fun(type t);
```

, that is meant to accept *any* type. By limiting the match to only types whose associated most-derived meta-object type is `type`, we would not be able to easily pass e.g., `reflexpr(my_enum)` (going via `info`→`enum_type`→`type` requires two user-defined conversions, which isn't valid).

We present a couple alternatives for making this work properly.

4.2.1 Prefer More-Constrained Conversion Candidates

In this solution we would allow `meta::info` objects to be converted to any base classes *in addition* to the most-derived class.

```
namespace meta::cpp20 {

struct type {
  consteval type(info i) requires is_type(i);
  // ...
};
struct enum_ {
  consteval enum_(info i) requires is_type(i) && is_enum(i);
  // ...
};

}
```

Overload resolution rules would be modified such that candidates with a more specific set of requires clauses are preferred. For example, conversion of `reflexpr(my_enum)` would match conversion to `enum_` better than conversion to `type`.

4.2.2 Use Actual Inheritance for Standard Conversion

Conversion of a value one of its base classes is not considered a user-defined conversion so it can be used to work around the "two user-defined conversions" issue.

```
namespace meta::cpp20 {

struct object { /*...*/ };
struct named : public object { /*...*/ };
struct type : public named { /*...*/ };
struct enum_ : public type {
  consteval enum_(info i) requires is_type(i) && is_enum(i);
  // ...
};

}
```

Calling `reflective_fun` above with an argument `reflexpr(my_enum)` will result in a user-defined conversion to `enum_` and then a standard conversion to `type`.

The slicing here is safe, but `meta::cpp20::downcast` must still be used instead of `dynamic_cast` to downcast.

The benefit of this approach is that another language feature is not required.

4.3 Evolution

The structure of C++'s abstract syntax tree (AST) can change drastically over time while the language itself retains backwards compatibility. It is important that reflection doesn't hinder evolution of C++'s AST. The design presented here, which relegates the AST view to a library feature, allows C++'s AST to evolve while retaining backwards compatibility of reflection-based code.

The are two types of AST evolutions to consider:

— **API backwards compatible changes**. Additions of new AST nodes or adding additional functions that operate on AST classes fall into this category. These types of changes can be made safely to the user-level API between revisions of the programming language. Most AST modifications fall into this category.

— **API non-backwards compatible changes**. These kinds of changes involve a reorganization of or significant meaning changes in the AST hierarchy. For these kinds of changes a new namespace would be created (e.g. `meta::cpp29`) containing the new hierarchy. The old namespace and classes would continue to be functional with existing code although they wouldn't be expected to work with newer features in the language. The expectation is that older AST variants would be deprecated and eventually removed from the standard library.

Decoupling the type hierarchy (`meta::cpp20`) from the reflection language facility (`reflexpr`) provides a means for the language to continue its evolution and provides reflection users a reasonable migration path.

5 Conclusion

Reflection facilities provide an interesting design challenge in both balancing flexibility in future migration of the language and providing an API that is intuitive and simple to use. The solution proposed here intends to strike a balance between the two that is sufficient for both these aims.

6 References

[ClangAttributes] 2019. Attributes in Clang.
http://clang.llvm.org/docs/AttributeReference.html#enable-if

[N4766] Matúš Chochlík, Axel Naumann, and David Sankel. 2018. Working Draft, C++ Extensions for Reflection.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf

[P0953R2] Matúš Chochlík, Axel Naumann, and David Sankel. 2019. constexpr reflexpr.
http://wg21.link/P0953R2

[P1240R0] Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2018. Scalable Reflection in C++.
http://wg21.link/P1240R0

[P1477R0] Matúš Chochlík, Axel Naumann, and David Sankel. 2019. constexpr C++ is not constexpr C.
http://wg21.link/P1477R0

[ReflexprRebuttle] 2019. P1447 & P0953: A Rebuttal. Presentation delivered in 2019 Kona meeting.
http://wiki.edg.com/pub/Wg21kona2019/SG7/reflexpr_rebuttal.pdf