Document: P2203R0
Authors: Tomasz Kamiński <tomaszkam@gmail.com>
Ryan McDougall <mcdougall.ryan@gmail.com>
Conor Hoekstra <hoekstra101@hotmail.com>
Bryan St. Amour <bryan@bryanstamour.com>
Audience: LEWG
Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C+

LEWG Executors Customization Point Report

Abstract

In order to make review of the proposal tractable, P0443 was broken into components and reviewed offline with the assistance of primary authors. This report is the result of investigations into Section 2.2.3: Customization Points.

Our impressions are summarized, and open questions are presented for LEWG consideration.

Scope

```
2.2.3 Customization points
2.2.3.1 execution::set_value
2.2.3.2 execution::set_done
2.2.3.3 execution::set_error
2.2.3.4 execution::execute
2.2.3.5 execution::connect
2.2.3.6 execution::start
2.2.3.7 execution::submit
2.2.3.8 execution::schedule
2.2.3.9 execution::bulk_execute
```

The report should address:

1. A summary of what you were asked to look at.
2. Smaller/non-controversial changes you agreed on.
3. Open questions/decisions Library Evolution needs to make.

Method

Aside from reviewing the paper, the reviewers were tasked with implementing P1895 tag_invoke: A general pattern for supporting customisable functions, which is understood to be the authors' preferred solution to implementing the customization points in P0443, as a means of familiarizing ourselves with customization points.

Summary

Two themes emerged from discussion:

1. Substantial increase in (possibly globally reserved) customization points.
2. The semantics of fallback (uncustomized) implementations.

Recapitulation

All errors are mine. Consult source material for detail and correctness.

Arthur O'Dwyer provides a helpful definition of customization points in his blog post:

[A] customization point is a place where you-the-library-programmer are deliberately delegating the behavior of some operation to your user-programmer [, maybe] overriding a behavior that you would otherwise provide by default. [...]

Every well-designed customization point has two pieces:

- *A, the piece the user is required to specialize; and*
- *B, the piece the user is required to invoke[.]*

The standard provides a helpful note on the ramifications of customization point objects:

Many of the customization point objects in the library evaluate function call expressions with an unqualified name which results in a call to a program-defined function found by argument dependent name lookup.

This may result in either needing to globally reserve such names, requiring intended overloads to always be better matches, or requiring unintended overloads to be explicitly "poison pilled'.

Customizing Receivers

```
R receiver; // R models std::execution::receiver_of<R,Ts...>
```

Then R should support the following "completion signal channels" with the following contract:

1. None of a receiver's completion-signal operations shall be invoked before execution has started.
2. Once an execution has started, exactly one of the receiver's completion-signal operations shall complete non-exceptionally.
3. If `set_value` exits with an exception, it is still valid to call `set_error` or `set_done` on the receiver.

Set Value

`std::execution::set_value(receiver, ts...);`
Customized to deliver result values to the receiver, not unlike `std::promise::set_value`.

Default behavior is expression equivalent to:

```
1. R.set_value(Ts…);
2. set_value(R, Ts…);
3. // ill-formed
```

Set Error

`std::execution::set_error(receiver, error);`
Customized to deliver error values to the receiver, such as `std::error_code` or `std::exception_ptr`.

Default behavior is expression equivalent to:

```
1. R.set_error(E);
2. set_value(R, E);
3. // ill-formed
```

Set Done

`std::execution::set_done(receiver);`
Customized to notify the receiver that work is completed without result value or error.

Default behavior is expression equivalent to:

```
1. R.set_done();
2. set_done(R);
3. // ill-formed
```

Customizing Senders

`S sender; // models std::execution::sender_to<S,R>`
Senders can begin their execution via two equivalent ways:

1. Connect + Start
2. Submit

Connect

`auto op = std::execution::connect(sender, receiver);`
Customized to notify the sender of its intended receiver.

Default behavior is expression equivalent to:

```
1. S.connect(R);
2. connect(S,R);
```

```
3. // wrapper of S and R chained execute/set_value calls
4. // ill-formed
```

Start

`std::execution::start(op);`
Customized to notify the sender to begin its execution.

Default behavior of connect is expression equivalent to:

```
1. O.start();
2. start(O);
3. // ill-formed
```

Submit

`std::execution::submit(sender, receiver);`
Is simply a fused connect and start.

Default behavior of connect is expression equivalent to:

```
1. S.submit(R);
2. submit(S,R);
3. // starts a wrapper of S and R chained execute/set_value
   calls
4. // ill-formed
```

Customizing Executors

```
E executor; // models std::execution::executor<E>
F f;        // models std::invocable<F>
```

Execute

`execution::execute(executor, f);`
Runs f eagerly on the executor's execution agent.

Default behavior is expression equivalent to:

```
1. E.execute(F);
2. execute(E,F);
3. // submit E as sender, and F as receiver where set_value
   invokes and set_error terminates
4. // ill-formed
```

Customizing Schedulers

```
S scheduler; // models std::execution::scheduler<S>
```

Schedule

`auto sender = execution::schedule(scheduler);`
Constructs senders for the scheduler's execution agent.

Default behavior is expression equivalent to:

```
1. S.schedule();
2. schedule(S);
3. // wrapper of S that models executor
4. // ill-formed
```

Open Questions

1. CPOs may "land grab" common names
    a. Does LEWG want to proceed with P1895 `tag_invoke` which reserves only one name globally, and uses private friends for ADL?
    b. Does LEWG want to reduce the number of CPOs?
    c. Does LEWG want to rename them to be less generic?
    d. It is still open if member functions (non-ADL, ie. `Foo::set_value`) are reserved for special meaning in a CPO
2. Relationship Hierarchy of Scheduler and Executor
    a. Should terminating behavior require an explicit opt-in (ie. `submit(schedule(sched), as_terminating_receiver(work))`)?
    b. Should all implicit behavior (in connect, submit, execute, schedule CPOs) require an explicit opt-in (ie. not use the `as-` adapters)?

Specifically when executing something that wants to be scheduled: `schedule(exec, work);` is fine because executor has strictly less communication channels. However `execute(sched, work);` is problematic because it wants to return value/error/done, which the current CPO by default calls terminate.

For example:

Program Version 1

```
deadline_executor exec(ctx, 20s); // start task within 20s
execute(exec, work);              // drop work otherwise
```

Program Version 2

```
deadline_scheduler sched(ctx, 20s); // now calls set_error
execute(sched, work);               // now calls terminate
// unless set_error is overloaded for decltype(work)
```

More discussion can be found here:
https://github.com/atomgalaxy/review-executor-sendrecv/issues/4