Document Number: N4046
Date: 2014-05-26
Reply To Christopher Kohlhoff <chris@kohlhoff.com>

Executors and Asynchronous Operations

1 Introduction

N3785 *Executors and schedulers, revision 3* describes a framework for executors. Unfortunately, this framework is built around some deliberate design choices that make it unsuited to the execution of fine grained tasks and, in particular, asynchronous operations. Primary among these choices is the use of abstract base classes and type erasure, but it is not the only such issue. The sum effect of these choices is that the framework is unable to exploit the full potential of the C++ language.

In this document, we will look at an alternative executors design that uses a lightweight, template-based policy approach. To describe the approach in a nutshell:

> *An executor is to function execution as an allocator is to allocation.*

This proposal builds on the type traits described in N4045 *Library Foundations for Asynchronous Operations, Revision 2* to outline a design that unifies the following areas:

- Executors and schedulers.
- Resumable functions or coroutines.
- A model for asynchronous operations.
- A flexible alternative to `std::async()`.

In doing so, it takes concepts from Boost.Asio, many of which have been unchanged since its inclusion in Boost, and repackages them in a way that is more suited to C++14 language facilities.

2 Reference implementation

A reference implementation of the proposed library can be found at:

```
http://github.com/chriskohlhoff/executors
```

3 Design issues in N3785

Before we begin, let us briefly review some of the design choices of N3785 executors.

3.1 Use of inheritance and polymorphism

> *The interface is based on inheritance and polymorphism, rather than on templates, for two reasons. First, executors are often passed as function arguments, often to functions that have no other reason to be templates, so this makes it possible to change executor type without code restructuring. Second, a non-template design makes it possible to pass executors across a binary interface: a precompiled library can export a function one of whose parameters is an executor.*

As we will see in this proposal, a template-based design does not preclude the inclusion of a runtime polymorphic wrapper. Such a wrapper still allows users to write non-template code

for use with executors, and makes it possible to pass executors across a binary interface. On the other hand, it is not possible to undo the performance impact of a type-erased interface.

> *The cost of an additional virtual dispatch is almost certainly negligible compared to the other operations involved.*

This claim might be true when passing coarse-grained tasks across threads. However, use cases for executors are not limited to this. As outlined in N4045, composition of asynchronous operations may entail multiple layers of abstraction. The ability to leverage function inlining is a key part of delivering a low abstraction penalty, but the compiler is unable to see through a virtual interface.

3.1.1 Use of std::function<void()>

> *Most fundamentally, of course, executor is an abstract base class and add() is a virtual member function, and function templates can't be virtual. Another reason is that a template parameter would complicate the interface without adding any real generality. In the end an executor class is going to need some kind of type erasure to handle all the different kinds of function objects with void() signature, and that's exactly what std::function already does.*

By forcing type erasure at the executor interface, an executor implementer is denied the opportunity to choose a more appropriate form of type erasure. For example, an implementer may wish to store pending work items in a linked list. With a template-based approach, the function object and the "next pointer" can be stored in the same object. This is not possible if type erasure has already occurred[1].

> *One theoretical advantage of a template-based interface is that the executor might sometimes decide to execute the work item inline, rather than enqueueing it for asynchronous, in which case it could avoid the expense of converting it to a closure. In practice this would be very difficult, however: the executor would somehow have to know which work items would execute quickly enough for this to be worthwhile.*

There is a key use case for wanting to execute work items inline: delivering the result of an asynchronous operation, possibly across multiple layers of abstraction. Rather than relying on the executor to "somehow have to know", we can allow the user to choose. This is the approach taken in this proposal.

Finally, another disadvantage of `std::function` is that it prevents the use of move-only function objects.

Of course, as N3785 states, the need for a type-erased function object is itself a consequence of the use of inheritance and polymorphism.

3.2 Scheduled work

> *There are several important design decisions involving that time-based functionality. First: how do we handle executors that aren't able to provide it? The issue is that add_at and add_after involve significant implementation complexity. In Microsoft's experience it's important to allow very simple and lightweight executor classes that don't need such complicated functionality.*

N3785 couples timer-based operations to executors via the `scheduled_executor` base class. This proposal avoids this coupling by distinguishing between the executor as a lightweight policy object, and an execution context where the "implementation complexity" of timers can

[1] Unless a small-object optimisation is employed by `std::function`, but this is not guaranteed.

be housed in a reusable way. Thus, timer operations are independent of executor types, and can be used with any executor.

> *Second, how should we specify time? [..] Some standard functionality, like sleep_until and sleep_for, is templatized to deal with arbitrary duration and time_point specializations. That's not an option for an executor library that uses virtual functions, however, since virtual member functions can't be function templates. There are a number of possible options:*
>
> 1. *Redesign the library to make executor a concept rather than an abstract base class. We believe that this would be invention rather than existing practice, and that it would make the library more complicated, and less convenient for users, for little gain.*
> 2. *Make executor a class template, parameterized on the clock. As discussed above, we believe that a template-based design would be less convenient than one based on inheritance and runtime polymorphism.*
> 3. *Pick a single clock and use its duration and time_point.*
>
> *We chose the last of those options, largely for simplicity.*

Unfortunately, N3785 chooses `system_clock` as that single clock. As the system clock is susceptible to clock changes it may be inappropriate for use cases that require a periodic timer. In those instances, `steady_timer` is the better choice.

In any case, by decoupling timers from executors, this proposal provides timer operations that are indeed templates, and can work with arbitrary clock types.

3.3 Exception handling

> *A more interesting question is what happens if a user closure throws an exception. The exception will in general be thrown by a different thread than the one that added the closure or the thread that started the executor, and may be far separated from it in time and in code location. As such, unhandled exceptions are considered a program error because they are difficult to signal to the caller. The decision we made is that an exception from a closure is ill-formed and the implementation must call std::terminate.*

Rather than apply a blanket rule, this proposal includes exception handling as part of an executor's policy. While `std::terminate` is likely to be the best choice for unhandled exceptions inside a thread pool, users may prefer greater control when using something like `loop_executor`. For example, the approach taken by Boost.Asio's `io_service` and this proposal's `loop_scheduler` is to allow exceptions to escape from the "event loop", where the user can handle them as appropriate.

3.4 Inline executors and the single add function

> *[...] Inline executors, which execute inline to the thread which calls add(). This has no queuing and behaves like a normal executor, but always uses the caller's thread to execute.*

Since the executor interface is defined by an abstract base class, code that calls the `add()` function has to assume that the underlying executor may execute the function object inline. As a consequence, extra care must be taken in situations such as:

- If using mutexes, avoiding calls to `add()` while holding the lock.
- If iterating over a container and calling `add()` for each element, ensuring the added function object cannot invalidate the iterators.

This proposal instead makes an explicit distinction between operations that can execute inline, and those that cannot. The library user is then able to choose the appropriate operation for their use case.

4 Library overview

The central concept of this library is the *executor* as a policy. An executor embodies a set of rules about where, when and how to run a function object. For example:

Type of executor	**Where, when and how**
System	Any thread in the process.
Thread pool	Any thread in the pool, and nowhere else.
Strand	Not concurrent with any other function object sharing the strand, and in FIFO order.
Future / Promise	Any thread. Captures any exception that is thrown by the function object and stores it in the promise.

Executors are ultimately defined by a set of type requirements, so the set of executors isn't limited to those listed here. Like allocators, library users can develop custom executor types to implement their own rules.

To submit a function object to an executor, we can choose from one of three fundamental operations: *dispatch*, *post* and *defer*. These operations differ in the eagerness with which they run the submitted function.

A dispatch operation is the most eager, and used when we want to run a function object according to an executor's rules, but in the cheapest way available:

```
void f1()
{
  std::cout << "Hello, world!\n";
}

// ...

dispatch(ex, f1);
```

By performing a dispatch operation, we are giving the executor `ex` the option of having `dispatch()` run the submitted function object before it returns. Whether an executor does this depends on its rules:

Type of executor	**Behaviour of dispatch**
System	Always runs the function object before returning from `dispatch()`.
Thread pool	If we are inside the thread pool, runs the function object before returning from `dispatch()`. Otherwise, adds to the thread pool's work queue.
Strand	If we are inside the strand, or if the strand queue is empty, runs the function object before returning from `dispatch()`. Otherwise, adds to the strand's work queue.

Future / Promise	Wraps the function object in a try/catch block, and runs it before returning from `dispatch()`.

The consequence of this is that, if the executor's rules allow it, the compiler is able to inline the function call.

A post operation, on the other hand, is not permitted to run the function object itself.

```
post(ex, f1);
```

A posted function is scheduled for execution as soon as possible, but according to the rules of the executor:

Type of executor	**Behaviour of post**
System	Adds the function object to an unspecified system-wide thread pool's work queue.
Thread pool	Adds the function object to the thread pool's work queue.
Strand	Adds the function object to the strand's work queue.
Future / Promise	Wraps the function object in a try/catch block, and delegates the post operation to the system executor.

Finally, the defer operation is the least eager of the three.

```
defer(ex, f1);
```

A defer operation is similar to a post operation, except that it implies a relationship between the caller and the function object being submitted. It is intended for use when submitting a function object that represents a continuation of the caller.

Type of executor	**Behaviour of defer**
System	If the caller is executing within the system-wide thread pool, saves the function object to a thread-local queue. Once control returns to the system thread pool, the function object is scheduled for execution as soon as possible. If the caller is not inside the system thread pool, behaves as a post operation.
Thread pool	If the caller is executing within the thread pool, saves the function object to a thread-local queue. Once control returns to the thread pool, the function object is scheduled for execution as soon as possible. If the caller is not inside the specified thread pool, behaves as a post operation.
Strand	Adds the function object to the strand's work queue.
Future / Promise	Wraps the function object in a try/catch block, and delegates to the system executor for deferral.

4.1 Posting functions to a thread pool

As a simple example, let us consider how to implement the Active Object design pattern using the library. In the Active Object pattern, all operations associated with an object are run on its own private thread.

```
class bank_account
{
  int balance_ = 0;
  std::experimental::thread_pool pool_{1};
  mutable std::experimental::thread_pool::executor_type ex_
    = pool_.get_executor();

public:
  void deposit(int amount)
  {
    std::experimental::post(ex_, [=]
      {
        balance_ += amount;
      });
  }

  void withdraw(int amount)
  {
    std::experimental::post(ex_, [=]
      {
        if (balance_ >= amount)
          balance_ -= amount;
      });
  }
};
```

First, we create a private thread pool with a single thread:

```
std::experimental::thread_pool pool_{1};
```

A thread pool is an example of an *execution context*. An execution context represents a place where function objects will be executed. This is distinct from an executor, which, as an embodiment of a set of rules, is intended to be a lightweight policy object that is cheap to copy and wrap for further adaptation.

Therefore, to inject function objects into the thread pool, we must obtain an executor for it using the member function `get_executor()`:

```
mutable std::experimental::thread_pool::executor_type ex_ =
  pool_.get_executor();
```

To add the function to the queue, we then use a post operation:

```
std::experimental::post(ex_, [=]
  {
    if (balance_ >= amount)
      balance_ -= amount;
  });
```

4.2 Waiting for function completion

When implementing the Active Object pattern, we will normally want to wait for the operation to complete. To do this we can re-implement our `bank_account` member functions to pass an additional *completion token* to the free function `post()`. A completion token specifies how we want to be notified when the function finishes[2]. For example:

```
void withdraw(int amount)
{
  std::future<void> fut = std::experimental::post(ex_, [=]
    {
      if (balance_ >= amount)
        balance_ -= amount;
    },
    std::experimental::use_future);
  fut.get();
}
```

Here, the `use_future` completion token is specified. When passed the `use_future` token, the free function `post()` returns the result via a `std::future` object.

Other types of completion token include plain function objects (used as callbacks), resumable functions or coroutines, and even user-defined types. If we want our active object to accept any type of completion token, we simply change the member functions to accept the token as a template parameter:

```
template <class CompletionToken>
auto withdraw(int amount, CompletionToken&& token)
{
  return std::experimental::post(ex_, [=]
    {
      if (balance_ >= amount)
        balance_ -= amount;
    },
    std::forward<CompletionToken>(token));
}
```

The caller of this function can now choose how to receive the result of the operation, as opposed to having a single strategy hard-coded in the `bank_account` implementation. For example, the caller could choose to receive the result via a `std::future`:

```
bank_account acct;
// ...
std::future<void> fut = acct.withdraw(10, std::experimental::use_future);
fut.get();
```

or callback:

[2] See N4045 for a complete description of completion tokens.

```
acct.withdraw(10, []{ std::cout << "withdraw complete\n"; });
```

or any other type that meets the completion token requirements. This approach also works for functions that return a value:

```
class bank_account
{
  // ...

  template <class CompletionToken>
  auto balance(CompletionToken&& token) const
  {
    return std::experimental::post(ex_, [=]
      {
        return balance_;
      },
      std::forward<CompletionToken>(token));
  }
};
```

When using `use_future`, the future's value type is determined automatically from the executed function's return type:

```
std::future<int> fut = acct.balance(std::experimental::use_future);
std::cout << "balance is " << fut.get() << "\n";
```

Similarly, when using a callback, the function's result is passed as an argument:

```
acct.balance([](int bal){ std::cout << "balance is " << bal << "\n"; });
```

4.3 Limiting concurrency using strands

Clearly, having a private thread for each `bank_account` object is not going to scale well to thousands or millions of objects. We may instead want all bank accounts to share a thread pool. The `system_executor` object provides access to a system thread pool that we can use for this purpose:

```
std::experimental::system_executor ex;
std::experimental::post(ex, []{ std::cout << "Hello, world!\n"; });
```

However, the system thread pool uses an unspecified number of threads, and the posted function could run on any of them. The original reason for using the Active Object pattern was to limit the `bank_account` object's internal logic to run on a single thread. Fortunately, we can also limit concurrency by using the `strand<>` template.

The `strand<>` template is an executor that acts as an adapter for other executors. In addition to the rules of the underlying executor, a strand adds a guarantee of non-concurrency. That is, it guarantees that no two function objects submitted to the strand will run in parallel.

We can convert the `bank_account` class to use a strand very simply:

```
class bank_account
{
  int balance_ = 0;
  mutable std::experimental::strand<
    std::experimental::system_executor> ex_;

public:
  // ...
};
```

4.4 Lightweight, immediate execution using dispatch

As noted above, a post operation always submits a function object for later execution. This means that when we write:

```
template <class CompletionToken>
auto withdraw(int amount, CompletionToken&& token)
{
  return std::experimental::post(ex_, [=]
    {
      if (balance_ >= amount)
        balance_ -= amount;
    },
    std::forward<CompletionToken>(token));
}
```

we will always incur the cost of a context switch (plus an extra context switch if we wait for the result using a future). This cost can be avoided if we use a dispatch operation instead. The system executor's rules allow it to run a function object on any thread in the process, so if we change the withdraw function to:

```
template <class CompletionToken>
auto withdraw(int amount, CompletionToken&& token)
{
  return std::experimental::dispatch(ex_, [=]
    {
      if (balance_ >= amount)
        balance_ -= amount;
    },
    std::forward<CompletionToken>(token));
}
```

then the enclosed lambda object can be executed before `dispatch()` returns. The only condition where it will run later is when the strand is already busy on another thread. In this case, in order to meet the strand's non-concurrency guarantee, the function object must be added to the strand's work queue. In the common case there is no contention on the strand and the cost is minimised.

4.5 Composition using variadic dispatch and wrap

Let us now modify the `bank_account` class to add a function to transfer balance from one bank account to another. To implement this function we must coordinate code on two distinct executors: the `strand<>` executors that belong to each of the bank accounts.

A first attempt at solving this might use a `std::future`:

```
class bank_account
{
  // ...

  template <class CompletionToken>
  auto transfer(bank_account& to_acct, CompletionToken&& token)
  {
    return std::experimental::dispatch(ex_, [=, &to_acct]
      {
        if (balance_ >= amount)
        {
          balance_ -= amount;
          std::future<void> fut = to_acct.deposit(
             amount, std::experimental::use_future);
          fut.get();
        }
      },
      std::forward<CompletionToken>(token));
  }
};
```

While correct, this approach has the side effect of blocking the thread until the future is ready. If the `to_acct` object's strand is busy running other function objects, this might take some time.

In the examples so far, you might have noticed that sometimes we call `post()` or `dispatch()` with just one function object, and sometimes we call them with both a function object and a completion token.

The `dispatch()`, `post()` and `defer()` functions are variadic templates that can accept a number of completion tokens. For example, the library specifies the `dispatch()` function as:

```
template <class... CompletionTokens>
  auto dispatch(CompletionTokens&&... tokens);

template <class Executor, class... CompletionTokens>
  auto dispatch(Executor e, CompletionTokens&&... tokens);
```

When we call `dispatch()`, the library turns each completion token into a function object[3] and calls these functions in sequence. The return value of any given function is passed as an argument to the next one. For example:

```
std::future<std::string> fut = std::experimental::dispatch(ex_,
  []{ return 1; },
  [](int i) { return i + 1; },
  [](int i) { return i * 2; },
  [](int i) { return std::to_string(i); },
  [](std::string s) { return "value is " + s; },
  std::experimental::use_future);
std::cout << fut.get() << std::endl;
```

will output the string `value is 4`.

[3] Using N4045's `handler_type` trait.

For our bank account example, what is more important is that the variadic `dispatch()`, `post()` and `defer()` functions let you run each function object on a different executor. This is accomplished using the executor's `wrap()` member function, which associates the executor with a function object. We can then write our `transfer()` function as follows:

```
template <class CompletionToken>
auto transfer(int amount, bank_account& to_acct, CompletionToken&& token)
{
  return std::experimental::dispatch(
    ex_.wrap([=]
      {
        if (balance_ >= amount)
        {
          balance_ -= amount;
          return amount;
        }

        return 0;
      }),
    to_acct.ex_.wrap(
      [&to_acct](int deducted)
      {
        to_acct.balance_ += deducted;
      }),
    std::forward<CompletionToken>(token));
}
```

Here, the first function object:

```
ex_.wrap([=]
  {
    if (balance_ >= amount)
    {
      balance_ -= amount;
      return amount;
    }

    return 0;
  }),
```

is run on the source account's strand `ex_`. We wrap the function object using `ex_.wrap(...)` to tell `dispatch()` which executor to use.

The amount that is successfully deducted is then passed to the second function object:

```
to_acct.ex_.wrap(
  [&to_acct](int deducted)
  {
    to_acct.balance_ += deducted;
  }),
```

which, again thanks to `wrap()`, is run on the `to_acct` object's strand. By running each function object on a specific executor, we ensure that both `bank_account` objects are updated in a thread-safe way.

When an executor is not explicitly specified, a function object defaults to the `unspecified_executor`. The unspecified executor delegates its dispatch, post and defer operations to the system executor.

4.6 Composition using resumable functions

Variadic `dispatch()`, `post()` and `defer()` are useful for strictly sequential task flow, but for more complex control flow the executors library is extensible to other approaches, including resumable functions, or coroutines. To illustrate them we will now add a function to find the bank account with the largest balance.

Stackful coroutines are identified by a last argument of type `yield_context`[4]:

```
template <class Iterator, class CompletionToken>
auto find_largest_account(
    Iterator begin, Iterator end,
    CompletionToken&& token)
{
  return std::experimental::dispatch(
    [=](std::experimental::yield_context yield)
    {
      auto largest_acct = end;
      int largest_balance;

      for (auto i = begin; i != end; ++i)
      {
        int balance = i->balance(yield);
        if (largest_acct == end || balance > largest_balance)
        {
          largest_acct = i;
          largest_balance = balance;
        }
      }

      return largest_acct;
    },
    std::forward<CompletionToken>(token));
}
```

The yield object is a completion token that means that, when the call out to a bank account object is reached[5]:

```
int balance = i->balance(yield);
```

the library implementation automatically suspends the current function. The thread is not blocked and remains available to process other function objects. Once the `balance()` operation completes, the `find_largest_account` function resumes execution at the following statement.

These stackful resumable functions are implemented entirely as a library construct, and require no alteration to the language in the form of new keywords. Consequently, they can

[4] Detected using a SFINAE-enabled `handler_type` trait specialisation.

[5] The `yield_context` completion token type is discussed in N4045.

utilise arbitrarily complex control flow constructs, including stack-based variables, while still retaining concise, familiar C++ language use.

4.7 Polymorphic executors

Up to this point, our `bank_account` class's executor has been a private implementation detail. However, rather than limit ourselves to the system executor, we will now alter the class to be able to specify the executor on a case-by-case basis.

Ultimately, executors are defined by a set of type requirements, and each of the executors we have used so far is a distinct type. For optimal performance we can use compile-time polymorphism, and specify the executor as a template parameter:

```
template <class Executor>
class bank_account
{
  int balance_ = 0;
  mutable std::experimental::strand<Executor> ex_;

public:
  explicit bank_account(const Executor& ex)
    : ex_(ex)
  {
  }

  // ...
};
```

On the other hand, in many situations runtime polymorphism will be preferred. To support this, the library provides the `executor` class, a polymorphic wrapper:

```
class bank_account
{
  int balance_ = 0;
  mutable std::experimental::strand<
    std::experimental::executor> ex_;

public:
  explicit bank_account(
    const std::experimental::executor& ex =
      std::experimental::system_executor())
    : ex_(ex)
  {
  }

  // ...
};
```

The `bank_account` class can then be constructed using an explicitly specified thread pool:

```
std::experimental::thread_pool pool;
bank_account acct(pool.get_executor());
```

or any other object that meets the executor type requirements.

4.8 Coordinating concurrent operations

To illustrate the tools that this library provides for managing parallelism and concurrency, let us now turn our attention to a different use case: sorting large datasets. Consider an example where we want to sort a very large vector of doubles:

```
std::vector<double> vec(a_very_large_number);
...
std::sort(vec.begin(), vec.end());
```

If we are running this code on a system with two at least CPUs then we can cut the running time by splitting the array into halves, sorting each half in parallel, and finally merging the two now-sorted halves into a sorted whole. This executors library lets us do this easily using the `copost()` function:

```
std::experimental::copost(
  [&]{ std::sort(vec.begin(), vec.begin() + (vec.size() / 2)); },
  [&]{ std::sort(vec.begin() + (vec.size() / 2), vec.end()); },
  std::experimental::use_future).get();
std::inplace_merge(vec.begin(), vec.begin() + (vec.size() / 2), vec.end());
```

The function name `copost()` is short for *co*ncurrent *post*. In the above example, it posts the two lambda objects:

```
[&]{ std::sort(vec.begin(), vec.begin() + (vec.size() / 2)); },
[&]{ std::sort(vec.begin() + (vec.size() / 2), vec.end()); },
```

to the system thread pool, where they can run in parallel. When both have finished, the caller is notified via the final completion token (in this case, `use_future`).

Like `post()`, `copost()` (and its counterparts `codispatch()` and `codefer()`) is a variadic template function that can accept a number of completion tokens. In pseudo-code, the library declares `copost()` as:

```
auto copost(t0, t1, ..., tN-1, tN);
auto copost(executor, t0, t1, ..., tN-1, tN);
```

where t_0 to t_N are completion tokens. When we call `copost()`, the library turns each of the tokens into the function objects f_0 to f_N. The functions f_0 to f_{N-1} are then posted for execution, and only when all are complete will f_N be invoked. The return values of f_0 to f_{N-1} are passed as arguments to f_N as in the following example:

```
std::experimental::copost(ex_,
  []{ return 1; },
  []{ return "hello"s; },
  []{ return 123.0; },
  [](int a, std::string b, double c) { ... });
```

4.9 Composition using chain

To facilitate reusability, we will now wrap our parallel sort implementation in a function and let the user choose how to wait for completion. However, once the two halves have been sorted we now need to perform two separate actions: merge the halves, *and* deliver the completion notification according to the user-supplied completion token. We can combine these two actions into a single operation using `chain()`:

```
template <class Iterator, class CompletionToken>
auto parallel_sort(Iterator begin, Iterator end, CompletionToken&& token)
{
  const std::size_t n = end - begin;
  return std::experimental::copost(
    [=]{ std::sort(begin, begin + (n / 2)); },
    [=]{ std::sort(begin + (n / 2), end); },
    std::experimental::chain(
      [=]{ std::inplace_merge(begin, begin + (n / 2), end); },
      std::forward<CompletionToken>(token)));
}
```

Here, the call to `chain()`:

```
std::experimental::chain(
```

takes the two completion tokens:

```
  [=]{ std::inplace_merge(begin, begin + (n / 2), end); },
  std::forward<CompletionToken>(token)));
```

turns them into their corresponding function objects, and then combines these into a single function object. This single function object meets the requirements for a final argument to `copost()`.

The `chain()` function is a variadic template similar to `dispatch()`, `post()` and `defer()`:

```
template <class... CompletionTokens>
  auto chain(CompletionTokens&&... tokens);

template <class Signature, class... CompletionTokens>
  auto chain(CompletionTokens&&... tokens);
```

When we call `chain()`, the library turns the completion tokens into their corresponding handler objects. These are chained such that they execute serially, and the return value of any given function in the chain is passed as an argument to the next one. However, unlike `dispatch()`, `post()` or `defer()`, the chained functions are not executed immediately, but are instead returned as a function object which we can save for later execution. For example, the continuation created by:

```
auto c = std::experimental::chain(
  []{ return 1; },
  [](int i) { return i + 1; },
  [](int i) { return i * 2; },
  [](int i) { return std::to_string(i); },
  [](std::string s) { std::cout << "value is " << s << "\n"; });
```

can be called directly like this:

```
std::move(c)();
```

taking care to note that is only safe to invoke the function once, as the call may have resulted in `c` being in a moved-from state.

A more typical way to invoke the continuation would be to pass it to an operation like `dispatch()`, `post()` or `defer()`:

```
std::experimental::dispatch(std::move(c));
```

Unlike a direct call, this has the advantage of preserving completion token behaviour with respect to the deduced return type. This means that the following works as expected:

```
auto c = std::experimental::chain(
  []{ return 1; },
  [](int i) { return i + 1; },
  [](int i) { return i * 2; },
  [](int i) { return std::to_string(i); },
  [](std::string s) { return "value is " + s; },
  std::experimental::use_future);
std::future<std::string> fut = std::experimental::dispatch(std::move(c));
std::cout << fut.get() << std::endl;
```

and outputs the string `value is 4`.

4.10 Convenience functions for timer operations

When working with executors, we will often need to schedule functions to run at, or after, a time. This library provides several high-level operations we can use for this purpose.

First of these is the `post_after()` function, which we can use to schedule a function to run after a delay:

```
std::experimental::post_after(std::chrono::seconds(1),
  []{ std::cout << "Hello, world!\n"; });
```

If we want to be notified when the function has finished, we can specify a completion token as well:

```
std::future<void> fut = std::experimental::post_after(
  std::chrono::seconds(1),
  []{ std::cout << "Hello, world!\n"; },
  std::experimental::use_future);
fut.get();
```

Both of the above examples use the system executor. We can of course specify an executor of our own:

```
std::experimental::thread_pool pool;
auto ex = pool.get_executor();
std::experimental::post_after(ex, std::chrono::seconds(1),
  []{ std::cout << "Hello, world!\n"; });
```

The `post_at()` function can instead be used to run a function object at an absolute time:

```
auto start_time = std::chrono::steady_clock::now();
// ...
std::experimental::post_after(start_time + std::chrono::seconds(1),
  []{ std::cout << "Hello, world!\n"; });
```

The library also provides `dispatch_after()` and `dispatch_at()` as counterparts to `post_after()` and `post_at()` respectively. As dispatch operations, they are permitted to run the function object before returning, according to the rules of the underlying executor. Likewise, `defer_after()` and `defer_at()` may be used to schedule function objects that represent a continuation of the caller.

4.10.1 Timer operations in resumable functions

These high-level convenience functions can easily be used in resumable functions to provide the resumable equivalent of `std::this_thread::sleep()`:

```
std::experimental::dispatch(
  [](std::experimental::yield_context yield)
  {
    auto start_time = std::chrono::steady_clock::now();
    for (int i = 0; i < 10; ++i)
    {
      std::experimental::dispatch_at(
        start_time + std::chrono::seconds(i + 1), yield);
      std::cout << i << std::endl;
    }
  });
```

Here, the yield object is passed as a completion token to `dispatch_at()`. The resumable function is automatically suspended and resumes once the absolute time is reached.

4.11 Timer objects and cancellation of timer operations

The convenience functions do not provide a way to cancel a timer operation. For this level of control we want to use one of the timer classes provided by the library: `steady_timer`, `system_timer` or `high_resolution_timer`. To support arbitrary clock types, the class template `basic_timer<>` may also be used directly.

A timer object has an associated expiry time, which can be set using either a relative value:

```
timer.expires_after(std::chrono::seconds(60));
```

or an absolute one:

```
timer.expires_at(start_time + std::chrono::seconds(60));
```

Once the expiry time is set, we then wait for the timer to expire:

```
timer.wait([](std::error_code ec){ std::cout << "Hello, world!\n"; });
```

Finally, if we want to cancel the wait, we simply use the `cancel()` member function:

```
timer.cancel();
```

If the cancellation was successful, the function object is called with an `error_code` equivalent to the condition `std::errc::operation_canceled`.

4.12 Execution context as a container of services

As an implementation detail of the library, pending timers may be managed using a timer queue, such as a heap or wheel. This timer queue is implemented as a *service* that is associated with an execution context.

The base class `execution_context` implements an extensible, type-safe, polymorphic set of services, indexed by service type. Services exist to manage the resources that are shared across an execution context, a timer queue being one such example.

Access to the services of an `execution_context` is via free function templates, `use_service()`, `add_service()` and `has_service()`.

In this example:

```
Service& service = use_service<Service>(my_context);
```

the `Service` type argument chooses a service, and a reference to the corresponding service object is returned. If `Service` is not present in an `execution_context`, an object of type `Service` is created and added to the context. A program can check if the

`execution_context` already implements a particular service with the function `has_service<Service>()`, or explicitly add a service instance using `add_service()`.

When constructing a timer object, we can choose whether it uses the system-wide execution context:

```
std::experimental::steady_timer timer;
```

or a specific execution context, such as a thread pool:

```
std::experimental::thread_pool pool;
std::experimental::steady_timer timer(pool);
```

In the latter case, the timer cannot be used once its execution context ceases to exist.

4.13 Developing an executor-aware asynchronous operation

N4045 includes an example of a simple wrapper around the Windows API function `RegisterWaitForSingleObject`. This function registers a callback to be invoked once a kernel object is in a signalled state, or if a timeout occurs. The callback is invoked from the system thread pool.

In N4045, the example shows how to convert this wrapper from a simple callback-based one into one that supports arbitrary completion tokens. In this paper, we will look at the additional changes required to make the wrapper executor-aware.

The minimal changes shown in this example are:

- A call to `get_executor()` to obtain the completion handler's associated executor.
- An `executor_work` object to record pending work against the executor.
- Rather than invoking the handler directly, use of `defer()` and `dispatch()` to ensure that the handler is correctly executed via the executor.

Traits-enabled wrapper

```
template <class Handler>
struct wait_op {
  atomic<HANDLE> wait_handle_;
  Handler handler_;



  explicit wait_op(Handler handler)
    : wait_handle_(0),
      handler_(move(handler)) {}

};

template <class Handler>
void CALLBACK wait_callback(
    void* param, BOOLEAN timed_out)
{
  unique_ptr<wait_op<Handler>> op(
    static_cast<wait_op<Handler>*>(param));

  while (op->wait_handle_ == 0)
    SwitchToThread();

  const error_code ec = timed_out
    ? make_error_code(errc::timed_out)
    : error_code();

  op->handler_(ec);

}

template <class CompletionToken>
auto wait_for_object(
    HANDLE object, DWORD timeout,
    DWORD flags, CompletionToken&& token)
{
  async_completion<CompletionToken,
    void(error_code)> completion(token);

  typedef handler_type_t<CompletionToken,
    void(error_code)> Handler;

  unique_ptr<wait_op<Handler>>
    op(new wait_op<Handler>(
      move(completion.handler)));

  HANDLE wait_handle;
  if (RegisterWaitForSingleObject(
        &wait_handle, object,
        &wait_callback<Handler>,
        op.get(), timeout,
        flags | WT_EXECUTEONLYONCE))
  {
    op->wait_handle_ = wait_handle;
    op.release();
  }
  else
  {
    DWORD last_error = GetLastError();
    const error_code ec(
        last_error, system_category());

    op->handler_(ec);

  }

  return completion.result.get();
}
```

Executor-aware wrapper

```
template <class Handler, class Executor>
struct wait_op {
  atomic<HANDLE> wait_handle_;
  Handler handler_;
  typedef decltype(get_executor(
    declval<Handler>())) Executor;
  executor_work<Executor> work_;
  explicit wait_op(Handler handler)
    : wait_handle_(0),
      handler_(move(handler)),
      work_(get_executor(handler_)) {}
};

template <class Handler>
void CALLBACK wait_callback(
    void* param, BOOLEAN timed_out)
{
  unique_ptr<wait_op<Handler>> op(
    static_cast<wait_op<Handler>*>(param));

  while (op->wait_handle_ == 0)
    SwitchToThread();

  const error_code ec = timed_out
    ? make_error_code(errc::timed_out)
    : error_code();

  dispatch(op->work_.get_executor(),
    [h=move(op->handler_), ec]{ h(ec); });
}

template <class CompletionToken>
auto wait_for_object(
    HANDLE object, DWORD timeout,
    DWORD flags, CompletionToken&& token)
{
  async_completion<CompletionToken,
    void(error_code)> completion(token);

  typedef handler_type_t<CompletionToken,
    void(error_code)> Handler;

  unique_ptr<wait_op<Handler>>
    op(new wait_op<Handler>(
      move(completion.handler)));

  HANDLE wait_handle;
  if (RegisterWaitForSingleObject(
        &wait_handle, object,
        &wait_callback<Handler>,
        op.get(), timeout,
        flags | WT_EXECUTEONLYONCE))
  {
    op->wait_handle_ = wait_handle;
    op.release();
  }
  else
  {
    DWORD last_error = GetLastError();
    const error_code ec(
        last_error, system_category());

    defer(op->work_.get_executor(),
      [h=move(op->handler_), ec]{ h(ec); });
  }

  return completion.result.get();
}
```

5 Impact on the standard

This proposal consists only of library additions and does not require any language features beyond those that are already available in C++14.

6 Relationship to other proposals

This proposal builds on the type traits defined in N4045 *Library Foundations for Asynchronous Operations*. This paper is intended as an alternative proposal to N3785 *Executors and schedulers*.

7 Conclusion

The type traits introduced in N4045 *Library Foundations for Asynchronous Operations* define an extensible asynchronous model that can support:

- Callbacks, where minimal runtime penalty is desirable.
- Futures, and not just `std::future` but also future classes supplied by other libraries.
- Coroutines or resumable functions, without adding new keywords to the language.

The library introduced in this paper applies this asynchronous model, and its design philosophy, to executors. Rather than a design that is restricted to runtime polymorphism, we can allow users to choose the approach that is appropriate to their use case.

Future work will aim to develop guidance on the development of asynchronous operations that participate in an executor-aware model, such as those that integrate operating system services, for example networking support.

8 Acknowledgements

The author would like to thank Jamie Allsop for providing feedback, corrections and suggestions on both the library implementation and this proposal.

9 Proposed text

Note: This is not intended as complete proposed text, but rather as a brief sketch of the library's key components.

9.1 Executors

Executor type requirements

An archetype for the executor type requirements looks like:

```
class Executor
{
public:
  Executor(const Executor&) noexcept; — #1

  execution_context& context() noexcept; — #2

  void work_started() noexcept;
  void work_finished() noexcept; } #3

  template <class Function, class Allocator>
    void dispatch(Function&& f, const Allocator& a);
  template <class Function, class Allocator>
    void post(Function&& f, const Allocator& a);
  template <class Function, class Allocator>
    void defer(Function&& f, const Allocator& a); } #4

  template <class T>
  auto wrap(T&& t) const;
};
```

The key elements of the executor type requirements are:

1. Like allocators, executors are CopyConstructible, and the copy and move constructors shall not exit via an exception.
2. All executors have an associated execution context. This may be used to access the execution context's services, such as a timer queue.
3. The `work_started()` and `work_finished()` functions are used to inform the executor of outstanding work. An example of outstanding work is a pending receive operation on a socket.
4. The `dispatch()`, `post()` and `defer()` functions submit function objects for execution according to the executor's rules. The executor may use the supplied allocator to allocate any objects required for bookkeeping, e.g. a linked-list element to store the function object in a queue.

continuation_of trait

```
template <class> continuation_of; // not defined

template <class _Func, class... _Args>
struct continuation_of<_Func(_Args...)>
{
  typedef see below signature;

  template <class _F, class _C> static auto chain(_F&& __f, _C&& __c);
};
```

continuation_of::signature

Type:

- If `_Func` and `decay_t<_Func>` are different types, `continuation_of<decay_t<_Func>(_Args...)>::signature`;
- Let `R` be the type produced by `result_of_t<_Func(_Args...)>`. If `R` is `void`, `void()`. If `R` is non-void, `void(R)`.
- Otherwise, if `result_of<_Func(_Args...)>` does not contain a nested type named `type`, the program is ill formed.

continuation_of::chain

```
template <class _F, class _C> static auto chain(_F&& __f, _C&& __c);
```

If `_Func` and `decay_t<_Func>` are different types, returns:

```
continuation_of<decay_t<_Func>(_Args...)>::chain(forward<_F>(__f), forward<_C>(__c)).
```

Otherwise, returns a function object that, when invoked, calls a copy of `__f`, and then passes the result to a copy of `__c`.

Class execution_context

```
class execution_context
{
public:
  class service;

  // construct / copy / destroy:

  execution_context();
  execution_context(const execution_context&) = delete;
  execution_context& operator=(const execution_context&) = delete;
  virtual ~execution_context();

protected:

  // execution context operations:

  void shutdown_context();
  void destroy_context();
};

class service_already_exists;
template <class _Service> _Service& use_service(execution_context& __c);
template <class _Service, class... _Args> _Service&
  make_service(execution_context& __c, _Args&&... __args);
template <class _Service> bool has_service(execution_context& __c) noexcept;
```

A collection of services, indexed by type.

Class execution_context::service

```
class execution_context::service
{
protected:
  explicit service(execution_context& __c);
  virtual ~service();

  execution_context& context();

private:
  virtual void shutdown_service() = 0;
};
```

The class `execution_context::service` is the base class for all services within an execution context.

is_executor trait

```
template <class _T> struct is_executor : false_type {};
```

Executor argument tag

```
struct executor_arg_t {};
constexpr executor_arg_t executor_arg = executor_arg_t{};
```

The `executor_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading.

uses_executor trait

```
template <class _T, class _Executor> struct uses_executor : false_type {};
```

Class template executor_wrapper

```
template <class _T, class _Executor> class executor_wrapper
{
public:
  typedef _Executor executor_type;

  // construct / copy / destroy:

  executor_wrapper(const executor_wrapper& __w);
  executor_wrapper(executor_wrapper&& __w);
  template <class _U> executor_wrapper(const executor_wrapper<_U, _Executor>& __w);
  template <class _U> executor_wrapper(executor_wrapper<_U, _Executor>&& __w);
  template <class _U> executor_wrapper(executor_arg_t, const _Executor& __e, _U&& __u);
  executor_wrapper(executor_arg_t, const _Executor& __e, const executor_wrapper& __w);
  executor_wrapper(executor_arg_t, const _Executor& __e, executor_wrapper&& __w);
  template <class _U> executor_wrapper(executor_arg_t, const _Executor& __e,
    const executor_wrapper<_U, _Executor>& __w);
  template <class _U> executor_wrapper(executor_arg_t, const _Executor& __e,
    executor_wrapper<_U, _Executor>&& __w);

  ~executor_wrapper();

  // executor wrapper operations:

  executor_type get_executor() const noexcept;
  template <class... _Args> result_of_t<_T(_Args&&...)> operator()(_Args&&...);
};

template <class _T, class _Executor>
  struct uses_executor<executor_wrapper<_T, _Executor>, _Executor> : true_type {};

template <class _T, class _Executor, class _Signature>
  struct handler_type<executor_wrapper<_T, _Executor>, _Signature>;

template <class _T, class _Executor>
  class async_result<executor_wrapper<_T, _Executor>>;
```

Class template executor_work

```
template <class _Executor>
class executor_work
{
public:
  typedef _Executor executor_type;

  // construct / copy / destroy:
```

```
  explicit executor_work(const executor_type& __e) noexcept;
  executor_work(const executor_work& __w) noexcept;
  executor_work(executor_work&& __w) noexcept;

  executor_work operator=(const executor_type&) = delete;

  ~executor_work();

  // executor work operations:

  executor_type get_executor() const noexcept;
  void reset() noexcept;
};
```

Class system_executor

```
class system_executor
{
public:
  // executor operations:

  execution_context& context();

  void work_started() noexcept;
  void work_finished() noexcept;

  template <class _Func, class _Alloc> void dispatch(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void post(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void defer(_Func&& __f, const _Alloc& a);

  template <class _Func> auto wrap(_Func&& __f) const;
};

template <> struct is_executor<system_executor> : true_type {};
```

Class unspecified_executor

```
class unspecified_executor
{
public:
  // executor operations:

  execution_context& context();

  void work_started() noexcept;
  void work_finished() noexcept;

  template <class _Func, class _Alloc> void dispatch(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void post(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void defer(_Func&& __f, const _Alloc& a);

  template <class _Func> auto wrap(_Func&& __f) const;
};

template <> struct is_executor<unspecified_executor> : true_type {};
```

get_executor

```
template <class _T> auto get_executor(const _T&) noexcept;
```

The `get_executor` function is used to obtain an object's associated executor. For function objects, this is the executor that should be used to invoke the given function. This default implementation behaves as follows:

- if the object type has a nested type `executor_type`, returns the result of the object's `get_executor()` member function;
- if the object is callable, returns an `unspecified_executor` object;
- otherwise, this function does not participate in overload resolution.

Polymorphic executor wrapper

```
class bad_executor;

class executor
{
public:
  // construct / copy / destroy:

  executor() noexcept;
  executor(nullptr_t) noexcept;
  executor(const executor& __e) noexcept;
  executor(executor&& __e) noexcept;
  template <class _Executor> executor(_Executor __e);
  template <class _Alloc> executor(allocator_arg_t, const _Alloc&) noexcept;
  template <class _Alloc> executor(allocator_arg_t, const _Alloc&, nullptr_t) noexcept;
  template <class _Alloc> executor(allocator_arg_t, const _Alloc&, const executor& __e);
  template <class _Alloc> executor(allocator_arg_t, const _Alloc&, executor&& __e);
  template <class _Executor, class _Alloc>
    executor(allocator_arg_t, const _Alloc& __a, _Executor __e);

  executor& operator=(const executor& __e) noexcept;
  executor& operator=(executor&& __e) noexcept;
  executor& operator=(nullptr_t) noexcept;
  template <class _Executor> executor& operator=(_Executor&& __e);

  ~executor();

  // executor operations:

  execution_context& context();

  void work_started() noexcept;
  void work_finished() noexcept;

  template <class _Func, class _Alloc> void dispatch(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void post(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void defer(_Func&& __f, const _Alloc& a);

  template <class _Func> auto wrap(_Func&& __f) const;

  // executor capacity:

  explicit operator bool() const noexcept;

  // executor target access:

  const type_info& target_type() const noexcept;
  template <class _Executor> _Executor* target() noexcept;
  template <class _Executor> const _Executor* target() const noexcept;
};

template <> struct is_executor<executor> : true_type {};

bool operator==(const executor& __e, nullptr_t) noexcept;
bool operator==(nullptr_t, const executor& __e) noexcept;
bool operator!=(const executor& __e, nullptr_t) noexcept;
bool operator!=(nullptr_t, const executor& __e) noexcept;

template<class _Alloc>
```

```
  struct uses_allocator<executor, _Alloc>
    : true_type {};
```

chain

```
template <class... _CompletionTokens>
  auto chain(_CompletionTokens&&... __tokens);
template <class _Signature, class... _CompletionTokens>
  auto chain(_CompletionTokens&&... __tokens);
```

dispatch

```
template <class... _CompletionTokens>
  auto dispatch(_CompletionTokens&&... __tokens);
template <class _Executor, class... _CompletionTokens>
  auto dispatch(const _Executor& __e, _CompletionTokens&&... __tokens);
```

post

```
template <class... _CompletionTokens>
  auto post(_CompletionTokens&&... __tokens);
template <class _Executor, class... _CompletionTokens>
  auto post(const _Executor& __e, _CompletionTokens&&... __tokens);
```

defer

```
template <class... _CompletionTokens>
  auto defer(_CompletionTokens&&... __tokens);
template <class _Executor, class... _CompletionTokens>
  auto defer(const _Executor& __e, _CompletionTokens&&... __tokens);
```

codispatch

```
template <class... _CompletionTokens>
  auto codispatch(_CompletionTokens&&... __tokens);
template <class _Executor, class... _CompletionTokens>
  auto codispatch(const _Executor& __e, _CompletionTokens&&... __tokens);
```

copost

```
template <class... _CompletionTokens>
  auto copost(_CompletionTokens&&... __tokens);
template <class _Executor, class... _CompletionTokens>
  auto copost(const _Executor& __e, _CompletionTokens&&... __tokens);
```

codefer

```
template <class... _CompletionTokens>
  auto codefer(_CompletionTokens&&... __tokens);
template <class _Executor, class... _CompletionTokens>
  auto codefer(const _Executor& __e, _CompletionTokens&&... __tokens);
```

Class template strand

```
template <class _Executor>
class strand
{
public:
  typedef _Executor executor_type;

  // construct / copy / destroy:

  strand();
  explicit strand(_Executor __e);
  strand(const strand& __s);
  strand(strand&& __s);
```

```
  template <class _OtherExecutor> strand(const strand<_OtherExecutor>& __s);
  template <class _OtherExecutor> strand(strand<_OtherExecutor>&& __s);

  strand& operator=(const strand& __s);
  strand& operator=(strand&& __s);
  template <class _OtherExecutor> strand& operator=(const strand<_OtherExecutor>& __s);
  template <class _OtherExecutor> strand& operator=(strand<_OtherExecutor>&& __s);

  ~strand();

  // executor operations:

  executor_type get_executor() const noexcept;
  execution_context& context();

  void work_started() noexcept;
  void work_finished() noexcept;

  template <class _Func, class _Alloc> void dispatch(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void post(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void defer(_Func&& __f, const _Alloc& a);

  template <class _Func> auto wrap(_Func&& __f) const;
};

template <class _Executor> struct is_executor<strand<_Executor>> : true_type {};
```

9.2 Thread pools

Class thread_pool

```
class thread_pool
  : public execution_context
{
public:
  class executor_type;

  // construct / copy / destroy:

  thread_pool();
  explicit thread_pool(size_t __num_threads);
  thread_pool(const thread_pool&) = delete;
  thread_pool& operator=(const thread_pool&) = delete;
  ~thread_pool();

  // thread pool operations:

  executor_type get_executor() const noexcept;

  void stop();
  void join();
};
```

Class thread_pool::executor_type

```
class thread_pool::executor_type
{
public:
  // construct / copy / destroy:

  executor_type(const executor_type& __e) noexcept;
  executor_type& operator=(const executor_type& __e) noexcept;
  ~executor_type();

  // executor operations:
```

```
  execution_context& context();

  void work_started() noexcept;
  void work_finished() noexcept;

  template <class _Func, class _Alloc> void dispatch(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void post(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void defer(_Func&& __f, const _Alloc& a);

  template <class _Func> auto wrap(_Func&& __f) const;
};

template <> struct is_executor<thread_pool::executor_type> : true_type {};
```

9.3 Loop schedulers

Class loop_scheduler

```
class loop_scheduler
  : public execution_context
{
public:
  class executor_type;

  // construct / copy / destroy:

  loop_scheduler();
  explicit loop_scheduler(size_t __concurrency_hint);
  loop_scheduler(const loop_scheduler&) = delete;
  loop_scheduler& operator=(const loop_scheduler&) = delete;
  ~loop_scheduler();

  // scheduler operations:

  executor_type get_executor() const noexcept;

  size_t run();
  size_t run_one();
  template <class _Rep, class _Period>
    size_t run_for(const chrono::duration<_Rep, _Period>& __rel_time);
  template <class _Clock, class _Duration>
    size_t run_until(const chrono::time_point<_Clock, _Duration>& __abs_time);
  size_t poll();
  size_t poll_one();

  void stop();
  bool stopped() const;
  void reset();
};
```

Class loop_scheduler::executor_type

```
class loop_scheduler::executor_type
{
public:
  // construct / copy / destroy:

  executor_type(const executor_type& __e) noexcept;
  executor_type& operator=(const executor_type& __e) noexcept;
  ~executor_type();

  // executor operations:

  execution_context& context();
```

```
  void work_started() noexcept;
  void work_finished() noexcept;

  template <class _Func, class _Alloc> void dispatch(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void post(_Func&& __f, const _Alloc& a);
  template <class _Func, class _Alloc> void defer(_Func&& __f, const _Alloc& a);

  template <class _Func> auto wrap(_Func&& __f) const;
};

template <> struct is_executor<loop_scheduler::executor_type> : true_type {};
```

9.4 Timers

timer_traits trait

```
template <class _Clock>
struct timer_traits
{
  static typename _Clock::duration to_duration(
    const typename _Clock::duration& __d);

  static typename _Clock::duration to_duration(
    const typename _Clock::time_point& __t);
};
```

Class template basic_timer

```
template <class _Clock, class _TimerTraits = timer_traits<_Clock>>
class basic_timer
{
public:
  typedef _Clock clock_type;
  typedef typename clock_type::duration duration;
  typedef typename clock_type::time_point time_point;
  typedef _TimerTraits traits_type;

  // construct / copy / destroy:

  basic_timer();
  explicit basic_timer(const duration& __d);
  explicit basic_timer(const time_point& __t);
  explicit basic_timer(execution_context& __c);
  basic_timer(execution_context& __c, const duration& __d);
  basic_timer(execution_context& __c, const time_point& __t);
  basic_timer(const basic_timer&) = delete;
  basic_timer(basic_timer&& __t);

  basic_timer& operator=(const basic_timer&) = delete;
  basic_timer& operator=(basic_timer&& __t);

  ~basic_timer();

  // timer operations:

  execution_context& context();

  void cancel();
  void cancel_one();

  time_point expiry() const;
  void expires_at(const time_point& __t);
  void expires_after(const duration& __d);
```

```
  void wait();
  void wait(error_code& __ec);
  template <class _CompletionToken> auto wait(_CompletionToken&& __token);
};

typedef basic_timer<chrono::system_clock> system_timer;
typedef basic_timer<chrono::steady_clock> steady_timer;
typedef basic_timer<chrono::high_resolution_clock> high_resolution_timer;
```

dispatch_at

```
template <class _Clock, class _Duration, class... _CompletionTokens>
  auto dispatch_at(const chrono::time_point<_Clock, _Duration>& __abs_time,
    _CompletionTokens&&... __tokens);
template <class _Clock, class _Duration, class _Executor, class... _CompletionTokens>
  auto dispatch_at(const chrono::time_point<_Clock, _Duration>& __abs_time,
    const _Executor& __e, _CompletionTokens&&... __tokens);
```

post_at

```
template <class _Clock, class _Duration, class... _CompletionTokens>
  auto post_at(const chrono::time_point<_Clock, _Duration>& __abs_time,
    _CompletionTokens&&... __tokens);
template <class _Clock, class _Duration, class _Executor, class... _CompletionTokens>
  auto post_at(const chrono::time_point<_Clock, _Duration>& __abs_time,
    const _Executor& __e, _CompletionTokens&&... __tokens);
```

defer_at

```
template <class _Clock, class _Duration, class... _CompletionTokens>
  auto defer_at(const chrono::time_point<_Clock, _Duration>& __abs_time,
    _CompletionTokens&&... __tokens);
template <class _Clock, class _Duration, class _Executor, class... _CompletionTokens>
  auto defer_at(const chrono::time_point<_Clock, _Duration>& __abs_time,
    const _Executor& __e, _CompletionTokens&&... __tokens);
```

dispatch_after

```
template <class _Rep, class _Period, class... _CompletionTokens>
  auto dispatch_after(const chrono::duration<_Rep, _Period>& __rel_time,
    _CompletionTokens&&... __token);
template <class _Rep, class _Period, class _Executor, class... _CompletionTokens>
  auto dispatch_after(const chrono::duration<_Rep, _Period>& __rel_time,
    const _Executor& __e, _CompletionTokens&&... __tokens);
```

post_after

```
template <class _Rep, class _Period, class... _CompletionTokens>
  auto post_after(const chrono::duration<_Rep, _Period>& __rel_time,
    _CompletionTokens&&... __token);
template <class _Rep, class _Period, class _Executor, class... _CompletionTokens>
  auto post_after(const chrono::duration<_Rep, _Period>& __rel_time,
    const _Executor& __e, _CompletionTokens&&... __tokens);
```

defer_after

```
template <class _Rep, class _Period, class... _CompletionTokens>
  auto defer_after(const chrono::duration<_Rep, _Period>& __rel_time,
    _CompletionTokens&&... __token);
template <class _Rep, class _Period, class _Executor, class... _CompletionTokens>
  auto defer_after(const chrono::duration<_Rep, _Period>& __rel_time,
    const _Executor& __e, _CompletionTokens&&... __tokens);
```

9.5 Resumable Functions

Class template basic_yield_context

```
template <class _Executor>
class basic_yield_context
{
public:
  typedef _Executor executor_type;

  // construct / copy / destroy:

  template <class _OtherExecutor>
    basic_yield_context(const basic_yield_context<_OtherExecutor>&);

  // basic_yield_context operations:

  executor_type get_executor() const noexcept;
  basic_yield_context operator[](error_code& __ec) const;
};

template <class _Executor, class _R, class... _Args>
  struct handler_type<basic_yield_context<_Executor>, _R(_Args...)>;

typedef basic_yield_context<executor> yield_context;
```