Document: P2049R0
Date: 2020-01-12
Audience: SG7
Authors: Andrew Sutton
(asutton@lock3software.com)
Wyatt Childers
(wchilders@lock3software.com)

Constraint refinement for special-cased functions

Abstract

This paper builds on P1733 (User-friendly and Evolution-friendly Reflection: A Compromise). In particular, we propose to extend concepts to allow refinable constraints on special-cased functions. These concepts extensions include:

- Function parameters for concepts
- Non-template concepts
- Constraints on non-template functions

Taken with P1733, this provides a general and powerful feature that extends well beyond the scope of static reflection and metaprogramming.

Introduction

P1733 proposes to allow function constraints (as in concepts) on constant function arguments. The purpose of this extension is to provide implementation support for a class-based reflection facility (P0953) built on top of a weakly typed, handle-based reflection facility (P1240). The proposed feature supports the following use:

```
constexpr meta::class_info c = reflexpr(some_class);
```

The `reflexpr` operator returns a `meta::info` object: a scalar value that designates the compiler's internal representation of `some_class`. The `class_info` class is initialized by the `meta::info` value and exposes a meaningful interface for querying class properties. This is a standard technique for building coherent abstractions on low-level facilities.

But what if the reflection designates something other than a class? In more conventional libraries, we might throw an exception from the constructor. However, P1733 allows these requirements to be expressed as constraints (as in concepts):

```
struct class_info {
  consteval class_info(info x) requires is_class(x)
    : refl(x)
  {
    if (!is_class_type(x))
      throw runtime_error("not a class");
  }
};
```

In the example above, if `c` were to be initialized by something other than a class reflection, the program would be ill-formed, with a diagnostic similar to: "`no matching constructor; is_class(x) evaluated to false`".

Details

P1733 extends the constraint (as in concept) mechanism to allow overloading based on the value of function arguments. For example:

```
double pow(double base, int exp);                      // #1
double pow(double base, int exp) requires (exp == 2); // #2

// Elsewhere
double pi = 3.14;
pow(pi, 3); // calls #1
pow(pi, 2); // calls #2
int n = 2;
pow(pi, n); // calls #1
```

The mechanism is relatively straightforward. Function parameters initialized by constant expressions are made available as constants during constraint satisfaction. A candidate whose constraints are not satisfied is not viable. In the call `pow(pi, 3)`, only #1 is viable because `exp == 3` is false. In the call `pow(pi, 2)`, both #1 and #2 are viable, and #2 is more specialized. In the call `pow(pi, n)`, only #1 is viable because `exp == 2` is not a constant expression.

Note that this feature does *not* support the more general notion of case-based function definitions that are popular in some functional languages (e.g., Haskell). This is a limited approach that "carves out" special cases for constant function arguments.

The semantics of this feature are: For each function parameter used in the trailing *requires-clause*, synthesize a new, implicit template parameter of the same type and name, and replace references to function parameters in the original expression with their corresponding template parameter. If a function parameter cannot be used as a template parameter (i.e., not a structural type), the program is ill-formed.[1] The "promotion" of function parameters to template parameters means we don't need to reformulate the rules for constraint normalization, satisfaction, or substitution in order to make this feature work. Everything remains defined in terms of template parameters.

To be clear, synthesized template parameters *do not replace* their corresponding function parameters. The scope of synthesized template parameters begins at the `requires` keyword and ends at the end of its expression.

When forming the template arguments needed to satisfy the associated constraint of a declaration, initialize each synthesized template parameter with the expression used to initialize the corresponding function parameter. If the template parameter cannot be initialized, the constraints are not satisfied.

Extensions for concepts

The following example is given in P1733. The `most_derived` down-casts an object to its most-derived type.

```
template<bool = true>
consteval type_info most_derived(object o)
  requires (is_type(o.info())
            && !is_class_type(o.info())
            && !is_union_type(o.info())
            && !is_enum_type(o.info()));

template<typename = true>
consteval enum_info most_derived(object o)
  requires is_enum_type(o.info());
```

The functions are templated since constraints are not allowed on non-template functions. While the constraints of functions would achieve the intended result, their authoring is a bit fragile. Any significant extension to the type system would require "re-juggling" the constraints on (potentially many) overloads in this set of functions in order to ensure the desired outcome.

The constraints in the example define a refinement hierarchy, albeit using exclusion instead of strengthening predicates. The constraints on the first overload are the most general; it works for anything

[1] Equivalently, synthesize a template parameter for each function parameter of structural type before parsing the *requires-clause*, then discard parameters that are not used in the trailing *requires-clause*. If an identifier in the *requires-clause* refers to a function parameter, the program is ill-formed.

that's a type, but not a more specific kind of type. The constraints of the second overload are more specific than those of the first--they refine the constraints of the original (all `enum` types are types). Ideally, we should be able to take advantage of the functionality provided by concepts in order to define these functions like so:

```
consteval type_info most_derived(object o)
  requires is_type(o.info());

consteval enum_info most_derived(object o)
  requires is_enum_type(o.info());
```

In other words, the `is_type` and `is_enum_type` predicates behave more like concepts than `consteval` or `constexpr` functions. Note that we have also removed the requirement to make these functions templates. Beyond the semantics proposed by P1733R0, this feature would require the following extensions:

- Extend concepts so they can take function arguments.
- Potentially allow concepts to be defined as non-templates.
- Allow constraints on non-template functions.

Each extension is described in the following sections.

Function concepts

Here, we want to allow concepts to accept function arguments in addition to template arguments. The `is_type` and `is_enum_type` concepts could be defined as:

```
template<typename Refl>
concept is_type(Refl x) = __meta_is_type(x);

template<typename Refl>
concept is_enum_type(Refl x) = is_type(x) && __meta_is_enum(x);
```

Function concepts can behave (more or less) like regular functions; they can be overloaded, they can be called (kind of), template argument deduction works in the usual way, etc. However, the function parameters here are interpreted as non-type template parameters, and as such, are restricted to being structural types.

Because a function concept is a concept, its "evaluation" follows the usual rules for evaluating concepts: it is first normalized into a logical constraint, which is then checked for satisfaction (interleaving template substitution and constant expression evaluation). We want them to have the following behavior:

```
int n;
```

```
cout << is_type(n); // prints 0
cout << is_type(reflexpr(int)); // prints 1
```

The rules for satisfying a called function concept are similar to those for satisfying constraints involving function parameters: the function parameters (actually template parameters) are initialized by their arguments. If initialization fails, the constraints of the concept are not satisfied (i.e., the expression evaluates to false). This causes the first `cout` statement to print 0; initialization of `Refl x` by the variable `n` fails because `n` is not a constant expression.

Non-template concepts

We could write `is_type` and `is_enum_type` as non-templates. In fact, this is preferable since their current formulation allows them to be used with undesirable function arguments (like `int` in the example above). Ideally, the definitions of those predicates should be:

```
concept is_type(info x) = __meta_is_type(x);
concept is_enum_type(info x) = is_type(x) && __meta_is_enum_type(x);
```

A non-template concept should work in exactly the same ways as a template concept since their parameters are interpreted as template parameters.

Constraints on non-template functions

For P1733 to be useful, it is essential that we allow constraints on non-template functions—even the first example in this paper requires this feature. The concepts TS did include the ability to constrain non-template functions, but that was removed when the TS was merged into the working paper. (Also, the implementation didn't really work.)

The reason for removing the feature is that our current declaration model readily makes constrained non-template overloads ill-formed, no diagnostic required. For example:

```
void f() requires (VERSION == 1);
void f() requires (VERSION == 2);
void f() requires (VERSION == 3);
```

If `VERSION == 1`, then the 3rd overload makes the program ill-formed because the constraints of the 2nd and 3rd overload are functionally equivalent but not equivalent; they have different spellings, but always evaluate to false.

This paper allows for a limited set constrained non-template functions, specifically those whose constraints refer to function parameters. In other words, the example above would still be rejected, although not necessarily diagnosed. This paper should allow the following:

```
void f(int version) requires (version == 1);
void f(int version) requires (version == 2);
void f(int version) requires (version == 3);
```

No two constraints are functionally equivalent.

Conclusions

If we adopt P1733, then we should also adopt these extensions to concepts. Combined, these provide a general and powerful feature that extends well beyond the scope of static reflection and metaprogramming. Although the feature in P1733 was conceived to support a tiered architecture for standard reflection facilities, the first examples demonstrate it's utility well outside the scope of metaprogramming: we can provide special cases for complex functions that might not be optimizable to the extent we can achieve with more direct approaches. The features in this paper are intended to improve our ability to declare such special cases.

That said, these features are a far cry from generalized predicate-based dispatch (e.g., Haskell functions). Pursuing that design goal introduces a slew of problems that should be considered separately from this proposal.