Asynchronous `main`

Document Number: P4333R0
Date: 2026-09-20
Reply-to: Ian Petersen <ispeters@gmail.com>
Reply-to: Jessica Wong <jesswong2011@gmail.com>
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: EWG, LEWG, SG1, SG17, SG18

Abstract

This paper proposes that `main` should have an alternate, sender factory form.

Background

Coroutines (C++20) and `std::execution` (C++26) bring powerful support for asynchronous programming to C++. Unfortunately there are limitations to how far a programmer can render a C++ program fundamentally asynchronous as the following cannot be a coroutine or a sender factory:

- Constructors,
- Destructors, and
- `main`

Discussion

Completion Signatures

Given that senders project synchronous functions into the asynchronous domain [1], and that therefore the completion signatures of a sender are the projection of the modalities by which a synchronous function returns control to its caller[1], naively projecting `main` into the asynchronous domain produces the following completion signatures:

```
std::execution::completion_signatures<
  std::execution::set_value_t(int),
  std::execution::set_error_t(std::exception_ptr)>
```

[1] Note that the modalities in question are not fully described by the return type of the function because synchronous functions may throw exceptions.

Or, for `noexcept main` [2]:

```
std::execution::completion_signatures<
  std::execution::set_value_t(int)>
```

This might immediately suggest that the completion signatures of the sender returned by a `main` sender factory ought to be one of the above. However there are two further possibilities:

```
std::execution::completion_signatures<
 std::execution::set_error_t(std::execution_ptr)>
```

Since senders need not advertise a successful completion modality. And:

```
std::execution::completion_signatures<>
```

This formulation is allowed by the `std::execution::completion_signatures` class template (i.e. it admits zero arguments) and the exposition-only *`valid-completion-signatures`* concept. It's debatable if the blanket wording in `std::execution` allows it, however (§33.3 [exec.async.ops], emphasis added):

"An asynchronous operation *is a distinct unit of program execution that [...] once started, eventually* ***completes exactly once*** *with a (possibly empty) set of result datums and in exactly one of three dispositions: success, failure, or cancellation [...]"*

The above formulation implicitly assumes the legitimacy of the completion modalities of `main`. This warrants considering each completion disposition supported by `std::execution` separately.

Value

A synchronous program which completes successfully does so by:

- Flowing off the end of main without returning anything, which is, as a special case, handled as if `return EXIT_SUCCESS` were executed, or by
- Returning a value equal to `EXIT_SUCCESS`

Returning any other value indicates the program failed. This despite the fact the return channel is arguably the semantically-discriminated success channel of a synchronous function.

The reason for the above-described confusion is twofold:

- The legacy of C, and
- The overhead of throwing exceptions

`std::execution` not only suffers from neither of those, but also stands to gain significant compositional power from modeling successful completion of the main asynchronous operation as it actually is: Nullary, i.e. `std::execution::set_value_t()`.

To demonstrate the compositional power gained through the above, consider a `main` sender factory which returns a sender representing a task which encapsulates the program's sole responsibility. If we do not accept the above argument, successful completion of this sender is modelled by `std::execution::set_value_t(int)`. Now imagine the author of the program wishes to add a second, concurrent responsibility. `std::execution::when_all` over senders representing each responsibility is a straightforward way to achieve this. But if both of those senders have a `std::execution::set_value_t(int)` completion the `std::execution::when_all` sender will advertise `std::execution::set_value(int, int)`, which isn't a completion signature which is being considered, or which could be made to make sense.

But the above is predicated on the idea that value completion of the main asynchronous operation is properly nullary. What about, for example `std::execution::set_value(rcvr, EXIT_FAILURE)` rather than `std::execution::set_value(rcvr, EXIT_SUCCESS)`? But the former isn't a successful completion, it's an erroneous completion, and abusing the completion dispositions of `std::execution` in this way has negative consequences for composability.

Consider the above `std::execution::when_all` example. If we suppose its children send `std::execution::set_value_t(int)`, and that the success channel is abused (as it is in the synchronous domain) to transmit failures, what happens when a child fails? It completes "successfully" sending a value which contextually reclassifies the completion as an error. But `std::execution::when_all` is a generic algorithm, it doesn't have this contextual information, and therefore when a child fails in this way `std::execution::when_all` allows the other children to continue running, which is unlikely to be what was expected or intended by the author.

Error

Synchronous functions have one way to report failure in a semantically-discriminated manner: Throwing an exception. Not only is this poorly differentiated (there's no program-observable information about what types are thrown) but it has overhead which gives programmers pause when using it. Hence, along with the legacy of C, the proliferation of abuses of the return channel to report error information (rather than leveraging a semantically-discriminated channel).

`std::execution` does not suffer from the above. There is no legacy of C as `std::execution` bootstraps an entirely new, separate, asynchronous function call protocol. Senders can advertise exactly the types they send to represent errors by adding completion signatures of the form `std::execution::set_error_t(E)` to the instantiation of `std::execution::completion_signatures` they return from `get_completion_signatures`.

Unlike in the synchronous domain this has no performance overhead as compared to value completion.

Synchronous `main` uses non-`EXIT_SUCCESS` return values to indicate failure. Therefore asynchronous main should be allowed to expose `std::execution::set_error_t(int)`. Unfortunately the design of synchronous `main` doesn't just use the return channel to indicate errors, it also has a sentinel value (i.e. `EXIT_SUCCESS`). This raises the question of what, if any, meaning or behavior `std::execution::set_error(rcvr, EXIT_SUCCESS)` ought to have or exhibit. There are three immediately-obvious options:

- The program ends with `EXIT_SUCCESS`,
- The program ends in error, or
- Undefined behavior

The last option has some platonic appeal since the situation should never happen, however it is user hostile and doesn't really offer a compelling upside. Optimizing away one branch inside the error completion path of the entire program doesn't seem worth the trade off of potential UB (especially in our more safety-conscious age).

The first option seems intuitive, but belies the semantic meaning of the error channel in the same way `std::execution::set_value_t(int)` would belie the meaning of the value channel.

Stopped

Because there is no synchronous analogue of the stopped completion disposition, consideration thereof is less anchored to synchronous analogues than with the other two completion dispositions.

That said there has been thought put into what stopped means. Particularly repudiation of the idea (implicit in, for example, `ECANCELED`) that stopped is an error [3]. Under this model stopped is "serendipitous-success." But this requires additional thinking about what, exactly, it is that stopped means.

When a function, asynchronous or otherwise, completes successfully that means that its postconditions have been established. When a function completes in error that means that its postconditions have not been established because there was some impediment to the establishment thereof. When an asynchronous operation completes with stopped this means that its postconditions have not been established because the operation didn't even try to establish them. The operation observed something in its environment (e.g. a stop request) and concluded therefrom that accomplishment of its goals was no longer necessary or desired.

But what does the above mean for completion of an entire program? In principle there could be operating systems which provide a differentiated stopped exit pathway from programs, but the

authors are unaware of any such system. All systems the authors are aware of have the usual dichotomous conception of a program's exit: Success or failure encapsulated in an integer.

Given that dichotomy it seems clear from the above that stopped is success, albeit "serendipitous," but is this user-friendly behavior? Under composition operations can gain stopped completions for reasons that are subtle or unclear at first glance [4]. Allowing those to lead to the program ending with success, but without accomplishing its actual goals, seems as though it provides an easy way for users to write bugs.

Foreground Thread

Unlike synchronous functions, asynchronous functions don't necessarily complete on the thread on which they were initiated. This means that the asynchronous operation resulting from connecting and starting the sender returned by a `main` sender factory could complete on a different thread from the main thread of execution.

The above is typically mediated by causing the foreground thread to wait for the asynchronous task to complete (e.g. `std::this_thread::sync_wait`). However in the case of `main` there is, in some sense, nothing to wait. The main thread of execution could simply end (via, for example, `pthread_exit(nullptr)`) and completion of the asynchronous operation could call `std::exit` with whatever a synchronous `main` would've returned.

The above is plausible, and has been implemented by one of the authors of this paper. However, ending the main thread of execution without ending the program, while possible, is heterodox. Moreover, calling std::exit when handling a completion signal does not unwind the stack, which can lead to the program ending without allowing destructors to run for variables with automatic storage duration.

The above is therefore not proposed.

Parameters of `main`

`main` receives as arguments at least:

- The number of arguments plus one (i.e. `argc`)
- The invocation and arguments (i.e. `argv`)

This formulation is very much a result of the legacy of C. A modern, C++-native realization of the signature of `main` would accept a sole parameter of type `std::span<const char* const>` or `std::span<std::string_view>`.

Changing the signature of `main` has nothing to do with allowing `main` to be asynchronous. Therefore it is not proposed.

Proposed Design

`std::execution::sender auto main(int, const char* const*)` should be an acceptable signature for `main` (note this implies that `main` can be a coroutine). When such a signature is used the compiler should provide scaffolding that:

1. Invokes `main`
2. Adapts the returned sender such that it generates the exit code of the program, that is:
    - Nullary value completions are coalesced to `std::execution::set_value(..., EXIT_SUCCESS)`
    - Integer error completions are coalesced to `std::execution::set_value(..., i)` where `i` is the integer sent on the error channel
3. Invokes `std::this_thread::sync_wait` with the adapted sender
4. Ends the program with the exit code obtained by dereferencing the returned `std::optional<int>`

Examples

Hello World

```
std::execution::sender auto main(int, const char* const*) {
  return
    std::execution::just() |
    std::execution::then([] { std::cout << "Hello world!\n"; });
}
```

Scheduling Multiple Work Items

```
std::execution::sender auto main(int, const char* const*) {
  return
    std::execution::read_env(std::execution::get_scheduler) |
    std::execution::let_value([](std::execution::scheduler auto sch) {
      return std::execution::when_all(
        std::execution::schedule(sch) | std::execution::then([] {
          std::cout << "Hello world!\n";
        }),
        std::execution::schedule(sch) | std::execution::then([] {
          std::cout << "Running on run_loop on main thread!\n";
        }));
```

```
    });
}
```

Failure

```
std::execution::sender auto main(int, const char* const*) {
  return std::execution::just_error(EXIT_FAILURE);
}
```

Proposed Wording

[basic.start.main]

A program shall contain exactly one function called `main` that belongs to the global scope. Executing a program starts a main thread of execution ([intro.multithread], [thread.threads]) in which the `main` function is invoked. It is implementation-defined whether a program in a freestanding environment is required to define a `main` function.

[*Note 1*: In a freestanding environment, startup and termination is implementation-defined; startup contains the execution of constructors for non-local objects with static storage duration; termination contains the execution of destructors for objects with static storage duration. — *end note*]

An implementation shall not predefine the `main` function. Its type shall have C++ language linkage and it shall have a declared return type of:

- `int`,
- a *placeholder-type-specifier* of the form *type-constraint*$_{opt}$ `auto`, or
- a type that satisfies the constraints of `execution::main_sender` ([exec.snd.concepts])

but otherwise its type is implementation-defined. If the declared return type of `main` contains a placeholder type, the return type deduced for `main` shall be:

- `int`, or
- a type that satisfies the constraints of `execution::main_sender`

It is implementation-defined whether a program in a freestanding environment may define a `main` function whose return type satisfies the constraints of `execution::main_sender`.

For each declared return type `R` that the implementation is required to permit for `main`, a~~A~~n implementation shall allow both

- an “optionally `noexcept` function of `()` returning ~~`int`~~`R`” and

- an “optionally `noexcept` function of `(int, pointer to pointer to char)` returning ~~`int`~~R”

as the type of `main` ([dcl.fct]). In the latter form, for purposes of exposition, the first function parameter is called `argc` and the second function parameter is called `argv`, where `argc` shall be the number of arguments passed to the program from the environment in which the program is run. If `argc` is nonzero these arguments shall be supplied in `argv[0]` through `argv[argc - 1]` as pointers to the initial characters of null-terminated multibyte strings (NTMBSS) ([multibyte.strings]) and `argv[0]` shall be the pointer to the initial character of an NTMBS that represents the name used to invoke the program or `""`. The value of `argc` shall be non-negative. The value of `argv[argc]` shall be `0`.

Recommended practice: Any further (optional) parameters should be added after `argv`.

If the return type of `main` satisfies the constraints of `execution::main_sender`, let *`main-impl`*`()` denote an expression that invokes the program-defined `main` function with the arguments supplied by the implementation. The program is treated as if the program-defined `main` function were replaced by the following function:

```
int main() {
  return get<0>(*this_thread::sync_wait(
    execution::upon_error(
      execution::then(
        main-impl(),
        []<class... Int>(Int... args) noexcept {
          return (0 + ... + args);
        }),
      []<class E>(E e) noexcept(is_same_v<E, int>) -> int {
        if constexpr (is_same_v<E, int>) {
          return e;
        } else {
          rethrow_exception(std::move(e));
        }
      })));
}
```

The function `main` shall not be named by an expression. The linkage ([basic.link]) of `main` is implementation-defined. A program that defines `main` as deleted or that declares `main` to be `inline`, `static`, `constexpr`, or `consteval` is ill-formed. ~~The function `main` shall not be a coroutine ([dcl.fct.def.coroutine]).~~ The `main` function shall not be declared with a *linkage-specification* ([dcl.link]) other than `"C++"`. A program that declares

- a variable `main` that belongs to the global scope, or
- a function `main` that belongs to the global scope and is attached to a named module, or
- a function template `main` that belongs to the global scope, or

- an entity named `main` with C language linkage (in any namespace)

is ill-formed. The name `main` is not otherwise reserved.

[...]

[exec.snd.concepts]

[...]

```
namespace std::execution {

  [...]

  template<class Sndr, class Rcvr>
    concept sender-to =                                                    // exposition only
      sender_in<Sndr, env_of_t<Rcvr>> &&
      receiver-of<Rcvr, completion_signatures_of_t<Sndr, env_of_t<Rcvr>>> &&
      requires (Sndr&& sndr, Rcvr&& rcvr) {
        connect(std::forward<Sndr>(sndr), std::forward<Rcvr>(rcvr));
      };

  struct main-receiver {                                                   // exposition only
    using receiver_concept = receiver_tag;

    void set_value() && noexcept;
    void set_value(int) && noexcept;
    void set_error(int) && noexcept;
    void set_error(exception_ptr) && noexcept;
  };

  template<class Sndr>
  concept main_sender = sender-to<Sndr, main-receiver>;
}
```

[...]

Suggested Polls

`set_value_t(int)`

Asynchronous main should allow `set_value_t(int)` completions

Disallowing this would avoid the semantic ambiguity of “value” completions that don’t send `EXIT_SUCCESS`.

`set_error_t(int)`

Asynchronous main should require that an integer sent as an error not be `EXIT_SUCCESS`

This would avoid the semantic ambiguity of “error” completions that actually indicate success.

Implementation Experience

This has been implemented against Nvidia’s stdexec. Users define `async_main` instead of `main`, and then `#include <exec/async_main.hpp>` (which defines `main`).

References

[1] K. Shoop. async-object - aka async-RAII P2849R0
[2] R. Leahy. Throwing Violation Handlers Are Post Hoc Library Design P4254R0
[3] K. Shoop et al. Cancellation is serendipitous-success P1677R2
[4] R. Leahy. when_all Oughtn't Hallucinate set_stopped P4269R04