`tag_invoke` in 0443

- D2220R0 - *redefine properties in P0443*
- D2221R0 - *define P0443 cpos with* `tag_invoke`

these papers are implemented in a fork of asio (github)

`tag_invoke` (http://wg21.link/P1895R0)

> *abstract* .. 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)`.

`tag_invoke` solves two problems

Creating a new CPO as described in *[customization.point.object]* solves problems related to ADL customization. There are two additional problems solved by creating new CPOs in terms of `tag_invoke`

problem	description
identifier collisions:	Each CPO 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.
no generic forwarding:	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.

D2221R0 - *define P0443 cpos with* `tag_invoke`

changes not intended to change functionality

D2221R0 - *define P0443 cpos with* `tag_invoke`

changes to the wording of a CPO in 0443

Before

- `O.`*`name`*`()`, if that expression is valid.
- Otherwise, *`name`*`(O)`, if that expression is valid, with overload resolution performed in a context that includes the declaration

```
void name();
```

  and that does not include a declaration of `execution::`*`name`*.

After

- `tbd::tag_invoke(`*`name`*`, O)`, if that expression is valid.

D2221R0 - *define P0443 cpos with* `tag_invoke`

changes to the definition of a CPO in 0443

Before

```
namespace _name {
  void name();

  struct _free_fn {
    template <typename Target, typename... Args>
    constexpr auto operator()(
      Target&& t, Args&&... args) const
        noexcept(noexcept(
          name(
            (Target &&) t, (Args &&) args...)))
        -> decltype(
          name(
            (Target &&) t, (Args &&) args...)) {
      return name(
        (Target &&) t, (Args &&) args...);
    }
  };

  struct _member_fn {
    template <typename Target, typename... Args>
      requires (!std::invocable<
        _free_fn, Target, Args...>)
    constexpr auto operator()(
      Target&& t, Args&&... args) const
        noexcept(noexcept(
          ((Target &&) t).name(
            (Args &&) args...)))
        -> decltype(
          ((Target &&) t).name(
            (Args &&) args...)) {
      return ((Target &&) t).name(
        (Args &&) args...);
    }
  };
  struct _fn : _free_fn, _member_fn {
    using _free_fn::operator();
    using _member_fn::operator();
  };
}  // namespace _name

namespace _name_cpo {
  inline constexpr _name::_fn name{};
}
using namespace _name_cpo;
```

After

```
namespace _name_cpo {
  inline const struct _fn {
    template<typename Target, typename... Args>
      requires tag_invocable<
        _fn, Target, Args...>
    auto operator()(
      Target&& t, Args&& args...) const
        noexcept(
          is_nothrow_tag_invocable_v<
            _fn, Target, Args...>) ->
        tag_invoke_result_t<
          _fn, Target, Args...> {
      return tbd::tag_invoke(
        _fn{}, (Target&&)t, (Args&&)args...);
    }
  } name{};
} // namespace _name

using _name_cpo::name;
```

er)

D2221R0 - *define P0443 cpos with `tag_invoke`*

changes to the `inline_executor` example in 0443

Before

```
struct inline_executor {
  // define execute as friend
  template<class F>
  friend void execute(
    //
    const inline_executor&,
    F&& f) noexcept {
    std::invoke(std::forward<F>(f));
  }

  // enable comparisons
  auto operator<=>(
    const inline_executor&) const = default;
};
```

After

```
struct inline_executor {
  // define execute as friend
  template<class F>
  friend void tag_invoke(
    tag_t<execution::execute>,
    const inline_executor&,
    F&& f) noexcept {
    std::invoke(std::forward<F>(f));
  }

  // enable comparisons
  auto operator<=>(
    const inline_executor&) const = default;
};
```

D2221R0 - *define P0443 cpos with `tag_invoke`*

summary

P0443R13 defines the following cpos: `set_value`, `set_done`, `set_error`, `execute`, `connect`, `start`, `submit`, `schedule`, and `bulk_execute`.

D2221R0 has the changes to P0443R13 wording needed to define these cpos in terms of `tag_invoke`.

- removes the globally reserved names
- adds support for generic forwarding

Any questions?

'properties' (http://wg21.link/P1393R0)

> *In general* When the property customization mechanism is being employed for some library facility, an object's behavior and effects on that facility in generic contexts may be determined by a set of applicable properties, and each property imposes certain requirements on that object's behavior or exposes some attribute of that object. As well as modifying the behavior of an object, properties can be applied to an object to enforce the presence of an interface, potentially resulting in an object of a new type that satisfies some concept associated with that property.

'properties' were designed to satisfy requirements of executors

Requirement	Definition
Extensible	users are able to define their own
Survivable	passing objects around does not lose information
Forwardable	it is possible to write a catch-all that forwards to a contained object
Overridable	it is possible to modify or block forwarding
Defaultable and **Ignorable**	must be able to define a default when not customized and ignore when not customized

(from: https://github.com/executors/executors/issues/191)

`tag_invoke` and 'properties' satisfy those requirements

Requirement	`tag_invoke`	'properties'
Extensible	define a new tag function type in a namespace	define a new traits struct in a new namespace
Survivable	defined as overloads of the `tag_invoke()` function	defined as overloads of `require()`, `require_concept()`, `prefer()`, and `query()` functions
Forwardable	a `tag_invoke()` function that does not constrain the namespaced tag function argument	`require()`, `require_concept()`, `prefer()`, and `query()` functions that do not constrain the namespaced tag struct argument
Overridable	a `tag_invoke()` function constrained to a specific namespaced tag function	`require()`, `require_concept()`, `prefer()`, and `query()` functions constrained to a specific namespaced tag struct
Defaultable and **Ignorable**	the namespaced tag function type has an overload of `operator()` constrained with `tag_invocable<..> == false`	the namespaced tag struct overloads `require()`, `require_concept()`, `prefer()`, and `query()` hidden friend functions with the target unconstrained and constrained with `can_require<..>`, `can_require_concept<..>`, `can_prefer<..>`, and `can_query<..> == false`

definition

property

```
struct my_property
{
  template<class... Ps>
    static constexpr bool
      is_applicable_property_v;

  /* optional */
  static constexpr bool
    is_requirable_concept = /* ... */;

  /* optional */
  static constexpr bool
    is_requirable = /* ... */;

  /* optional */
  static constexpr bool
    is_preferable = /* ... */;

  /* optional */
  template<class... Ps>

    class polymorphic_wrapper_type;

  /* optional */
  using
    polymorphic_query_result_type = /*..*/;

  /* optional */
  template<class T>
    static constexpr /* ... */
      static_query_v = /* ... */;

  /* optional */
  static constexpr /* ... */
    value() const { return /* ... */; }
};
```

tag_invoke function

```
inline constexpr struct my_function_fn {
  template<typename... Tn>
  auto operator()(Tn&&... tn) const
    noexcept(noexcept(
      tbd::tag_invoke(*this, (Tn&&)tn...)))
    -> decltype(
      tbd::tag_invoke(*this, (Tn&&)tn...)) {
    return
      tbd::tag_invoke(*this, (Tn&&)tn...);
  }
} my_function{};
```

'properties' has additional functionality

Meaning

- A fixed set of functions are defined (`require`, `prefer`, `query`, `require_concept`)
- These functions are defined to have the same meaning across all properties.
- Some things are not good properties because they do not have a good mapping to the meanings available (eg. `execute()`).
- Each property author selects the subset of meanings that 'make sense' for that property.

> .. each property imposes certain requirements on that object's behavior or exposes some attribute of that object. As well as modifying the behavior of an object, properties can be applied to an object to enforce the presence of an interface, potentially resulting in an object of a new type that satisfies some concept associated with that property.

'properties' has additional functionality

function	best-guess meaning	trait members
`require`	make a new instance of the target instance that satisfies the supplied property *iff* the target instance does not satisfy the supplied property, otherwise return the original target instance unchanged	`is_requirable`, `polymorphic_wrapper_type`
`prefer`	make a new instance of the target instance that satisfies the supplied property *iff* the supplied property is supported by the target and the target instance does not satisfy the supplied property, otherwise return the original target instance unchanged	`is_preferable`, `polymorphic_wrapper_type`
`require_concept`	make a new instance of a type that satisfies a concept determined by the supplied property *iff* the target instance does not satisfy the concept determined by the supplied property, otherwise return the original target instance unchanged	`is_requirable_concept`, `polymorphic_wrapper_type`
`query`	return the value of the supplied property that this target instance satisfies	`polymorphic_query_result_type`, `static_query_v`, `value()`

'properties' has additional functionality

trait member	meaning
`is_applicable_property_v<T>`	`true` *iff* `T` is allowed to support this property. In [@P0443R13] - *executors* this is always derived from concepts
`is_requirable_concept`	`true` *iff* this property can be used with `require_concept()`
`is_requirable`	`true` *iff* this property can be used with `require()`
`is_preferable`	`true` *iff* this property can be used with `prefer()`
`static_query_v<T>`	accesses the value that would be returned from a call to `query()` that is targeted on T *iff* the value has been made available at compile-time.
`value()`	provides a value that will be compared to `static_query_v<T>` to "determine whether invoking `require` or `require_concept` would result in an identity transformation." - [@P1393R0] - *properties*
`polymorphic_wrapper_type< SupportableProperties...>`	defines the polymorphic type for a specific concept. The concept-specific expressions are built-in (like `execute()` for `any_executor<>`)

trait member	meaning
	defines the result type for `query` when used by any

'properties' has additional functionality

Polymorphism

- `is_applicable` to potentially many concepts (eg. all the properties in P0443)
- Each concept is expected to author a polymorphic type that implements that concept and supports forwarding of properties.
- Defines the polymorphic types to use when type-erasing the results of functions that operate on properties (`require`, `prefer`, `query`, and `require_concept`).
- The author of a property is unlikely to choose a polymorphic type implementation that all users of that property will find satisfying.

Behaviour properties in P0443R13

Behaviour properties are a pattern of using nested properties and bespoke polymorphism to define a group of mutually-exclusive behaviours.

Polymorphism

- A property `group_t` (eg. `mapping_t`) will be a value type that is *equality-comparable* and that can be constructed from values of each of the nested property types.
- Each of the nested property types are value types that are *equality-comparable*.
- `group_t` will be *equality-comparable* to the nested type that was used to construct it. The implementation of this polymorphism is unspecified - asio uses an integer and each constructor taking a nested property will set a specific value to the integer.
- users and libraries cannot add additional nested behaviours to a standarized property.

> open question: can any new nested behaviour type be standardized? what are the ABI implications?

D2220R0 - *redefine properties in P0443*

changes not intended to change functionality

D2220R0 - *redefine properties in P0443*

move `prefer_only` to P1393 - *'properties'*

- composes with a property and blocks support for `require`.
- used to allow a polymorphic type to support a property 'optionally'.
- nothing to do with executors

D2220R0 - *redefine properties in P0443*

remove `any_executor`

- There is an implementation of `any_ref<>`, with `tag_invoke` forwarding, in my changes to asio (github) that works to construct `any_executor<>` as a templated type alias.
- Polymorphism for `tag_invoke` functions is a separate concern from both `tag_invoke` and executors.
- There will be several `any_` types for `tag_invoke` (eg. `any_value`, `any_ref`, `any_unique`, `any_shared`, etc..)

D2220R0 - *redefine properties in P0443*

naming functions

query	requires	prefer		function
`query(t, allocator)`	`requires(t, allocator(a))`	`prefer(t, allocator(a))`	vs	`get_allocator(t)` & `make_with_allocator( t, a)`

D2220R0 - *redefine properties in P0443*

naming behaviours

requires	prefer		behaviour
`requires(t, mapping.thread)`	`prefer(t, mapping.thread)`	vs	`make_with_mapping(t, thread_mapping)`
`requires(t, mapping.new_thread)`	`prefer(t, mapping.new_thread)`	vs	`make_with_mapping(t, new_thread_mapping)`
`requires(t, mapping.other)`	`prefer(t, mapping.other)`	vs	`make_with_mapping(t, other_mapping)`

D2220R0 - *redefine properties in P0443*

usage example from P0443R13

Before

```
// obtain an executor
executor auto ex = ...;

// require the execute operation to block
executor auto blocking_ex =
  std::require(
    ex, execution::blocking.always);

// prefer to execute with a particular
// priority p
executor auto blocking_ex_with_priority =
  std::prefer(
    blocking_ex, execution::priority(p));

// execute my blocking, possibly prioritized
// work
execution::execute(
  blocking_ex_with_priority, work);
```

After

```
// obtain an executor
executor auto ex = ...;

// require the execute operation to block
executor auto blocking_ex =
  execution::make_with_blocking(
    ex, execution::always_blocking);

// prefer to execute with a particular
// priority p
executor auto blocking_ex_with_priority =
  tbd::prefer(execution::make_with_priority,
    blocking_ex, p);

// execute my blocking, possibly prioritized
// work
execution::execute(
  blocking_ex_with_priority, work);
```

D2220R0 - *redefine properties in P0443*

redefine properties in P0443R13

before	after
`context_t`	`get_context()`
`blocking_t`	`get_blocking()`, `make_with_blocking()`
`blocking_t::possibly_t`	`possibly_blocking_t`
`blocking_t::always_t`	`always_blocking_t`
`blocking_t::never_t`	`never_blocking_t`
`relationship_t`	`get_relationship()`, `make_with_relationship()`
`relationship_t ::continuation_t`	`continuation_relationship_t`
`relationship_t::fork_t`	`fork_relationship_t`

D2220R0 - *redefine properties in P0443*

redefine properties in P0443R13

before	after
`outstanding_work_t`	`get_outstanding_work()`, `make_with_outstanding_work()`
`outstanding_work_t::untracked_t`	`untracked_outstanding_work_t`
`outstanding_work_t::tracked_t`	`tracked_outstanding_work_t`
`mapping_t`	`get_mapping()`, `make_with_mapping()`
`mapping_t::thread_t`	`thread_mapping_t`
`mapping_t::new_thread_t`	`new_thread_mapping_t`
`mapping_t::other_t`	`other_mapping_t`
`allocator_t<Allocator>`	`get_allocator()`, `make_with_allocator()`

Any Questions?