Language Support for customisable Functions

Document #:	D2547R1
Date:	2022-03-14
Project:	Programming Language C++
Audience:	Evolution

Reply-to:	Lewis Baker (Woven-Planet) <lewissbaker@gmail.com> Corentin Jabot <corentinjabot@gmail.com> Gašper Ažman <gasper.azman@gmail.com>

Contents

1 Abstract **1**

2 Status of this proposal **2**

3 Short description of the proposed facilities **2**

4 Terminology **2**

5 Examples **3**
- 5.1 Declaring a customisable function . . . 3
- 5.2 Declaring a customisable function contains that has a default implementation . . . 3
- 5.3 Defining a customisation as hidden friend . . . 3
- 5.4 Defining a customisation at namespace scope . . . 3
- 5.5 Calling a customisable function . . . 4
- 5.6 Generic customisations . . . 4
- 5.7 Customizable function templates . . . 5

6 Background **6**

7 Motivation **7**
- 7.1 The problem with member-functions . . . 9
  - 7.1.1 Member functions all live in a single global "namespace" . . . 9
  - 7.1.2 We cannot customise different member names generically . . . 9
  - 7.1.3 Customisation-point member functions cannot be passed to higher-order functions . . . 10
  - 7.1.4 Implementing a concept requires modification of the type . . . 10

8 References **11**

1 Abstract

This paper proposes a language mechanism for defining customisable namespace-scope functions as a solution to the problems posed by [P2279R0] "We need a language mechanism for customisation points".

2 Status of this proposal

This work is a preliminary initial design. We intend this proposal to replace the use of `tag_invoke` in [P2300R4] in the C++26 time frame. We need directional guidance from EWG and LEWG to refine the design over the next few months.

3 Short description of the proposed facilities

This proposal seeks to improve over existing customisation point mechanisms in a number of ways:

- Allows customisation-point names to be namespace-scoped rather than names being reserved globally.
- Allows generic forwarding of customisable functions through wrapper-types, including generic type-erasing wrappers (like `std::optional`) and adapters that customise some operations and pass through others.
- Concise syntax for defining customisable function-objects and adding customisations for particular types.
- Support for copy-elision of arguments passed by-value to customisation points.
- Far better compile times compared to the `tag_invoke` mechanism (avoids 3 layers of template instantiations and cuts down on SFINAE required to separate implementation functions)
- Far better error messages compared to the `tag_invoke` mechanism (`tag_invoke` does not distinguish between different customisation point functions and results in sometimes hundreds of overloads)

This proposal improves on the semantics of the `tag_invoke` proposal [P1895R0], keeping namespace-scoped customisation points and generic customisation / forwarding, while providing a terser and cleaner syntax accessible to less experienced programmers.

The proposed syntax introduces use of:

- The `virtual` function-specifier for namespace-scope functions as a way of declaring that the function is a customisation-point (syntax idea borrowed from [P1292R0] "customisation Point Functions").
- The *virt-specifier* `override` for customisations of a customisable function-object.
- The *pure-specifier* `= 0` for declaring a customisable function without a default implementation
- The `default` keyword as an additional *virt-specifier* annotation for a customisable function's default implementation.
- A syntax for declaring customisations of a specific customisable function-object by using the fully-scoped name of the function-object as the function-name.
- The ability to deduce the customisable function object for the purposes of generic forwarding.

4 Terminology

This paper introduces the following terms:

- A function declaration for a *customisable function* is a **customisable function prototype**:

```
virtual void foo(auto&&) = 0;
```

- The set of *customisable function prototypes* naming the same entity represent a **customisable function**.
- A *customisable function* introduces a **customisable Function Object (CFO)** in the scope it is declared.
- The set of functions or template functions with the `override` keyword are **customisations** of the corresponding customisable function. They form the **customisation overload set**.
- The set of functions or template functions with the `default` keyword are **default implementations** of the corresponding customisable function. They form the **default overload set** for that customisable function object.
- A declaration of namespace-scope function of the form

```
template <auto func, typename T> auto func(T&& object) override;
```

declares a **generic customisation**.

5 Examples

5.1 Declaring a customisable function

The trailing `= 0` indicates this is a declaration without a default implementation.

```
namespace std::execution {
  template<sender S, receiver R>
  virtual operation_state auto connect(S s, R r) = 0;
}
```

5.2 Declaring a customisable function contains that has a default implementation

```
namespace std::ranges {
  template<input_range R, typename Value>
    requires equality_comparable_with<range_reference_t<R>, Value>
  virtual bool contains(R&& range, const Value& v) default {
    for (const auto& x : range) {
      if (x == v) return true;
    }
    return false;
  }
}
```

5.3 Defining a customisation as hidden friend

As an example we define a customisation of the above customisable function contains (as a hidden friend) for `std::set` using the `override` keyword. Note that the name of the customised function is qualified.

```
namespace std {
  template<class Key, class Compare, class Allocator>
  class set {
  // ...
  private:
    template<typename V>
      requires requires(const set& s, const V& v) { s.contains(v); }
    friend bool ranges::contains(const set& s, const V& v) override {
      return s.contains(v);
    }
  };
}
```

5.4 Defining a customisation at namespace scope

Alternatively, we can define a customisation at namespace scope using the `override` keyword. This can be useful when implicit conversions should be considered.

```
namespace std {
  template<class Key, class Hash, class KeyEq, class Allocator>
  class unordered_set { ... };

  // Defined outside class definition.
  template<class Key, class Hash, class Eq, class Allocator, class Value>
```

```
  requires(const unordered_set<Key,Hash,Eq, Allocator>& s, const Value& v) {
      s.contains(v);
  }
  bool ranges::contains(const unordered_set<K,H,Eq,A>& s,
                        const Value& v) override {
    return s.contains(v);
  }
}
```

5.5 Calling a customisable function

When calling a customisable function there is no need for the *two-step* `using`. They are safe to call fully qualified.

```
void example() {
  std::set<int> s = { 1, 2, 3, 4 };
  for (int x : { 2, 5 }) {
    if (std::ranges::contains(s, x)) // calls std::set customisation
      std::cout << x << " Found!\n";
  }
}
```

Note that, as with C++20 `std::ranges` CPOs, customisable-functions cannot be found by ADL when resolving unqualified call expressions.

```
void lookup_example() {
  std::set<int> s = { 1, 2, 3 };

  contains(s, 2); // normal ADL lookup for function declarations.
                  // will not find std::ranges::contains.

  using std::ranges::contains;
  contains(s, 2); // name lookup for 'contains' finds the std::ranges::contains
                  // "customisable function", and then follows overload resolution
                  // rules of customisable functions instead
                  // of [basic.lookup.argdep].
}
```

A customisable function prototype creates a name that identifies an empty object that can be passed around by value. This object represents the overload set of all overloads of that function, and so can be passed to higher-order functions without having to wrap it in a lambda.

```
template<typename T>
virtual void frobnicate(T& x) = 0;

struct X {
  friend void frobnicate(X& x) override { ... }
};
void example() {
  std::vector<X> xs = ...;              // 'frobnicate' is a callable-object that
  std::ranges::for_each(xs, frobnicate); // can be passed to a higher-order function.
}
```

5.6 Generic customisations

A type can customise a set of customisable-functions generically by defining namespace-scope **generic customisation**.

```
template<typename Obj, typename Member>
using member_t = decltype((std::declval<Obj>().*std::declval<Member Obj::*>()));

template<typename Inner>
struct logging_wrapper {
  Inner inner;
  // Forward calls with the first argument as 'logging_wrapper' to the inner object if callable
  // on the inner object after printing out the name of the CPO that is being called.
  template<auto cpo, typename Self, typename... Args>
    requires std::derived_from<std::remove_cvref_t<Self>, logging_wrapper> &&
             std::invocable<decltype(cpo), member_t<Self, Inner>, Args...>
  friend decltype(auto) cpo(Self&& self, Args&&... args) noexcept(/* ... */) override {
    std::print("Calling {}\n", typeid(cpo).name());
    return cpo(std::forward<Self>(self).inner, std::forward<Args>(args)...);
  }
};
```

A single `override` declaration is able to provide an overload for an open set of customisable functions by allowing the non-type template parameter `cpo` to be deduced to an instance of whichever customisation point object resolution is being performed for.

5.7 Customizable function templates

A customisable function template is declared with explicit template parameters. Calling the customisable function requires explicitly passing the template parameters and each specialisation of the customisable function results in an independent customisation point object.

Note: the ability to declare multiple variable templates of the same name but different parameter kinds is novel.

```
namespace std {
  // get for types
  template<typename T, typename Obj>
  virtual auto get(Obj&& obj) = 0;

  // get for numeric indices
  template<size_t N, typename Obj>
  virtual auto get(Obj&& obj) = 0;

  // get that deduces numeric indices, no template args
  template<size_t N, typename Obj>
  virtual auto get(Obj&& obj, std::integral_constant<size_t, N>) default
    -> decltype(auto) {
       return std::get<N>(std::forward<Obj>(obj);
  }
}

struct my_tuple {
  int x;
  float y;

  friend int&   std::get<int>(my_tuple& self) noexcept override { return self.x; }
  friend float& std::get<float>(my_tuple& self) noexcept override { return self.y; }

  friend int&   std::get<0>(my_tuple& self) noexcept override { return self.x; }
  friend float& std::get<1>(my_tuple& self) noexcept override { return self.y; }
```

```
};
```

Note: unlike variables and variable templates, CFOs are not less-than-comparable, which means `cfo-name<token>` is unambiguously a template name and not a comparison. This allows CFOs and CFO-templates to coexist in the same namespace.

6 Background

One of the main purposes of defining customisation points is to enable the ability to program generically. By defining a common set of operations that many different types can customize with their own type-specific behaviour we can write generic algorithms defined in terms of that common set of operations, and have them work on many different types. This is the cornerstone of generic programming.

The state of the art for defining customisable functions has evolved over time.

Early customisation points such as `std::swap()` make use of raw argument-dependent-lookup (ADL) but require a two-step process to call them (`using std::swap; swap(a, b);`) to ensure the customisation is found if one exists but with a fallback to a default implementation. It is a common programmer error to forget to do this two-step process and just call `std::swap(a, b)` which results in always calling the default implementation. Raw ADL calls can also give different results depending on the context in which you call them, which can lead to some hard to track down bugs.

The `std::ranges` customisation-point objects added in C++20 improved on this by encapsulating the two-step ADL call into a function object, making the customisation point easier to call correctly, but also making it much more complex to define a customisation point, as one needs to define two nested namespaces, poison pill declarations, `inline constexpr` objects, and a class with a constrained `operator()` overload.

The `tag_invoke` proposal [P1895R0] further refines the concept of customisation-point objects to use a single ADL name `tag_invoke` and instead distinguish customisations of different CPOs by passing the CPO itself as the first argument, using tag-dispatch to select the right overload. This simplifies definitions of customisation-point objects, enables generically customising many CPOs, and eliminates the issue of name conflicts inherent in ADL-based approaches when different libraries use the same function name for customisation points with different semantics by allowing names to be namespace-scoped.

Adding a first-class language solution for defining customisation points has been suggested before; Matt Calabrese's paper [P1292R0] *"Customization Point Functions"* suggests adding a language syntax for customisation points similar to the syntax proposed here.

Barry Revzin's paper [P2279R0] *"We need a language mechanism for customization points"* discusses what he sees as the essential properties of "proper customisation" in the context of `tag_invoke` and also seems to come to the conclusion that `tag_invoke`, despite being an improvement on previous solutions, still leaves much to be desired and that we should pursue a language solution.

"`tag_invoke` is an improvement over customization point objects as a library solution to the static polymorphism problem. But I don't really think it's better enough, and we really need a language solution to this problem. …"

A discussion of [P2279R0] in a joint library/language evolution session had strong consensus for exploring a language solution to the problem of defining customisation points. This is that paper.

POLL: *We should promise more committee time to exploring language mechanism for customization points ([P2279R0]), knowing that our time is scarce and this will leave less time for other work.*

SF	WF	N	WA	SA
30	12	2	0	0

The paper [P2300R4] *"`std::execution`"* proposes a design that heavily uses customisable functions which are currently based on `tag_invoke` as the customisation mechanism. If [P2300R4] is standardised with customisation

points defined in terms of `tag_invoke()`, retrofitting them to support the language-based solution for customisable functions proposed in this paper will still carry all the downsides of `tag_invoke` due to backwards compatibility requirements.

The added complexity of CPOs and abstractions to support both `tag_invoke` and a language solution may negate much of the benefit of using a language feature.

The committee should consider whether it is preferable to first standardise a language-based approach to customisable functions before adding a large number of customisable functions to the standard library based on `tag_invoke`.

7 Motivation

A primary motivation for writing this paper was based on experience building libraries such as libunifex, which implement the sender/receiver concepts from [P2300R4] and are heavily based on `tag_invoke` customisation point objects.

While the `tag_invoke` mechanism for implementing customisation points is functional and powerful, there are a few things that make it less than ideal as the standard-blessed mechanism for customisable functions.

- It requires a lot of boiler-plate when defining a new customisation point (see example below)
- Customisations are harder to read due to the function-name being found in the first argument position instead of in the function-name position (which is always `tag_invoke`).
- The extra layer of forwarding prevents copy-elision of arguments even if the customisation takes a parameter by-value.
- Every customisation uses the name `tag_invoke`, which can make for a large overload-set that the compiler has to consider. Types that customise a large number of CPOs have all of those customisations added to the overload-set for every call to a CPO with that type as an argument. This can impact compile times when used at scale, and heavily impacts user-visible error messages.
- The `tag_invoke` forwarding mechanism requires a lot of template instantiations when instantiating the call operator (`tag_invocable` concept for constraint, `tag_invoke_result_t` for return-type, `is_nothrow_tag_invocable_v` to forward `noexcept` clause, the `std::tag_invoke_t::operator()` template function. This can potentially impact compile-time in code where there are a large number of calls to CPOs.

For example, defining a hypothetical `std::ranges::contains` customisable function with a default implementation requires a lot of boiler-plate with `tag_invoke`.

tag_invoke ([P1895R0])

```
namespace std::ranges {

struct contains_t {
  template<input_range R, typename Value>
    requires
      tag_invocable<contains_t, R, const Value&> &&
      equality_comparable_with<
        range_reference_t<R>, Value>
  auto operator()(R&& range, const Value& v) const
    noexcept(is_nothrow_tag_invocable_v<
               contains_t, R, const Value&>)
    -> tag_invoke_result_t<contains_t, R,
                           const Value&>
  {
    return std::tag_invoke(contains_t{}, (R&&)range, value);
  }

  template<input_range R, typename Value>
    requires
      !tag_invocable<contains_t, R, const Value&> &&
      equality_comparable_with<range_reference_t<R>, Value>
  bool operator()(R&& range, const Value& v) const
  {
    for (const auto& x : range) {
      if (x == v) return true;
    }
    return false;
  }
};
inline constexpr contains_t contains{};
} // namespace std::ranges
```

This proposal

```
namespace std::ranges {

template<input_range R, typename Value>
  requires
    equality_comparable_with<range_reference_t<R>, Value>
virtual bool contains(R&& range, const Value& v) default
{
  for (const auto& x : range) {
    if (x == v) return true;
  }
  return false;
}

} // namespace std::ranges
```

When reading code that customises a function, it is difficult for the eye to scan over the declaration to see which function is being customised. You need to look for the template argument to `std::tag_t` in the first argument of the customization instead of in the function-name position, where most editors highlight the name.

Barry's discussion paper [P2279R0] contains further critique of `tag_invoke` and other customisation point mechanisms along several axes:

1. The ability to see clearly, in code, what the interface is that can (or needs to) be customised.
2. The ability to provide default implementations that can be overridden, not just non-defaulted functions.
3. The ability to opt in *explicitly* to the interface.
4. The inability to *incorrectly* opt in to the interface (for instance, if the interface has a function that takes an `int`, you cannot opt in by accidentally taking an `unsigned int`).
5. The ability to easily invoke the customised implementation. Alternatively, the inability to accidentally invoke the base implementation.
6. The ability to easily verify that a type implements an interface.
7. The ability to present an atomic group of functionality that needs to be customised together (and diagnosed early).
8. The ability to opt in types that you do not own to an interface non-intrusively.
9. The ability to provide associated types as part of the customisation of an interface.

10. The ability to customise multiple CPOs generically (e.g. for forwarding to a wrapped object)

The proposal in this paper addresses most of these axes, improving on [P1292R0] customisation point functions by adding better support for diagnosis of incorrect customisations and adding the ability to generically customise multiple customisation-points and forward them to calls on wrapped objects.

This proposal is not attempting to solve the "atomic group of functionality" or the "associated types" aspects that [P2279R0] discusses. Although in combination with C++20 concepts it does a reasonable job of matching the simplicity of Rust Traits (see Comparison to Rust Traits). The authors do not believe that this proposal would prevent the ability to define such an atomic group of functionality as a future extension to the language, or even as a library feature.

While [P2279R0] does a great job of surveying the existing techniques used for customisation points, we want to further elaborate on some limitations of those techniques not discussed in that paper.

7.1 The problem with member-functions

7.1.1 Member functions all live in a single global "namespace"

— (from the perspective of generic programming)

If a generic concept defined by one library wants to use the `foo()` member function name as a customisation point, it will potentially conflict with *another* library that also uses the `foo()` member function name as a customisation point with different implied semantics.

This can lead to types accidentally satisfying a given concept syntactically even if the semantics of the implementations don't match because they are implementing a different concept.

It can also make it impossible to implement a type that satisfies two concepts if both of those concepts use conflicting customisation point names. e.g. the somewhat contrived example:

```
namespace GUI {
  struct size { int width; int height; };

  template<typename T>
  concept widget =
    requires (const T& w) {
      { w.size() } -> std::same_as<GUI::size>;
    };
}

namespace CONTAINER {
  template<typename T>
  concept sized_container =
    requires (const T& c) {
      { c.size() } -> std::same_as<typename T::size_type>;
    };
}
```

A `composite_widget` type that wants to be a `sized_container` of widgets but also a widget itself would not be able to simultaneously satisfy both the concepts as it can only define one `size()` member function.

7.1.2 We cannot customise different member names generically

If one wants to build a wrapper type that customises only one customisation point and *forwards the rest*, or that type-erases objects that support a given concept, one needs to implement a new class for each set of member names they want to forward.

e.g. For each concept we need to define a new wrapper type instead of being able to define a wrapping pattern generically.

```
template<foo_concept Foo>
class synchronised_foo {
  Foo inner;
  std::mutex mut;

  void foo() { std::unique_lock lk{mut}; inner.foo(); }
};

template<bar_concept Bar>
class synchronised_bar {
  Bar inner;
  std::mutex mut;

  void bar() { std::unique_lock lk{mut}; inner.bar(); }
  void baz() { std::unique_lock lk{mut}; inner.baz(); }
};
```

7.1.3 Customisation-point member functions cannot be passed to higher-order functions

Generally, we need to wrap up the call to an object in a generic lambda so we can pass it as an object representing an overload set.

```
const auto foo = [](auto&& x) -> decltype(auto) {
  return static_cast<decltype(x)>(x).foo();
};
```

7.1.4 Implementing a concept requires modification of the type

We cannot define new customisations of functions for types we cannot modify if they must be defined as member functions.

A common workaround for this is to use the CRTP pattern (or, since C++23, [P0847R7] *"Deducing this"*) to have each type inherit from some base class that provides default implementations for common operations, but this is effectively providing a way to modify types that inherit from it, and not purely unrelated types.

e.g. We can define two types that implement a concept and that both inherit from `foo_base`:

```
struct foo_base {
  void thing1(this auto& self) {
    std::print("default thing1\n");
  }
};

struct foo_a : foo_base {};

struct foo_b : foo_base {
  void thing1() { std::print("foo_b thing1\n"); }
};
```

We can then later extend `foo_base` to add additional members with default implementations in a non-breaking fashion.

```
struct foo_base {
  void thing1(this auto& self) {
    std::print("default thing1\n");
```

```
  }

  void thing2(this auto& self, int n) {
    std::print("default thing2\n");
    while (n-- > 0) self.thing1();
  }
};
```

It may not be possible to retrofit or enforce that all types that satisfy a concept use the CRTP base, however.

8 References

[P0847R7] Barry Revzin, Gašper Ažman, Sy Brand, Ben Deane. 2021-07-14. Deducing this.
https://wg21.link/p0847r7

[P1292R0] Matt Calabrese. 2018-10-08. Customization Point Functions.
https://wg21.link/p1292r0

[P1895R0] Lewis Baker, Eric Niebler, Kirk Shoop. 2019-10-08. tag_invoke: A general pattern for supporting customisable functions.
https://wg21.link/p1895r0

[P2279R0] Barry Revzin. 2021-01-15. We need a language mechanism for customization points.
https://wg21.link/p2279r0

[P2300R4] Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. 2022-01-19. std::execution.
https://wg21.link/p2300r4