FR-031-319: Defer sender adaptors to C++29

Jonathan Müller | WG21 | 2025-11-03

What `std::execution` provides

- Common vocabulary for asynchronous work (“senders/receivers”)

What `std::execution` provides

- Common vocabulary for asynchronous work (“senders/receivers”)
- Traits and concepts

What `std::execution` provides

- Common vocabulary for asynchronous work ("senders/receivers")
- Traits and concepts
- Pre-defined execution contexts (`ex::run_loop`, `ex::parallel_scheduler`)

What `std::execution` provides

- Common vocabulary for asynchronous work (“senders/receivers”)
- Traits and concepts
- Pre-defined execution contexts (`ex::run_loop`, `ex::parallel_scheduler`)
- A coroutine task type (`ex::task`)

What `std::execution` provides

- Common vocabulary for asynchronous work (“senders/receivers”)
- Traits and concepts
- Pre-defined execution contexts (`ex::run_loop`, `ex::parallel_scheduler`)
- A coroutine task type (`ex::task`)
- Execution scopes for structured concurrency (`ex::counting_scope`)

What `std::execution` provides

- Common vocabulary for asynchronous work (“senders/receivers”)
- Traits and concepts
- Pre-defined execution contexts (`ex::run_loop`, `ex::parallel_scheduler`)
- A coroutine task type (`ex::task`)
- Execution scopes for structured concurrency (`ex::counting_scope`)
- Pre-defined sender factories (`ex::just`, `ex::schedule`)

What `std::execution` provides

- Common vocabulary for asynchronous work (“senders/receivers”)
- Traits and concepts
- Pre-defined execution contexts (`ex::run_loop`, `ex::parallel_scheduler`)
- A coroutine task type (`ex::task`)
- Execution scopes for structured concurrency (`ex::counting_scope`)
- Pre-defined sender factories (`ex::just`, `ex::schedule`)
- Pre-defined sender consumers (`this_thread::sync_wait`, `ex::spawn`)

What `std::execution` provides

- Common vocabulary for asynchronous work ("senders/receivers")
- Traits and concepts
- Pre-defined execution contexts (`ex::run_loop`, `ex::parallel_scheduler`)
- A coroutine task type (`ex::task`)
- Execution scopes for structured concurrency (`ex::counting_scope`)
- Pre-defined sender factories (`ex::just`, `ex::schedule`)
- Pre-defined sender consumers (`this_thread::sync_wait`, `ex::spawn`)
- Pre-defined sender adaptors (`ex::then`, `ex::let_value`, `ex::when_all`)

Most value for library authors: Vocabulary and concepts

Finally, a common vocabulary for async work in C++!

```
ex::sender auto my_async_algorithm()
{
    ex::sender auto data = library_a::read_data_async();
    ex::sender auto processed = library_b::process_data(data, fn);
    ex::sender auto result = library_c::async_write_async_data(processed);
    return result;
}
```

The “iterators” of asynchronous code.

Most value for average users: Usable coroutines

Finally, coroutine support in the standard library!

```cpp
ex::task<Data> my_coroutine()
{
    auto data = co_await library_a::read_data_async();
    fn(data);
    auto result = co_await library_c::async_write_data(data);
    return result;
}
```

The “range-based for loop” of asynchronous code.

Also: Sender adaptors

Generic algorithms on senders.

```
ex::sender auto my_async_algorithm()
{
    return library_a::read_data_async()
      | ex::then(fn)
      | ex::let_value([&](const auto& data) {
          return library_c::async_write_data(data);
      });
}
```

The `std::views` of asynchronous code.

Sender adaptors are useful

- Provide many common operations on senders
- Declarative way of composing senders
- Avoid coroutine overhead

But: Sender adaptors are hard to design

Papers that need to be considered in C++26:

- P3718: Fixing Lazy Sender Algorithm Customization, Again (Eric Niebler)

But: Sender adaptors are hard to design

Papers that need to be considered in C++26:

- P3718: Fixing Lazy Sender Algorithm Customization, Again (Eric Niebler)
- P3826: Fix or Remove Sender Algorithm Customization (Eric Niebler)

But: Sender adaptors are hard to design

Papers that need to be considered in C++26:

- P3718: Fixing Lazy Sender Algorithm Customization, Again (Eric Niebler)
- P3826: Fix or Remove Sender Algorithm Customization (Eric Niebler)
- P3425: Reducing operation-state sizes for subobject child operations (Lewis Baker)

But: Sender adaptors are hard to design

Papers that need to be considered in C++26:

- P3718: Fixing Lazy Sender Algorithm Customization, Again (Eric Niebler)
- P3826: Fix or Remove Sender Algorithm Customization (Eric Niebler)
- P3425: Reducing operation-state sizes for subobject child operations (Lewis Baker)
- P3373: Of Operation States and Their Lifetimes (Robert Leahy)

ISO standards are supposed to be stable

- API can only be changed if it essentially hasn't been implemented yet

ISO standards are supposed to be stable

- API can only be changed if it essentially hasn’t been implemented yet
- ABI can only be changed before implementations ship a version that promises ABI stability

ISO standards are supposed to be stable

- API can only be changed if it essentially hasn’t been implemented yet
- ABI can only be changed before implementations ship a version that promises ABI stability
- DRs are supposed to be for minor issues not fundamental design changes

ISO standards are supposed to be stable

- API can only be changed if it essentially hasn’t been implemented yet
- ABI can only be changed before implementations ship a version that promises ABI stability
- DRs are supposed to be for minor issues not fundamental design changes

It is a process failure if we ship something that we know is not yet right.

The C++ standard does not have an experimental channel.

Are we sure we can get it done in C++26?

- This is the third (?) approach to sender adaptor Customization

Are we sure we can get it done in C++26?

- This is the third (?) approach to sender adaptor Customization
- Reducing operation-state sizes is currently UB and probably requires core language changes

Are we sure we can get it done in C++26?

- This is the third (?) approach to sender adaptor Customization
- Reducing operation-state sizes is currently UB and probably requires core language changes
- Is there consensus on how the lifetime of operation states should work?

Are we sure we can get it done in C++26?

- This is the third (?) approach to sender adaptor Customization
- Reducing operation-state sizes is currently UB and probably requires core language changes
- Is there consensus on how the lifetime of operation states should work?

Do we really have sufficient implementation experience for something designed *this year*?

Adopting it now is risky.

Luckily: Sender adaptors are also somewhat annoying to use

- Slow build times
- Hard to debug code
- Poor error messages

Luckily: Sender adaptors are also somewhat annoying to use

- Slow build times
- Hard to debug code
- Poor error messages

The average programmer barely uses `std::views`!

I don’t foresee widespread adoption of the sender adaptors.

Therefore: Defer sender adaptors to C++29

- We still have the concepts and vocabulary, unblocking standardized networking

Therefore: Defer sender adaptors to C++29

- We still have the concepts and vocabulary, unblocking standardized networking
- We still have usable coroutines

Therefore: Defer sender adaptors to C++29

- We still have the concepts and vocabulary, unblocking standardized networking
- We still have usable coroutines
- We still have execution contexts and structured concurrency

Therefore: Defer sender adaptors to C++29

- We still have the concepts and vocabulary, unblocking standardized networking
- We still have usable coroutines
- We still have execution contexts and structured concurrency
- We still have composition of senders using coroutines or third-party libraries

Therefore: Defer sender adaptors to C++29

- We still have the concepts and vocabulary, unblocking standardized networking
- We still have usable coroutines
- We still have execution contexts and structured concurrency
- We still have composition of senders using coroutines or third-party libraries

We also have three more years to get sender adaptors right.

This is precisely why we switched to a train model.