P3845R0: Make std::execution's monadic operations naming scheme consistent

Jonathan Müller | WG21 | 2025-11-07

Consistent naming is important

- When we have a naming scheme, we should stick to it.
- It makes it easy to guess what something is done.

Consistent naming is important

- When we have a naming scheme, we should stick to it.
- It makes it easy to guess what something is done.

`std::execution`'s monadic operations don't follow the consistent naming scheme.

We should make them consistent.

Counterpoints

It is too late!

- The design has to settle before we can name it.
- This is a no risk cosmetic change.

Counterpoints

It is too late!

- The design has to settle before we can name it.
- This is a no risk cosmetic change.

People are used to the names!

- Very few people follow committee papers in that detail.
- It first appeared in P1897R3 “Towards C++23 executors: A proposal for an initial set of algorithms” in May 2020.
  - `let_error` was initially called `handle_error`.
- stdexec implemented it in Q3 of 2021.
- Unifxed had `then` and `let` and was published in late 2019.

=> The general public has at *best* 5 years of experience.

`std::execution::then`

- `std::views::transform` operates on a range of `T` and a function `T -> U` and returns a range of `U` by applying the function to each element.

`std::execution::then`

- `std::views::transform` operates on a range of `T` and a function `T -> U` and returns a range of `U` by applying the function to each element.
- `std::optional::transform` operates on an `optional<T>` and a function `T -> U` and returns an `optional<U>` by applying the function if the optional has a value.

std::execution::then

- `std::views::transform` operates on a range of `T` and a function `T -> U` and returns a range of `U` by applying the function to each element.
- `std::optional::transform` operates on an `optional<T>` and a function `T -> U` and returns an `optional<U>` by applying the function if the optional has a value.
- `std::expected::transform` operators on an `expected<T, E>` and a function `T -> U` and returns an `expected<U, E>` by applying the function if the expected has a value.

`std::execution::then`

- `std::views::transform` operates on a range of `T` and a function `T -> U` and returns a range of `U` by applying the function to each element.
- `std::optional::transform` operates on an `optional<T>` and a function `T -> U` and returns an `optional<U>` by applying the function if the optional has a value.
- `std::expected::transform` operators on an `expected<T, E>` and a function `T -> U` and returns an `expected<U, E>` by applying the function if the expected has a value.
- `std::execution::then` operates on a sender of `T` and a function `T -> U` and returns a sender of `U` by applying the function after receiving the value.

`std::execution::then`

- `std::views::transform` operates on a range of `T` and a function `T -> U` and returns a range of `U` by applying the function to each element.
- `std::optional::transform` operates on an `optional<T>` and a function `T -> U` and returns an `optional<U>` by applying the function if the optional has a value.
- `std::expected::transform` operators on an `expected<T, E>` and a function `T -> U` and returns an `expected<U, E>` by applying the function if the expected has a value.
- `std::execution::then` operates on a sender of `T` and a function `T -> U` and returns a sender of `U` by applying the function after receiving the value.

This is the ”map” operation of a monad.

We call it ”transform” everywhere else.

Counterpoints

Unlike ”transform”, ”then” implies temporal sequencing.

- True, but we necessarily have to compute the value before we can apply a transformation!
- Also: we immediately execute `f` when we have the value.

Counterpoints

Unlike ”transform”, ”then” implies temporal sequencing.

- True, but we necessarily have to compute the value before we can apply a transformation!
- Also: we immediately execute `f` when we have the value.

`sndr | ex::transform([](int i)  print(i); )` is weird.

Precedent in `expected.transform([](int i) { print(i); })`.

`std::execution::upon_error`, `std::execution::upon_stopped`

- `std::execution::upon_error` operates on a sender with error `E` and a function `E -> U` and returns a sender that completes successfully with value `U` if the original sender completes with error `E`.

`std::execution::upon_error`, `std::execution::upon_stopped`

- `std::execution::upon_error` operates on a sender with error `E` and a function `E -> U` and returns a sender that completes successfully with value `U` if the original sender completes with error `E`.
- `std::execution::upon_stopped` operates on a sender and a function `() -> U` and returns a sender that completes successfully with value `U` if the original sender was stopped.

`std::execution::upon_error`, `std::execution::upon_stopped`

- `std::execution::upon_error` operates on a sender with error `E` and a function `E -> U` and returns a sender that completes successfully with value `U` if the original sender completes with error `E`.
- `std::execution::upon_stopped` operates on a sender and a function `() -> U` and returns a sender that completes successfully with value `U` if the original sender was stopped.

Not equivalent:

- `std::expected::transform_error` only changes the error and does not turn it into a value.
- `std::expected::or_else` accepts a function `E -> expected<T, F>` and returns an `expected<T, F>`.

`std::execution::upon_error`, `std::execution::upon_stopped`

- `std::execution::upon_error` operates on a sender with error `E` and a function `E -> U` and returns a sender that completes successfully with value `U` if the original sender completes with error `E`.
- `std::execution::upon_stopped` operates on a sender and a function `() -> U` and returns a sender that completes successfully with value `U` if the original sender was stopped.

Not equivalent:

- `std::expected::transform_error` only changes the error and does not turn it into a value.
- `std::expected::or_else` accepts a function `E -> expected<T, F>` and returns an `expected<T, F>`.

Good names, keep them!

Suggested naming scheme for missing operations

Functions `T x E x stopped -> U x F x stopped`

Suggested naming scheme for missing operations

Functions `T x E x stopped -> U x F x stopped`

- `set_value -> set_value : transform(f)`
- `set_error -> set_value : upon_error(f)`
- `set_stopped -> set_value : upon_stopped(f)`

Suggested naming scheme for missing operations

Functions `T x E x stopped -> U x F x stopped`

- `set_value -> set_value : transform(f)`
- `set_error -> set_value : upon_error(f)`
- `set_stopped -> set_value : upon_stopped(f)`

- `set_error -> set_error : transform_error(f)`
- `set_stopped -> set_error : stopped_as_error | transform_error(f)`
- `set_value -> set_error : value_as_error | transform_error(f)` (or `fail | transform_error(f)`)

Suggested naming scheme for missing operations

Functions `T x E x stopped -> U x F x stopped`

- `set_value -> set_value : transform(f)`
- `set_error -> set_value : upon_error(f)`
- `set_stopped -> set_value : upon_stopped(f)`

- `set_error -> set_error : transform_error(f)`
- `set_stopped -> set_error : stopped_as_error | transform_error(f)`
- `set_value -> set_error : value_as_error | transform_error(f)` (or `fail | transform_error(f)`)

- `set_stopped -> set_stopped : transform_stopped(f)`
- `set_value -> set_stopped : value_as_stopped | transform_stopped(f)` (or `stop | transform_stopped(f)`)
- `set_error -> set_stopped : error_as_stopped | transform_stopped(f)` (or `stop_on_error | transform_stopped(f)`)

std::execution::let_value

- `std::optional::and_then` operates on an `optional<T>` and a function `T -> optional<U>` and returns an `optional<U>` by applying the function if the optional has a value.

std::execution::let_value

- `std::optional::and_then` operates on an `optional<T>` and a function `T -> optional<U>` and returns an `optional<U>` by applying the function if the optional has a value.
- `std::expected::and_then` operates on an `expected<T, E>` and a function `T -> expected<U, E>` and returns an `expected<U, E>` by applying the function if the expected has a value.

`std::execution::let_value`

- `std::optional::and_then` operates on an `optional<T>` and a function `T -> optional<U>` and returns an `optional<U>` by applying the function if the optional has a value.
- `std::expected::and_then` operates on an `expected<T, E>` and a function `T -> expected<U, E>` and returns an `expected<U, E>` by applying the function if the expected has a value.
- `std::execution::let_value` operates on a sender of T and a function `T -> sender` and returns a sender by applying the function after receiving the value.

`std::execution::let_value`

- `std::optional::and_then` operates on an `optional<T>` and a function `T -> optional<U>` and returns an `optional<U>` by applying the function if the optional has a value.
- `std::expected::and_then` operates on an `expected<T, E>` and a function `T -> expected<U, E>` and returns an `expected<U, E>` by applying the function if the expected has a value.
- `std::execution::let_value` operates on a sender of T and a function `T -> sender` and returns a sender by applying the function after receiving the value.

This is the ”bind” operation of a monad.

We call it ”and_then” everywhere else.

`std::execution::let_value`

The name is confusing to everybody at first.

If it were `and_then`, people would be able to guess it.

Counterpoint

```
sndr | let_value([](T x) { … })
```

`let_value` also allocates the value and keeps it alive. (“let x = result of sndr; f(x)”)

- Naively reads as `let x = fn(result of sndr);` though…
- Sort of implied because `sndr | let_value([&](T& x) { ... })` compiles
- Is this really the essence of the operation?
- Or is it merely something that has to be done to avoid a pitfall?

Observation: Three use cases for `let_value`

1. Monadic ”bind”, where we don’t care about the lifetime extension part.

```
// extract the input images from the request
just(req) | then(extract_images)
// process images in parallel with bulk.
| let_value([](std::vector<image> imgs) {
    return /* bulk process images */;
})
// transform the resulting 3 images into an HTTP response
| then(imgvec_to_response)
; // done; error and cancellation handling is performed outside
```

Perfectly served by the name `and_then`.

Observation: Three use cases for `let_value`

2. Monadic ”bind”, where we do care about the lifetime extension part.

```
// get a sender when a new request comes
schedule_request_start(the_read_requests_ctx)
// make sure the request is valid; throw if not
| let_value(validate_request)
// process the request in a function
| let_value(handle_request)
```

Less perfectly served by the name `and_then`; the “let” part wouldn’t be highlighted.
… but there is no alternative that doesn’t do let.

Observation: Three use cases for `let_value`

3. We want to declare a variable for the lifetime of our operation.

```
return just(dynamic_buffer{})
  | let_value([handle] (dynamic_buffer& buf) {
      …
  })
  | …;
```

Awkwardly served by `and_then`.

The `just(x) | let_value(fn)` pattern

Status quo: `let(x, fn)`.

```
just(x)
  | let_value([](auto& x) {
      …
  });
```

- The `just(x)` looks awkward.
- x has to be movable.

The `just(x) | let_value(fn)` pattern

Better solution (C++29?): `let(x, fn)`.

```
let(x, [](auto& x) {
    …
})
```

- No awkward `just(x)`.
- We can extend it to support immovable values.

The `just(x) | let_value(fn)` pattern

Better solution (C++29?): `let(x, fn)`.

```
let(x, [](auto& x) {
    …
})
```

- No awkward `just(x)`.
- We can extend it to support immovable values.

And then it doesn't matter that `just(x) | and_then(fn)` is less clear about the "let".

Also: both `let` and `let_value` is confusing.

`std::execution::let_error`, `std::execution::let_stopped`

- `std::optional::or_else` operates on an `optional<T>` and a function `() -> optional<T>` and returns an `optional<T>` by calling the function if the optional is empty.

`std::execution::let_error`, `std::execution::let_stopped`

- `std::optional::or_else` operates on an `optional<T>` and a function `() -> optional<T>` and returns an `optional<T>` by calling the function if the optional is empty.
- `std::expected::or_else` operates on an `expected<T, E>` and a function `E -> expected<T, F>` and returns an `expected<T, F>` by applying the function if the expected has an error.

`std::execution::let_error`, `std::execution::let_stopped`

- `std::optional::or_else` operates on an `optional<T>` and a function `() -> optional<T>` and returns an `optional<T>` by calling the function if the optional is empty.
- `std::expected::or_else` operates on an `expected<T, E>` and a function `E -> expected<T, F>` and returns an `expected<T, F>` by applying the function if the expected has an error.
- `std::execution::let_error` operates on a sender with error `E` and a function `E -> sender` and returns a sender by applying the function if the original sender completes with an error.

`std::execution::let_error`, `std::execution::let_stopped`

- `std::optional::or_else` operates on an `optional<T>` and a function `() -> optional<T>` and returns an `optional<T>` by calling the function if the optional is empty.
- `std::expected::or_else` operates on an `expected<T, E>` and a function `E -> expected<T, F>` and returns an `expected<T, F>` by applying the function if the expected has an error.
- `std::execution::let_error` operates on a sender with error `E` and a function `E -> sender` and returns a sender by applying the function if the original sender completes with an error.
- `std::execution::let_stopped` operates on a sender and a function `() -> sender` and returns a sender by calling the function if the original sender was stopped.

`std::execution::let_error`, `std::execution::let_stopped`

- `std::optional::or_else` operates on an `optional<T>` and a function `() -> optional<T>` and returns an `optional<T>` by calling the function if the optional is empty.
- `std::expected::or_else` operates on an `expected<T, E>` and a function `E -> expected<T, F>` and returns an `expected<T, F>` by applying the function if the expected has an error.
- `std::execution::let_error` operates on a sender with error `E` and a function `E -> sender` and returns a sender by applying the function if the original sender completes with an error.
- `std::execution::let_stopped` operates on a sender and a function `() -> sender` and returns a sender by calling the function if the original sender was stopped.

This is the ”bind” operation on the error channel of a monad.

We call it ”or_else” everywhere else.

Two error channels for senders/receiver

Naming pattern: `or_else` + channel

1. `error_or_else`, `stopped_or_else`
2. `or_error_else`, `or_stopped_else`
3. `or_else_error`, `or_else_stopped`

Two error channels for senders/receiver

Naming pattern: `or_else` + channel

1. `error_or_else`, `stopped_or_else`
2. `or_error_else`, `or_stopped_else`
3. `or_else_error`, `or_else_stopped`

`or_else_error` and `or_else_stopped` are clearly the best!

Counterpoints

`let_error` also allocates the error and keeps it alive. (“let x = error result of sender; f(x)”)

- How often do you actually need that?
- Should it really be the primary feature of the name?

Counterpoints

`let_error` also allocates the error and keeps it alive. (“let x = error result of sender; f(x)”)

- How often do you actually need that?
- Should it really be the primary feature of the name?

Also: `let_stopped` doesn’t do that:

```
let_stopped([](/* no arguments! */) {
    …
})
```

We’re not “let”-ing anything, so it shouldn’t be called “let”!

Proposal

- Rename `std::execution::then` to `std::execution::transform`
- Rename `std::execution::let_value` to `std::execution::and_then`
- Rename `std::execution::let_error` to `std::execution::or_else_error`
- Rename `std::execution::let_stopped` to `std::execution::or_else_stopped`

Proposal

- Rename `std::execution::then` to `std::execution::transform`
- Rename `std::execution::let_value` to `std::execution::and_then`
- Rename `std::execution::let_error` to `std::execution::or_else_error`
- Rename `std::execution::let_stopped` to `std::execution::or_else_stopped`

Optionally, in the future:

- Add something better than `just(x) | and_then(f)`.

Example: Request processing

Status quo:

```
// The whole flow for transforming incoming requests into responses
sender auto snd =
    // get a sender when a new request comes
    schedule_request_start(the_read_requests_ctx)
    // make sure the request is valid; throw if not
    | let_value(validate_request)
    // process the request in a function asynchronously
    | let_value(handle_request)
    // If there are errors transform them into proper responses
    | let_error(error_to_response)
    // If the flow is cancelled, send back a proper response
    | let_stopped(stopped_to_response)
    // write the result back to the client
    | let_value(send_response);
```

Example: Request processing

Renamed:

```
// The whole flow for transforming incoming requests into responses
sender auto snd =
    // get a sender when a new request comes
    schedule_request_start(the_read_requests_ctx)
    // make sure the request is valid; throw if not
    | and_then(validate_request)
    // process the request in a function asynchronously
    | and_then(handle_request)
    // If there are errors transform them into proper responses
    | or_else_error(error_to_response)
    // If the flow is cancelled, send back a proper response
    | or_else_stopped(stopped_to_response)
    // write the result back to the client
    | and_then(send_response);
```

Example: async_read_array

Status quo:

```
sender_of<dynamic_buffer> auto async_read_array(auto handle) {
    return just(dynamic_buffer{})
         | let_value([handle] (dynamic_buffer& buf) {
             return just(std::as_writeable_bytes(std::span(&buf.size, 1)))
                  | async_read(handle)
                  | then([&buf](std::size_t bytes_read) { … })
                  | async_read(handle)
                  | then([&buf](std::size_t bytes_read) { … });
         });
}
```

Example: async_read_array

Renamed:

```
sender_of<dynamic_buffer> auto async_read_array(auto handle) {
    return just(dynamic_buffer{})
         | and_then([handle] (dynamic_buffer& buf) {
             return just(std::as_writeable_bytes(std::span(&buf.size, 1)))
                  | async_read(handle)
                  | transform([&buf](std::size_t bytes_read) { … })
                  | async_read(handle)
                  | transform([&buf](std::size_t bytes_read) { … });
         });
}
```

Example: `async_read_array`

Renamed + future `let` extension:

```
sender_of<dynamic_buffer> auto async_read_array(auto handle) {
    return let(dynamic_buffer{}, [handle](dynamic_buffer& buf) {
         return just(std::as_writeable_bytes(std::span(&buf.size, 1)))
              | async_read(handle)
              | transform([&buf](std::size_t bytes_read) { … })
              | async_read(handle)
              | transform([&buf](std::size_t bytes_read) { … });
     });
}
```