Document	P2547R0
Date	2022-02-15
Reply To	Lewis Baker (Woven-Planet) <lewissbaker@gmail.com> Corentin Jabot <corentinjabot@gmail.com> Gašper Ažman <gasper.azman@gmail.com>
Audience	Evolution

Language support for customisable functions

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 customization points”.

Status of this proposal

This work is a preliminary initial design. We intend this proposal to replace the use of `tag_invoke` in P2300 in the C++26 time frame. We need directional guidance from EWG and LEWG to refine the design over the next few months.

Table of contents

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 Customization Point Objects 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 CPOs 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 customization 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 P1292 “Customization 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 customizable function object for the purposes of generic forwarding.

Terminology

This paper introduces the following terms

- A function declaration for a CPO is a **customisable function prototype (CFP):**
  ```
  virtual void foo(auto&&) = 0;
  ```
- The set of **customisable function prototypes** naming the same entity represent a **customisable function.**
- A **customisable function** introduces a **Customizable Function Object** in the scope it is declared.
- The set of functions or template functions with the `override` keyword are **customizations** of the corresponding CPO. They form the **customizations overload set.**
- The set of functions or template functions with the `default` keyword are **default implementations** of the corresponding CPO. They form the **default overload set** for that CPO.
- A declaration of namespace-scope function of the form
  ```
  template <auto cpo, typename T> auto cpo(T&& object) override;
  ```
  declares a **generic customisation.**

Examples

Declaring a customisable function using the ‘`virtual`’ keyword.
The trailing `= 0` indicates that 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;
}
```

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;
  }
}
```

Defining a customisation of the above customisable function `contains` (as a hidden friend) for `std::set` using the `override` keyword. the name of the customised CPO 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);
    }
  };
}
```

Defining a customisation of the above customisable function `contains` (at namespace scope) using the override keyword. This can be useful when you want implicit conversions to 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);
  }
}
```

Calling a customisable function (no need for two-step using when calling, 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, like 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" declared above. This 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.
}
```

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)...);
  }
};
```

Here 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.

A template customisable function 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.

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

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

  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, CPOs are not less-than-comparable, which means `cpo-name<token>` is unambiguously a template name and not a comparison. This allows CPOs and CPO-templates to coexist in the same namespace.

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 implement to have type-specific behaviour then we can write generic algorithms that are defined in terms of that common set of operations and have them work on many different types. This is the cornerstone approach 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 (need to define two nested namespaces, poison pill declarations, inline constexpr objects, and a class with a constrained operator() overload).

The `tag_invoke` proposal P1895 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 find the right overload. This simplifies the 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” suggested 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 the 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.

> 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.

Strongly Favor	Weakly Favor	Neutral	Weakly Against	Strongly Against
30	12	2	0	0

The paper P2300R3 “std::execution” proposes a design that heavily uses customisable functions which are currently based on the use of `tag_invoke` as the customisation mechanism. If P2300 is standardised with customisation points defined in terms of `tag_invoke()`, it may be difficult (or impossible) to later retrofit them to support the language-based solution for customisable functions proposed in this paper.

Even if we find a way to make it possible, the added complexity of CPOs and abstractions around them having 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`.

Motivation

A primary motivation for writing this paper was based on experience building libraries such as libunifex, which implement the sender/receiver concepts from P2300 and are heavily based on tag_invoke-based 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.
- 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 harder for the eye to scan over the declaration to see which function is being customised as you need to look for the template argument to `std::tag_t` in the first argument 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). I 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.

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.

The problem with member-functions

Member functions all live in a single global namespace.
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.

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()` method.

Unable to customise different member names generically.
If I want 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 then I need to implement a new class for each set of member names I 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(); }
};
```

Unable to directly pass a customisation-point as a parameter to a higher-order function.
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();
};
```

Requires modification of the type
We cannot define new customisations of functions for types we cannot modify if they must be defined as a member function.

A common workaround for this is to use the CRTP pattern (or in C++23, 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.

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.

The problem with raw argument-dependent lookup (ADL)

This is the technique used by facilities like `std::swap()`.

A default implementation is defined in some namespace and then customisations are placed in an associated namespace of the arguments to the type. e.g.

```
namespace std {
  template<typename T>
  void swap(T& a, T& b) {
    T tmp = move(a);
    a = move(b);
    b = move(tmp);
  }
}

namespace customising_lib {
  struct X {
    friend void swap(X& a, X& b) { /* More efficient implementation */ }
  };
}

namespace consumer {
  template<typename T>
  void reverse(std::vector<T>& v) {
    for (size_t i = 0; i < v.size() / 2; ++i) {
      using std::swap;                 // Step 1. Make default available.
      swap(v[i], v[v.size() - i - 1]); // Step 2. Call unqualified.
    }
  }
}
```

ADL function names all live within a single global namespace
Similar to the limitation of member functions, which have a single global namespace that all member function names conceptually live within, there is also a single global namespace that all functions intended to be called using ADL conceptually live within.

If two libraries decide to use the same ADL name as a customisation point then it is possible that the usages of those libraries may conflict. This can lead to either an inability for a type to implement concepts from both libraries or the potential for a type to implement the concept from one library and accidentally match the concept from another library.

Unable to customise different ADL names generically
Similar to the limitation of member functions, customisation points defined in terms of raw ADL need to know the name of the ADL function in order to customise it.

This means we cannot build wrapper types that generically customise and forward on multiple customisation point calls to a child object. We need to explicitly customise and forward on the calls for each ADL name. This necessitates

Unable to pass a customisation-point as a parameter to a higher-order function
We cannot just pass an ADL function name to a higher-order function as a parameter. A name that names a function must resolve to a specific overload when used outside of a call-expression, so that the name can resolve to a function pointer.

The issues here are covered in more detail in P1170R0 “Overload sets as function parameters”. One of the motivating examples from that paper is reproduced here for convenience.

e.g. While we can write:

```
namespace N {
  struct X { ... };
  X getX();
}

foo(N::getX()); // which calls some function 'foo'
```

we can’t necessarily write:

```
template <typename F>
void algorithm(F f, N::X x) {
  f(x);
}

algorithm(foo, N::getX());
```

As `foo` could be a function template, name an overload-set or be a function that was only found by ADL, or any of several other situations, this code may or may not be valid code.

If ‘foo’ was intended to be a customisation point, it would almost always be an overload set.

One common workaround to this is to wrap the call in a lambda. e.g.

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

However, this is a lot of boiler-plate to have to remember to write at every call site where you want to pass ‘foo’ as an overload-set.

Need for two-step using/call to call functions with default implementations (e.g. swap())
The need to perform the two-step using/call is unfortunate due to the extra line of code you need to write.

This also complicates deducing what the return-type of a call to the customisation point is. e.g. to determine the iterator type returned by begin() - we can't add the 'using std::begin;' default implementation to a decltype() expression.

```
// This won't find the default implementation.
template<typename Rng>
using range_iterator_t = decltype(/* what to write here? */);
```

Instead, you need to wrap up the two-step call mechanism in a namespace that has brought in the default implementation. e.g.

```
namespace _begin {
  using std::begin; // Also shadows 'begin' defined in parent namespace.

  template<typename Rng>
  using range_iterator_t = decltype(begin(std::declval<Rng&>()));
}
using range_iterator_t;
```

A bigger problem, however, is that callers might accidentally call the default function explicitly.
i.e. instead of writing

```
using std::swap;
swap(a, b);
```

a programmer might write

```
std::swap(a, b);
```

The compiler will no longer perform ADL when resolving overloads of this call and so it will generally not find any customisations of `swap()` for the types passed to it, instead always calling the default implementation. For many types, the default implementation of swap() will still silently compile just fine - it may just run a lot slower than if it called the custom implementation.

ADL is location-dependent and therefore nondeterministic and difficult to debug when wrong
When a raw ADL customisation-point is defined that is intended to be called unqualified so that ADL finds the right customisation but where there is no default implementation, the two-step using+call process is generally not required when calling the customisation-point.

However, this means that the call-site will consider any overloads declared in parent namespaces which can make the overload that a call dispatches to context-sensitive.

For example: Two call expressions to the same ADL name from different contexts resolve to different overloads despite the arguments to those expressions having the same type.

```
namespace ns1 {
  struct X {};

  int foo(X const&) { return 1; }
}

namespace ns2 {
  int foo(ns1::X& x) { return 2;}
}

namespace ns3 {
  void bar() {
    ns1::X x;
    foo(x); // calls ns1::foo(ns1::X const&)
  }
}

namespace ns2 {
  void baz() {
    ns1::X x;
    foo(x); // calls ns2::foo(ns1::X&)
  }
}
```

This problem does not usually come up when using the two-step using, however, as the ‘using’ declaration shadows the declarations from the parent namespace.

The problem with CPOs defined in ranges

The `std::ranges` library defines a number of customisation point objects which build on ADL but that make them easier to use correctly by encapsulating the two-step using approach.

For example: A rough approximation of the `std::ranges::swap()` CPO defined

```
namespace std::ranges
{
  namespace __swap {
    void swap(); // poison-pill so we don't find std::swap

    template<typename T, typename U>
    concept __has_swap = requires(T&& t, U&& u) {
      swap(static_cast<T&&>(t), static_cast<U&&>(u));
    };

    struct __fn {
      template<typename T, typename U>
        requires __has_swap<T, U>
      void operator()(T&& t, U&& u) const
          noexcept(noexcept(swap(static_cast<T&&>(t), static_cast<U&&>(u))) {
        swap(static_cast<T&&>(t), static_cast<U&&>(u));
      }

      template<typename T>
        requires (!__has_swap<T>) && movable<T>
      void operator()(T& a, T& b) const
          noexcept(std::is_nothrow_move_constructible_v<T> &&
                   std::is_nothrow_move_assignable_v<T> &&
                   std::is_nothrow_destructible_v<T>) {
        T tmp(std::move(a));
        a = std::move(b);
        b = std::move(tmp);
      }

      // etc. for other default implementations (e.g. for swapping arrays)
    };
  }
  // Function object needs to be in a separate namespace
  // And imported using 'using namespace' to avoid conflicts with
  // hidden-friend customisations defined for types in std::ranges.
  namespace __swap_cpo {
    inline constexpr __swap::__fn swap{};
  }
  using namespace __swap_cpo;
}
```

Defining customisation-points as objects has a number of benefits:

- Callers no longer need to use the two-step using+call to call it correctly.
  Simplifies usage and makes the API safer to use.
- The function is an object so can be passed as an argument to a higher-order function as an overload set (although current std::ranges specifications prohibit this).
- Still lets customisations be found by ADL - customisations are defined as function overloads with the name ‘swap’.

However, there are still some problems with this approach.

Still relies on ADL for customisation. Single global namespace.
This approach still generally relies on the same approach as raw ADL which has a single-global namespace for customisation-point names and so is susceptible to conflicts between libraries.

Unable to generically customise and forward implementations in wrapper types.
Wrapper types that want to forward customisations to wrapped objects still need to know the names of all CPOs to be forwarded as they need to explicitly define customisations using those names.

It is not possible to define a generic customisation that can forward calls to many different customisation-points.

Lots of boiler-plate needed to define a CPO
Defining a CPO in this way requires writing a lot of boiler-plate. This can often obscure the intent of the CPO.

E.g. the std::ranges::swap() CPO above requires:

- defining two private namespaces
- adding a poison pill declaration
- defining a function object type
- defining multiple `operator()` overloads
- manually constraining `operator()` overloads to prefer a customisation over a default.
- inline constexpr instance of callable object

Many of the reasons for defining this way are subtle and require a high-level of understanding of ADL and various corner-cases of the language.

The extra layer of indirection through the operator() overload inhibits copy-elision
Even if the caller of a CPO invokes the function with a prvalue and this resolves to calling a customisation that takes the parameter by-value, the fact that the call goes through an intermediate call to operator(), which typically takes its arguments by universal-reference and “perfectly forwards” those arguments to the customisation, means that copy-elision of the argument will be inhibited.

For example:

```
namespace _foo {
  void foo();
  struct _fn {
    template<typename T, typename V>
      requires requires (T& obj, V&& value) {
        foo(obj, (V&&)value);
      }
    void operator()(T& obj, V&& value) const {
      foo(obj, (V&&)value);
    }
  };
}
inline namespace _foo_cpo {
  inline constexpr _foo::_fn foo{};
}

struct my_type {
  friend void foo(my_type& self, std::string value);
};

void example() {
  my_type t;
  foo(t, std::string{"hello"}); // will result in an extra call to
                                // std::string's move constructor.
}
```

Whereas, if we were using raw ADL then the foo() call would have resolved directly to a call to the customisation and copy-elision would have been performed.

Related to this extra level of indirection are some (minor) additional costs:

- stepping-through a call to a CPO when debugging means having to step through the intermediate `operator()`.
- Debug builds that do not inline this code have to execute the intermediate function, slowing debug execution performance (some game developers have cited debug performance as an issue)
- There are compile-time costs to instantiating the intermediate `operator()` function template.

The problem with `tag_invoke` CPOs

The paper P1895R0 first introduced the tag_invoke technique for defining customisation points, which tries to solve some of the limitations of customisation-point objects.

In particular it tries to address the following issues:

- removing need for a single global namespace for customisation point names by customising based on a tag-type, which can be namespace scoped
- allowing defining customisations to generically customise multiple CPOs e.g. for forwarding on to an wrapped object

Defining a CPO using `tag_invoke` is much like defining a `std::ranges` customisation-point object, however instead of dispatching to a named ADL call, we instead dispatch to a call to `tag_invoke`(), with the CPO object itself as the first parameter.

For example:

```
namespace N {
  struct contains_t {
    template<range R, typename V>
      requires std::tag_invocable<contains_t, R&&, const V&>
    bool operator()(R&& rng, const V& value) const
        noexcept(std::is_nothrow_tag_invocable_v<contains_t, R&&, const V&>) {
      return std::tag_invoke(contains_t{}, (R&&)rng, value);
    }

    // A default implementation if no custom implementation is defined.
    template<range R, typename V>
      requires (!std::tag_invocable<foo_t, const T&, const U&>) &&
                std::equality_comparable<
                  std::ranges::range_reference_t<R>, V>
    bool operator()(R&& rng, const V& value) const {
      for (const auto& x : rng) {
        if (x == value) return true;
      }
      return false;
    }
  };

  inline constexpr foo_t foo{};
}
```

Note that there is no longer a need to define the CPO and the instance of the function object in a nested namespace and no longer a need to define a poison-pill overload of the customisation-point. The `std::tag_invoke` CPO now handles these aspects centrally.

Then a type can customise this CPO by defining an overload of `tag_invoke`. e.g.

```
class integer_set {
  std::uint32_t size_;
  std::unique_ptr<std::uint32_t[]> bits_;
public:
  // ... iterator, begin(), end() etc. omitted for brevity.
  template<typename U>
  friend bool tag_invoke(std::tag_t<N::contains>,
                         const bit_set& self, std::uint32_t value) noexcept {
    if (value >= size) return false;
    return (bits[value / 32] >> (value & 31)) != 0;
  }
};
```

Also, with `tag_invoke`, a generic wrapper can customise and forward calls to many CPOs generically. e.g.

```
template<typename Inner>
struct wrapper {
  Inner inner;

  template<typename CPO, typename... Args>
    requires std::invocable<CPO, Inner, Args...>
  friend decltype(auto) tag_invoke(CPO cpo, wrapper&& self, Args&&... args)
    noexcept(std::is_nothrow_invocable_v<CPO, Inner, Args...>) {
    return cpo(std::move(self).inner, std::forward<Args>(args)...);
  }
};
```

This can be used for building generic type-erasing wrappers, or types that generically forward through queries to wrapped types (e.g. this is used often when writing P2300 receiver types).

There are still some problems / limitations with `tag_invoke`, however.

Still lots of boiler-plate to define a new CPO
While the amount of boiler-plate has been reduced compared to std::ranges-style CPOs, there is still a need to define a struct with operator() overloads that detect whether customisations are defined and conditional forward to the customisation or to the default implementation.

Syntax for customisations is more cumbersome
Defining customisations requires defining a tag_invoke() overload rather than defining a overload of a function named after the operation you are customising.

Potential for very large overload-set
As every customisation of `tag_invoke`-based CPOs necessarily defines a `tag_invoke()` overload, this means that all `tag_invoke()` overloads defined for all associated types of all arguments to a CPO will be considered when resolving the appropriate overload to a call to a CPO.

This can potentially lead to increased compile-times for types that add a lot of `tag_invoke`-based customisations.

Overload set sizes can be reduced somewhat by careful use of hidden-friend definitions and ADL isolation techniques[1] for class templates. However, types that need to customise a large number of CPOs will still have an overload-resolution cost proportional to the number of CPOs that they customise.

Note that raw ADL does not usually have the same problem as different names are typically used for each customisation point and so overload resolution only needs to consider functions with the same name as the function currently being called and can ignore overloads for other customisation-points.

Forwarding layers inhibit copy-elision
The tag_invoke approach inherits the down-sides of CPOs regarding inhibiting copy-elision of arguments as it also introduces a layer of forwarding through the `operator()` function.

Also, as implementations generally dispatch to `tag_invoke()` through the std::tag_invoke() CPO rather than doing direct ADL calls, this usually introduces two levels of forwarding, increasing the debug runtime and function template instantiation costs compared with std::ranges-style CPOs.

- Syntax is cumbersome
- Lots of boiler-plate to declare a CPO
- Syntax for customising is more verbose / harder to follow than ADL
- Potentially very large overload set
    - Requires everyone to play nice and define `tag_invoke` overloads as hidden friends

Overload sets can get large because of the single name `tag_invoke` unless great care is taken with template parameter cleanliness and hidden friends to keep them small.
Even still, some types may customise a large number of CPOs meaning each of the overloads for the customised CPOs are considered for every call to a CPO taking that type.

Compile Errors are difficult to diagnose
If you attempt to call a tag_invoke-based CPO with arguments that do not match an available customisation / default-implementation, the compile-errors can be difficult to wade-through.

There are two levels of forwarding and indirection which add extra lines to the diagnostic output, plus the compiler needs to list all of the tag_invoke overloads it considered when looking for a valid call - which might be a large list containing may `tag_invoke()` overloads unrelated to the call.

[1] https://quuxplusone.github.io/blog/2019/04/09/adl-insanity-round-2/

Use-cases

This section discusses some of the use-cases for customisable functions proposed in this paper.

Basis operations

We can use customisable functions to declare a set of basis operations that are included in a generic concept.

Basis operations are operations that do not have any default implementation and that need to be customised for a given type (or set of types) before they can be called.

For example, the `std::ranges::begin` and `std::range::end` CPOs are basis operations for implementing the range concept. They have no default implementation and they can be customised either by defining a member function or a namespace-scope function found by ADL.

For the receiver concept defined in P2300R0 std::execution paper, there are three separate basis operations; `set_value(receiver, values…)`, `set_error(receiver, error)` and `set_done(receiver)`.

Customisable algorithms

Customisable algorithms are another major use-case for this proposal.

A customisable algorithm differs from a basis-function in that it has a default implementation that is implemented in terms of some other set of basis operations. If a customisation of the algorithm is not provided for a particular set of argument types then the call to a customisable algorithm dispatches to a default implementation implemented generically in terms of some basis operations implemented by the arguments (which are typically constrained to require such basis operations for all overloads).

Wrapper types

One of the use-cases that needs to be supported for P2300-style customisation points is the ability to generically customise many customisation points and forward them on to a wrapped object.

This is used extensively in the definition of receiver types, which often need to customise an open-set of queries about the calling context and forward them on to the parent receiver if the answer is not known locally.

For example, a receiver defined using `tag_invoke` proposed by P2300 often looks like this:

```
template<typename T, typenname... Ts>
concept one_of = (std::same_as<T, Ts> || ...);
template<typename T>
concept completion_signal = one_of<T, std::tag_t<std::execution::set_value>,
                                      std::tag_t<std::execution::set_error>,
                                      std::tag_t<std::execution::set_done>>;

template<typename ParentReceiver>
struct my_receiver {
  my_operation_state<ParentReceiver>* op;

  friend void tag_invoke(tag_t<std::execution::set_value>,
                         my_receiver&& r, int value) noexcept {
    // transform the result and forward to parent receiver
    std::execution::set_value(std::move(r.op->receiver), value * 2);
  }

  // Forward through error/done unchanged
  template<typename Error>
  friend void tag_invoke(tag_t<std::execution::set_error>,
                         my_receiver&& r, Error&& error) noexcept {
    std::execution::set_error(std::move(r.op->receiver), std::forward<Error>(error));
  }

  friend void tag_invoke(tag_t<std::execution::set_done>, my_receiver&& r) noexcept {
    std::execution::set_done(std::move(r));
  }

  // Generically forward through any receiver query supported upstream.
  template<typename CPO>
    requires (!completion_signal<CPO>) &&
             std::invocable<CPO, const ParentReceiver&>
  friend auto tag_invoke(CPO cpo, const my_receiver& r)
      noexcept(std::is_nothrow_invocable_v<CPO, const ParentReceiver&>)
      -> std::invoke_result_t<CPO, const ParentReceiver&> {
    return cpo(std::as_const(r.op->receiver));
  }
};
```

Another use-case for building generic forwarding wrappers types is building generic type-erasing wrappers.

For example, the `libunifex::any_unique<CPOs…>` type defines a type-erasing wrapper that can generically type-erase any type that satisfies the set of CPOs.

```
virtual float width(const T& x) = 0;
virtual float height(const T& x) = 0;
virtual float area(const T& x) = 0;

template<typename T>
concept shape = requires(const T& x) {
  width(x);
  height(x);
  area(x);
};

struct square {
  float size;
  friend float width(const square& self) override { return size; }
  friend float height(const square& self) override { return size; }
  friend float area(const square& self) override { return size * size; }
};

// Define a type-erased type in terms of a set of CPO signatures.
using any_shape = unifex::any_unique<
  unifex::overload<float(const unifex::this_&)>(width),
  unifex::overload<float(const unifex::this_&)>(height),
  unifex::overload<float(const unifex::this_&)>(area)>;

any_shape s = square{2.0f};
assert(width(s) == 2.0f);
assert(height(s) == 2.0f);
assert(area(s) == 4.0f);
```

Building a generic type-erasing wrapper that can be used with any set of customisable-functions is made possible by the ability to generically customise on a CPO without needing to know its name. This ability is discussed further in Generic forwarding.

Other type-erasing wrapper types are possible that make different choices on storage, copyability/movability, ownership, comparability axes.

Design

A **customisable function prototype** is a namespace-scope function declaration prefixed with the `virtual` keyword.

Calls to customisable functions dispatch to the appropriate customisation or default implementation based on the static types of the arguments to that call.

Declaring customisable functions

A namespace-scope function declarator declares a *customisable function prototype* if the declaration contains the `virtual` keyword in its function-specifier.

A declaration of a customisable function prototype that has no default (i.e. a basis operation) is introduced with the `'virtual'` keyword, indicating a customisable function prototype, and an '`= 0;`', indicating that there is no default defined inline.

For example, you can declare a `hash()` customisable function prototype that may be used by hash-table implementations in a container library but where there is no generic default implementation possible.

```
namespace containers
{
  template<typename T, std::unsigned_integral InitialValue>
  virtual InitialValue hash(const T& value, InitialValue iv) noexcept = 0;
}
```

Note that the declaration of a customisable function prototype is always a template (otherwise it wouldn't be customisable).

It is ill-formed to declare a normal function of the same name as a customisable function in the same namespace as the latter declares the name to be an object or variable template.

Customisable function CPO

Each customisable function introduces a Customisation Point Object, which is a constexpr object of implementation-defined trivial type with no data members.

A CPO exists as a way to name the customisable function. It allows the implementation of generic forwarders, and can more generally be used to pass an overload set generically.

Neither default implementations or customisations are member functions of CPOs, and a CPO doesn't have call operators either. Instead, a call expression on a CPO triggers lookup and overload resolutions specific to customisable functions.

It should be possible to inherit from CPO types. CPO types are default-constructible. All objects of a given CPO type are structurally identical and are usable as NTTPs. Members of CPO type are implicitly `[[no_unique_address]]`.

Declaring Customizations and Default Implementations

Customisations and Default Implementations are implementations of Customisable functions.

They form separate overload sets: only if overload resolution finds no suitable customisations do we look for a default implementation.

A customisable function can have multiple default implementations and multiple customisations.
Default implementations and customisations are never found by normal lookup. It is not possible to take their address.

Declaring default implementations

We can declare zero or more default implementations or implementation templates for any customizable function.

Default implementations are only considered during overload resolution if no viable customisation of the function is found.

Default implementations are overloads of the customisable function that are declared with the '`default`' keyword. The keyword is placed in the same syntactic location of the function definition that the '`override`' and '`final`' keywords are placed for member functions. i.e. after the noexcept and any trailing return-type.

A default implementation can either be defined inline with the declaration or can be defined later as a separate '`default`' declaration.

For example: Declaring a hypothetical customisable `std::ranges::contains()` algorithm with a default implementation provided at the point of declaration.

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

This definition is semantically equivalent to a separate declaration of the customisable function and a declaration with inline definition of the default implementation. i.e. is equivalent to the following:

```
namespace std::ranges
{
  template<input_range R, typename V>
  requires equality_comparable_with<range_reference_t<R>, const V&>
  virtual bool contains(R&& range, const V& value) = 0;

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

It is also valid to forward-declare a default implementation and define it later. e.g. we could write

```
namespace std::ranges {
  // Declaration of customisable function.
  template<input_range R, typename V>
  requires equality_comparable_with<range_reference_t<R>, const V&>
  virtual bool contains(R&& range, const V& value) = 0;

  // Forward declaration of default implementation.
  template<input_range R, typename V>
  requires equality_comparable_with<range_reference_t<R>, const V&>
  bool contains(R&& range, const V& value) default;
}

// ... later

namespace std::ranges {
  // Definition of default implementation.
  template<input_range R, typename V>
    requires equality_comparable_with<range_reference_t<R>, const V&>
  bool contains(R&& range, const V& value) default {
    for (auto&& x : range) {
      if (x == value) return true;
    }
    return false;
  }
}
```

Default implementations should be declared separately from the customisable function prototype in cases where we don't want to constrain all customisations to have the same return-type or noexcept specification as the default implementation, even though the default implementation has the same constraints on its parameters as the customisable function prototype and so is valid for all arguments that would be valid for the customisable function prototype.

For example, if we wanted to make a customisable sender algorithm, customisations might only be required to return some type that satisfies the `sender` concept, whereas the default implementation will need to return a particular sender-type (something that we wouldn't want all customisations to have to return).

In this case we need to declare the customisation point with general constraints on the return-type but with a pure virtual descriptor, and define a separate default implementation that returns a particular concrete type that satisfies those constraints. e.g.

```
namespace std::execution {
  // Declaration of customisation-point defines only general constraints.
  template<sender S, typename F>
  requires invocable-with-sender-value-results<F, S>
  sender auto then(S&& src, F&& func) = 0;

  // Class used by default implementation. Satisfies 'sender' concept.
  template<sender S, typename F>
  struct default_then_sender { ... };

  // Default implementation returns a particular implementation.
  template<sender S, typename F>
  requires invocable-with-sender-value-results<F, S>
  default_then_sender<remove_cvref_t<S>, remove_cvref_t<F>>
  then(S&& src, F&& func) default {
    return ...;
  }
}
```

We can also declare a customisable function with default implementations that are only valid for types that satisfy some additional constraints by declaring the customisable function first with the `= 0;` and then declaring additional overloads with the `'default'` keyword.

For example: Declaring the `swap()` customisable function (which permits swapping between any two types) but only providing a default that works with types that are move-constructible/move-assignable.

```
namespace std::ranges {
  template<typename T, typename U>
  virtual void swap(T&& t, U&& u) = 0;

  template<typename V>
  requires std::move_constructible<V> && std::assignable_from<V&, V&&>
  void swap(V& t, V& u)
      noexcept(std::is_nothrow_move_constructible_v<V> &&
               std::is_nothrow_assignable_v<V&, V>) default {
    V tmp = std::move(t);
    t = std::move(u);
    u = std::move(tmp);
  }
}
```

Multiple default overloads can be defined.
For example, we can extend the above `swap()` example to also define a default for swapping arrays.

```
namespace std::ranges {
  template<typename T, typename U>
  concept nothrow_swappable_with =
  requires(T&& t, U&& u) { { swap((T&&)t, (U&&)u); } noexcept };

  template<typename T, typename U, std::size_t N>
    requires nothrow_swappable_with<T&, U&>
  void swap(T(&t)[N], U(&u)[N]) noexcept default {
    for (std::size_t i = 0; i < N: ++i) {
      swap(t[i], u[i]);
    }
  }
}
```

Declaring default implementations in other namespaces

It is possible to declare a default implementation for a customisable function, even if the declaration is textually within a different namespace, by fully-qualifying the function name to name the customisable function that the default is being declared for.

This can be useful in cases where you want to define a default implementation of a customisable function in terms of some concept local to a library such that it is valid for all types that implement that concept.

e.g.

```
namespace containers {
  template<typename T, std::unsigned_integral IV>
  virtual IV hash(const T& value, IV iv) noexcept = 0;
}
namespace otherlib {
  // Hash function used for types in this library
  template<typename T>
  virtual std::unsigned_integral hash(const T& value) noexcept = 0;
  template<typename T>
  concept hashable = requires(const T& value) { hash(value); };

  // Default implementation of containers::hash() so that hash-containers
  // from containers:: library can use hash-function of types defined in
  // this library.
  template<hashable T, std::unsigned_integral IV>
  IV containers::hash(const T& value, IV iv) noexcept default {
    return static_cast<IV>(otherlib::hash(value)) ^ IV;
  }
}
```

Even though the declaration of this default is textually within the `otherlib` namespace, and name lookup of names evaluated within the definition are looked up in the scope of `otherlib`, the definition itself is only found when looking for a `containers::hash` default implementation.

Why do we need defaults to be looked up as fallback?

If we provide a customisation for a type that accepts a const-ref where we have a default that accepts a universal-reference, we would often prefer overload resolution to find the less-perfect match of the customisation rather than the default implementation.

For example: If we consider the `contains()` algorithm from the Examples section

It has a default implementation with signature:

```
template<range R, typename Value>
requires equality_comparable_with<range_reference_t<R>, Value>
bool contains(R&& range, const Value& v) default { ... }
```

and then we have a customisation that takes a first argument of type `const set&`.

```
template<class Key, class Compare, class Allocator>
class set {
  //...
  template<class Value>
  requires (const set& s, const Value& v) { s.contains(v); }
  friend ranges::contains(const set& s, const Value& v) override { ... }
};
```

Consider what would happen if, instead of having the two-stage overload resolution that first looks for customisations and then looks for default implementations, we just define the default implementation as a generic customisation - equivalent to changing the default implementation to an '`override`'.

In particular, consider:

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

bool result = std::ranges::contains(s, 2); // passes type 'std::set<int>&'
```

If the default were considered in overload resolution at the same time as customisations then the default implementation with a deduced template parameter `R` of type `std::set<int>&` would actually represent a better match for overload resolution than the customisation which has a parameter of type `const std::set<int>&` and so this call would resolve to the more expensive default implementation.

By allowing implementations to be declared as defaults, and by defining overload resolution as a two-phase process - first try to find an overload marked as an `override` and only if no such overload was found then try to find an overload marked as `default` - we can ensure that customisations marked as `override` are preferred to default implementations, even if a default implementation would be a better match.

Note that this approach closely matches the approach taken by `tag_invoke`-based CPOs defined in P2300, which will generally check if there is a valid `tag_invoke()` overload that it can dispatch to and only if there is no valid `tag_invoke()` overload, then fall back to a default implementation.

Calling an override with argument conversion is preferred over calling a default

One of the implications of this two-phase approach to name lookup is that it means that overload resolution can potentially prefer calling customisations found through implicit conversions over calling a default implementation that doesn't require implicit conversions.

For example, consider:

```
template<typename T, typename U>
virtual void swap(T&& t, U&& u) = 0;

template<typename T>
requires std::move_constructible<T> && std::assignable_from<T&, T&&>
void swap(T& a, T& b) noexcept(/*...*/) default {
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(a);
}

struct X {
  int value = 0;
  friend void swap(X& a, X& b) noexcept override { swap(a.value, b.value); }
};

void example1() {
  int a = 0;
  int b = 1;
  std::reference_wrapper<int> ref1 = a;
  std::reference_wrapper<int> ref2 = b;
  swap(ref1, ref2); // calls default implementation - swaps references.
  assert(&ref1.get() == &b);
  assert(&ref2.get() == &a);
  assert(a == 0 && b == 1);
}

void example2() {
  X a{ 0 };
  X b{ 1 };
  std::reference_wrapper<X> ref1 = a;
  std::reference_wrapper<X> ref2 = b;
  swap(ref1, ref2); // calls X's customisation. swaps contents of 'a' and 'b'
  assert(&ref1.get() == &a); // ref1/ref2 still point to same objects
  assert(&ref2.get() == &b);
  assert(a.value == 1 && b.value == 0); // contents of 'a' and 'b' have been swapped.
}
```

In the case where we call `swap()` on `std::reference_wrapper<int>` (example 1) the overload resolution finds no overrides and so falls back to the default implementation which then swaps the `reference_wrapper` values, swapping `ref1` to now reference `b` and `ref2` to now reference `a`.

However, when calling `swap()` on `std::reference_wrapper<X>` (example 2), the overload resolution finds the override defined for `X` (since `X` is an associated entity of `std::reference_wrapper<X>`) and this overload is viable because `std::reference_wrapper<X>` is implicitly convertible to `X&`. So overload resolution ends up preferring to call `swap(X&, X&)` instead of `swap(reference_wrapper<X>&, reference_wrapper<X>&)`.

This difference in behaviour may be surprising for some. It could be made consistent in this case by explicitly defining an overload of `swap()` for `std::reference_wrapper<T>` to ensure that this always swaps the references rather than swapping the contents of the referenced objects.

Declaring multiple forms of a customisable function

Sometimes we would like to use a given name for multiple forms of a customisable function. For example, to add overloads that take different numbers of arguments, or that take arguments of different types or concepts.

It is possible to declare multiple customisation point overloads with the same name by declaring multiple 'virtual' functions with the same name in the same namespace.

For example, we might want to define two flavours of the sender algorithm `stop_when()`.

```
namespace std::execution
{
  // Create a sender that runs both input senders concurrently.
  // Requests cancellation of the other operation when one completes.
  // Completes with result of 'source' once both operations have completed.
  template<sender Source, sender Trigger>
  virtual sender auto stop_when(Source source, Trigger trigger) default;

  // Create a sender that sends a stop-request to 'source' if a stop-request
  // is delivered on the input stop-token.
  template<sender Source, stoppable_token ST>
  virtual sender auto stop_when(Source source, ST stop_token) default;
}
```

This ends up creating a single customisation point object named '`std::execution::stop_when`' that is callable with and that allows customisation of either of these signatures. i.e. that forms a single overload-set.

Customising customisable functions

If a namespace-scope function declaration contains the `override` specifier then the (possibly scoped) function name must name a previously declared customisable function. In this case, the declaration adds an overload to that customisable function's customisations overload-set.

For example: Given the following customisation point declaration

```
namespace shapes {
  template<typename T>
  virtual float area(const T& shape) noexcept = 0;
}
```

We can declare a customisation of this function for our own data-type as follows:

```
namespace mylib {
  struct circle {
    float radius;
  };

  inline float shapes::area(const circle& c) noexcept override {
    return c.radius * c.radius * std::numbers::pi_v<float>;
  }
}
```

Customisations can also be declared as hidden-friends, in which case the overloads are only considered if the containing type is considered an associated type of an argument to the call to a customisable function. e.g.

```
namespace mylib {
  struct circle {
    float radius;

    friend float shapes::area(const circle& c) noexcept override {
      return c.radius * c.radius * std::numbers::pi_v<float>;
    }
  };
}
```

Template customisable functions

Some customisable functions are intended to be called with explicit template arguments.

For example, if we were to hypothetically define `std::get` as a customisable function so that user-defined types can customise it. e.g. for supporting structured bindings, then we might declare it as follows:

```
namespace std {
  template<typename T, typename U>
  concept reference_to = std::is_reference_v<T> &&
                         std::same_as<std::remove_cvref_t<T>, U>;
  // Get the element of 'obj' with type T
  template<typename T, typename Obj>
  virtual reference_to<T> auto get<T>(Obj&& obj) = 0;

  // Get the Nth element of 'obj'
  template<size_t N, typename Obj>
  virtual auto&& get<N>(Obj&& obj) = 0;
}
```

This ends up declaring two variable templates with the same name. Each instantiation of which represents a customisable function object with a distinct type and thus a distinct overload-set.

This is intended to mimic the behaviour of normal template functions, such as `std::get`, which permit different sets of template function declarations to coexist alongside each other.

Note that it is similarly valid for a template customisable function and a non-template customisable function of the same name to coexist beside each other as well, similar to normal functions.

e.g. The following creates an object named `N::foo` and a variable template named `N::foo<T>`. These represent independent CPOs that can be customised separately.

```
namespace N {
  virtual void foo(auto& obj, const auto& value) = 0;

  template<typename T>
  virtual void foo<T>(auto& obj, const T& value) = 0;
}

struct X {
  friend void N::foo(X&, const int& x) override;      // 1
  friend void N::foo<int>(X&, const int& x) override; // 2
};

X x;
N::foo(x, 42);      // calls 1.
N::foo<int>(x, 42); // calls 2.
```

Given the example declaration of `std::get` above we could write:

```
// Assume that std::tuple has customised std::get<T> and std::get<N>
std::tuple<int, float, bool> t{42, 1.0f, true};

assert(std::get<0>(t) == 42);
assert(std::get<float>(t) == 1.0f);

// A customisable-function object that when called gets the first element
auto get_first = std::get<0>;
auto get_second = std::get<1>;
static_assert(!std::same_as<decltype(get_first), decltype(get_second)>);

// A customisable-function object that when called gets the 'float' element
auto get_float = std::get<float>;

assert(get_first(t) == 42);
assert(get_float(t) == 1.0f);
```

Note that the template arguments of `std::get` are never deduced, which is why we can allow multiple CPOs of the same name to coexist.

A type can define non-template customisations of template customisable functions as follows:

```
template<typename First, typename Second>
struct pair {
  First first;
  Second second;

  friend First& std::get<0>(pair& self) noexcept override { return self.first; }
  friend Second& std::get<1>(pair& self) noexcept override { return self.second; }

  friend First& std::get<First>(pair& self) noexcept
  requires (!std::same_as<First, Second>) override {
    return self.first;
  }

  friend Second& std::get<Second>(pair& self) noexcept
  requires (!std::same_as<First, Second>) override {
    return self.second;
  }
};
```

A type can also define a generic customisation of a template customisable function that allows deduction of the template arguments from the customisable-function template mentioned in the function declarator. e.g.

```
template<typename T, std::size_t N>
struct array {
  T data[N];

  template<size_t Idx>
    requires (Idx < N)
  friend T& std::get<Idx>(array& self) noexcept { return data[Idx]; }
};
```

Generic forwarding

Often when we are building wrapper types we want to be able to forward through a set of CPOs to the wrapped object.

Examples of such wrapper types include:

- Type-erasing wrappers, where the list of CPOs to forward is specified via template parameters. e.g. `unifex::any_unique<cpo1, cpo2, cpo3>` where the generic type `any_unique` needs to be able to customise the arbitrary set of user-provided CPOs.
- Types that customise one or more CPO to have different behaviour and forward calls to other CPOs onto the wrapped value (if it supports them). This technique is used heavily in sender/receiver-based algorithm implementations (e.g. a receiver customising `get_allocator()` to return an allocator owned by the current operation, while forwarding other queries to the parent receiver).

The ability to be able to generically customise different CPOs is built on the ability to define an override with a declarator-id that is deducible.

Example: A receiver type that customises the `get_allocator()` query and forwards other queries.

```
template<typename T, auto CPO, typename... Args>
concept receiver_of =
  receiver<T> &&
  completion_signal<CPO> &&
  requires(CPO cpo, T&& r, Args&&... args) {
    { cpo((T&&)r, (Args&&)args...) } noexcept;
  };

template<typename Receiver, typename Allocator>
struct allocator_receiver {
  Receiver receiver;
  Allocator alloc;

  // Customise the get_allocator query
  friend Allocator std::execution::get_allocator(const allocator_receiver& r)
      noexcept override {
```

```
    return r.alloc;
  }

  // Forward completion-signals
  template<auto cpo, typename... Args>
  requires receiver_of<allocator_receiver, CPO, Args...>
  friend void cpo(allocator_receiver&& r, Args&&... args) noexcept override {
    cpo(std::move(r.receiver), std::forward<Args>(args)...);
  }

  // Forward query-like calls onto inner receiver
  template<auto cpo>
  requires (!completion_signal<CPO>) && std::invocable<decltype(cpo), const Receiver&>
  friend decltype(auto) cpo(const allocator_receiver& r)
      noexcept(std::is_nothrow_invocable<CPO, const Receiver&>) override {
    return cpo(r.receiver);
  }
};
```

Overload resolution for customisable function calls

In a function-call expression, if the *postfix-expression* operand of the call expression denotes a Customizable Function Object, then overload resolution is performed as follows:

- Let `F` be the decayed type of the customisable function object O and `args...` be the sequence of argument expressions for the call expression
- Start with an empty set of customisation overloads
- For each associated entity of arguments passed to the call-expression look in each of the associated namespaces for
    - non-member functions marked with `override` where either;
        - the declarator-id names an object that has type `F`; or
        - the declarator-id is a dependent name deducible to O
    - if the override is a function template then attempt to deduce the template arguments based on the types of the argument expressions in the post-fix call expression.
        - if template argument deduction fails the discard this override
    - if any of the constraints of the override declaration are not met then discard this override.

  Add each overload found to the *customisation overload set*
- form an argument list `(args...)` and attempt to perform overload resolution using this argument-list on the overloads in the *customisation overload set*
- if overload resolution finds a single, best overload then call-expression resolves to a call to this overload
- otherwise, if there were viable candidates; then the call-expression is ill-formed
  [[Note: Because the call is ambiguous]]

- otherwise, if there were no viable candidates in the overload set, perform a second overload resolution with the same argument list, this time on the set of function definitions marked marked with '`default`' (which are found in the customisable function's private associated namespace)
- if overload resolution finds a single, best overload then the call-expression resolves to a call to this overload
- otherwise, the call expression is ill-formed
  [[Note: Because either there is no viable candidate, or the call is ambiguous]]

Once the single, best overload is selected the next step is to check that the selected overload is a valid customisation of the customisable function prototype, described in the Validating customisations section.

Validating customisations

The process of checking that a particular customisation of a customisable function is a valid customisation helps to eliminate bugs caused by defining a customisation that don't match the expectation of a caller of the customisation point.

One of the points from P2279R0 states that customisation-points should have:

> 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`)

For example, if I declare the following customisable function:

```
template<typename Obj>
virtual void resize(Obj& obj, size_t size) = 0;
```

Then if I had a type, say:

```
struct size_wrapper {
  operator size_t() const { std::print("size_t"); return 1; }
  operator int() const { std::print("int"); return 2; }
};
```

You would never expect a call to `resize(someObj, size_wrapper{})` as the second argument to invoke the `operator int()` conversion operator. However, if we were to allow a type to define the override `resize(some_type&, int)` then a call that resolves to this overload might end up calling `operator int()` unexpectedly.

To avoid this situation, the signature of customisations and default implementations are checked against the customisable function prototypes to ensure that they are consistent with the signature of at least one customisable function prototype.

This is done by attempting to match the signature of the selected overload to the signature of each of the customisable function prototypes

- for each customisable function prototype for the current customisable function

    - if it was a template customisable function prototype then first deduce the template arguments required to match the declarator-id such that decltype(declarator-id) is F.
    - Then, for each function parameter of the declaration in-order from left to right
        - Attempt to deduce template arguments so that the parameter type exactly matches the corresponding parameter type from the selected overload.
        - If this step fails then ignore this customisable function prototype
    - Then, attempt to deduce template arguments such that the return type exactly matches the return type of the selected overload [[Example: `copyable auto f() = 0` deduces an implicit template argument to the return type of the overload and then checks the copyable constraint on the deduced implicit template argument -- end example]].
        - If this step fails then ignore this customisable function prototype
    - Then evaluate the constraints on the declaration with the deduced template arguments.
        - If the constraints are not met then ignore this customisable function prototype
    - Then evaluate the noexcept-specifier of the customisable function prototype using the deduced template arguments.
        - If the noexcept-specifier evaluates to `noexcept(true)` then if the implementation does not have a noexcept(true) specifier then the program is ill-formed.
    - If the consteval-ness of the prototype does not match the consteval-ness of the selected overload, discard the prototype.
- If there were no customisable function prototypes that were not discarded for which the overload was able to successfully match then the program is ill-formed. [[Note: Because the implementation was an invalid overload]].

Parts of this process are described in more detail Noexcept specifications, Constraining parameter-types and Constraining return-types.

Noexcept specifications

When a customisable function is declared, the declaration may contain a `noexcept` specification.

If the customisable function prototype has a `noexcept(true)` specification then all customisations and default implementations of that function must also be declared as `noexcept(true)`.

```
namespace somelib {
  template<typename T>
  virtual void customisable_func(const T& x) noexcept = 0;
}

namespace mylib {
  struct type_a { /*...*/ };
  struct type_b { /*...*/ };

  // OK, customisation is noexcept
  friend void somelib::customisable_func(const type_a& x) noexcept {
    std::cout << "type_a";
  };

  // Ill-formed: customisation not declared noexcept
  friend void somelib::customisable_func(const type_b& x) {
    std::cout << "type_b";
  }
}
```

If the customisable function prototype has a `noexcept(false)` specification or if the noexcept specification is absent then customisations and default implementations may either be declared as `noexcept(true)` or `noexcept(false)`. These rules are the same as for function pointer convertibility around `noexcept`.

When a noexcept-expression contains a call-expression in its argument that resolves to a call to a customisable function, the result is evaluated based on the noexcept-specification of the selected overload of the customisable function rather than the noexcept-specification of the declaration of the customisable function.

```
namespace std::ranges {
  template<typename T, typename U>
  virtual void swap(T&& t, U&& u) = 0;
}

struct type_a {
  friend void std::ranges::swap(type_a& a, type_a& b) noexcept override;
};

struct type_b {
  friend void std::ranges::swap(type_b& a, type_b& b) override;
};
```

```
type_a a1, a2;
type_b b2, b2;
static_assert(noexcept(std::ranges::swap(a1, a2)));
static_assert(!noexcept(std::ranges::swap(b1, b2)));
```

For example: Consider a hypothetical `make_stop_callback()` inspired from P2175

```
template<stoppable_token ST, typename F>
  requires std::invocable<std::decay_t<F>>
virtual auto make_stop_callback(ST st, F func)
  noexcept(std::is_nothrow_constructible_v<std::decay_t<F>, F>) = 0;

struct never_stop_token { ... };
struct never_stop_callback {};
template<typename F>
  requires std::invocable<std::decay_t<F>>
never_stop_callback make_stop_callback(never_stop_token, F&&) noexcept override {
  return {};
}

void example() {
  auto cb = make_stop_callback(never_stop_token{}, []() { do_something(); });
}
```

In this case, the overload resolution for the call to `make_stop_callback` finds the `never_stop_token` overload which has argument types: '`never_stop_token`' and '`lambda&&`'.

The compiler then looks at the customisable function prototype and deduces the template arguments:

- `ST` is deduced to be '`never_stop_token`'
- `F` is deduced to be '`lambda&&`'

Next, the compiler evaluates noexcept specification:

- `noexcept(std::is_nothrow_constructible_v<lambda, lambda&&>)` evaluates to `noexcept(true)`

Next, as the declaration evaluated the noexcept specification to `noexcept(true)`, the compiler then checks the noexcept specification of the selected overload:

- The selected overload is declared `noexcept` unconditionally, so noexcept specification of customisation is considered compatible.

Constraining parameter-types

A customisation point may put constraints on the parameter types that it is designed to handle.

For example, a customisable range-based algorithm may want to constrain a `contains()` algorithm so that its first argument is a `range` and the second argument is a value that is `equality_comparable_with` the elements of that range.

Given:

```
template<std::ranges::range R,
         std::equality_comparable_with<std::ranges::range_reference_t<R>> V>
virtual bool contains(R range, V value) = 0;

template<std::ranges::range R, typename V>
  requires std::equality_comparable_with<
             std::ranges::range_reference_t<R>, const V&>
bool contains(R&& range, const V& value) default {
  for (auto&& x : range) {
    if (x == value) return true;
  }
  return false;
}



template<typename T>
requires std::equality_comparable<T>
bool contains(const my_container<T>& c, const T& value) noexcept override {
  return c.find(value) != c.end();
}
```

Then when evaluating the following call to `contains()`:

```
my_container<int> c;
bool result = contains(c, 42);
```

The compiler performs the following steps:

- using the argument types `my_container<int>&` and `int` performs overload resolution on overloads of the `contains`() customisable function (see section on overload resolution).
- If the selected overload corresponds to an overload with an '`override`' specifier, then we look at the types of the parameters of the selected overload. In our example, the parameter types are `(const my_container<int>&, const int&)`.
- For each associated customisable function prototype, attempt to deduce template arguments to make the parameter types of the prototype identical to the parameter types of the selected overload.
  Note: This is exactly the mechanism used to deduce arguments to partial class template specialisations.
  Note: This means that non-dependent parameter types must match exactly.
  Note: Conversions are not considered at this stage (they were already considered during overload resolution)

- If template argument deduction fails then this declaration is removed from the considered set of declarations.
- If template argument deduction succeeds then the compiler evaluates the constraints to determine if the deduced template arguments satisfy the constraints. In this example we evaluate the constraints:
  - `std::ranges::range<const my_container<int>&>`
  - `std::equality_comparable_with<`
    `std::ranges::range_reference_t<const my_container<int>&>,`
    `const int&>`
- If the deduced template arguments do not satisfy the constraints, then this declaration is removed from the considered set of declarations.
- If, after evaluating the constraints of all prototype declarations there are no declarations remaining in the considered set of declarations then the call expression is considered ill-formed.

Important to note here is that the customisable function prototype having parameters types that are unqualified template parameters that appear to be prvalues does not necessarily mean that all customisations must define those parameters as prvalues. If you want to require customisations to accept parameters by-value then you will need to add additional constraints for this. e.g.

```
template<typename T>
concept prvalue = (!std::is_reference_v<T>);

// Force customisations to accept parameters by-value
virtual auto combine(prvalue auto first, prvalue auto second) = 0;
```

It is also important to note that we can constrain the signature of the selected overload to have particular concrete parameter types, or to have a parameter that is an instantiation of a particular template. e.g.

```
template<typename Shape>
virtual Shape scale(Shape s, float factor) = 0; // 'factor' param must be
                                                // 'float' to match signature.

// The 'request' parameter can be constrained to be a shared_ptr of some
// type that satisfies the 'httplib::request' concept.
template<typename Processor, httplib::request Request>
virtual void process_request(Processor& p,
                             std::shared_ptr<Request> request) = 0;
```

Constraining return-types

Often, a customisation point wants to allow the return type of the customisation point to be deduced from the customisation, but still wants to be able to constrain customisations to require that they return types that satisfy some concept or other constraints.

For example, the `std::ranges::begin()` customisation point allows customisations to determine the return-type (different types of ranges usually need different iterator types) but requires customisations to return a type that satisfies `std::ranges::input_or_output_iterator`.

Where there is an existing concept that describes the return type requirements you can use the concept-auto syntax for the return type to require that the return-type satisfies some concept.

For example, the `std::ranges::begin()` customisation point might be defined as follows:

```
namespace std::ranges {
  template<typename R>
  virtual input_or_output_iterator auto begin(R range) = 0;

  template<typename T, std::size_t N>
  T* begin(T(&range)[N]) noexcept override {
    return range;
  }

  template<typename R>
  requires (R& r) { { auto(r.begin()); } -> input_or_output_iterator }
  auto begin(R&& r) noexcept(noexcept(r.begin())) default {
    return r.begin();
  }
}
```

Much like the act of constraining parameters in the previous section, the return-type is also deduced from the signature of the selected overload and then constraints are then applied to the deduced template parameters.

The begin() customisable function prototype above uses the concept-auto syntax, which this paper is defining as a syntactic sugar for the following equivalent code:

```
namespace std::ranges
{
  template<typename R, typename It>
    requires input_or_output_iterator<It>
  virtual It begin(R range) = 0;
}
```

When a call to a customisable function is made, the compiler looks at the signature of the selected overload and then uses this signature (both the return-type and the parameter-types) to deduce the template arguments of the customisable function prototype. Constraints can then be applied to a deduced return-type.

This more explicit syntax can be used to apply multiple constraints to the return-type, or apply constraints to a function of the return-type without having to define a new concept that composes those constraints. e.g.

```
template<typename R, typename T>
requires decay_copyable<R> && some_concept<std::decay_t<R>>
virtual R some_getter(const T& x) = 0;
```

Additional Design Discussions

Default arguments

Default arguments are not currently permitted for customisable function prototypes or for declarations of default implementations or customisations of customisable functions.

Support for default arguments may be explored as an extension in future if desired.

Attributes

It should be possible to place attributes like `[[deprecated]]` and `[[noreturn]]` on customisable function prototypes, this has not been fully fleshed out yet.

Controlling the set of associated namespaces

This proposal makes use of argument-dependent lookup for finding customisations of customisable functions.

The current argument dependent lookup rules can often result in the compiler searching many more associated namespaces for overloads than desired, hurting compile-times and sometimes resulting in unexpected overloads being called.

It would be great if we had better control over which set of associated namespaces the compiler considers when performing argument-dependent lookup.

There are two main features that could be considered as companions to this paper:

- Declaring that certain parameters of a customisable function should not be considered when constructing the list of associated namespaces (this builds on top of this paper)
- Declaring that certain template parameters of a class template should not be considered when constructing the list of associated namespaces (this is discussed in another paper).

Comparison to Rust Traits

This is an example from Barry's blog post on customisation points, which compares C++ to Rust at the end when discussing ideals.

Rust	This Proposal

`trait PartialEq {`
`  fn eq(&self, rhs: &Self) -> bool;`
 	
`  fn ne(&self, rhs: &Self) -> bool {`
`  !self.eq(rhs)`
`  }`
`}`
	`template<typename T>`
	`virtual bool eq(const T& x, const T& y) = 0;`
	
	`template<typename T>`
	`requires requires(const T& x, const T& y) {`
	`  eq(x, y);`
	`}`
	`virtual bool ne(const T& x, const T& y)`
	`  noexcept(eq(x, y)) default {`
	`  return !eq(x, y);`
	`};`
	
	`template<typename T>`
	`concept PartialEq =`
	`requires(const T& x, const T& y) {`
	`  eq(x, y);`
	`  ne(x, y);`
	`  };`

`struct Point {`
`  x: f64,`
`  y: f64`
`}`
 	
`impl PartialEq for Point {`
`  fn eq(&self, rhs: &Self) -> bool {`
`  eq(self.x, rhs.x) && eq(self.y, rhs.y)`
`  }`
`}`
	`struct Point {`
	`  double x;`
	`  double y;`
	
	`  friend bool eq(const Point& lhs,`
	`  const Point& rhs) override {`
	`  return eq(lhs.x, rhs.x) && eq(lhs.y, rhs.y);`
	`  }`
	`};`
	
	`static_assert(PartialEq<Point>);`

`let a = Point { x: 1.0, y: 2.0 };`
`let b = Point { x: 1.0, y: 3.0 };`
`if a.eq(b) {`
`  println!("equal");`
`}`
	`Point a{1.0, 2.0};`
	`Point b{1.0, 3.0};`
	
	`if (eq(a, b)) {`
	`  std::puts("equal");`
	`}`

We are able to express the same semantics in C++, although without some of the explicitness that we get from the Rust implementation of PartialEq. The C++ code that customises eq() does not make any reference to the PartialEq concept to indicate that we are customising a CPO associated with that trait.

The implementation on the C++ side is slightly more involved:

- The default implementation of ne() is constrained to only be valid if there is an eq().
  We could also consider putting this requirement on the ne() declaration instead of just on the default implementation if we always wanted people to provide an eq() if they provide a ne()
- The default implementation of ne() deals with forwarding noexcept-ness

Redefining existing customisation points in terms of this solution

Our intent is to replace all `tag_invoke` usages in P2300 by this proposal. (P2300 makes extensive use of CPOs and forwarding)

We also cover other usage patterns of the standard library like swap, std::begin, ranges::begin, std::get, etc.

However we have not investigated whether it would be possible to respecify the existing facilities in terms of language cpos without ABI breaks or subtle change of behaviour and we are not proposing to do that.

Some facilities, like ranges for loops and structured binding are currently specified in terms of raw ADL calls.

Implementation

This proposal has not yet been implemented in a compiler and so does not yet have implementation or usage experience. The author is seeking directional guidance from the language evolution group at this point in time.

Acknowledgements

Thank you to Barry Revzin for generously providing his time to read and give feedback on early drafts of this paper.

References

- We need a language mechanism for customization points - Barry Revzin <https://wg21.link/P2279>
- Why tag_invoke is not the solution I want - Barry Revzin - <https://brevzin.github.io/c++/2020/12/01/tag-invoke/>
- std::execution <https://wg21.link/P2300>
- Customization Point Functions - Matt Calabrese - <https://wg21.link/p1292r0>
- tag_invoke: A general pattern for supporting customisable functions - Lewis Baker, Eric Niebler, Kirk Shoop <https://wg21.link/p1895r0>