define P0443 cpos with tag_invoke

Document #:	D2221R0
Date:	2020-09-11
Project:	Programming Language C++
Audience:	LEWG Library Evolution

Reply-to:	Kirk Shoop <kirk.shoop@gmail.com>

Contents

1 Introduction

[P0443R13] - *A Unified Executors Proposal for C++* defines several Customization-Point-Objects (cpos).

The definition of a cpo is a carefully crafted function object that reserves a new name in the global scope for the purposes of finding customizations via ADL.

This paper will list the changes to [P0443R13] - *executors* needed to define the cpos using `tag_invoke` as defined in *[P1895R0]* - *A general pattern for supporting customisable functions* and remove the global name reservations.

The changes in this paper are implemented in a fork of asio (github)

2 Motivation

2.1 simplify definition of cpos

Cpos can be implemented in terms of `tag_invoke` without the careful crafting required in the definition of `tag_invoke` itself.

2.2 remove global name reservation

The cpos in [P0443R13] - *executors* reserve the following global names: `set_value`, `set_done`, `set_error`, `execute`, `connect`, `start`, `submit`, `schedule`, and `bulk_execute`.

[P1895R0] - *`tag_invoke`* reserves a single global name, `tag_invoke`, as a carefully crafted function object.

Whether a cpo also reserves a name in the global scope becomes a choice. In this paper the global name reservations are removed. Given a good rational, the names of any particular cpo can be reserved in a member-scope, and independently, be reserved in the global scope.

Table 1: definition of a customization

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;
};
```

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

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

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

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

2.3 support generic forwarding

The shared name `tag_invoke` allows generic code to forward calls to nested objects. The support for forwarding becomes a tool that supports generic type-erasure and a variety of composition patterns.

3 Changes

3.1 Modify section *2.1.2 Header <execution> synopsis*

```
  // Customization points:

-  inline namespace unspecified{
-    inline constexpr unspecified set_value = unspecified;
+  inline constexpr set_value_t set_value = set_value_t{};

-    inline constexpr unspecified set_done = unspecified;
+  inline constexpr set_done_t set_done = set_done_t{};

-    inline constexpr unspecified set_error = unspecified;
+  inline constexpr set_error_t set_error = set_error_t{};

-    inline constexpr unspecified execute = unspecified;
+  inline constexpr execute_t execute = execute_t{};

-    inline constexpr unspecified connect = unspecified;
+  inline constexpr connect_t connect = connect_t{};

-    inline constexpr unspecified start = unspecified;
+  inline constexpr start_t start = start_t{};

-    inline constexpr unspecified submit = unspecified;
+  inline constexpr submit_t submit = submit_t{};

-    inline constexpr unspecified schedule = unspecified;
+  inline constexpr schedule_t schedule = schedule_t{};

-    inline constexpr unspecified bulk_execute = unspecified;
+  inline constexpr bulk_execute_t bulk_execute = bulk_execute_t{};
-  }
```

3.2 Modify section *2.2.3.1 execution::set_value*

? `execution::set_value`

where `set_value_t` is an implementation-defined class template equivalent to

```
inline constexpr struct set_value_t {
  template<typename T, typename... VN>
  auto constexpr operator()(T&& t, VN&&... vn) const
    noexcept(noexcept(
      tbd::tag_invoke(*this, (T&&)t, (VN&&)vn...)))
    -> decltype(
```

```
      tbd::tag_invoke(*this, (T&&)t, (VN&&)vn...)) {
    return
      tbd::tag_invoke(*this, (T&&)t, (VN&&)vn...);
  }
} set_value{};
```

The name `execution::set_value` denotes a customization point object. The `set_value(t, vn...)` function passes a pack of values that represent a result to the object `t`.

For some subexpression `t`, let `T` be a type such that `decltype((t))` is `T` and for some subexpression `vn...`, let `VN...` be a type pack such that `decltype((vn))...` is `VN...`. The expression `execution::set_value(t, vn...)` is expression-equivalent to:

— `tbd::tag_invoke(set_value, t, vn...)`, if that expression is valid. If the function selected does not send the value(s) `vn...` to the receiver `t`'s value channel, the program is ill-formed with no diagnostic required.

— Otherwise, `execution::set_value(t, vn...)` is ill-formed.

The name execution::set_value denotes a customization point object. The expression execution::set_value(R, Vs...) for some subexpressions R and Vs... is expression-equivalent to:

R.set_value(Vs...), if that expression is valid. If the function selected does not send the value(s) Vs... to the receiver R's value channel, the program is ill-formed with no diagnostic required.

Otherwise, set_value(R, Vs...), if that expression is valid, with overload resolution performed in a context that includes the declaration

void set_value(); and that does not include a declaration of execution::set_value. If the function selected by overload resolution does not send the value(s) Vs... to the receiver R's value channel, the program is ill-formed with no diagnostic required.

Otherwise, execution::set_value(R, Vs...) is ill-formed.

3.3 Modify section *2.2.3.2* `execution::set_done`

? `execution::set_done`

where `set_done_t` is an implementation-defined class template equivalent to

```
inline constexpr struct set_done_t {
  template<typename T>
  auto constexpr operator()(T&& t) const
    noexcept
    -> decltype(
      tbd::tag_invoke(*this, (T&&)t)) {
    static_assert(
      is_nothrow_tag_invocable_v<set_done_t, T>,
      "set_done() invocation is required to be noexcept.");
    return
      tbd::tag_invoke(*this, (T&&)t);
  }
} set_done{};
```

The name `execution::set_done` denotes a customization point object. The `set_done(t)` function signals a termination signal, with no value or error, to the object `t`.

For some subexpression `t`, let `T` be a type such that `decltype((t))` is `T`. The expression `execution::set_done(t)` is expression-equivalent to:

— `tbd::tag_invoke(set_done, t)`, if that expression is valid. If the function selected does not signal the receiver `t`'s done channel, the program is ill-formed with no diagnostic required.

— Otherwise, `execution::set_done(t, vn...)` is ill-formed.

The name execution::set_done denotes a customization point object. The expression execution::set_done(R) for some subexpression R is expression-equivalent to:

R.set_done(), if that expression is valid. If the function selected does not signal the receiver R's done channel, the program is ill-formed with no diagnostic required.

Otherwise, set_done(R), if that expression is valid, with overload resolution performed in a context that includes the declaration

void set_done(); and that does not include a declaration of execution::set_done. If the function selected by overload resolution does not signal the receiver R's done channel, the program is ill-formed with no diagnostic required.

Otherwise, execution::set_done(R) is ill-formed.

3.4 Modify section *2.2.3.3 `execution::set_error`*

? `execution::set_error`

where `set_error_t` is an implementation-defined class template equivalent to

```
inline constexpr struct set_error_t {
  template<typename T, typename E>
  auto constexpr operator()(T&& t, E&& e) const
    noexcept
    -> decltype(
      tbd::tag_invoke(*this, (T&&)t, (E&&)e)) {
    static_assert(
      is_nothrow_tag_invocable_v<set_error_t, T, E>,
      "set_error(E) invocation is required to be noexcept.");
    return
      tbd::tag_invoke(*this, (T&&)t, (E&&)e);
  }
} set_error{};
```

The name `execution::set_error` denotes a customization point object. The `set_error(t, e)` function passes an error result to the object `t`.

For some subexpression `t`, let `T` be a type such that `decltype((t))` is `T` and for some subexpression `e`, let `E` be a type pack such that `decltype((e))` is `E`. The expression `execution::set_error(t, e)` is expression-equivalent to:

— `tbd::tag_invoke(set_error, t, e)`, if that expression is valid. If the function selected does not send the error `e` to the receiver `t`'s error channel, the program is ill-formed with no diagnostic required.

— Otherwise, `execution::set_error(t, e)` is ill-formed.

The name execution::set_error denotes a customization point object. The expression execution::set_error(R, E) for some subexpressions R and E are expression-equivalent to:

R.set_error(E), if that expression is valid. If the function selected does not send the error E to the receiver R's error channel, the program is ill-formed with no diagnostic required.

Otherwise, set_error(R, E), if that expression is valid, with overload resolution performed in a context that includes the declaration

void set_error(); and that does not include a declaration of execution::set_error. If the function selected by overload resolution does not send the error E to the receiver R's error channel, the program is ill-formed with no diagnostic required.

Otherwise, execution::set_error(R, E) is ill-formed.

3.5 Modify section *2.2.3.4 `execution::execute`*

? `execution::execute`

where `execute_t` is an implementation-defined class template equivalent to

```
inline constexpr struct execute_t {
  template<typename T, typename F>
    requires invocable<remove_cvref_t<F>&> &&
      constructible_from<remove_cvref_t<F>, F> &&
      move_constructible<remove_cvref_t<F>> &&
      tag_invocable<execute_t, T, F>
  auto constexpr operator()(const T& t, F&& f) const
    noexcept(noexcept(
      tbd::tag_invoke(*this, t, (F&&)f)))
    -> decltype(
      tbd::tag_invoke(*this, t, (F&&)f)) {
    return
      tbd::tag_invoke(*this, t, (F&&)f);
  }
  template<typename T, typename F>
    requires invocable<remove_cvref_t<F>&> &&
      constructible_from<remove_cvref_t<F>, F> &&
      move_constructible<remove_cvref_t<F>> &&
      !tag_invocable<execute_t, T, F> &&
      invocable<execution::submit, T, F>
  auto constexpr operator()(const T& t, F&& f) const
    noexcept(noexcept(
      execution::submit(t, as-receiver<F>((F&&)f))))
    -> decltype(
      execution::submit(t, as-receiver<F>((F&&)f))) {
    return
      execution::submit(t, as-receiver<F>((F&&)f));
  }
} execute{};
```

The name `execution::execute` denotes a customization point object.

For some subexpressions `e` and `f`, let `E` be a type such that `decltype((e))` is `E` and let `F` be a type such that `decltype((f))` is `F`. The expression `execution::execute(e, f)` is ill-formed if `F` does not model `invocable`, or if `E` does not model either `executor` or `sender`. Otherwise, it is expression-equivalent to:

— e.execute(f), if that expression is valid. If the function selected does not execute the function object f on the executor e, the program is ill-formed with no diagnostic required.

— Otherwise, execute(e, f), if that expression is valid, with overload resolution performed in a context that includes the declaration

  void execute();

  and that does not include a declaration of execution::execute. If the function selected by overload resolution does not execute the function object f on the executor e, the program is ill-formed with no diagnostic

required.

— `tbd::tag_invoke(execute, e, f)`, if that expression is valid. If the function selected by overload resolution does not execute the function object `f` on the executor `e`, the program is ill-formed with no diagnostic required.

— Otherwise, if `F` is not an instance of *`as-invocable`*`<R, E>` for some type `R`, and `invocable<remove_cvref_t<F>&> && sender_to<E,`*`as-receiver`*`<remove_cvref_t<F>, E>>` is `true`, `execution::submit(e,` *`as-receiver`*`<remove_cvref` where *`as-receiver`* is some implementation-defined class template equivalent to:

```
          template<class F, class>
          struct as-receiver {
            F f_;
-            void set_value() noexcept(is_nothrow_invocable_v<F&>) {
+            void tag_invoke(execution::set_value_t)
+              noexcept(is_nothrow_invocable_v<F&>) {
              invoke(f_);
            }
            template<class E>
-            [[noreturn]] void set_error(E&&) noexcept {
+            [[noreturn]] void tag_invoke(execution::set_error_t, E&&) noexcept {
              terminate();
            }
-            void set_done() noexcept {}
+            void tag_invoke(execution::set_done_t) noexcept {}
          };
```

3.6 Modify section *2.2.3.4 `execution::connect`*

? `execution::connect`

where `connect_t` is an implementation-defined class template equivalent to

```
inline constexpr struct connect_t {
  template<typename T, typename R>
    requires sender<T> && receiver_of<R>
      tag_invocable<connect_t, T, R>
  auto constexpr operator()(const T& t, R&& r) const
    noexcept(noexcept(
      tbd::tag_invoke(*this, t, (R&&)r)))
    -> decltype(
      tbd::tag_invoke(*this, t, (R&&)r)) {
    return
      tbd::tag_invoke(*this, t, (R&&)r);
  }
  template<typename T, typename R>
    requires receiver_of<R> &&
      !tag_invocable<connect_t, T, R> &&
      invocable<execution::execute, T, as-invocable<T, R>>
  auto constexpr operator()(const T& t, R&& r) const
    noexcept(noexcept(
      as-operation<T, R>(t, (R&&)r)))
    -> as-operation<T, R> {
    return as-operation<T, R>(t, (R&&)r);
  }
} connect{};
```

The name `execution::connect` denotes a customization point object. For some subexpressions `s` and `r`, let `S` `decltype((s))` and let `R` be `decltype((r))`. If `R` does not satisfy `receiver`, `execution::connect(s, r)` is ill-formed; otherwise, the expression `execution::connect(s, r)` is expression-equivalent to:

— `s.connect(r)`, if that expression is valid, if its type satisfies `operation_state`, and if `S` satisfies `sender`.

— Otherwise, `connect(s, r)`, if that expression is valid, if its type satisfies `operation_state`, and if `S` satisfies `sender`, with overload resolution performed in a context that includes the declaration

```
void connect();
```

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

— `tbd::tag_invoke(connect, s, r)`, if that expression is valid, if its type satisfies `operation_state`, and if `S` satisfies `sender`.

— Otherwise, *as-operation*`{s, r}`, if

— `r` is not an instance of *as-receiver*`<F, S'>` for some type `F` where `S` and `S'` name the same type ignoring cv and reference qualifiers, and

— `receiver_of<R> && executor-of-impl<remove_cvref_t<S>,`*as-invocable*`<remove_cvref_t<R>, S>>` is true, where *as-operation* is an implementation-defined class equivalent to

```
struct as-operation {
  remove_cvref_t<S> e_;
  remove_cvref_t<R> r_;
  void start() noexcept try {
  void tag_invoke(execution::start_t) noexcept try {
    execution::execute(std::move(e_), as-invocable<remove_cvref_t<R>, S>{r_});
  } catch(...) {
    execution::set_error(std::move(r_), current_exception());
  }
};
```

and *as-invocable* is a class template equivalent to the following:

```
template<class R, class>
struct as-invocable {
  R* r_;
  explicit as-invocable(R& r) noexcept
    : r_(std::addressof(r)) {}
  as-invocable(as-invocable && other) noexcept
    : r_(std::exchange(other.r_, nullptr)) {}
  ~as-invocable() {
    if(r_)
      execution::set_done(std::move(*r_));
  }
  void operator()() & noexcept try {
    execution::set_value(std::move(*r_));
    r_ = nullptr;
  } catch(...) {
    execution::set_error(std::move(*r_), current_exception());
    r_ = nullptr;
  }
};
```

— Otherwise, `execution::connect(s, r)` is ill-formed.

3.7 Modify section *2.2.3.6 execution::start*

? `execution::start`

where `start_t` is an implementation-defined class template equivalent to

```
inline constexpr struct start_t {
  template<typename T>
  auto constexpr operator()(const T& t) const
    noexcept(noexcept(
      tbd::tag_invoke(*this, t)))
    -> decltype(
      tbd::tag_invoke(*this, t)) {
    return
      tbd::tag_invoke(*this, t);
  }
} start{};
```

The name `execution::start` denotes a customization point object. The expression `execution::start(O)` for some lvalue subexpression `O` is expression-equivalent to:

— `O.start()`, if that expression is valid.

— Otherwise, `start(O)`, if that expression is valid, with overload resolution performed in a context that includes the declaration

```
    void start();
```

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

— `tbd::tag_invoke(start, O)`, if that expression is valid.

— Otherwise, `execution::start(O)` is ill-formed.

3.8 Modify section *2.2.3.7 execution::submit*

? `execution::submit`

where `submit_t` is an implementation-defined class template equivalent to

```
inline constexpr struct submit_t {
  template<typename T, typename R>
    requires sender<T> &&
      tag_invocable<submit_t, T, R>
  auto constexpr operator()(const T& t, R&& r) const
    noexcept(noexcept(
      tbd::tag_invoke(*this, t, (R&&)r)))
    -> decltype(
      tbd::tag_invoke(*this, t, (R&&)r)) {
    return
      tbd::tag_invoke(*this, t, (R&&)r);
  }
  template<typename T, typename R>
    requires sender_to<T, R> &&
      !tag_invocable<submit_t, T, R> &&
      invocable<start, connect_result_t<T, submit-receiver<T, R>>>
  auto constexpr operator()(const T& t, R&& r) const
    noexcept(noexcept(submit-state<T, R>(t, (R&&)r)))
```

```
    -> decltype(
      execution::start(new submit-state<T, R>(t, (R&&)r)->state_)) {
    return
      execution::start(new submit-state<T, R>(t, (R&&)r)->state_);
  }
} submit{};
```

The name `execution::submit` denotes a customization point object.

For some subexpressions `s` and `r`, let `S` be `decltype((s))` and let `R` be `decltype((r))`. The expression `execution::submit(s, r)` is ill-formed if `sender_to<S, R>` is not `true`. Otherwise, it is expression-equivalent to:

- `s.submit(r)`, if that expression is valid and `S` models `sender`. If the function selected does not submit the receiver object `r` via the sender `s`, the program is ill-formed with no diagnostic required.
- Otherwise, `submit(s, r)`, if that expression is valid and `S` models `sender`, with overload resolution performed in a context that includes the declaration

  ```
  void submit();
  ```

  and that does not include a declaration of `execution::submit`. If the function selected by overload resolution does not submit the receiver object `r` via the sender `s`, the program is ill-formed with no diagnostic required.
- `tbd::tag_invoke(submit, s, r)`, if that expression is valid. If the function selected by overload resolution does not submit the receiver object `r` via the sender `s`, the program is ill-formed with no diagnostic required
- Otherwise, `execution::start((new`*`submit-state`*`<S, R>{s,r})->state_)`, where *`submit-state`* is an implementation-defined class template equivalent to

```
      template<class S, class R>
      struct submit-state {
        struct submit-receiver {
          submit-state * p_;
          template<class...As>
            requires receiver_of<R, As...>
-           void set_value(As&&... as) && noexcept(is_nothrow_receiver_of_v<R, As...>) {
+           void tag_invoke(execution::set_value_t, As&&... as) &&
+             noexcept(is_nothrow_receiver_of_v<R, As...>) {
            execution::set_value(std::move(p_->r_), (As&&) as...);
            delete p_;
          }
          template<class E>
            requires receiver<R, E>
-           void set_error(E&& e) && noexcept {
+           void tag_invoke(execution::set_error_t, E&& e) && noexcept {
            execution::set_error(std::move(p_->r_), (E&&) e);
            delete p_;
          }
-           void set_done() && noexcept {
+           void tag_invoke(execution::set_done_t) && noexcept {
            execution::set_done(std::move(p_->r_));
            delete p_;
          }
        };
        remove_cvref_t<R> r_;
        connect_result_t<S, submit-receiver> state_;
```

```
      submit-state(S&& s, R&& r)
        : r_((R&&) r)
        , state_(execution::connect((S&&) s, submit-receiver{this})) {}
    };
```

3.9 Modify section *2.2.3.8 execution::schedule*

? `execution::schedule`

where `schedule_t` is an implementation-defined class template equivalent to

```
inline constexpr struct schedule_t {
  template<typename T>
    requires tag_invocable<schedule_t, T>
  auto constexpr operator()(const T& t) const
    noexcept(noexcept(
      tbd::tag_invoke(*this, t))
    -> decltype(
      tbd::tag_invoke(*this, t)) {
    return
      tbd::tag_invoke(*this, t);
  }
  template<typename T>
    requires !tag_invocable<schedule_t, T>
  auto constexpr operator()(const T& t) const
    noexcept(noexcept(
      as-sender<T>(t)))
    -> as-sender<T> {
    return as-sender<T>(t);
  }
} schedule{};
```

The name `execution::schedule` denotes a customization point object. For some subexpression `s`, let `S` be `decltype((s))`. The expression `execution::schedule(s)` is expression-equivalent to:

— `s.schedule()`, if that expression is valid and its type models `sender`.

— Otherwise, `schedule(s)`, if that expression is valid and its type models `sender` with overload resolution performed in a context that includes the declaration

```
    void schedule();
```

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

— `tbd::tag_invoke(schedule, s)`, if that expression is valid and its type models `sender`.

— Otherwise, *as-sender*`<remove_cvref_t<S>>{s}` if `S` satisfies `executor`, where *as-sender* is an implementation-defined class template equivalent to

```
    template<class E>
    struct as-sender {
    private:
      E ex_;
    public:
      template<template<class...> class Tuple, template<class...> class Variant>
        using value_types = Variant<Tuple<>>;
      template<template<class...> class Variant>
```

```
            using error_types = Variant<std::exception_ptr>;
          static constexpr bool sends_done = true;

          explicit as-sender(E e) noexcept
            : ex_((E&&) e) {}
          template<class R>
            requires receiver_of<R>
-          connect_result_t<E, R> connect(R&& r) && {
+          connect_result_t<E, R> tag_invoke(execution::connect_t, R&& r) && {
            return execution::connect((E&&) ex_, (R&&) r);
          }
          template<class R>
            requires receiver_of<R>
-          connect_result_t<const E &, R> connect(R&& r) const & {
+          connect_result_t<const E &, R> tag_invoke(execution::connect_t, R&& r) const & {
            return execution::connect(ex_, (R&&) r);
          }
        };
```

— Otherwise, `execution::schedule(s)` is ill-formed.

3.10 Modify section *2.5.4.5 `static_thread_pool` sender execution functions*

```
class C
{
  public:
    template<template<class...> class Tuple, template<class...> class Variant>
      using value_types = Variant<Tuple<>>;
    template<template<class...> class Variant>
      using error_types = Variant<exception_ptr>;
    static constexpr bool sends_done = true;

    template<receiver_of R>
-      see-below connect(R&& r) const;
+      friend see-below tag_invoke(execution::connect_t, const C&, R&& r);
};
```

```
template<receiver_of R>
-  see-below connect(R&& r) const;
+  friend see-below tag_invoke(execution::connect_t, const C&, R&& r);
```

Effects: When `execution::start` is called on the returned operation state, the receiver `r` is submitted for execution on the `static_thread_pool` according to the ~~the properties established for~~behaviours requested for `*this`. let `e` be an object of type `exception_ptr`; then `static_thread_pool` will evaluate one of `execution::set_value(r)`, `execution::set_error(r, e)`, or `execution::set_done(r)`.

3.11 Modify section *2.5.5.5 `static_thread_pool` executor execution functions*

```
class C
{
  public:
    template<class Function>
```

```
-       void execute(Function&& f) const;
+       friend see-below tag_invoke(execution::execute_t, const C&, Function&& f);

    template<class Function>
-       void bulk_execute(Function&& f, size_t n) const;
+       friend see-below tag_invoke(execution::bulk_execute_t, const C&, Function&& f);
};
```

```
template<class Function>
-  void execute(Function&& f) const;
+  friend see-below tag_invoke(execution::execute_t, const C&, Function&& f);
```

Effects: Submits the function `f` for execution on the `static_thread_pool` according to the ~~the properties established for~~behaviours requested for `*this`. If the submitted function `f` exits via an exception, the `static_thread_pool` invokes `std::terminate()`.

```
template<class Function>
-  void bulk_execute(Function&& f, size_t n) const;
+  friend see-below tag_invoke(execution::bulk_execute_t, const C&, Function&& f);
```

Effects: Submits the function `f` for bulk execution on the `static_thread_pool` according to ~~properties established for~~the behaviours requested for `*this`. If the submitted function `f` exits via an exception, the `static_thread_pool` invokes `std::terminate()`.

3.12 Modify section *1.3 Executors Execute Work*

Authoring executors. Programmers author custom executor types by defining a type with a tag_invoke~~execute~~ function. Consider the implementation of an executor whose tag_invoke~~execute~~ function executes the client's work "inline":

```
struct inline_executor {
  // define execute
  template<class F>
-  void execute(F&& f) const noexcept {
+  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;
};
```

4 References

[P0443R13] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, D. S. Hollman, Lee Howes, Kirk Shoop, Lewis Baker, Eric Niebler. 2020. A Unified Executors Proposal for C++. https://wg21.link/p0443r13

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