Document	P1895R0
Date	2019-10-07
Reply To	Lewis Baker <lbaker@fb.com> Eric Niebler <eniebler@fb.com> Kirk Shoop <kirkshoop@fb.com>
Audience	LEWG
Target	C++23

tag_invoke: A general pattern for supporting customisable functions

Abstract

Modern customization point objects ([customization.point.object]) were a step forward over raw ADL for making libraries customizable. However, there are a couple of problems they leave unsolved:

1. Each one internally dispatches via ADL to a free function of the same name, which has the effect of globally reserving that identifier (within some constraints). Two independent libraries that pick the same name for an ADL customization point still risk collision.
2. There is occasionally a need to write wrapper types that ought to be transparent to customization. (Type-erasing wrappers are one such example.) With C++20's CPOs, there is no way to generically forward customizations through the transparent wrappers.

Point (1) above is an immediate and pressing concern in the executors design, where we would like to give platform authors the ability to directly customize parallel algorithms. Using the C++20 CPO design would explode the number of uniquely-named ADL customization points from a handful to potentially hundreds, which would create havoc for the ecosystem.

Point (2) is also a concern for executors, where platform authors would like to decorate executor types with platform-specific "properties" (extra-standard affinities and thresholds of all sorts) that can be exposed even through transparent layers of adaptation, such as the polymorphic executor wrapper. This need led to the properties system (P1393) which LEWG has already reviewed.

It's important to note that, although the problems in C++20 CPOs are exposed by the executors work, the problems are not specific to executors.

This paper presents a solution: a single ADL customization point named `tag_invoke` that takes as its first argument a CPO that is used as a tag to select an overload. A new CPO, `std::is_fooable(t)`, rather than dispatching via ADL to `is_fooable(t)`, would dispatch instead to

`tag_invoke(std::is_fooable, t)`. As will be shown below, this neatly solves both problems described above without the need for a separate properties system.

Overview

This paper proposes defining a standard, uniform mechanism for customisable functions exposed as customisation-point-objects (CPOs) to allow user-defined types to customise the behaviour of those operations without requiring the use of a separate ADL name for each CPO.

This would add a new meta-customisation-point-object called `std::tag_invoke` that abstracts away the mechanism of evaluating an unqualified call to `tag_invoke()` that finds the appropriate overload by ADL.

Customisation-point-objects are then implemented in terms of calls to overloads of `tag_invoke()` that pass the CPO itself as the first argument, allowing it to tag-dispatch to an appropriate overload for that CPO based on the type of the first parameter. CPOs can optionally dispatch to some default implementation if no overload of `tag_invoke()` could be found.

Customisations of a CPO define overloads of `tag_invoke()` found by ADL, typically defined as hidden friend functions to avoid making the overload-set unnecessarily large, and that take the CPO object as the first argument and specialise on one or more of the other argument types.

For example:

```
// Define a new CPO named mylib::foo()
namespace mylib
{
  inline constexpr struct foo_fn {
    template<typename T>
    auto operator()(const T& x) const ->
      std::tag_invoke_result_t<foo_fn,const T&> {
      // CPO dispatches to tag_invoke() call
      // passes the CPO itself as first argument.
      return std::tag_invoke(*this, x);
    }
  } foo{};
}

// Use the mylib::foo() CPO
template<typename T>
  requires std::invocable<decltype(mylib::foo),
                          const T&>
bool print_foo(const T& value) {
  // Just call the CPO like an ordinary function
  std::cout << mylib::foo(value) << std::endl;
}
```

```
// Customise the mylib::foo() CPO for my_type
namespace otherlib
{
  class other_type {
    ...

  private:
    // Provide an overload of tag_invoke() with
    // the CPO as the first argument.
    friend int tag_invoke(std::tag_t<mylib::foo>,
                          const other_type& x) {
      return x.value_;
    }

    int value_;
  };
}

// Can now call print_foo() function.
void example() {
  otherlib::other_type x;
  print_foo(x);
}
```

Note: Code that just wants to use/call a customisable function does not need to know anything about `tag_invoke()`. It can just call the CPO as if it were an ordinary function. The `tag_invoke()`

mechanism is used only in the implementation of customisable functions and by types that want to customise the implementation of the function.

This approach to defining CPOs has a number of important properties:

It centralises and hides away the mechanics of defining customisation-points that allow customisation by defining functions found by ADL.

> This greatly simplifies the amount of boiler-plate needed to define a new customisation-point. No longer do you need to:
> - define the CPO type in an unspecified, private namespace
> - optionally define a poison-pill overload of the ADL function for unqualified calls in that context from finding the ADL name in parent namespaces
> - define the constexpr CPO in a separate inline namespace that will not be found by the CPO type's `operator()` so that it doesn't result in recursively calling itself, and add a 'using namespace' declaration to bring the CPO into the parent namespace without conflicting with hidden friends that might be also defined in that namespace.

It solves the problem of CPOs defined by different libraries potentially conflicting if they happen to use the same name and signature for the ADL function but have different semantics.

> By incorporating the type of the CPO itself in the signature of the ADL function it becomes possible to allow two CPOs with the same name but defined in different namespaces, possibly in unrelated libraries, to be disambiguated by having customisations explicitly specify the type of the CPO that they intended to customise.
>
> This prevents a type from accidentally providing a customisation for a CPO that they did not intend to, and also allows a type to simultaneously customise/implement an arbitrary set of CPOs without worrying about potential naming conflicts that might otherwise arise were CPO-specific names used to dispatch to ADL-overloads.
>
> This has the added benifit that we only need to reserve a single ADL name, `tag_invoke`, which can then be used by all CPOs.

It provides a single ADL name with a uniform structure for the customisations.

> Having a single ADL name to customise, ie. `tag_invoke()`, and a uniform structure to these overloads, always taking the CPO itself as the first argument, makes it possible to build adapters that generically forward through calls to CPOs involving the adapter type to an underlying object.
>
> For example, one such adapter that can be built using this is a generic type-erasing adapter that can forward through calls to an arbitrary list of CPOs through to the concrete implementations.

It provides a simpler and more direct representation of customisable functions compared to using query/prefer/require as described in P1393R0.

> The use of `tag_invoke`-based CPOs provides many of the same benefits that P1393 properties provide but does so with a more general facility that also naturally supports customisable algorithms.
>
> With P1393 properties, full support for customizable parallel algorithms would require a new standard property for every overload of every standard parallel algorithm. With `tag_invoke`, no additional properties are needed; the algorithm CPO itself, which needs to exist anyway, suffices. We could even reuse the ones that already exist in the `std::ranges` namespace.
>
> We believe that `tag_invoke`-based CPOs meet the needs of the P1393 properties mechanism while being simpler, more general, and more in keeping with the existing design of the STL. If, however, it is decided that the query/prefer/require/require_concept CPOs are still desired, they can trivially be implemented on top of `tag_invoke`.

Motivation / Goals

The high-level goals of this facility are:

- To make it simpler to define customisable functions as CPOs
- To provide a uniform interface for customising these functions
- To allow adapter types, such as type-erasing adapters, to forward through calls to CPOs involving the adapter type to the equivalent call of the CPO with the wrapped type in a generic way.
- Provide a mechanism for customisable functions that avoids the potential for naming conflicts inherent in picking names from a global scope (names found by ADL, names of member functions)

While the main driver for adding this facility has been in support of customisation of async algorithms in the domain of executors and sender/receiver, it is expected that many other domains will also want to define interfaces in terms of customisable functions or CPOs.

This facility is entirely general and is not specific to any particular domain.

Customisable functions/algorithms

The primary motivation for this capability is to allow libraries to define functions/algorithms that can be customised externally by user-defined types, allowing them to define alternative implementations when the function is invoked with particular argument types.

A given customisable function can provide a default implementation, typically in terms of some base concept that it constrains its arguments on. If someone invokes the function with types that have not customised the function then the call will dispatch to the default implementation.

If a customisable function has not defined a default implementation then it effectively acts as a basis operation and user-defined types need to explicitly implement a customisation of this function for it to be callable with those types.

Thus customisable functions can be broadly split into two main categories:

- *basis operations* - These are customisable functions with no default implementation. Customisations for these operations must be defined for them to be callable. User-defined types will typically customise basis operations in order to satisfy some higher-level concept.
- *customisable algorithms* - Customisable functions that have a default implementation defined in terms of some set of basis operations, but that allow user-defined customisations of the algorithm. Customisable algorithms are not used in the definition of new concepts.

The ability to allow user-defined types to customise standard library algorithms for their types allows generic code to continue calling standard library algorithm functions, while allowing those calls to dispatch to more efficient implementations than the generic implementation provided by the standard library when called with certain combinations of argument types.

For example, the paper P1171R0 proposes a function `sync_wait()` that blocks the current thread until some asynchronous operation, represented as a Sender passed as its only parameter, completes. This `sync_wait()` function can be implemented as a generic algorithm in terms of the Sender interface by attaching a callback to the Sender and using standard thread-synchronisation primitives to block until the callback is invoked. However, there may be certain types of Senders that may have a more efficient implementation of `sync_wait()`.

For example a Sender that represents the execution of a CUDA kernel on a GPU might be able to implement the semantics of `sync_wait()` more efficiently by calling the `cudaEventSynchronize()` function. Thus an implementer of such a sender might want to customise the implementation of `sync_wait()` so that if generic code were to call `sync_wait()` with their sender type that it would dispatch the call to the more efficient implementation.

Simplifying definition of CPOs

The paper N4381 by Eric Niebler introduced the design of customisation-point objects (CPOs) as a better abstraction for implementing customisation-points that find customisations by ADL. This is the approach that has been adopted by the recent `std::ranges` CPOs added in C++20.

Say we want to define a CPO, `mylib::foo`, that will dispatch to an overload of `foo()` that is found by ADL, if such an overload exists. If we were to follow the same approach as the `std::ranges` CPOs then, to avoid all of the issues outlined in N4381, we would need to define our CPO as follows:

```
namespace mylib
{
  namespace foo_cpo_detail
  {
    // "poison pill" to hide overloads of foo() that might be found in parent namespace.
    // We want to limit to only finding overloads by ADL.
    void foo() = delete;

    // Define function object with operator() that forwards to call to unqualified 'foo()'
    struct foo_fn {
      template<typename T>
      auto operator()(T&& x) const
          noexcept(noexcept(foo((T&&)x)))
          -> decltype(foo((T&&)x)) {
        return foo((T&&)x);
      }
    };
  }

  // Define 'foo' object in an inline nested namespace.
  // This is necessary to avoid potential conflicts that can arise from having types in
  // 'mylib' that define customisations of 'foo()' as hidden friends, thus implicitly
  // adding functions with name 'foo' in the enclosing namespace,
  // at the same time as having an object with name 'foo' in the same namespace.
  inline namespace foo_cpo
  {
    inline constexpr foo_cpo_detail::foo_fn foo{};
  }
}
```

This is a lot of boiler-plate that we need to go through to be able to define a new CPO.

We would ideally be able to reduce the amount of boiler-plate and magic incantations required to define a CPO to something more concise and with fewer pitfalls than the above. It should be easy for applications to define their own CPOs without needing to write a lot of code.

With this proposal we can write a CPO more simply in terms of tag_invoke as follows:

```
namespace mylib
{
  inline constexpr struct foo_fn {
    template<typename T>
    auto operator()(T&& x) const
        noexcept(noexcept(std::tag_invoke(*this, (T&&)x)))
        -> decltype(std::tag_invoke(*this, (T&&)x)) {
      return std::tag_invoke(*this, (T&&)x);
    }
  } foo{};
}
```

The fact that the ADL name is different from the name of the CPO also means that we do not have to put the CPO in a different namespace to avoid potential conflicts with friend functions declared by types in the same namespace that are trying to customise the CPO.

Avoiding problems with name-based ADL customisation

The existing approach to defining CPOs, such as the approach taken by `std::ranges::begin()`, et.al., is to have the CPO dispatch to a call to a particular function name with the overload found by ADL.

While this approach works well for well-known idioms and concepts, like that of a `std::ranges::range`, this approach does not scale well if we start applying it to making a large number of algorithms customisable.

Each time we use a name found by ADL we are effectively imbuing that name with special meaning globally. A call to an unqualified function name can result in that name being found in any namespace associated with the arguments to the call, regardless of whether the overloads that are found were intended as customisations of the CPO that is calling them or not.

This can lead to surprising behaviour if an unrelated function is found that just happens to be callable with the same name and arguments.

It can also lead to conflicts if two libraries independently define customisation points that use the same name and signature but that have different semantics.
e.g. A library defines `shapelib::size(const T& shape)` CPO that dispatches to the `size(shape)` function found by ADL and is intended to return the size of a shape in pixels, whereas `std::ranges::size(const T& range)` also dispatches to `size(range)` function but is intended to return the number of elements in the range. If I have a shape type that is a container of shapes then it may not be possible to simultaneously customise both of these CPOs for the same type. And worse, a call to one of the CPOs may end up finding an overload intended as a customisation of the other.

The same argument about conflicting names also applies to member functions. I can't have a type that has both a `.size()` method that returns the number of pixels and a `.size()` method that returns the number of elements in the container.

These problems will become more prevalent as more libraries, the standard library included, seek to define customisation points, particularly if libraries start to allow algorithms to be customisable, not just basis operations.

This proposal addresses this problem by disambiguating ADL calls by having overloads specify the type of the CPO as the first argument. Different CPOs with the same name can be defined in different namespaces and so we can disambiguate between customisations of two CPOs with the same name by explicitly mentioning the type of the CPO that a customisation is intended for.

For example:

```
namespace shapelib
{
  template<shape S>
  class composite_shape
  {
  public:
    ...

    using iterator = typename std::vector<S>::iterator;
    iterator begin() { return shapes_.begin(); }
    iterator end() { return shapes_.end(); }

  private:
    // Separately customise std::ranges::size and shapelib::size

    friend size_t tag_invoke(std::tag_t<std::ranges::size>, const composite_shape& s) {
      // Return number of elements
      return shapes_.size();
    }

    friend size_t tag_invoke(std::tag_t<shapelib::size>, const composite_shape& s) {
      // Return the max size of the child objects.
      size_t size = 0;
      for (auto& child : s.shapes_) {
        size = std::max(shapelib::size(child), size);
      }
      return size;
    }

    std::vector<S> shapes_;
  };
```

This approach is similar to that proposed independently by P1665R0, but has a few important differences.

Supporting generic adapters that pass-through CPO calls

One of the limitations of using name-based customisation points is that it becomes difficult to customise a set of CPOs in a generic way. This makes it difficult to build adapter types that would otherwise be transparent to certain CPO calls, passing the call through to a corresponding call to the CPO on the wrapped type.

For example, say we wanted to write an adapter that customised calls to a `get_executor()` CPO to return a particular executor object but that otherwise forwarded all other CPOs invoked with the adapter as the first argument to call the CPO on the wrapped object. If every CPO required declaring a hidden friend with a different name then it is not possible to generically forward through customisations of other CPOs.

However, by having all CPOs customisable using the same name, `tag_invoke`, we can define a generic overload of `tag_invoke()` that accepts any CPO and have it generically forward calls to these CPOs to the wrapped object.

This capability is a key benefit of this approach over the "Customisation Point Functions" approach discussed in P1292R0.

For example:

```
template<typename T, typename Executor>
struct executor_wrapper
{
  explicit executor_wrapper(T object, Executor ex)
  : object_(object), executor_(ex) {}

private:
  friend Executor tag_invoke(std::tag_t<get_executor>, const executor_wrapper& ew) {
    return ew.executor_;
  }

  // Forward through calls to other CPOs to the wrapped object.
  // Done in a SFINAE-friendly way that won't hook unsupported operations
  template<typename CPO, typename... Args>
  friend auto tag_invoke(CPO cpo, const executor_wrapper& ew, Args&&... args)
      noexcept(std::is_nothrow_tag_invocable_v<CPO, const T&, Args...>)
      -> std::tag_invoke_result_t<CPO, const T&, Args...> {
    // We can just call the 'cpo' object itself
    return cpo(ew.object_, (Args&&)args...);
  }

  T object_;
  Executor executor_;
};
```

If required, the set of CPOs to be forwarded through can be constrained by adding an additional '`requires`' constraint on some attribute of the CPO.

For example, a library that defines a number of domain-specific CPOs could categorise those CPO types by annotating them with a nested '`category`' type alias. Then a particular adapter could choose to filter which CPOs are forwarded based on whether they match a particular category or not.

Type-erasing Wrappers

An important feature for the design of executors is the ability to be able to define a type-erased executor type. i.e. a type that satisfies the executor concept but that can hold a value that is any concrete executor type. However, the need to be able to define a type-erasing adapter is far more general than that required by executors.

It is possible to define a concept in terms of the set of its CPOs. For example, the receiver concept can be defined in terms of calls to three CPOs; `set_value`, `set_error` and `set_done`.

```
template<typename T, typename V, typename E = std::exception_ptr>
concept receiver_of =
  requires(T r, V value, E error) {
    set_value((T&&)r, (V&&)value);
    set_error((T&&)r, (E&&)error);
    set_done((T&&)r);
  };
```

What we would like to be able to do is define a type-erased adapter type that also satisfies this concept and that can wrap any other value that can implement this concept.

```
template<typename V, typename E = std::exception_ptr>
using any_receiver_of = ...;

any_receiver_of<int> r = some_concrete_receiver{...};
set_value(std::move(r), 123);  // Dispatches to set_value(some_concrete_receiver&&, int)
set_done(std::move(r));        // Dispatches to set_done(some_concrete_receiver&&)
set_error(std::move(r), eptr); // Dispatches to set_error(some_concrete_receiver&&,
                               //                         std::exception_ptr)
```

The fact that we can generically customise a CPO by defining a hidden-friend overload of `tag_invoke` makes it possible to build generic type-erased wrappers that can type-erase any object whose concept is defined in terms of calls to CPOs using common template metaprogramming techniques.

For example, it is possible to build a type-erased adapter type that takes a variadic list of CPOs and that customises certain overloads of those CPOs so that when the CPO is invoked with the type-erased adapter that it dispatches via an indirect call to the corresponding call to the CPO with the type-erased adapter parameter replaced with the underlying concrete object.

```
template<typename... CPOs>
class any_unqiue {
  // See godbolt.org link below for implementation.
};

template<auto&... CPOs>
using any_unique_t = any_unique<std::tag_t<CPOs>...>;

template<typename V, typename E = std::exception_ptr>
using any_receiver_of = any_unique_t<
  overload<void(this_&&, V)>(set_value),
  overload<void(this_&&) noexcept>(set_done),
  overload<void(this_&&, E) noexcept>(set_error)>;

template<typename V, typename E = std::exception_ptr>
using any_sender_of = any_unique_t<
  overload<void(this_&&, any_receiver_of<V, E>)>(submit)
  >;
```

Internally, the `any_unique` type builds a vtable with an entry for each CPO overload listed as a template parameter. It then declares a hidden friend that customises that particular overload of the CPO and dispatches via an indirect call to the function pointer stored in the corresponding vtable entry. The

concrete implementation of this then invokes the CPO again, this time with the `any_unique` parameter replaced with the concrete object.

The type `this_` is used as a placeholder to indicate the position, value category and qualifiers of the parameter for the `any_unique` object and the hidden friend simply substitutes the `this_` parameter in the signature. This allows the type-erased object to be in positions other than the first argument if required.

For an example implementation of the above see https://godbolt.org/z/3TvO4f

Note that the type-erased wrappers do not have to be limited to forwarding through basis operations. A type-erased wrapper can also support forwarding through calls to algorithms, allowing calls to that algorithm with the type-erased type to dispatch through to a specialisation of that algorithm for the concrete type rather than having the algorithm implemented in terms of the type-erased basis operations.

This can allow applications to trade-off allowing certain algorithms to have specialised implementations even when the objects have been type-erased, at the expense of additional function instantiations that may never be called, and the cost of additional vtable entries for each type-erased type.

For example, a sender that represents a CUDA operation might have customised `sync_wait()` to be implemented in terms of `cudaEventSynchronize()`. However, if we type-erase the sender and then invoke the `sync_wait()` algorithm on the type-erased sender then by default it will not be able to find the customisation for the CUDA sender and will dispatch to the default implementation defined in terms of the `submit()` basis operation and some thread-synchronisation primitive such as `std::binary_semaphore`.

However, if we extend the type-erased sender type to also include an entry for the `sync_wait()` CPO then it can customise the call to `sync_wait()` on the type-erased sender to dispatch through a vtable entry to call the concrete implementation of `sync_wait()`, including any customisations there might have been for that concrete type.

For example:

```
inline constexpr unspecified sync_wait{};

template<typename V, typename E = std::exception_ptr, auto&... OtherCPOs>
using any_sender_of = any_unique_t<
  overload<void(this_&&, any_receiver_of<V, E>)>(submit),
  OtherCPOs...
  >;

using my_float_sender = any_sender_of<float, std::exception_ptr,
                                      overload<float(this_&&)>(sync_wait)>;

void example(my_float_sender s) {
  // Call to sync_wait() will dispatch to concrete implementation.
  float result = sync_wait(std::move(s));
```

```
  std::cout << "Result was: " << result << "\n";
}
```

Another example is a type-erasing wrapper that wraps a container-like thing that exposes a `get(c, idx)` operation for member access and a `sort(c)` operation to sort the members:

```
// Declare two CPOs:
// - get(C& container, size_t index) -> T&
// - sort(C& container) -> void

// get() CPO
inline constexpr struct get_fn {
  // Default implementation
  template<typename Container>
    requires requires (Container c, size_t idx) { c[idx]; }
  friend auto tag_invoke(get_fn, Container& c, idx) -> decltype(c[idx]) {
    return c[idx];
  }

  template<typename Container>
    requires std::tag_invocable<get_fn, Container&, size_t>
  auto operator()(Container& c, size_t idx) const
    -> std::tag_invoke_result_t<get_fn, Container&, size_t> {
    return std::tag_invoke(*this, c, size_t(idx));
  }
} get{};

// sort() CPO
inline constexpr struct sort_fn {
  template<typename Container>
    requires requires (Container c) { std::ranges::sort(c); }
  friend void tag_invoke(sort_fn, Container& c) { std::ranges::sort(c); }

  template<typename Container>
    requires std::tag_invocable<sort_fn, Container&>
  void operator()(Container& c) const { std::tag_invoke(*this, c); }
} sort{};

// Declare a type-erased sortable container.
// Allows member access by calling get() and sorting by calling sort()
template<typename T>
using sortable_container = any_unique_t<
  overload<const T&(const this_&, size_t)>(get),
  overload<T&(this_&, size_t)>(get),
  overload<void(this_&)>(sort)>;

// Example usage.
sortable_container<int> c = std::vector<int>{4, 7, 2, 9, 3};
assert(get(c, 0) == 4);
get(c, 2) = 1;
sort(c); // Dispatches to concrete implementation of sort()
assert(get(c, 0) == 1);
assert(get(c, 1) == 3);
```

See https://godbolt.org/z/JJ5TcZ for an example implementation of this approach.

Different type-erasing adapters can be defined with different strategies for storage, ownership and copy/move semantics. But all of the adapters can use the same technique for customising CPOs using `tag_invoke`.

This paper is not proposing any particular type-erasure abstraction for the standard library - that should be the subject of a separate paper. These examples are presented here primarily to demonstrate that it is possible to build generic type-erased wrappers once we have a generic way to customise any customisation point.

Simplifying use of Properties

The paper P1393R0 "A General Property Customization Mechanism" proposes a facility that lets applications define "properties" that can optionally support one or more of the basis operations: `query()`, `prefer()`, `require()` and `require_concept()`.

The `query()` operation allows the caller to retrieve the value of a property for a given object.
e.g. `std::query(someExecutor, std::execution::concurrency)` might retrieve the maximum number of tasks an executor can execute concurrently.

The `prefer()` and `require()` operations allow the caller to adapt/transform an object into a new object that has the same interface (ie. supports the same set of operations) as the original object but that has a given property value. This can be done to either enforce particular semantic behaviours or provide performance-tuning hints to an algorithm.
e.g. `std::require(someExecutor, std::execution::continuation)` could return an executor that will treat tasks executed on it as continuations of the current task, and so should be executed immediately on the same thread when the current task completes to take advantage of the current task's data likely already being in CPU caches.

The `require_concept()` operation allows the caller to adapt/transform an object into a new object that is based on the original object but that has a different interface.

The facilities proposed by P1393R0 have an overlap in capability with this paper in that it is also attempting to provide a mechanism for allowing types to customise the behaviour of some operations through the use of customisation points.

Properties, basis operations and customisable algorithms

The properties system described by P1393 draws a distinction between two axes of customization: properties and algorithms/basis operations. However, we note that in practice it is very difficult to define a line between what is a property, a basis operation and a customisable algorithm; they are all some flavour of customizable functions.

By treating these things consistently, we can make it possible for algorithm customizations, basis operation customizations, and property customizations to follow a type generically through transparent layers of adaptation.

While these facilities can also be used to provide support for implementing general customisable functions, doing so via property customisation is less-direct and more restrictive compared to the `tag_invoke()`-based approach described by this paper.

The property representation of customisable algorithms would need a separate property for each algorithm and would either use `query()` or `require_concept()` to obtain an invocable that represented the corresponding customisation for that algorithm.

Some examples of different techniques that use properties to implement customisable algorithms can be found here: https://github.com/chriskohlhoff/propria/tree/8d4762f85da8dc83d46cd53e4a5fa412f4c40862/examples/cpp17/algorithms

CPOs as an alternative to properties

Many of the use-cases for properties can also potentially be represented directly using CPOs implemented in terms of tag_invoke.

Query-only properties could be defined as customisable getter-functions that can be directly called to retrieve the result of the query.

P1393 properties also have the ability to perform a compile-time query by querying the `Property::static_query_v<T>` member. This facility is used by the `require()` customisation point to allow it to return the object unmodified it if it already has the given property value.

Example: A query-only property, `concurrency`, from P1436R0 that lets you query the maximum available concurrency for an executor. This query has no default implementation.

P1393 properties approach	**tag_invoke() CPO approach**

`// Defining the 'concurrency' property`
`struct concurrency_t {`
`  template<typename T>`
`  static constexpr bool is_applicable_property_v =`
`  std::executor<T>;`
 	
`  static constexpr bool is_preferable = false;`
`  static constexpr bool is_requirable = false;`
 	
`  using polymorphic_query_result_type = size_t;`
 	
`  template<typename Executor>`
`  constexpr auto static_query_v =`
`  Executor::query(concurrency_t{});`
	`// Defining the 'get_concurrency()' function.`
	`struct get_concurrency_fn {`
	`  template<std::executor E>`
	`  requires std::tag_invocable<`
	`  get_concurrency_fn, const E&>`
	`  auto operator()(const E& executor) const`
	`  noexcept(std::nothrow_tag_invocable<`
	`  get_concurrency_fn, const E&>)`
	`  -> std::tag_invoke_result_t<`
	`  get_concurrency_fn, const E&> {`
	`  return std::tag_invoke(*this, executor);`
	`  }`
	`};`

```
};

inline constexpr concurrency_t concurrency{};
```

```
inline
constexpr get_concurrency_fn get_concurrency{};
```

```
// Customising query of property for a type.
class my_thread_pool_executor {
  ...
private:
  friend
  size_t query(concurrency_t,
               const my_thread_pool& tp) {
    return tp.threadCount_;
  }

  size_t threadCount_;
  ...
};
```

```
// Customising the get_concurrency() function.
class my_thread_pool_executor {
  ...
private:
  friend
  size_t tag_invoke(std::tag_t<get_concurrency>,
                    const my_thread_pool& tp) {
    return tp.threadCount_;
  }

  size_t threadCount_;
  ...
};
```

```
// Querying runtime value of an executor
template<std::executor E>
  requires
    std::can_query_v<concurrency_t, E>
void some_algorithm(E ex) {
  size_t concurrency = std::query(concurrency,
ex);
  // ... create 'concurrency' tasks.
}
```

```
// Querying runtime value of an executor
template<std::executor E>
  requires
    std::invocable<decltype(get_concurrency), E>
void some_algorithm(E ex) {
  size_t concurrency = get_concurrency(ex);
  // ... create 'concurrency' tasks.
}
```

```
// Customising a static query of this property
class my_strand_executor {
  ...

public:
  static constexpr size_t query(concurrency_t) {
    // Only executes 1 task at a time.
    return 1;
  }


  ...
};
```

```
// Customising static get_concurrency()
function.
class my_strand_executor {
  ...

private:
  friend constexpr size_t tag_invoke(
      std::tag_t<get_concurrency>,
      any_instance_of<my_strand_executor>) {
    // Only executes 1 task at a time.
    return 1;
  }
  ...
};
```

NOTE: The RHS implementation of the equivalent of `static_query` makes use of an empty helper type `any_instance_of<T>` that is implicitly constructible from any type from which a `T` can be implicitly constructed. This allows this overload to be chosen when passed either an `any_instance_of<T>`, as in a static-query, or a value of type `T`, as in a runtime-query.

See https://godbolt.org/z/MERH5I for an example implementation of static queries using `tag_invoke`-based CPOs.

It is also possible to represent the `prefer/require/require_concept` operations on properties as customisable algorithms that perform adaption/transformation of objects.

For example, an algorithm that takes an arbitrary executor and returns an executor that satisfies the "strand executor" concept (i.e. that never schedules work concurrently) could be represented as a '`strand_executor`' property or using CPOs for querying the strand-ness, `is_strand_executor(e)`, and for turning an executor into a strand, `make_strand_executor(e)`.

Properties	CPOs

`// Define property`
`struct strand_executor_t {`
`  static constexpr bool is_requirable = true;`
`  static constexpr bool is_preferable = false;`
 	
`  template<typename T>`
`  static constexpr bool static_query_v = ...;`
 	
`  ...;`
`};`
 	
`inline constexpr`
`strand_executor_t strand_executor{};`
	`// Define CPOs - with default implementations.`
	`inline constexpr unspecified is_strand_executor;`
	`inline constexpr unspecified`
	`make_strand_executor;`
	
	`// Static query`
	`template<typename T>`
	`inline constexpr auto is_strand_executor_v =`
	`  is_strand_executor(any_instance_of_v<T>);`

`some_executor e;`
 	
`auto strand = std::require(e, strand_executor);`
 	
`static_assert(`
`  std::static_query_v<decltype(strand),`
`  strand_executor>);`
 	
`execute(strand, [] { do_something(); });`
	`some_executor e;`
	
	`auto strand = make_strand_executor(e);`
	
	`static_assert(`
	`  is_strand_executor_v<decltype(strand)>);`
	
	`execute(strand, []{ do_something(); });`

One of the limitations of the properties based approach using prefer/require is that it assumes that there is only one way to transform an object into another object that has a particular property. However, there might be several such strategies for doing this.

With P1393 properties, the primary strategy for implementing a property is defined by the overload selected by a call to `std::require(object, property)`. Other strategies can still be defined using algorithms but these cannot use the `require()` syntax since there can be only a single overload selected for this call.

Property CPOs in terms of tag_invoke

If the committee determines that the P1393 property mechanisms are still desired then it would also be possible to build the property mechanism on top of `tag_invoke()`.

The `std::query`, `std::prefer`, `std::require` and `std::require_concept` names described in P1393 are CPOs and so could be defined as being customisable via `std::tag_invoke()` without needing to reserve the ADL names `query`, `prefer`, `require` and `require_concept` globally.

Doing so would allow properties to make use of generic type-erasure and other adapter facilities built on top of `tag_invoke()` rather than having to build that capability into the properties mechanism itself. This would also allow type-erasing wrappers to forward through calls to particular overloads of customised algorithms in addition to forwarding through property queries and calls to `prefer()`/`require()`.

Proposal Details

This section describes the additions proposed for the standard library to be able to support defining `tag_invoke`-based CPOs.

std::tag_invoke

```
// <functional>
namespace std
{
  inline namespace unspecified {
    inline constexpr unspecified tag_invoke = unspecified;
  }
}
```

The `std::tag_invoke` name defines a constexpr object that is invocable with one or more arguments, the first argument being the 'tag' (typically a CPO), if and only if an overload of `tag_invoke()` that accepts the same arguments could be found by ADL.

Evaluation of the expression `std::tag_invoke(tag, args...)` is equivalent to evaluating the unqualified call to `tag_invoke(`*`decay-copy`*`(tag), args...)` with overload resolution performed in a context that includes the declaration:

```
void tag_invoke();
```

and that does not include the `std::tag_invoke` name.

[[Editorial note: The `std::tag_invoke` CPO has been nominally placed in the <functional> header as its name suggests an association with `std::invoke` which is also defined in <functional>. Other names and other headers could also be considered.]]

Type traits

This section defines some type-traits that simplifies the definition of customisation-point objects that are defined in terms of `std::tag_invoke`, allowing them to more easily constrain overloads, deduce return-types and forward noexcept qualification of the CPO's `operator()`.

```
// <type_traits>
namespace std
```

```
{
  template<auto& Tag>
  using tag_t = decay_t<decltype(Tag)>;

  template<class Tag, class... Args>
  concept tag_invocable =
    invocable<decltype(tag_invoke), Tag, Args...>;

  template<class Tag, class... Args>
  concept nothrow_tag_invocable =
    tag_invocable<Tag, Args...> &&
    is_nothrow_invocable_v<decltype(tag_invoke), Tag, Args...>;

  template<class Tag, class... Args>
  using tag_invoke_result = invoke_result<decltype(tag_invoke), Tag, Args...>;

  template<class Tag, class... Args>
  using tag_invoke_result_t = invoke_result_t<decltype(tag_invoke), Tag, Args...>;
}
```

Question: Should `nothrow_tag_invocable` be a concept or a `constexpr bool nothrow_tag_invocable_v` type-trait?

Example usage of type-traits

An example usage of these facilities in definition of a CPO:

```
inline constexpr struct schedule_fn {
  template<typename T>
    requires std::tag_invocable<schedule_fn, T>
  auto operator()(T&& x) const
      noexcept(std::nothrow_tag_invocable<schedule_fn, T>)
      -> std::tag_invoke_result_t<schedule_fn, T> {
    return std::tag_invoke(*this, (T&&)x);
  }
} schedule{};
```

and in definition of a customisation of this CPO:

```
struct some_scheduler {
  ...
private:
  struct schedule_sender { /*...*/ };
  friend schedule_sender tag_invoke(std::tag_t<schedule>,
                                    const some_scheduler& s) noexcept {
    return schedule_sender{s};
  }
};
```

Note the use of the `std::tag_t` helper to simplify obtaining the type of the CPO from the CPO name. This allows writing `std::tag_t<some_cpo>` instead of `std::decay_t<decltype(some_cpo)>`.

Allowing CPOs to be defined in such a way that the type of the CPO is left unspecified gives the implementation more flexibility over implementation strategy and also can avoid needing to introduce two names into the `std::` namespace for each CPO.

We only specify the name of the object itself, we don't specify the name of the type. But this means we need to provide some convenient way of accessing the type of a CPO so that user-defined types can define overloads of `tag_invoke()` that customise that CPO.

Using the type-alias, `std::tag_t`, which extracts that type directly from the CPO itself creates a stronger tie between the `tag_invoke()` overloads and the CPO compared to explicitly naming the CPO type, which would otherwise need to rely on a naming convention to provide the association between the CPO and its type.

```
// Exposing the CPO type name
// Using type name directly in customisations
namespace somelib
{
  inline constexpr struct foo_fn
  {
    ...
  } foo;
}

struct my_type
{
  ...
  friend void tag_invoke(somelib::foo_fn,
                         const my_type& x);
};
```

```
// Hiding CPO type name
// Using std::tag_t<CPO> in customisation instead
namespace somelib
{
  inline constexpr struct unspecified
  {
    ...
  } foo;
}

struct my_type
{
  ...
  friend void tag_invoke(std::tag_t<somelib::foo>,
                         const my_type& x);
};
```

The use of the object name in the signature should also allow customisations of the CPO to be found using an IDE's "find all references" feature.

Design Discussion

Strategies for defining default implementations of CPOs

When defining a customisable algorithm that has a default implementation in terms of some concept there are a couple of ways in which the default implementation can be defined. Each with differing tradeoffs.

1) Define an unconstrained tag_invoke() method

This approach involves defining a default implementation of the CPO as an unconstrained `tag_invoke` hidden-friend overload within the CPO type itself.

For example:

```
inline constexpr struct contains_fn {
  // Unconstrained default implementation.
```

```
  template<typename Range, typename Value>
  friend bool tag_invoke(contains_fn, Range&& r, const Value& value) {
    return std::ranges::find(r, value) != std::ranges::end(r);
  }

  template<std::range R, typename Value>
    requires std::tag_invocable<contains_fn, R, const Value&>
  auto operator()(R&& r, const Value& value) const
    noexcept(std::nothrow_tag_invocable<contains_fn, R, const Value&>)
    -> std::tag_invoke_result_t<contains_fn, R, const Value&> {
    return std::tag_invoke(*this, (R&&)r, value);
  }
} contains{};
```

The hidden friend declared here will be found by the ADL call to `tag_invoke()` as the tag argument is of type `contains_fn` and so its hidden friends will be considered in the overload set.

This approach has the benefit that we can just use the same `operator()` definition as for basis-operation CPOs that forwards on to `std::tag_invoke()`. The return-type of `operator()` can be declared using `std::tag_invoke_result_t` and doesn't need to be modified to handle either being a

One problem with this approach, though, is that the unconstrained overload can sometimes be a better match than a more appropriate customisation.

For example:

```
template<typename T>
class my_hash_set {
  ...
private:
  friend bool tag_invoke(
    std::tag_t<contains>, const my_container& c, const value_type& value);
};

void example() {
  my_hash_set<int> a;
  const auto& ca = a;

  bool result1 = contains(ca, 123); // calls custom version.
  bool result2 = contains(c, 123);  // calls default version!
  bool result3 = contains(ca, convertibleToInt); // calls default version!
}
```

The reason for this is that the unconstrained version is a better match than the version that customised implementation when passed slightly different argument types.

This can often be addressed by appropriately constraining the custom `tag_invoke` overload. eg. by writing a generic `tag_invoke()` overload that accepts both l-value, r-value and const/non-const arguments for the container type.

However, this can be difficult to get right and can lead to additional template instantiations of the algorithm.

Ideally, the CPO would dispatch to the custom version if a custom version is callable with the provided arguments and only fall back to the default one if no valid custom overload exists. Which leads us to approach #2.

2) Use if constexpr to dispatch to tag_invoke() if overload is defined, otherwise fall back to some default implementation

This approach has the implementation of `operator()` detect whether there is a customised version of the algorithm that is callable with the arguments and if so dispatches to that version, otherwise uses some default implementation.

```
inline constexpr unspecified via;
inline constexpr unspecified transform;

inline constexpr struct then_execute_fn {
  template<sender S, executor E, typename F>
  decltype(auto) operator()(S&& sender, E&& executor, F&& func) const {
    if constexpr (std::tag_invocable<then_execute_fn, S, E, F>) {
      // There is a customisation defined for this overload set. Call it.
      static_assert(sender<std::tag_invoke_result_t<then_execute_fn, S, E, F>>);
      return std::tag_invoke(*this, (S&&)sender, (E&&)executor, (F&&)func);
    } else {
      // Fall-back to default implementation.
      // Works for any args that satisfy the function's constraints.
      return transform(via((S&&)sender, (E&&)executor), (F&&)func);
    }
  }
} then_execute;
```

This approach allows the CPO to define a chain of successive fallback implementations that are checked in a particular order. This could allow different default implementations to be used depending on properties of the arguments.

3) Use overload resolution with negative constraints to define order.

Another approach to defining CPOs with a default implementation is to define separate overloads of operator(). This can, however, require additional constraints on the overload set to define a particular order of preference in cases where the overloads would otherwise be ambiguous.

```
inline constexpr unspecified via;
inline constexpr unspecified transform;

inline constexpr struct then_execute_fn {
  // Dispatch to customisation if one is defined.
  template<sender S, executor E, typename F>
    requires std::tag_invocable<then_execute_fn, S, E, F>
  auto operator()(S&& sender, E&& executor, F&& func) const
    noexcept(std::nothrow_tag_invocable<then_execute_fn, S, E, F>)
    -> std::tag_invoke_result_t<then_execute_fn, S, E, F> {
    return std::tag_invoke(*this, (S&&)sender, (E&&)executor, (F&&)func);
  }

  // Fall-back to default implementation if no customisation found.
  // Works for any args that satisfy the function's constraints.
  template<sender S, executor E, typename F>
    requires (!std::tag_invocable<then_execute_fn, S, E, F>)
  decltype(auto) operator()(S&& sender, E&& executor, F&& func) const
      noexcept(noexcept(transform(via((S&&)sender, (E&&)executor), (F&&)func))) {
    return transform(via((S&&)sender, (E&&)executor), (F&&)func);
  }
} then_execute;
```

This approach makes it easier to perfectly forward noexcept and return-types but requires some careful crafting of constraints to avoid creating ambiguous overloads.

Note that it may be difficult to add extra overloads in a non-ABI-breaking way in future releases if taking this approach as the requires-clause generally forms part of the mangled name of a function overload.

Providing default implementations in terms of other concepts

In some cases we may want to provide a generic implementation of a CPO for a category of types defined by some other concept than the default implementation of an algorithm is in terms of.

One example for this might be the implementation of the submit() customisation-point, which is a basis operation for the Sender concept, in terms of the coroutines Awaitable concept. It is possible to treat any Awaitable like a Sender so we should be able to define a generic implementation of submit() in terms of the Awaitable interface.

For example: A generic implementation of `submit()` for any awaitable type.

```
template<awaitable A, receiver R>
void tag_invoke(std::tag_t<submit>, A&& awaitable, R&& receiver) {
```

```
  [](std::decay_t<A> awaitable, std::decay_t<R> receiver) -> oneway_task {
    try {
      if constexpr (std::is_void_v<await_result_t<decltype(awaitable)>>) {
        co_await std::move(awaitable);
        set_value((R&&)receiver);
      } else {
        set_value((R&&)receiver, co_await std::move(awaitable));
      }
    } catch (...) {
      set_error((R&&)receiver, std::current_exception());
    }
  }((A&&)awaitable, (R&&)receiver);
}
```

However, even if we define such an overload, ensuring that this overload is found by ADL for all possible awaitables can be a challenge.

We would need to have this overload of `tag_invoke()` placed in a namespace that was associated with *every* ADL call to `tag_invoke()` for this CPO with an awaitable. The obvious approach for this is to place these overloads in a namespace associated with the `submit` CPO type itself.

To enable this use-case, CPOs would need to advertise an associated namespace that generic tag_invoke() overloads could be added to and permit applications to add overloads of `tag_invoke()` for that CPO to that namespace.

For example, we could define the `submit` CPO as follows:

```
namespace std::execution {
  namespace submit_ns {
    struct __submit_base {};
  }

  // Inherit from a type defined in submit_ns to make it an associated
  // namespace.
  inline constexpr struct __submit_fn : __submit_base {
    template<typename Sender, typename Receiver>
      requires std::tag_invocable<__submit_fn, Sender, Receiver>
    void operator()(Sender&& s, Receiver&& r) const {
      std::tag_invoke(*this, (Sender&&)s, (Receiver&&)r);
    }
  } submit;
}
```

Then third-party libraries can add a generic overload of `tag_invoke()` to that namespace to have them findable by the implementation of the CPO.

For example, the generic `tag_invoke()` overload for the submit CPO for all awaitables shown above could be defined in this associated namespace to enable any Awaitable to automatically implement satisfy the Sender concept by virtue of implementing the `submit()` CPO.

This capability is something that should be considered when defining new CPOs. However, it is not without its dangers.

This can lead to ODR issues if that generic definitions are not available everywhere they might be used. It can also lead to an increase in compile times if a large number of generic overloads are added to the associated namespace.

It can also potentially lead to ambiguous calls if multiple generic overloads are added for different concepts and we try to call the CPO with a type that implements both concepts.

More research and deployment experience is required to fully understand the implications of providing such a facility.

Potential for ODR issues

Whenever we have a customisation point that is defined in terms of ADL calls, or even template specialization, it is possible that some translation units will have visibility of a different set of those overloads or template specialisations than other translation units based on which headers were included or which modules were imported.

This can lead to ODR-violations if an application is not careful to ensure the same set of customisations of tag_invoke() overloads for types are visible consistently wherever those types are used throughout a codebase.

This can introduce problems if, for example, new overloads of `tag_invoke()` are introduced for a given type and the application does not recompile all code that could possibly have invoked that new overload.

The risk of ODR violations can be mitigated somewhat by declaring all `tag_invoke` customisations as friend functions of the associated type. This ensures that whenever a type is used that its customisation of algorithms for that type are also available.

These ODR problems are not new but may become more common if CPOs encounter more widespread use.

Potential for ABI breaks

Calls to CPOs that dispatch to `std::tag_invoke()` might return different types depending on whether particular `tag_invoke()` customisations were found.

This means that adding a `tag_invoke()` overload later might cause a CPO call to change its return-type to a different type when called with the same arguments.

These ABI problems are not new but may become more common if CPOs encounter more widespread use.

The name-mangling for a function template on some platforms does not encode the concrete return-type that was deduced, only the expression used to deduce the template. This will typically be in terms of something like `decltype(std::tag_invoke(args...))`.

If an invocation of a CPO originally dispatches to the default implementation and then later adds a new `tag_invoke()` customisation such that some TUs use one implementation and other TUs use a different implementation then can end up with undiagnosed ODR-violations as the new instantiation of the CPO's operator() might have the same mangled name but a different return-type.

We can mitigate this somewhat by having CPOs encode the chosen return-type as a defaulted template parameter to the `operator()` function. This way, the chosen return-type would at least be encoded in the mangled name and so if you have different TUs that find different overloads then you at least have a chance of detecting this at the linker stage.

For example:

```
inline constexpr struct some_cpo {
  template<
    typename A,
    typename B,
    typename Result = std::tag_invoke_result_t<some_cpo, A, B>>
  Result operator()(A&& a, B&& b) const {
    return std::tag_invoke(*this, (A&&)a, (B&&)b);
  }
};
```

It may not catch all such ODR violations, however. And this approach would need to be applied consistently across all functions whose return-types may depend on the result of a customisation point.

The wider implication here, however, is that it could be an ABI-break to later add any overload of `tag_invoke()` that would change the return-type of a CPO after adding the type. For some third-party libraries this may not be a problem, but it could be a problem for standard library async algorithms and standard library types.

Again, these issues are not specific to this proposal. Any customisation mechanism we come up with is going to suffer from the same challenges as they are inherent to C++'s separate compilation mechanism.

Compile-time impacts

One concern that needs to be investigated further is the impact that having all CPOs dispatch to a single ADL name would have on compile times.

By having every CPO customisable using the same '`tag_invoke`' name, if we end up with a large number of types all customising a large number of CPOs then the number of overloads of `tag_invoke` in the overload-set that the compiler needs to consider when resolving the correct overload has the potential to be large and this could potentially impact compile times.

The compile-time impacts can be mitigated somewhat by limiting the number of `tag_invoke` overloads that are considered by a given `tag_invoke` ADL call.

This can be achieved by:

- defining CPO types in separate namespaces that do not contain and `tag_invoke()` overloads
- declaring `tag_invoke()` customisations for particular types as hidden-friends of those types so that these overloads are only considered when that type is an argument to a CPO

In cases where these steps are not enough (*e.g.*, when a CPO takes a large number of arguments), we can further limit the number of associated namespaces and types to inspect by having the CPO dispatch to `std::tag_invoke()` with only a subset of the arguments. The `tag_invoke` overload thus found would return an invocable object that curries those arguments out. The CPO, after obtaining this invocable by dispatching to `tag_invoke()`, would then invoke the invocable with the rest of the arguments.

This should, in the vast majority of cases, present the compiler with only a handful of `tag_invoke` overloads in the overload set for it to sort through during overload resolution.

Conclusion

There is a growing need for being able to define customisable functions in the standard library. The most pressing use-case for them at the moment is in support of the design of executors.

This paper proposes that CPOs should in general be defined in terms of ADL calls to overloads of the `tag_invoke()` function and proposes adding a `std::tag_invoke` helper and some associated concepts and type-traits to help definition of this style of CPO.

This approach to defining CPOs has some advantages over the existing approach of using a separate ADL name for each CPO: it enables building generic adapters that forward through CPO calls, and avoids potential conflicts due to libraries choosing conflicting ADL names.

This approach also has the potential to either replace the need for the P1393R0 properties facilities or complement it as a potential foundational building block on which the higher-level properties APIs can be built, enabling general abstractions like type-erased wrappers to be able to work with both properties and customisable algorithms at the same time.