Paper no.	P1341R0
Date	2018-11-25
Reply To	Lewis Baker <lbaker@fb.com>
Audience	SG1, LEWG

UNIFYING ASYNCHRONOUS APIS IN C++ STANDARD LIBRARY

Unifying executors, sender/receiver, coroutines, parallel algorithms and networking.

ABSTRACT

The C++ language is currently at a unique junction point where we see the convergence of several major pieces of functionality related to concurrent and parallel programming potentially being merged into the language at the same time.

We have the Executors proposal P0443 and the associated proposal for a Sender/Receiver-based executors API in P1194, the Coroutines TS, the Concurrency TS, Ranges and the Networking TS. Each of these pieces of functionality is in some way related to concurrency, parallelism and asynchronous programming and so these pieces would ideally all be designed to fit together nicely into a coherent design.

Unfortunately, perhaps due in part to the history of the independent development of each of these features, we find that as they come together there are some incompatibilities that mean that these features do not work as well together as we would like them to.

This paper seeks to sketch out a proposed vision for these components that ties their designs together with a unified asynchronous programming model for the C++ standard library.

Please note that this paper is exploratory and concepts in here are still evolving.

The key components of this design direction are:

- Unifying coroutines and Sender/Receiver by creating a new `Task` concept that implements both `Awaitable` and `SingleTypedSender` concepts. This allows an asynchronous operation to be consumed either using callbacks or by using operator co_await from within a coroutine.
- Defining a minimal interface for an `Executor` concept in terms of an `executor.schedule()` method that returns a `TaskOf<SubExecutor>`.
- Defining generic algorithms that compose `Executor` and `Task` objects and that have default implementations that are defined in terms of the `executor.schedule()` method.
- Allowing these algorithms to be customised for a given `Executor` type to provide more efficient implementations where appropriate/possible.
- Defining bulk/parallel algorithms to be implemented in terms of `Executor` objects and that return `Task` objects that allow chaining units of parallel work with dependencies.
- Allowing bulk/parallel algorithms to be customised by `Executors` for particular execution policies.

- Modifying async methods from Networking TS that accept a `CompletionHandler` parameter to drop this parameter and instead return a `Task` object so that they can be more easily composed with `Executors`, used by generic parallel algorithms and be more efficiently consumed by coroutines.

MOTIVATION

In the upcoming versions of the C++ standard we are hoping to introduce a raft of new asynchronous programming facilities and APIs covering areas such as executors, coroutines, networking, parallel algorithms and async ranges.

These asynchronous facilities should ideally all integrate well together to provide an efficient and composable foundation that applications can use to build asynchronous applications. However, there are some issues with the current designs that either limit the composability or that limit the efficiency of these asynchronous APIs.

COMPOSABILITY OF ASYNCHRONOUS OPERATIONS

The Sender/Receiver model for representing asynchronous operations as described in [P1194] provides a good basis for composable asynchronous operations, drawing on many years of experience from RxCpp and the reactive extensions community.

By reifying an asynchronous operation as a Sender object and by providing a uniform interface for attaching a generalised callback to an asynchronous operation it allows us to write generic higher-order functions, such as `when_all()` from [P1316], that can compose arbitrary asynchronous operations.

Also, by separating of the creation of the object representing the asynchronous operation from the step of attaching a continuation this allows these higher-order functions to compose lazy asynchronous operations together efficiently without introducing extra overhead for synchronising and storing intermediate results such as would be required with an eager `std::experimental::future`-based API.

COMPOSABILITY OF NETWORKING TS ASYNCHRONOUS OPERATIONS

The design of asynchronous APIs in the Networking TS requires that a continuation is passed in as the CompletionHandler parameter together with other parameters to the asynchronous operation. The operation is started immediatly and there is no opportunity to defer starting the operation until later without deferring the call to the initiating function itself.

While this design does allow composition of lower-level asynchronous operations into higher-level asynchronous operations, doing so requires each composition of lower-level operations to be written separately in an imperative fashion. This style of API does not seem to easily support composition by using generic higher-order algorithms without first wrapping each initiating function in another function that returns an object that captures the parameters but defers calling the operation's initiating function until a continuation is attached.

For an example of this approach to wrapping the Networking TS APIs see Gor Nishanov's talk "Naked coroutines live (with networking)" from CppCon 2017.

We can potentially improve the composability of asynchronous APIs in the Networking TS if they are modified to return an asynchronous object that satisfies the Sender interface and that lazily starts execution when a continuation is later attached to the object rather than requiring a continuation to be provided immediately.

COMPOSABILITY OF PARALLEL ALGORITHMS

The C++17 standard introduced overloads of the standard library algorithms that accept an additional execution-policy parameter and that allow the algorithm to execute in parallel across multiple threads.

These algorithms are natural extensions to the existing APIs and its blocking semantics makes it easy for existing synchronous applications to take advantage of the parallelism of their CPUs with the simple addition of an extra execution-policy parameter to the existing standard-library algorithm calls.

However, with the introduction of executors and executor-customised execution policies like `std::par.on(e)` it now becomes possible to specify that the algorithm should run on an execution context other than the current execution context. This means that to avoid blocking the current thread while waiting for the algorithm to complete the parallel algorithms should ideally be exposed as an asynchronous operation rather than a synchronous blocking operation.

If the parallel algorithms can be exposed as asynchronous operations that have the same uniform interface used by other asynchronous operations then it becomes possible to use the same higher-order functions to compose parallel algorithms.

EFFICIENCY GAINS POSSIBLE THROUGH INVERTED OWNERSHIP MODEL

With the traditional callback-based asynchronous model used in both Networking TS and in the Sender/Receiver-based APIs the initiating function that accepts the callback cannot typically assume that the callback object passed to it will live beyond the call to the initiating function. This means that if the operation does not complete synchronously then the initiating function will typically need to take a copy of the callback object to ensure that it can be safely called when the operation does eventually complete.

In this model, the producer of the asynchronous result owns the consumer state and is responsible for ensuring the consumer stays alive until the operation completes. This is often implemented by placing the consumer state along with other per-operation state in heap-allocated storage. While the cost of this heap-allocation can often be amortised by allowing the caller to provide a custom allocator (e.g. a recycling allocator), the current API design of P0443 executors and the Networking TS does not expose the required size of the allocation and so the caller cannot in general pre-allocate enough memory to guarantee the operation will succeed without additional knowledge about the implementation.

However, there is an alternative model where the ownership model is inverted and instead, the consumer is made responsible for ensuring the producer remains alive until the operation completes.

This inverted ownership model is naturally and safely expressible when writing the consumer of an operation as a coroutine. The producer and per-operation state can be placed as a local variables in the coroutine consuming the operation and so when the coroutine awaits the operation the coroutine is suspended until the operation completes, naturally keeping both the consumer state (the coroutine frame) and the producer state (local variables) alive until the operation completes and the coroutine is resumed.

This model allows the compiler to statically guarantee that enough memory is reserved within the coroutine frame for the per-operation state required by an asynchronous operation and this can eliminate the need for an additional per-operation heap-allocation altogether in some cases. Note that there is often still a heap-allocaton present here, it’s just that it’s a single heap allocation for the coroutine frame which can then be reused for many individual operations initiated by that coroutine.

With this inverted ownership model it is possible to implement executors that can schedule a coroutine onto another execution context without needing to perform any additional heap allocations. This in turn can allow these operations to be implemented as noexcept, an important property for implementing certain classes of algorithms.

It is also possible to implement asynchronous networking APIs that make use of this inverted ownership model to avoid the need to heap-allocate per-operation state.

Working towards a unified asynchronous model for C++

There are potentially large benefits to adopting an asynchronous model that integrates executors, parallel algorithms and networking in a unified and composable way and that works efficiently with both callbacks and coroutines.

This paper sketches out a proposed design direction that attempts to tie these aspects together.

Please note that this paper is exploratory and concepts in here are still evolving.

Background

Sender/Receiver

The “Compromise executors proposal” paper P1194R0 was presented at the ad-hoc Executors meetings in Bellevue as a formalization of the concept of callbacks.

P1194R0 introduces the concepts of Sender and Receiver as fundamental building blocks of asynchronous operations. A Sender is a producer of a value and a Receiver is a generalisation of a callback that can receive a value, error or done signal.

The consensus at the Bellevue meeting was that the Sender/Receiver design was preferred as the long-term direction for Executors.

Receivers

A "receiver" is a concept that represents a callback or continuation that can be passed a value result, a done signal or an error result.

A receiver has three possible operations you can perform on it:

- `void op::set_value(Receiver& r, Values&&... values);`
- `void op::set_done(Receiver& r);`
- `void op::set_error(Receiver& r, Error&& error) noexcept;`

A void value result can be sent by calling `set_value()` with zero value arguments.

The protocol for calling these methods on a receiver is one of the following sequences:

1. A call to `set_value()` which returns normally followed by a call to `set_done()` which returns normally (the success case)
2. A call to `set_done()` which returns normally.
3. A call to `set_error()` with an error value.

A call to `op::set_done()` that returns, or a call to `op::set_error()`, which is always noexcept, terminates the sequence of calls to the receiver.

Note also that it is the caller's responsibility to ensure that two calls to the receiver methods from different threads do not overlap. The caller must wait until one call returns before calling a subsequent method on the receiver.

These operations are customisation points on the `Receiver` type. The default implementation of these methods will call onto the `.value()`, `.done()` and `.error()` member functions of the receiver.

SENDERS

A sender represents an asynchronous operation that produces either a single value result or an error result. The result of the asynchronous operation is obtained by attaching a receiver to the sender by calling `op::submit(sender, receiver)`.

The sender has a single customisation point:

- `void op::submit(Sender&& s, Receiver&& r) noexcept;`

The default implementation of `op::submit()` calls `s.submit(r)`. Although this operation can be customised for particular senders or sender/receiver pairs.

Once `op::submit()` has been called, the sender is responsible for ensuring that the `op::set_value()`, `op::set_done()` or `op::set_error()` method is called on the receiver passed to it once the result of the operation is available. The sender is also responsible for ensuring that the receiver passed to it remains alive until the operation is complete. This means that the sender may need to make a copy of the receiver by move/copy-construction if it will not be completed by the time the submit function returns.

Note that the actual computation/operation associated with a sender may have already been eagerly started or may be lazy and only start execution once the continuation has been attached by a call to `op::submit()`. The only way generic client code can guarantee the operation has been started is to call `op::submit()`.

CONCEPT DEFINITIONS

```
concept Receiver =
  MoveConstructible<T> &&
  requires (T& r)
  {
    op::set_done(r);
  };

template<typename T, typename... Values>
concept ValueReceiver =
  Receiver<T> &&
  requires(T& r, Values&&... values)
  {
    op::set_value(r, static_cast<Values&&>(values)...);
  };

template<typename T, typename Error>
concept ErrorReceiver =
  Receiver<T> &&
  requires(T& r, Error&& e)
  {
    { op::set_error(r, static_cast<Error&&>(e)) } noexcept;
  };

template<typename T, typename Error, typename... Values>
concept ReceiverOf =
  ErrorReceiver<T, Error> &&
```

```
    ValueReceiver<T, Values...>;

struct sender_tag {};

template<typename S>
struct sender_traits;

template<typename T>
concept Sender =
  MoveConstructible<std::remove_cvref_t<S>> &&
  requires { typename sender_traits<S>::sender_concept; } &&
  DerivedFrom<typename sender_traits<S>::sender_concept, sender_tag>;

template<typename S, typename R>
concept SenderTo =
   Sender<S> &&
   requires(S&& sender, R&& receiver)
   {
     op::submit(
       static_cast<S&&>(sender),
       static_cast<R&&>(receiver));
   };

template<template<template<class...> class,
                  template<class...> class> class>
struct __value_types;
template<template<template<class...> class> class>
struct __error_types;

template<typename T>
concept TypedSender =
   Sender<T> &&
   requires()
   {
     typename __value_types<sender_traits<T>::template value_types>;
     typename __error_types<sender_traits<T>::template error_types>;
   };

template<typename T>
struct __one_value_only { using type = T; };

template<typename... Ts>
struct __zero_or_one_value {};
template<>
struct __zero_or_one_value<> { using type = void; };
template<typename T>
struct __zero_or_one_value<T> { using type = T; };

template<typename T>
concept SingleTypedSender =
  TypedSender<T> &&
  requires()
  {
    typename sender_traits<T>
      ::template value_types<__one_value_only, __zero_or_one_value>
      ::type::type;
```

```
  };
template<typename T>
using sender_value_type_t =
  typename sender_traits<T>
    ::template value_types<__one_value_only, __zero_or_one_value>
    ::type::type;

template<typename... Ts>
struct __is_exception_ptr_or_empty : std::false_type {};
template<>
struct __is_exception_ptr_or_empty<std::exception_ptr>
: std::true_type {};
template<>
struct __is_exception_ptr_or_empty<> : std::true_type {};

// Query whether the sender sends only exceptions via error channel.
template<typename T>
concept ExceptionErrorSender =
  TypedSender<T> &&
  sender_traits<T>::
    template error_types<__is_exception_ptr_or_empty>::value;
```

Awaitable concepts and coroutines

With the coroutines language feature as specified in the Coroutines TS we can write asynchronous code that looks sequential. The coroutine body is divided up into parts that execute sequentially in-between suspension points. Suspension points are identified by the `co_await` and `co_yield` keywords.

Whereas, with the Sender/Receiver concepts, the Receiver represents a callback or continuation, with coroutines, the coroutine itself represents the continuation. When a coroutine co_awaits some type that satisfies the Awaitable concept, the coroutine is suspended and it submits itself as the continuation to the operation by calling the await_suspend() method.

For a deeper understanding of the Awaitable concepts please see paper [P1288R0](P1288R0) or the blog post "Understanding operator co_await."[1]

Paper P1288R0 proposes to add some new concept definitions and template meta-functions to the standard library which we will reference here. The important facilities from P1288R0 are:

```
template<typename T>
concept Awaitable = ...; // See P1288R0 for definition

template<typename T>
using await_result_t = ...; // See P1288R0 for definition
```

A comparison of Sender/Receiver and Awaitable/Coroutine

[1] https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await

Both the Sender/Receiver and Awaitable/Coroutine abstractions can represent an asynchronous operation that eventually produces a result, but there are some differences in how they work.

Sender/Receiver is explicitly a callback model where results are pushed to the consumer by calling the callback. Awaitable/Coroutine is a model that can operate in either a pull or a push model. From the consumer side, the code looks like a pull model.

The Sender concept is more flexible in several aspects:

- A sender can produce a variadic number of values rather than a single value whereas an awaitable would need to wrap up multiple values into a single return-type, e.g. using a std::tuple
- A sender can produce different types of values and dispatch to different overloads of receiver.value() to handle different cases, whereas an awaitable would need to type-erase different types into a single return-type. e.g. using a std::variant.
- The type of the value being produced does not need to be known a priori. We can construct a sender that defers instantiation of the calls to the receiver until later when some method is called and is given a concrete type. With an Awaitable, the result-type of the operation is equal to the return-type of the await_resume() method and so needs to be known statically ahead of time.

This means that we need to further constrain the Sender concept if we are to find something that can be considered equivalent to an Awaitable type. The Sender needs to be able to report statically the type of value that it will send and the value either needs to have arity-0 (the void case) or arity-1 (the single-value case). We can define this refinement of the Sender concept to be a SingleTypedSender.

A correspondence table comparing SingleTypedSender/Receiver and Awaitable/Coroutine:

Sender/Receiver	**Awaitable/Coroutine**
SingleTypedSender	Awaitable
Receiver	Coroutine
sender_value_type_t<SingleTypedSender>	await_result_t<Awaitable>
op::submit(task, receiver);	co_await task;
op::set_error(receiver, e);	coroutine_handle::resume() + T await_ready() { std::rethrow_exception(e); }
op::set_value(receiver, v);	coroutine_handle::resume() + T await_ready() { return v; }

op::set_value(receiver, a, b, c);	coroutine_handle::resume() +
	std::tuple<A, B, C> await_resume() {
	  return std::make_tuple(a, b, c);
	}

if (cond) {
  op::set_value(receiver, a);
} else {
  op::set_value(receiver, b);
}
	coroutine_handle::resume() +
	std::variant<A, B> await_resume() {
	  if (cond) { return a; }
	  else { return b; }
	}

Lifetime of consumer (receiver) owned by the producer (sender)	Lifetime of producer (awaitable) owned by the consumer (coroutine)
Execution context propagated from producer to consumer	Execution context propagated from consumer to producer

ADAPTING SINGLETYPEDSENDERS TO AWAITABLES AND AWAITABLES TO SINGLETYPEDSENDERS

As SingleTypedSender and Awaitable types are effectively duals of each other, we can write adapters that convert between them.

For example, we can write an `operator co_await()` for a `SingleTypedSender` that allows us to `co_await` the `SingleTypedSender` object from within the coroutine.

This allows us to write a `SingleTypedSender` class and then consume that from within a coroutine. e.g.

```
template<typename T>
struct delayed {
  std::chrono::milliseconds delay;
  T value;

  template<template<typename... Ts> class Variant,
           template<typename... Ts> class Tuple>
  using value_types = Variant<Tuple<T>>;

  template<template<typename... Es> class Variant>
  using error_types = Variant<std::exception_ptr>;

  template<ReceiverOf<std::exception_ptr, T> R>
  void submit(R&& receiver) && noexcept;
};

task<> usage_example(delayed<std::string> op)
{
  // Calls the default operator co_await() for SingleTypedSender args.
  // This in turn calls op.submit().
  std::string s = co_await std::move(op);
}
```

And similarly, we can implement the `op::submit()` customisation point for an arbitrary `Awaitable` type that allows us to attach a `Receiver` to the `Awaitable` object as if it were a `Sender`.

```
task<std::string> get_message(int messageId);

struct cout_receiver {
  template<typename T>
  void value(T value) { std::cout << value << "\n"; }

  void error(std::exception_ptr ePtr) {
    try {
       std::rethrow_exception(ePtr);
    }
    catch (const std::exception& e) {
      std::cout << "error: " << e.what() << "\n";
    }
    catch (...) {
     std::cout << "error\n";
```

```
    }
  }

  void done() {}

};

void usage_example() {
  op::submit(get_message(123), cout_receiver{});
}
```

See Appendix A for implementations of the Awaitable/Sender adapters.

Exploring the ownership models of Senders and Awaitables

One of the key differences between Senders and Awaitables is their ownership model.

When `op::submit()` is called with a sender and receiver it cannot assume that either the receiver or the sender objects will continue to exist after the call to `submit()` returns. This means it will generally need to take a copy of the `Receiver` object and of any state from the `Sender` object needed to perform the operation.

This will often then be wrapped up in another `Receiver` object that is then passed to the next `Sender` in the pipeline. At the terminal nodes in the pipeline, a `Sender` will typically need to type-erase and heap-allocate the copy of the `Receiver`. In many cases, the terminal point in the pipeline is typically some sort of executor which schedules the operation and the executor needs to be able to add different types of Receiver objects into a single queue of pending work.

This means that there is often only a single heap-allocation that is needed to store state for a given pipeline's continuation chain.

However, when using a coroutine, the ownership model is reversed. An `Awaitable` object is typically allocated as a local variable within the coroutine frame of the consumer and so the coroutine frame keeps the `Awaitable` object alive while it is suspended waiting for the operation to complete.

A pipeline of operations is typically represented within coroutines by calls to nested coroutine functions where the calling coroutine owns the lifetime of the called coroutine. In this model, assuming the compiler can elide[2] the coroutine frame allocations of nested calls, the entire pipeline can be optimised to a single heap allocation for the top-level coroutine.

In both cases, each pipeline ends up with a single heap allocation. With coroutines the single heap allocation is performed by the top-level consumer before starting the operation and with senders the heap allocation is performed by the leaf-level producer at the time the operation is started.

As a result of these differences in ownership models, this means that every time we adapt from one model to the other model that we necessarily incur a heap allocation.

- When adapting an Awaitable to a Sender we need to allocate a new coroutine frame that holds ownership of the Awaitable and Receiver.
- When adapting a Sender as an Awaitable, the Sender will end up heap allocating a copy of the receiver because it cannot assume that the receiver will outlive the call to `op::submit()`, even though in this case the coroutine

[2] P0981R0 - "Halo: coroutine Heap Allocation eLision Optimization" (Gor Nishanov, Richard Smith)

frame awaiting the Sender will keep both the temporary Receiver and the Sender alive until the operation completes.

UNIFYING AWAITABLE/SENDER INTO A TASK CONCEPT

One of the issues with implementing asynchronous operations as either a Sender or an Awaitable is that regardless of which one you implement, you will require an adapter to use it in the opposite context and this adapter will incur an extra heap-allocation.

However, this then leads us to ask the question, "Can we eliminate this overhead by implementing both the Awaitable and Sender interfaces on the same type?". The answer is "yes, we can!"

We define a new concept, tentatively named `Task`, that requires the type to implement both `SingleTypedSender` and `Awaitable`.

```
template<typename T>
concept Task =
  Awaitable<T> &&
  SingleTypedSender<T> &&
  ExceptionErrorSender<T> &&
  ConvertibleTo<sender_value_type_t<T>, await_result_t<T>>;

template<typename T>
using task_result_t = await_result_t<T>;

template<typename T, typename Result>
concept TaskOf =
  Task<T> &&
  ConvertibleTo<task_result_t<T>, Result>;
```

When this concept is used in conjunction with the default adapters from the previous section, this effectively means that we can define a `Task` type by either implementing the `Awaitable` concept (i.e. by defining an `operator co_await()`) or by implementing the `SingleTypedSender` concept (i.e. by defining a `submit()` method) or by implementing both.

If the type defines only the `Awaitable` interface then when consuming the `Task` from a coroutine then its `operator co_await()` will be called and this should generally have minimal overhead. However, when consuming such a type using the Sender/Receiver interface, it would fall back to the default implementation of `op::submit()` for Awaitable types which uses an adapter (with some overhead).

For example: A simple implementation of the ‘Transform’ adapter that applies a function to the result and yields the result.

```
template<Awaitable Inner, Invocable<await_result_t<Inner>> Func>
class transform_op {
  Inner inner;
  Func func;

  std::task<std::invoke_result_t<Func, await_result_t<Inner>>>
  operator co_await() &&
  {
    co_return std::invoke(
```

```
      static_cast<Func&&>(func),
      co_await static_cast<Inner&&>(inner));
  }
};
```

If we wanted to eliminate the overhead of the adapter when consuming this operation through the Sender/Receiver API then we can also implement the `submit()` method on this class.

```
template<Task Inner, Invocable<task_result_t<Inner>> Func>
class transform_op {
  Inner inner;
  Func func;

  using value_type =
    std::invoke_result_t<Func, task_result_t<Inner>>;

  template<template<typename...> class Variant,
           template<typename...> class Tuple>
  using value_types = Variant<Tuple<value_type>>;

  template<template<typename...> class Variant>
  using error_types = Variant<std::exception_ptr>;

  std::task<value_type> operator co_await() && {
    co_return std::invoke(
      static_cast<Func&&>(func),
      co_await static_cast<Inner&&>(inner));
  }

  template<ReceiverOf<std::exception_ptr, value_type> R>
  void submit(R&& receiver) && noexcept {
    struct wrapped_receiver {
      Func func;
      std::remove_cvref_t<R> receiver;

      void value(sender_value_type_t<Inner> value) {
        op::set_value(
          receiver,
          std::invoke(static_cast<Func&&>(func),
                      static_cast<decltype(value)&&>(value)));
      }

      void done() {
        op::set_done(receiver);
      }

      void error(std::exception_ptr e) noexcept {
        op::set_error(receiver, std::move(e));
      }
    };

    op::submit(
      static_cast<Inner&&>(inner),
      wrapped_receiver{std::move(func), std::forward<R>(receiver)});
  }
```

```
};
```

This now allows this implementation of the transform adapter operation to have an efficient implementation regardless of whether the operation is consumed from a coroutine or consumed via `op::submit()`.

It does mean that you need to provide two different implementations of the operation for the different async result delivery mechanisms, but in cases where performance matters, this could be an acceptable tradeoff.

In cases where performance is not critical you can implement just one of these and the other variant will still be available through the adapters.

SEPARABILITY FROM COROUTINES

This design is also separable from the Coroutines TS in the case that adoption of coroutines into the language is delayed. Developers can initially write code in terms of SingleTypedSender and implement the op::submit() operation.

Then later, when coroutines become available, we can define a default `operator co_await()` for all `SingleTypedSender` types and we can then start using the `Task` concept. All types that have implemented the `SingleTypedSender` concept will automatically become `Task` types since they will also be `Awaitable`.

DESIGN TRADEOFFS

Forcing a `SingleTypedSender` to be used pessimises use-cases of Sender that would otherwise allow handling of different types to be handled with inline code. E.g. having different overloads of value() for different types would allow static dispatch to the right code-path for handling each potential value type.

It's possible, with future evolution of the design of coroutines, that we could generalise a coroutine to allow a co_await expression to also resume with different types depending on the type of the value produced by the producer. This would reduce the impedence mismatch between a Receiver and a Coroutine but could require a non-trivial design extension to the Coroutines TS design to support it.

EXECUTORS

Recent discussions on the Executors calls and mailings have been centred around defining something like `make_value_task(ex, predecessor, f)` as being (one of) the fundamental primitives of an Executor.

However, it seems cumbersome to have to pass in empty functions or 'null_sender' as predecessor to extract the desired behaviour. We realised that the `make_value_task()` operation can actually be decomposed into several individual pieces: waiting for a dependency to complete, switching execution contexts and applying a transform to the result.

Ideally we could define a simpler (preferably single) fundamental primitive that executors can implement that could then be composed into these higher-level operations.

We would also like to allow an executor to be able to efficiently schedule execution when used either under sender/receiver or under coroutines. It is possible to implement a zero-allocation, noexcept executor schedule operation when used under coroutines. See cppcoro::static_thread_pool, cppcoro::io_service for examples. It would be nice to come up with a design for executors that allows these kinds of executor implementations to be used within coroutines.

THE SCHEDULE() OPERATION AS THE FUNDAMENTAL PRIMITIVE

This paper proposes that the interface for an Executor should be to have a single `.schedule()` method that returns a `TaskOf<Executor>`.

This is a change from the earlier model to make an Executor a factory of Sender/Task rather than being a Sender/Task itself.

```
template<typename T>
concept Executor =
  CopyConstructible<T> &&
  std::is_nothrow_move_constructible_v<T> &&
  requires(T executor)
  {
    // Ideally TaskOf<Executor>.
    // Unfortunately we can't define concepts to be recursive.
    { executor.schedule() } -> Task;
  };
```

This allows us to then simply context-switch a coroutine from one executor by co_awaiting the Task returned from `executor.schedule()`:

```
template<Executor E>
task<> foo(E executor)
{
  // Initially executing on whatever execution context
  // the caller co_awaited this task on.

  // Switch executors
  Executor subExecutor = co_await executor.schedule();

  // Now executing on subExecutor (possibly a different type than E)

  // This will resume the coroutine that was awaiting
  // foo() inline on subExecutor.
}
```

With the `schedule()` operation returning a Task that is able to support both coroutines and sender/receiver natively, it becomes possible to then implement an Executor that can reschedule a coroutine onto its execution context with no memory allocations and that is noexcept.

See example code unifex.tar.bz attached to the LEWG wiki for a proof-of-concept of this design. See also `cppcoro::static_thread_pool::schedule()` and `cppcoro::io_service::schedule()` for coroutine implementations of zero-allocation, noexcept implementations of executor `schedule()` operations.

WHY NOT MAKE EXECUTOR A TASK ITSELF?

By making Executor a factory for Tasks rather than a Task itself, we open the door for extending the Executor concepts further to be factories of other kinds of tasks which take parameters.

The op::submit() customization point does not currently take extra arguments (although it could potentially be extended to do so). However, the operator co_await() customization point cannot be extended to support additional arguments. By currying any additional arguments into a Task object via a factory method we allow the Executor concept to be later refined to add support for returning different kinds of scheduling operations which may require additional parameters while retaining a uniform interface for attaching continuations via either op::submit() or operator co_await().

For example, we can extend the Executor concept to a TimedExecutor concept that allows scheduling work at or after a specific time:

```
template<typename T>
concept TimedExecutor =
  Executor<T> &&
  Regular<typename T::time_point> &&
  requires(T executor, typename T::time_point time)
  {
    { executor.now() } -> T::time_point;
    { executor.schedule_at(time) } -> Task; // TaskOf<TimedExecutor>
  };
```

We can imagine other executor concepts that may extend the scheduling parameters, eg. to include support for prioritization or cancellation.

An Executor also has different semantics from Task with regards to copyability. A Task is only guaranteed to be move-only and can only be submitted or awaited once (it only guarantees that the type Task&& is Awaitable, not that Task& is Awaitable). Whereas a single Executor is expected to be able to schedule and submit multiple units of work.

BUILDING HIGHER-LEVEL OPERATORS

Once we have the fundamental .schedule() primitive, we can build higher-level algorithms and/or operators on top of these.

For example, the one-way execute operation:

```
template<Executor E, Invocable Func>
void oneway_execute(E&& executor, Func&& f)
{
  std::invoke([](std::remove_cvref_t<E> executor,
                 std::remove_cvref_t<Func> f) -> oneway_task {
    co_await executor.schedule();
    std::invoke(std::move(f));
  }, static_cast<E&&>(executor), static_cast<Func&&>(f));
}
```

This is implemented in terms of a coroutine. It could equivalently have been implemented in terms of the op::submit() interface.

Other operators can also be implemented in terms of the `executor.schedule()` interface:

- `on(Executor, TaskOf<T>) -> TaskOf<T>`
- `via(Executor, TaskOf<T>) -> TaskOf<T>`
- ```
  make_value_task(E executor, P predecessor, F func)
    -> TaskOf<std::invoke_result_t<F, task_result_t<P>>>;
  ```

See Appendix B for example implementations of some of these operations.

Note that we are not suggesting that these operators are necessarily the set of operators that we should provide in the standard library – that should be up for discussion. Only that these operators are ones that have been discussed in Executor calls previously and that it is possible to provide default implementations of these operators in terms of an `executor.schedule()` primitive.

ALGORITHMS AS CUSTOMISATION POINTS

While we are able to define default implementations of these operators and algorithms in terms of `Executor::schedule()`, for some executors it may be possible to provide more efficient implementations of these operators.

The idea of making algorithms customization points is also discussed in [P1232](). The general idea is for the standard library to provide default implementations of algorithms that accept executors (or an execution_policy) but allow specific executors to customize these to provide implementations that are more efficient.

As some standard library algorithms may be implemented in terms of other standard library algorithms, executors may not need to customize all algorithms to benefit from more efficient implementations for that executor. eg. many algorithms may be implemented in terms of parallel-for-each or parallel-accumulate and so by customizing those algorithms for an executor, the other algorithms built on them would also benefit.

It is still an open question which set of algorithms should be customizable here. Should all algorithms that accept an execution policy be a customization point?

PARALLEL ALGORITHMS

Key points for parallel algorithms:

- Parallel algorithms parameterised on an executor or executor-bound execution policy should be asynchronous
  - This allows chaining and pipelining execution without needing a transition back to the host device between each algorithm invocation
- Blocking versions of parallel algorithms should be defined in terms of `sync_wait()` on the asynchronous versions, when passed a 'parallel' execution policy with a bound executor.
- Parallel algorithms should have default implementations that are defined in terms of `executor.schedule()`
  - This allows an author of an executor to only need to implement a single basis operation and still get the benefit of usability with all of the parallel algorithms.
- Parallel algorithms should be customisable by executors that can provide more efficient implementations of an algorithm than the generic version in terms of `executor.schedule()`
  - For example, a GPU executor can very efficiently implement parallel-`for_each` using an GPU kernel
  ```

- Consider adding a new `std::unseq` policy that allows unsequenced execution in a single thread without introducing parallelism.
    - This allows executor-specific implementations to still be able to leverage SIMD optimisations for intra-thread concurrency while letting the executor customise inter-thread concurrency.

Default implementations of parallel algorithms in terms of Executor concept

Once we have an `Executor.schedule()` operation, we can also use that as a basis operation to implement generic/default versions of parallel algorithms.

Example: The following code snippet shows how you could implement a generic parallel `for_each()` in terms of an arbitrary executor object that implements the Executor concept.

```
template<
  Executor Ex,
  RandomAccessIterator Iter,
  Sentinel<Iter> EndIter,
  Invocable<iter_value_type_t<Iter>> Func)
std::task<void>
for_each(Ex executor, Iter it, EndIter itEnd, Func func)
{
  using difference_type =
    typename std::iterator_traits<Iter>::difference_type;
  const difference_type count = std::distance(it, itEnd);

  // TODO: Query executor for max concurrency instead
  auto workerCount = std::min(
    std::thread::hardware_concurrency(),
    count);

  // Fall back to sequential if there is no concurrency possible.
  if (workerCount < 2) {
    std::for_each(it, itEnd, std::ref(func));
    co_return;
  }

  std::atomic<bool> interrupted = false;
  std::atomic<difference_type> next = 0;
  std::atomic<bool> startedParallel = false;

  auto makeWorker = [&]() -> std::task<void> {
    // Schedule execution onto the executor.
    co_await executor.schedule();

    startedParallel.store(true, std::memory_order_relaxed);

    try {
      difference_type chunkSize = 1;
      while (!interrupted.load(std::memory_order_relaxed))
      {
        // Acquire a chunk of work.
        auto offset = next.load(std::memory_order_relaxed);
        do {
          if (offset >= count) co_return;
          const auto remaining = count - offset;
```

```
        if (chunkSize > remaining) chunkSize = remaining;
      } while (!next.compare_exchange_weak(
                   offset,
                   offset + chunkSize,
                   std::memory_order_relaxed));

      // Time how long the chunk of work took so we can
      // adjust our chunk size for next time.
      auto start = std::chrono::high_resolution_clock::now();

      auto chunkIt = it + offset;
      const auto chunkEnd = chunkIt + chunkSize;
      std::for_each(chunkIt, chunkEnd, std::ref(func));

      auto end = std::chrono::high_resolution_clock::now();

      auto chunkTime = (end - start);

      // Try to keep next chunk of work to about 10ms
      if (chunkTime < 5us) {
        chunkSize *= 100;
      } else if (chunkTime < 1ms) {
        chunkSize *= 10;
      } else if (chunkTime > 15ms && chunkSize > 1) {
        chunkSize /= 2;
      }
    }
  } catch (...) {
    interrupted.store(true, std::memory_order_relaxed);
    throw;
  }
};

try {
  std::vector<std::task<void>> workers;
  workers.reserve(workerCount);
  while (workerCount-- > 0) {
    workers.emplace_back(makeWorker());
  }

  co_await when_all(std::move(workers));
  co_return;
} catch (...) {
  if (startedParallel.load(std::memory_order_relaxed)) throw;
}

// Failed to launch execution on specified executor.
// Fall back to sequential execution.
std::for_each(it, itEnd, std::ref(func));
}
```

Note that even this representation of a parallel algorithm is still not ideal as it requires that the algorithm finish processing the entire input range before the next algorithm in the chain can start executing. Thus we expect that a more fundamental basis operation for parallel algorithms on ranges is possible that allows stream-processing of chunks of the input range in a pipeline of algorithms. This is an area of future research.

Example: Parallel accumulate using recursive divide and conquer (example from P1316R0)

```
template<
  typename Executor, typename Iter, typename Sentinel,
  typename T, typename BinaryOp>
task<T> accumulate(Executor executor, Iter begin, Sentinel end,
                   T init, BinaryOp op, bool schedule = false)
{
  if (schedule) co_await executor.schedule();

  auto count = std::distance(begin, end);
  if (count < 512) {
    // Below some threshold just run single-threaded version
    co_return std::accumulate(begin, end, init, op);
  } else {
    // Divide range into two halves
    auto mid = begin + (count / 2);

    auto [left, right] = co_await when_all(
      accumulate(executor, begin, mid, init, op, true),
      accumulate(executor, mid + 1, end, *mid, op, false));

    co_return op(left, right);
  }
}
```

CUSTOMISING ALGORITHMS FOR AN EXECUTOR

Algorithms such as parallel for_each() and accumulate() would then ideally be customized for execution on a GPU to make use of the GPU-specific APIs and programming models to provide the most-efficient implementation when passed a GPU execution policy.

The GPU implementations of these algorithms would be free to return GPU-specific Task-types that allow the GPU executor to again provide a more efficient implementation when composing these tasks together into a pipeline.

```
class cuda_executor {
public:
  struct schedule_task;

  schedule_task schedule();

private:
  ...
};

template<typename T>
class cuda_task {
public:
  cuda_awaiter operator co_await() &&;

  template<ReceiverOf<T> R>
  void submit(R receiver) &&;
```

```
  friend T sync_wait(cuda_task t)
  {
    cudaStreamSynchronize(t.stream);
    return std::move(*t.result);
  }

private:
  cudaStream_t stream;
  cudaEvent_t event;
  T* result;
};

template<typename T, typename F>
cuda_task<std::invoke_result_t<F, T>>
make_value_task(
  cuda_executor ex,
  cuda_task<T> predecessor,
  F func) {
  // - Create new cudaEvent_t
  // - Create new cudaStream_t
  //   - Add wait for new cudaEvent_t

  //   - Add execution of make_kernel<<<1,1>>>(func,
  //                                            &predecessor.result);
  //   - Add signaling of new cudaEvent_t to predecessor.stream
  // - Return new cuda_task wrapping new event/stream.
}

template<typename T, typename Func>
cuda_task<void> for_each(
  cuda_executor ex,
  cuda_task predecessor,
  T* iter, T* iterEnd, Func func)
{
  // - Create new cudaEvent_t
  // - Create new cudaStream_t
  //   - Add wait for new cudaEvent_t
  //   - Add execution of make_kernel<<<128, 128>>>(func)
  //   - Add signalling of new cudaEvnet_t to predecessor.stream
  // - return new cuda_task wrapping new event/stream
}
```

NETWORKING TS

- The Networking TS design currently defines a "Universal Async Model" in terms of a generic CompletionHandler parameter to all async methods.
- CompletionHandler represents a combination of:
  - The continuation
  - An allocator for allocating state for the operation
  - An executor to call the continuation on
- When you call an async function the operation is started immediately and must be provided with a continuation.
- If you don't have a final continuation yet then you still need to provide a continuation internally. There are mechanisms within the CompletionHandler design that allow you to pass a placeholder, like std::use_future that manufactures a continuation that stores the result into some shared state and returns a std::future.
  - This means it will often require a heap allocation for the shared state, and also require synchronisation to arbitrate the race between eventually attaching the final continuation to the std::future and the result becoming available.
- If we were to build a similar std::use_await object we can pass into the CompletionHandler and have it return an Awaitable that could then be co_awaited then this would still need to heap-allocate and synchronise.
- It's possible to wrap every async method in the Networking TS in an awaitable object that defers the call to the initiating method until `await_suspend()` is called. See Gor Nishanov's CppCon 2017 talk for more details.
  - This solution is functional but cumbersome and duplicates the APIs.
  - There must be a better way.
- What if instead we had each of the `xxx_async()` functions in the Networking TS return an object that satisfied the `Task` concept instead?
  - Then it could be implemented to be synchronisation-free and allocation-free.
  - There would be no need to pass in an executor as a parameter to the function as the returned object can be adapted with the `via()` operator by the caller to schedule onto a particular execution context – the default would be to resume/execute the continuation on the `io_context` associated with the socket.
  - The async operations from the Networking TS would be composable with other operations/adapters built on top of Tasks and Executors using higher-level generic operators/algorithms like `when_all()`.
  - The async methods could then be consumed naturally from a coroutine with minimal overhead and without having to duplicate/wrap the APIs.
  - Existing callback-based code can still be accommodated by using the `op::submit()` extension point and building a Receiver that wraps the callback. This could be wrapped up in a helper function. eg. `op::submit_callback()`
    - Example:
      ```
      op::submit_callback(
        socket.read_some(buffer, size), callback);
      ```
    - This would be just as efficient as the equivalent code that uses `CompletionHandlers`.
      Example:
      ```
      socket.read_some(buffer, size, callback);
      ```

OPEN QUESTIONS

- Is `Sender` the right corresponding concept for `Awaitable`?
  Sender represents a sequence of either 0 or 1 elements whereas `Awaitable` always yields exactly a single element. This presents somewhat of an impedence mismatch. There may be a simpler concept that omits the `set_done()` operation and just supports `set_value()` and `set_error()`.
- Perhaps `Sender`, being a specialisation of the more-general ManySender concept, should have a correspondence to some kind of AsyncRange concept rather than to Awaitable.
- What constraints should we put on the relationship between the types that a Sender sends and the result-type of the `await_result_t` of the awaitable code-path?
  - The return value of `await_resume()` can only support a single value of a single type.
  - A Sender can support a number of different kinds of arities:
    - It can produce a variable number of values (in this case 0 or 1)
    - It can produce different types of values (eg. either type `T` or type `U`)
    - It can produce a value consisting of multiple parameter values
  - Should the Sender only support sending a single value argument?
    Or zero arguments for the `void` case.
  - Should the Sender only support sending a single value type?
    - Should this type be the same as `await_result_t`?
    - If we require it send a single value argument then perhaps we can just require that the value type it sends is convertible to the `await_result_t` type?
- How should we allow the caller to customise heap allocations that may be needed by an executor/Task implementation?

Appendix A – Adapting Sender and Awaitable

An implementation of operator co_await() for an arbitrary TypedSender that calls op::submit() on the sender, stores the result in an Awaiter object and then results the coroutine and returns the result from await_resume().

```
// This type implements both Awaiter and Receiver interfaces
template<SingleTypedSender S, typename T>
class sender_awaiter {
  S&& sender;
  std::experimental::coroutine_handle<> continuation;
  std::exception_ptr err;
  std::optional<T> value;

public:
  explicit sender_awaiter(S&& sender) noexcept : sender(sender) {}

  bool await_ready() noexcept { return false; }

  void await_suspend(std::experimental::coroutine_handle<> h) {
    continuation = h;
    op::submit(static_cast<S&&>(sender), std::ref(*this));
  }

  T await_resume() {
    if (err) std::rethrow_exception(err);
    else if (!value) throw operation_cancelled{};
    return std::move(*value);
  }

  template<typename U>
  void value(U&& v)
      noexcept(std::is_nothrow_constructible_v<T, U>) {
    value.emplace(static_cast<U&&>(v));
  }

  void error(std::exception_ptr e) noexcept {
    err = std::move(e);
    continuation.resume();
  }

  void done() noexcept {
    continuation.resume();
  }
};

template<SingleTypedSender S>
auto operator co_await(S&& sender) {
  return sender_awaiter<S, sender_value_type_t<S>>{
    static_cast<S&&>(sender)
  };
}
```

An implementation of the `op::submit()` customisation point for arbitrary Awaitable types. It is implemented by co_awaiting the Awaitable object in a new `oneway_task` coroutine and then forwarding the result on to the receiver passed to `submit()`.

```
// Fire-and-forget, eager coroutine type
struct [[maybe_unused]] oneway_task {
  struct promise_type {
    oneway_task get_return_object() noexcept { return {}; }
    suspend_never initial_suspend() noexcept { return {}; }
    suspend_never final_suspend() noexcept { return {}; }
    void return_void() noexcept {}
    void unhandled_exception() noexcept { std::terminate(); }
  };
};

template<Awaitable A, ReceiverOf<await_result_t<A>> R>
void submit(A&& awaitable, R&& receiver) noexcept {
  try {
    // Take care to move/copy the awaitable and only if that
    // succeeds do we try to then copy the receiver. This allows
    // us to deliver the receiver
    std::invoke([](std::remove_cvref_t<A> awaitable,
                   R&& receiver) -> oneway_task {
      try {
        std::remove_cvref_t<R> receiverCopy{
          static_cast<R&&>(receiver) };
        try {
          if constexpr (std::is_void_v<await_result_t<A>>) {
            co_await std::move(awaitable);
            op::set_value(receiverCopy);
          } else {
            op::set_value(receiverCopy,
                          co_await std::move(awaitable));
          }
          op::set_done(receiverCopy);
        } catch (...) {
          // Handle failures awaiting the result, calling
          // set_value() or set_done().
          op::set_error(receiverCopy, std::current_exception());
        }
      } catch (...) {
        // Handle failures copying the receiver
        op::set_error(receiver, std::current_exception());
      }
    }, std::forward<A>(awaitable), std::forward<R>(receiver));
  } catch (...) {
    // Handle failures creating the coroutine-frame or copying
    // the awaitable object.
    op::set_error(receiver, std::current_exception());
  }
}
```

APPENDIX B – EXECUTOR OPERATOR EXAMPLES

Example implementation of the via(executor, task) operator using Executor.schedule()

```
template<Executor E, Task T>
class via_task {
  E executor;
  T task;

  template<typename V, Receiver R>
  struct value_receiver {
    V value;
    R receiver;

    template<typename SubExecutor>
    void value([[maybe_unused]] SubExecutor subExecutor)
    {
      op::set_value(receiver, std::forward<V>(value));
    }

    void done()
    {
      op::set_done(receiver);
    }

    template<typename Error>
    void error(Error&& err) noexcept
    {
      op::set_error(receiver, std::forward<Error>(err));
    }
  };

  template<typename Error, Receiver R>
  struct error_receiver {
    Error error;
    R receiver;

    template<typename SubExecutor>
    void value([[maybe_unused]] SubExecutor subExecutor)
    {
      op::set_error(receiver, std::move(error));
    }

    void done()
    {
      op::set_done(receiver);
    }

    template<typename ScheduleError>
    void error(ScheduleError&& scheduleError) noexcept
    {
      op::set_error(receiver,
                    std::forward<ScheduleError>(scheduleError));
    }
  };
```

```
template<Receiver R>
struct done_receiver {
  R receiver;

  template<typename SubExecutor>
  void value([[maybe_unused]] SubExecutor subExecutor)
  {}

  void done()
  {
    op::set_done(receiver);
  }

  template<typename ScheduleError>
  void error(ScheduleError&& error) noexcept
  {
    op::set_error(receiver, static_cast<ScheduleError&&>(error));
  }
};

template<Receiver R>
struct task_receiver {
  E executor;
  R receiver;
  bool valueCalled = false;

  template<typename Value>
  void values(Value value)
  {
    op::submit(executor.schedule(),
               value_receiver<Value, R>{std::move(value),
                                        std::move(receiver)});
    valueCalled = true;
  }

  void done()
  {
    if (!valueCalled) {
      op::submit(executor.schedule(),
                 done_receiver<R>{std::move(receiver)});
    }
  }

  template<typename Error>
  void error(Error error) noexcept
  {
    try {
      op::submit(executor.schedule(),
                 error_receiver<Error, R>{std::move(error),
                                          std::move(receiver)});
    } catch(...) {
      op::set_error(receiver, std::current_exception());
    }
  }

};
```

```
public:

  via_task(E executor, T task)
  : executor(std::move(executor))
  , task(std::move(task))
  {}

  // Awaitable implementation for coroutines
  std::task<await_result_t<T>> operator co_await() &&
  {
    std::exception_ptr err;
    bool scheduleAttempted = false;
    try {
      auto&& awaiter = get_awaiter(std::move(task));
      auto&& result = co_await awaiter;
      scheduleAttempted = true;
      co_await executor.schedule();
      co_return static_cast<decltype(result)&&>(result);
    } catch (...) {
      err = std::current_exception();
    }
    if (!scheduleAttempted) {
      // Try to deliver the error on the specified execution context.
      co_await executor.schedule();
    }
    std::rethrow_exception(err);
  }

  // op::submit() implementation
  template<Receiver R>
    requires SenderTo<T, R>
  void submit(R receiver) && noexcept {
    try {
      op::submit(std::move(task),
                 task_wrapper<R>{std::move(receiver)});
    } catch (...) {
      op::set_error(receiver, std::current_exception());
    }
  }
};

template<Executor E, Task T>
via_task<E, T> via(E executor, T task) {
  return via_task<E, T>{ std::move(executor), std::move(task) };
}
```

Example implementation of transform(task, func)

```
template<Task T, Invocable<task_result_t<T>> F>
struct transform_task {
  T task;
  F func;

  std::task<std::invoke_result_t<F, task_result_t<T>>>
  operator co_await() && {
```

```
    co_return std::invoke(static_cast<F&&>(func),
                          co_await static_cast<T&&>(task));
  }

  template<Receiver R>
  struct wrapped_receiver {
    F func;
    R receiver;

    template<typename Value>
    void value(Value&& v) {
      op::set_value(receiver,
                    std::invoke(std::move(func),
                                static_cast<Value&&>(v)));
    }

    void done() { op::set_done(receiver); }

    template<typename Error>
    void error(Error&& error) noexcept {
      op::set_error(receiver, static_cast<Error&&>(error));
    }
  };

  template<Receiver R>
  void submit(R receiver) noexcept {
    try {
      op::submit(std::move(task),
                 wrapped_receiver<R>{ std::move(func),
                                      std::move(receiver) });
    } catch (...) {
      op::set_error(receiver, std::current_exception());
    }
  }
};

template<Task T, Invocable<task_result_t<T>> F>
auto transform(T task, F func)
{
  return transform_task<T, F>{std::move(task), std::move(func)};
}
```

Finally, an example make_value_task() operation can be implemented by composing via() and transform()

```
template<Executor E, Task T, Invocable<task_result_t<T>> F>
auto make_value_task(E executor, T predecessor, F func) {
  return transform(
    via(std::move(executor), std::move(predecessor)),
    std::move(func));
}
```