Document No.	P1745R0
Date	2019-06-17
Reply To	Lewis Baker <lbaker@fb.com>
Audience	Evolution
Targeting	C++20, C++Next

Coroutine changes for C++20 and beyond

Abstract 3

Overview 5

Changes for C++20 5

- Splitting coroutine_handle responsibilities 5
- Passing non-type-erased handles 8
- Type-erased handle types 10
  - The continuation_handle type 10
  - The suspend_point_handle type 11
    - Convertibility between different suspend_point_handle instantiations 13
    - Mapping from coroutine_handle usages to suspend_point_handle usages 13
- The noop_continuation 14

Incremental steps to incorporate proposed changes 15

- Merge initial_suspend() and get_return_object() 15
- Remove coroutine_handle::from_promise() 16
- Remove coroutine_handle::address()/from_address() 17
- Simplify final_suspend() 18
- Rename final_suspend() to done() 19
- Split coroutine_handle into suspend_point_handle and continuation_handle 19
- Split suspend_point_handle::destroy() into set_done() and destroy() 21
- Rename std::noop_coroutine() to std::noop_continuation() 21
- Limit suspend-point handles to only allow resuming a coroutine once 22
- Modify the compiler to pass compiler-generated, per-suspend-point handle types to get_return_object() and await_suspend() 23

Roadmap of changes for post-C++20 26

- Add support for async RAII 26
- Add support for async return-value optimisation 26

Add support for async range-based for-loops 28
Add support for co_return as a suspend-point 28
Add support for non-type-erased coroutines 29
Add support for delaying choice of promise-type 29
Add support for exposing coroutine-frames as types 29
Add support for heterogeneous resume 29
Implementation Experience 31
Bikeshedding 32
Bikeshedding coroutine resumption method names 32
References 33
Acknowledgements 34
Appendix A - Examples 35

// Abstract

The Coroutines TS was recently merged into the working draft and is on track for being included in C++20.

The Coroutines TS has been available for several years now, along with multiple compiler implementations, and in that time we have gained much usage experience with coroutines. There are multiple open-source libraries that provide abstractions for using coroutines (e.g. cppcoro[1], folly[2] and boost::asio). Facebook is now using coroutines successfully in production in a number of key services.

This usage experience has highlighted some limitations of the current design and given insight into ways we can improve the coroutines feature over time.

Some improvements, such as exposing non-type-erased coroutines, deferred coroutine-frame creation and coroutine-frames as types have been explored in [P1342R0, P1492R0, P1493R0] and we are reasonably confident that support for these capabilities can be added incrementally to the current coroutines design in the future.

We have also been exploring some other directions for evolving the design coroutines in the post-C++20 timeframe that we think are important for improving the expressiveness, safety and performance of code written using coroutines.

The main areas of exploration have been:

- **Async RAII** - the ability to `co_await` automatically at the end of a scope (expressiveness, safety)
- **RVO for co_await** - eliminate copies typically needed for `co_await` expressions (performance)
- **RVO for synchronous coroutines** - *E.g.*, allow `std::expected` coroutines to take advantage of return-value optimisation (performance)
- **Non-type-erased coroutine handles** - Allow the compiler to avoid indirect calls or branches when resuming a coroutine, improving the likelihood of inlining (performance)
- **Heterogeneous resume** - improve the expressiveness of a `co_await` expression to match what is currently possible with callback-based code where the call operator is overloaded by allowing a `co_await` expression to resume with one of several possible result types (expressiveness, performance)

We have been exploring potential designs for these capabilities in more detail over the last 6 months to determine, with a reasonable level of confidence, whether there is a viable path to adding them incrementally to the current design in a future version of the standard or whether changes to the coroutines design are required prior to C++20 being finalised to enable such a viable path.

---

[1] cppcoro - https://github.com/lewissbaker/cppcoro
[2] folly::coro - https://github.com/facebook/folly/tree/master/folly/experimental/coro

The papers "*Adding async RAII support to coroutines*" [P1662R0] and "*Supporting return value optimisation in coroutines*" [P1663R0] discuss the results of exploration of some of these areas in more detail, covering motivation, use-cases and design discussion.

All of these papers conclude that the current design of the `coroutine_handle` type either limits or prevents the evolution of coroutines to support these capabilities in future.

The core limitation is that a `coroutine_handle` currently only supports resuming a coroutine along a single resumption path - the `resume()` path - whereas these capabilities all require the ability to resume a coroutine along one of several possible paths.

Adding async RAII means that when cancelling the execution of the rest of a suspended coroutine we must ensure that any async cleanup work is run before calling `coroutine_handle::destroy()`. [P1662R0] proposes that we add a 'done' resumption path separate from `resume()` that will asynchronously unwind the coroutine from its current suspend-point back to the final suspend-point without needing to throw an exception instead of calling the synchronous `destroy()` method. The motivation for requiring a separate signal for cancellation is discussed in more detail in the paper "Cancellation is not an error" [P1667R0].

Supporting async RVO requires the ability to resume execution of the awaiting coroutine either along a 'value' path that produces the value or along an 'exception' path that immediately throws when it is resumed. The current design merges those two paths into the 'resume' path and lets `await_resume()` decide which path should be taken - however this indirection through `await_resume()` inhibits RVO.

Supporting heterogeneous resume requires further extending the async RVO capability, which has a 'value' and 'exception' resumption path, to support resuming on one of many possible 'value' paths and many possible 'exception' paths. This allows async operations to avoid needing to type-erase results and thus avoid the resulting branches or indirect calls in the same way that an async operation built on callbacks can by overloading `operator()`. This capability would further harmonize the design of coroutines with the design of async operations and callbacks proposed in [P1341R0, P1660R0].

> This paper proposes splitting the current responsibilities of `coroutine_handle` into two separate concepts, a **suspend-point-handle** and a **continuation-handle**, to enable support for resuming a coroutine on one of multiple possible resumption paths while also retaining the ability to symmetrically-transfer execution to another coroutine (See [P0913R0]).

The proposed design changes do not affect the syntax for authoring coroutines. The changes only affect the implementers of new coroutine `promise_types` and awaitable types and even then the changes required to these types are typically minor.

A prototype of the proposed design has been implemented in clang as an incremental change to clang's existing implementation of coroutines.
This implementation can be found at https://github.com/lewissbaker/llvm/tree/P1745

Overview

This document is structured into four main parts:

1. A description of the proposed revised design for C++20.
2. A step-by-step walk through of the incremental changes required to get from the current coroutines design to the proposed design for C++20
3. A description of some of the features that we propose adding post-C++20 that provides some context for the changes proposed for C++20.
4. A description of implementation experience for the proposed changes.

The appendix contains some code examples that show usage of the revised design.

Changes for C++20

This section describes the proposed revised design for coroutines that splits the `coroutine_handle` responsibilities into separate concepts: **SuspendPointHandle** and **ContinuationHandle**.

This section also describes the interface of the two new types, `suspend_point_handle` and `continuation_handle`, which replace `coroutine_handle` in the proposed design.

The following section titled "Incremental steps to incorporate proposed changes" walks through the set of incremental changes needed to go from the current design in the working draft to the proposed design and goes into more detail about the rationale for each of these changes.

Splitting `coroutine_handle` responsibilities

The main change to coroutines being proposed by this paper is to split the responsibilities of the `coroutine_handle` type into two distinct concepts: a **SuspendPointHandle** and a **ContinuationHandle**.

This change is necessary to support resuming a coroutine along one of several possible resumption paths while retaining the ability to symmetrically transfer to the chosen resumption path.

This enables adding a `set_done()` resumption path to support asynchronously cancelling/unwinding a coroutine without needing to throw an exception. This resumption path executes '`goto done;`' instead of calling `await_resume()`.

It also enables the ability to support async RVO in future, which requires the ability to resume a coroutine on either the 'value' path or on the 'exception' path.

The current design of `coroutine_handle` is restricted to resuming a coroutine along only a single path;

the 'resume()' path which executes await_resume().

A SuspendPointHandle represents a coroutine suspended at a particular suspend-point.
A ContinuationHandle represents a particular continuation path on which to resume the coroutine.

A coroutine suspended at a suspend-point can have one of several operations performed on it. The set of operations available can depend on what kind of suspend-point the coroutine is currently suspended at.

The SuspendPointHandle allows user-code to select a particular continuation path to resume the coroutine with by calling one of the continuation factory methods on the SuspendPointHandle object. Each of these methods returns a different ContinuationHandle which corresponds to the selected resumption path.

The ContinuationHandle can then either be invoked asymmetrically using operator() or can be invoked symmetrically (ie. with tail-recursion) by returning the handle from await_suspend().

The ContinuationHandle concept is defined as follows:

```
// A helper concept for defining traits common to the various handle concepts.
template<typename T>
concept _Handle =
  std::is_nothrow_default_constructible_v<T> &&
  std::is_nothrow_copy_constructible_v<T> &&
  std::is_nothrow_move_constructible_v<T> &&
  std::is_nothrow_copy_assignable_v<T> &&
  std::is_nothrow_move_assignable_v<T> &&
  std::is_trivially_destructible_v<T> &&
  requires (const T handle) {
    { handle ? (void)0 : (void)0 } noexcept;
    { !handle ? (void)0 : (void)0 } noexcept;
  };

template<typename T>
concept ContinuationHandle =
  _Handle<T> &&
  Invocable<T> &&
  unspecified;
```

Note that the ContinuationHandle concept contains an '*unspecified*' condition. This has the effect of preventing user-defined types from satisfying this concept. The rationale here is that to support the ability to perform tail-recursive resumption of a coroutine there will likely need to be some implementation-defined mechanics present on the handle to support appropriate calling-conventions and type-erasure of the handle type (see later section on type-erased handles).

There is not a single SuspendPointHandle concept but rather a number of lower-level concepts that can be composed together to make one of several different suspend-point handle concepts, one for each kind of suspend-point.

For C++20 this paper is proposing to define 3 operations that can be performed on a suspend-point handle corresponding to a `co_await`/`co_yield` expression:

- `resume()` - Returns a continuation-handle that will resume execution of the coroutine. This will immediately call `await_resume()` upon invocation of the returned continuation-handle.
- `set_done()` - Returns a continuation-handle that will resume execution of the coroutine by immediately executing '`goto done;`' upon invocation of the returned continuation-handle.
- `promise()` - Obtains a reference to the coroutine's promise object.

And 4 operations that can be performed on a coroutine-handle passed to `get_return_object()`:

- `destroy()` - Destroys the coroutine frame. Only valid to be called if the coroutine has either never been resumed or has run to completion and is currently suspended at the final suspend-point.
- `resume()` - Returns a continuation-handle that will start executing the coroutine body when invoked.
- `set_done()` - Returns a continuation-handle that will start execution of the coroutine by immediately executing '`goto done;`' upon invocation of the returned continuation-handle.
- `promise()` - Obtains a reference to the coroutine's promise object.

Different kinds of suspend-points within a coroutine will have different combinations of these operations.

Future versions of the language may define new kinds of suspend-points that support a different configuration of operations. For example, the `operator ~co_await()` suspend point described in P1662R0 to support async RAII. Or they may add new operations, such as the `set_value()`/`set_exception()` operations described in P1663R0 to support async return-value-optimisation.

The suspend-point concepts are defined as follows:

```
template<typename T>
concept SuspendPointHandleWithResume =
  _Handle<T> &&
  requires (const T sp) {
    sp.resume();
    requires ContinuationHandle<decltype(sp.resume())>;
  } &&
  unspecified;

template<typename T>
concept SuspendPointHandleWithDone =
  _Handle<T> &&
  requires (const T sp) {
    sp.set_done();
    requires ContinuationHandle<decltype(sp.set_done())>;
  } &&
  unspecified;

template<typename T>
concept SuspendPointHandleWithPromise =
  _Handle<T> &&
  requires (const T sp) {
```

```
    sp.promise();
  } &&
  unspecified;

template<typename T, typename Promise>
concept SuspendPointHandleWithPromiseOf =
  SuspendPointHandleWithPromise<T> &&
  requires (const T sp) {
    { sp.promise() } -> Promise&;
  };

template<typename T>
concept SuspendPointHandleWithDestroy =
  _Handle<T> &&
  requires (const T sp) {
    { sp.destroy() } -> void;
  } &&
  unspecified;
```

Given these definitions we can then build suspend-point handle concepts for the suspend-point kinds proposed for C++20:

The concept for suspend-point handles passed to the `promise.get_return_object()` method:

```
template<typename T>
concept InitialSuspendPointHandle =
  SuspendPointHandleWithDestroy<T> &&
  SuspendPointHandleWithResume<T> &&
  SuspendPointHandleWithSetDone<T> &&
  SuspendPointHandleWithPromise<T>;
```

An alternative name for this concept could be `CoroutineHandle` as it represents a handle to the coroutine frame that was created for a particular invocation.

The concept for suspend-point handles passed to the `await_suspend()` method in a `co_await` or `co_yield` expression:

```
template<typename T>
concept AwaitSuspendPointHandle =
  SuspendPointHandleWithResume<T> &&
  SuspendPointHandleWithSetDone<T> &&
  SuspendPointHandleWithPromise<T>;
```

Passing non-type-erased handles

When the compiler generates calls to the `get_return_object()` or `await_suspend()` it passes a suspend-point handle that represents the particular suspend-point.

This paper proposes that the handle types that the compiler passes to these methods are anonymous, per-suspend-point handle types which are not type-erased. The compiler generates a unique type

corresponding to that particular suspend-point in that particular coroutine and passes an instance of that type into the method call.

Note that the underlying coroutine frame type is still hidden from the program, the type of the handle only encodes knowledge of the location of the suspend-point within the coroutine body, not of the layout of the coroutine frame.

The `get_return_object()` and `await_suspend()` methods can then call the continuation factory methods on the suspend-point handle to select a continuation path to resume the coroutine on.

For example:

```
struct on_new_thread {
  bool await_ready() { return false; }

  auto await_suspend(SuspendPointHandleWithResume auto suspendPoint) {
    std::thread{[suspendPoint] { suspendPoint.resume()(); }}.detach();
    return std::noop_continuation();
  }

  void await_resume() {}
};

task<void> example() {
  co_await on_new_thread{};

  // Rest of the function is now executing on a new std::thread.
  // ...
}
```

When the `example()` coroutine is invoked the function will create a new coroutine frame, construct a new instance of a compiler-generated `InitialSuspendPointHandle` type that knows statically that it represents the initial suspend-point for the `example()` coroutine and pass this handle as a parameter to `promise.get_return_object()`.

When the coroutine is resumed and eventually reaches the `co_await` expression, the coroutine is suspended and the compiler passes a new instance of a compiler-generated `AwaitSuspendPointHandle` type that knows statically that it represents that particular suspend-point in that particular coroutine.

This means that when the thread executes the lambda the lambda can resume the coroutine without needing an indirect call to resume the coroutine. The compiler is therefore able to emit a direct jump to the resumption point in the coroutine.

However, to be able to store the `InitialSuspendPointHandle` passed to `get_return_object()` in the returned `task<T>` we need to be able to type-erase the handle. To avoid the need for libraries to type-erase the handle manually, and also to guarantee the ability to support symmetric-transfer to the

continuation, the standard library provides a type-erased handle type that can be used to store the type-erased handle.

Type-erased handle types

The standard library provides two type-erased handle types: `continuation_handle` and `suspend_point_handle<Ops...>`.

These types replace the existing `coroutine_handle` type described by the Coroutines TS.

The `continuation_handle` type

The public interface of the `continuation_handle` type is as follows:

```
class continuation_handle {
public:
  // Construct to the null continuation_handle.
  constexpr continuation_handle() noexcept;

  // Copy construction/assignment.
  continuation_handle(const continuation_handle&) noexcept;
  continuation_handle& operator=(const continuation_handle&) noexcept;

  // Trivial destructor
  ~continuation_handle() = default;

  // Implicit conversion from compiler-generated ContinuationHandle types.
  continuation_handle(ContinuationHandle auto handle) noexcept;

  // Test for the null coroutine_handle
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;

  // Invoke the continuation.
  void operator()() const;

private:
  // exposition only
  void* data;
  void (*fn)(void*);
};
```

The `continuation_handle` type satisfies the `ContinuationHandle` concept and is also implicitly constructible from any of the compiler-generated `ContinuationHandle` types returned from continuation methods on compiler-generated suspend-point handle types.

Note that the size of the `continuation_handle` is unspecified. Implementations that choose to store the resumption state in the coroutine frame may have a `continuation_handle` type that stores a single pointer to the coroutine frame. Whereas implementations that store the resumption state in the `continuation_handle` may store a pointer to the coroutine frame as well as either a function pointer or integer that encodes the resumption-path.

The `suspend_point_handle` type

The `suspend_point_handle` type is designed to support type-erasure of different kinds of suspend-points which support different sets of operations.

The intent is also to allow library code to specify the minimum set of operations that the type-erased suspend-point needs to support so that the compiler can avoid generating code for resumption paths that will never be called.

The `suspend_point_handle` type is also designed to be extensible to support type-erasing new kinds of operations that may be added in future versions of C++, such as a `set_value<T>()`, `set_error<E>()` and `set_exception()` operations, described in P1663R0, to support async return-value optimisation.

To support these goals, the `suspend_point_handle` has a variadic template argument which can be passed the set of operation types as template arguments that are to be made available on the type-erased handle.

The standard library also provides types that represent each of the operations supported by suspend-point handles which can be used as the template arguments to `suspend_point_handle`. Note that *only* these types are valid to pass as template arguments to `suspend_point_handle` - user-defined types are not supported.

The operation types simply serve as tag types and do not have any public API and are not intended to be used other than as template arguments to `suspend_point_handle`.

The public interface of `suspend_point_handle` is defined as follows:

```
using with_resume = unspecified;
using with_set_done = unspecified;
using with_destroy = unspecified;
template<typename Promise>
using with_promise = unspecified;

// Helper concepts to assist with definition.
template<typename T, typename... Values>
concept _OneOf = (Same<T, Values> || ...);

template<typename T>
concept _CompilerGeneratedSuspendPointHandle = unspecified;

template<typename... Operations>
struct __promise_type_helper {};

template<typename Promise, typename... Rest>
struct __promise_type_helper<with_promise<Promise>, Rest...> {
  using type = Promise;
};

template<typename First, typename... Rest>
```

```
struct __promise_type_helper<First, Rest...>
: __promise_type_helper<Rest...> {};

template<typename... Operations>
concept _HasWithPromise =
  requires() {
    typename __promise_type_helper<Operations...>::type;
  };

template<typename... Operations>
class suspend_point_handle {
public:

  // Construct to the null handle.
  constexpr suspend_point_handle() noexcept;

  // Copy construction/assignment
  suspend_point_handle(const suspend_point_handle&) noexcept;
  suspend_point_handle& operator=(const suspend_point_handle&) noexcept;

  // Implicit conversion from a compiler-generated suspend-point handle type.
  // Only valid if the handle supports all of the specified Operations.
  template<_CompilerGeneratedSuspendPointHandle Handle>
    requires /* ... see below */
  suspend_point_handle(Handle handle) noexcept;

  // Trivial destructor
  ~suspend_point_handle() = default;

  // Test for the null handle.
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;

  [[nodiscard]]
  continuation_handle resume() const
    requires _OneOf<with_resume, Operations...>;

  [[nodiscard]]
  continuation_handle set_done() const
    requires _OneOf<with_set_done, Operations...>;

  void destroy() const
    requires _OneOf<with_destroy, Operations...>;

  auto promise() const
    -> typename __promise_type_helper<Operations...>::type&
    requires _HasWithPromise<Operations...>;

private:
  // Exposition only.
  void* data;
  void* vtable;
};
```

Note that the implicit conversion from a compiler-generated suspend-point handle type is constrained to require the source handle type support all of the operations that have been requested by the type-erased handle.

The constraint on this constructor can be written using a concept requires clause as follows:

```
template<_CompilerGeneratedSuspendPointHandle Handle>
  requires
    (!_OneOf<with_resume, Operations...> ||
     SuspendPointHandleWithResume<Handle>) &&
    (!_OneOf<with_destroy, Operations...> ||
     SuspendPointHandleWithDestroy<Handle>) &&
    (!_OneOf<with_set_done, Operations...> ||
     SuspendPointHandleWithSetDone<Handle>) &&
    (!_HasWithPromise<Operations...> ||
     SuspendPointHandleWithPromiseOf<
       Handle, typename __promise_type_helper<Operations...>::type>)
suspend_point_handle(Handle handle) noexcept;
```

However, implementations may be able to define this constructor with equivalent constraints in a more direct way without relying on disjunctions in the requires clauses. e.g. by making use of the internal details of the `Operation` types.

Convertibility between different `suspend_point_handle` instantiations

Note also that this does not support conversion from one type-erased handle type to a different type-erased handle type.

Reference implementations of the handle type store a pointer to the frame and a pointer to a vtable which encodes information about the type-erased operations. Supporting conversion between type-erased handle types would require being able to generate a new vtable for the combination of the target subset of operations and the original handle type. However, as we have already type-erased the original handle type we cannot easily do that unless we pre-generate vtables for all possible combinations of sub-operations.

User code can still convert the same compiler-generated handle type to multiple different type-erased handle types as required, however.

Mapping from `coroutine_handle` usages to `suspend_point_handle` usages

To translate code written against the Coroutines TS from using `coroutine_handle` to using `suspend_point_handle`

Table 1: Handles representing initial and final suspend-points

`coroutine_handle<void>`	`suspend_point_handle<with_resume, with_destroy>`

`coroutine_handle<Promise>`	`suspend_point_handle<`
	`  with_resume, with_destroy,`
	`  with_promise<Promise>>`

Table 2: Handles representing `co_await`/`co_yield` suspend-points

coroutine_handle<void>	suspend_point_handle<with_resume, with_set_done>
coroutine_handle<Promise>	suspend_point_handle< with_resume, with_set_done, with_promise<Promise>>

Note that there is not a direct correspondence between these types for all use-cases due to the change in semantics of `destroy()`, which was previously allowed to be called at any suspend-point and is now only allowed at initial and final suspend points, and due to the introduction of `set_done()`, which can now be called at any suspend-point except the final suspend-point.

The `noop_continuation`

In cases where an awaitable does not have another continuation to transfer execution to it needs to be able to suspend and return execution back to the most-recent caller on the stack that asymmetrically invoked a continuation handle.

Under the Coroutines TS you can call `std::noop_coroutine()` to obtain a `coroutine_handle` that represents the no-op continuation (ie. when it was resumed it would immediately return to the caller). If such a handle was returned from `await_suspend()` then this would cause the top-most call to `resume()` on the stack to return to its caller.

The same facility is provided here, however it is named `std::noop_continuation()` and returns a `noop_continuation_handle` that satisfies the `ContinuationHandle` concept.

The synopsis for the `std::noop_continuation()` and `std::noop_continuation_handle` type:

```
class noop_continuation_handle {
public:
  // Construct to the null handle
  constexpr noop_continuation_handle() noexcept;

  // Copy construction/assignment
  noop_continuation_handle(const noop_continuation_handle&) noexcept;
  noop_continuation_handle& operator=(const noop_continuation_handle&) noexcept;

  // Trivial destructor
  ~noop_continuation_handle() = default;

  // Test for the null handle
  explicit operator bool() const noexcept;
  bool operator!() const noexcept;

  // Invoke the noop-continuation
  void operator()() noexcept {}

private:
  unspecified;
};

// Returns a non-null handle.
```

```
noop_continuation_handle noop_continuation() noexcept;
```

Incremental steps to incorporate proposed changes

The following changes describe a set of logical steps to get from the current wording in the C++20 working draft to wording that incorporates the changes proposed by this paper.

1. Merge `initial_suspend()` and `get_return_object()`.
2. Remove `coroutine_handle::from_promise()`
3. Remove `coroutine_handle::address()`/`from_address()`
4. Simplify `final_suspend()` method to return a `coroutine_handle`
5. Rename `final_suspend()` to `done()`
6. Split `coroutine_handle` into two separate types `suspend_point_handle` and `continuation_handle`
   a. Compiler passes a `suspend_point_handle` to `get_return_object()` and `await_suspend()`
   b. Library returns a `continuation_handle` from `await_suspend()` and `done()`
7. Refactor `destroy()` into `set_done()` and `destroy()`
8. Rename `std::noop_coroutine()` to `std::noop_continuation()`
9. Limit suspend-point handles to only allow resuming a coroutine once
10. Modify the compiler to pass compiler-generated, per-suspend-point handle types to `get_return_object()` and `await_suspend()`.
11. Modify `suspend_point_handle` to take template parameters indicating operations to type-erase and add `with_resume`, `with_set_done`, `with_promise<P>`, `with_destroy` operation types.

A followup paper with proposed wording for the changes will be made available post-Cologne.

Merge `initial_suspend()` and `get_return_object()`

This change is described in more detail in P1477R1, but the basic summary of the change is:

- No longer evaluate '`co_await promise.initial_suspend();`' at the start of the coroutine.
- The coroutine is always created initially suspended.
- The `promise.get_return_object()` method is passed the initial `coroutine_handle`

This results in the following change in definition of a promise_type:

<table>
<tr>
<td>

```
// Coroutines TS promise_type interface
struct promise_type
{
  T get_return_object();

  Awaitable<void> initial_suspend();
  ...
};
```

</td>
<td>

```
// Proposed promise_type interface
struct promise_type
{
  T get_return_object(coroutine_handle<promise_type>
h);
  ...
};
```

</td>
</tr>
</table>

The motivation for this change is:

- It reduces the amount of code needed to implement a coroutine's `promise_type`.
- It eliminates a number of specification issues relating to the behaviour of a coroutine if an exception is thrown from the `co_await promise.initial_suspend()` expression.
- It eliminates the need to call `coroutine_handle::from_promise()` to obtain a `coroutine_handle` in `get_return_object()`.
- It opens the door to making `coroutine_handle`'s resumption single-shot.
- It opens the door to having the compiler pass different kinds of handles for different types of suspend-points.
- This is an important step for later changes.

Remove `coroutine_handle::from_promise()`

The primary use for `coroutine_handle::from_promise()` is to allow a `coroutine_handle` that references the current coroutine to be manufactured from within the `get_return_object()` method so that a handle to the coroutine can be passed to the constructor of the return value. eg. `task<T>`

The motivation for this change is:

- `from_promise()` is no longer needed once we have merged `initial_suspend()` and `get_return_object()` as the compiler is now passing the `coroutine_handle` to `get_return_object()`.
- `from_promise()` is a potentially dangerous function:
    - You need to pass the exact promise-type used for the coroutine frame.
      Calling `coroutine_handle<promise_base>::from_promise(*this)` when the dynamic type of `*this` is a derived type has undefined-behaviour.
      Any derived promise type will need to override `get_return_object()` to ensure that the `coroutine_handle` is constructed using the most-derived type.
    - This method manufactures a `coroutine_handle` for a coroutine that is not necessarily suspended. Callers will need to be careful that they do not call methods on this handle unless the coroutine is actually suspended at a suspend-point.
- The existence of this method constrains implementations in how they can layout the coroutine frame:
    - The existence of this function requires the coroutine frame to be laid out such that the address of the coroutine's resumption state is able to be deterministically calculated from the address of the promise object. This typically means that the promise must be located at a fixed offset from the storage of the ABI-stable part of the coroutine resumption state.
    - This can potentially constrain what implementations are able to support in the future if we were to, at some stage, require different flavours of ABI-stable coroutine resumption state for different kinds of suspend-points.
- Removing this method is a step towards supporting different kinds of handles for different kinds of suspend-points.
    - To be able to support adding new kinds of suspend-points in future (eg. for async return-value optimisation or a 'for co_await' facility) we need the flexibility to be able to have

different kinds of suspend-points use different kinds of handles with different sets of operations available on them.
    - A factory function that allows you to manufacture a `coroutine_handle` for a coroutine at any point in time, regardless of kind of suspend-point it is suspended at, may constrain the design to only ever supporting a single kind of `coroutine_handle`.

Remove `coroutine_handle::address()`/`from_address()`

The `coroutine_handle::address()` and `coroutine_handle::from_address()` methods allow converting a `coroutine_handle` to/from a `void*` pointer.

This step removes those methods from the interface.

The motivation for this change is:

- Later steps introduce a type-erased `suspend_point_handle` where a reasonable implementation strategy requires a handle-type that is larger than a single pointer.
- The presence of the ability to convert to/from a `void*` either requires the handle type to be the size of a single pointer or requires the coroutine frame to reserve additional storage in the coroutine frame to store enough state allow reconstructing the handle from a `void*`.
- Removing these methods frees implementations to define handle types larger than a pointer without needing to reserve storage in the coroutine frame for the conversion to/from `void*`

This ability to convert a `coroutine_handle` to/from a `void*` was originally added to facilitate passing a `coroutine_handle` through existing async C APIs that accepted a `void*` data parameter which was then passed back to the user-provided callback.

However, in cases where the C API produces a result that is to be returned from the `co_await` expression, the `coroutine_handle` alone can be insufficient for resuming the coroutine. The result would need to be stored inside the awaitable object so that when the coroutine is resumed the `await_resume()` method can return this result.

In these cases it is typical to pass a pointer to the awaitable object as the `void*` parameter rather than passing `coroutine_handle::address()`.

For example: An awaitable wrapper around a fictional OS API with a `void*` data parameter

```
using callback_t = void(int /* result */, void* /* data */);
void os_async_start_foo(callback_t* cb, void* data);

struct foo_awaitable {
  bool await_ready() { return false; }

  void await_suspend(coroutine_handle<> h) {
    continuation_ = h;
    os_async_start_foo(&foo_awaitable::on_complete, /*data=*/this);
  }
```

```
  int await_resume() { return result_; }

private:
  static void on_complete(int result, void* data) {
    foo_awaitable* awaitable = static_cast<foo_awaitable*>(data);
    awaitable->result_ = result;
    awaitable->handle_.resume();
  }

  coroutine_handle<> continuation_;
  int result_;
};
```

The same approach can be used as an alternative to converting the `coroutine_handle` to/from `void*`. Just store the handle in the awaitable object and pass a pointer to it.

Simplify `final_suspend()`

This change is described in more detail in P1477R1, but the basic summary of the change is:

- Instead of executing `co_await promise.final_suspend();` at the end of the coroutine body, the coroutine is unconditionally suspended and the compiler calls `promise.final_suspend()` and then symmetrically-transfers to the coroutine identified by the returned `coroutine_handle`.
- The coroutine is no longer implicitly destroyed when execution runs off the end of the coroutine.
- Any exceptions thrown from `promise.final_suspend()` call propagate out of the current call to `coroutine_handle::resume()`.

Motivation for this change:

- Reduces the amount of code needed to implement `final_suspend()` methods.
  See P1477R1 for examples.
- Eliminates some of the specification issues relating to exceptions propagating from evaluating the `co_await promise.final_suspend();` statement.
- Encourages usage patterns that call `.destroy()` from the coroutine wrapper type rather than from `final_suspend()`. These patterns are more likely to have their coroutine frame allocations elided

Note that this paper has a further simplification to `final_suspend()` compared with the simplification described in P1477R1, which described the `final_suspend()` method as being passed a `coroutine_handle` which could be used to destroy the coroutine, whereas this paper proposes that `final_suspend()` is called without any arguments.

The rationale for this difference from P1477R1 is:

- It is a further simplification of the interface.
- The only operation that it is valid to perform on a coroutine suspended at the final-suspend point is to destroy the coroutine frame.

- If we add support for coroutine-frames-as-types in future (See [P1342R0]) then calling `destroy()` on a handle passed to `final_suspend()` would actually be a no-op / not be meaningful as it should be the destructor of the coroutine-frame object itself that would destroy the coroutine frame.
- If the frame was created by calling the constructor on a type then it should be destroyed by calling the destructor on that type rather than calling `.destroy()` on the coroutine frame.
- The only reason for passing a handle to `final_suspend()` was to support self-destructing coroutine types
  - This can be implemented by storing the handle passed to `get_return_object()` in the promise and calling `.destroy()` on that from `final_suspend()`.
    See `detached_task` example in Appendix A.

Rename `final_suspend()` to `done()`

Rename the `promise.final_suspend()` method to `promise.done()`.
Rename the label used by `co_return` definition from `'final_suspend:'` to `'done:'`.

Rationale:

- No need to maintain consistency of naming with now removed `initial_suspend()`
- Matches terminology from sender/receiver design for executors.
  See P1341R0, P1660R0

Split `coroutine_handle` into `suspend_point_handle` and `continuation_handle`

This change is the primary enabling change for some of the extensions to coroutines we anticipate adding in future versions of C++. These extensions include supporting async cancellation (needed for async RAII - See P1662R0), async return-value-optimisation (See P1663R0) and heterogeneous resume.

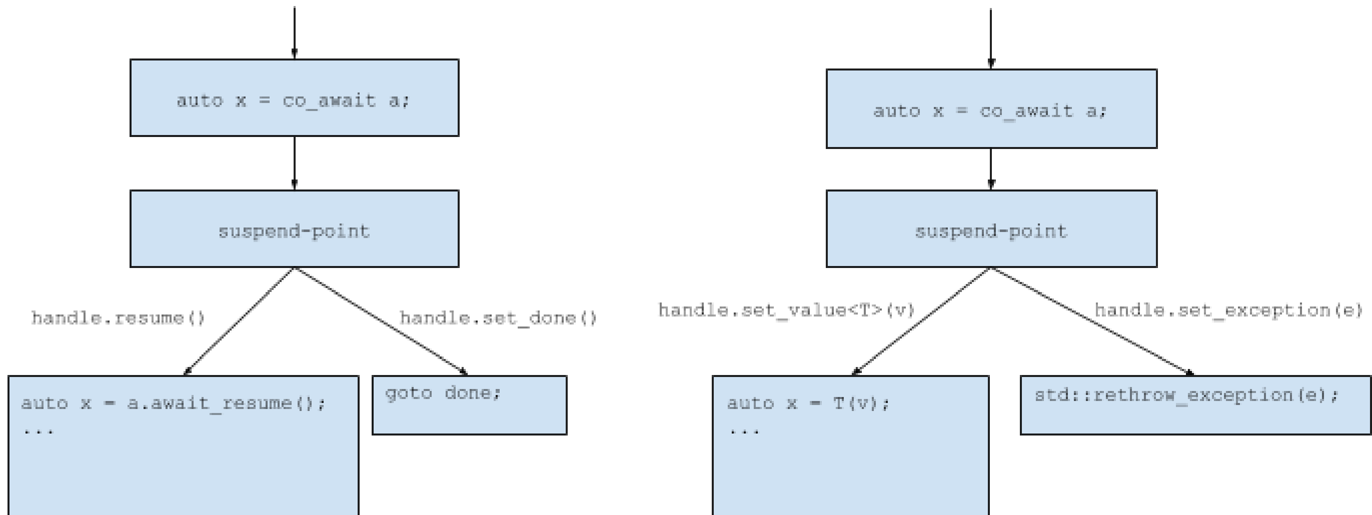
All of these features have in common that they require the ability for a suspended coroutine to be resumed along one of several possible continuation paths rather than just along a single '`resume()`' path.

For example, support for async RVO would require resumption on either a 'value' path, which continues with the result of the `co_await` expression having the provided value, or resumption on an 'error' path, which immediately throws an exception when the coroutine is resumed.

This step splits the `coroutine_handle` type into two separate types:

- A `suspend_point_handle`, which represents a coroutine suspended at a particular point in its execution.
- A `continuation_handle`, which represents a selected resumption path of a coroutine from a particular suspend-point.

If you were to represent the control-flow of a coroutine as a graph then a suspend-point represents a node in the control flow graph at which the coroutine can be suspended and this node has a single incoming edge. That node in the graph can have zero or more out-edges, each corresponding to a particular resumption path (eg. resuming with a value or resuming with cancellation).



A `suspend_point_handle` represents the suspend-point node in the control-flow graph. The interface of the `suspend_point_handle` allows the caller to select which outgoing edge the coroutine should resume on by calling one of the methods that returns a `continuation_handle`. The returned `continuation_handle` represents the chosen out-edge of the control-flow graph.

By splitting the `coroutine_handle` into `suspend_point_handle` and `continuation_handle` types we allow a coroutine to be resumed on one of several possible paths while still retaining the ability to perform symmetric transfer (See P0913R0).

If, instead, we were to simply add another method, say `set_done()`, to `coroutine_handle` which would, like `resume()`, asymmetrically resume the coroutine but along the path that executed '`goto done;`', then we would still need some other way to indicate that this path should be taken for the symmetric-transfer cases when the handle was returned from `await_suspend()`, which is currently defined to call `resume()` on the returned handle.

The interface of the `suspend_point_handle` and `continuation_handle` types are described above in the section on "Type-erased handle types".

Split `suspend_point_handle::destroy()` into `set_done()` and `destroy()`

This change separates the current responsibilities of the `suspend_point_handle::destroy()` method into two separate operations:

- `set_done()` - Returns a `continuation_handle` that, when invoked, will resume the coroutine and immediately execute '`goto done;`'.
- `destroy()` - Destroys the promise, parameters and coroutine frame.
  The `destroy()` method is only valid to call when the coroutine is suspended either at the initial or final suspend-points.

The `suspend_point_handle::destroy()` method is currently used by `generator` types to cancel execution of the rest of the coroutine when the consumer destroys the generator object, in addition to being used in general to destroy a coroutine that has run to completion.

The `destroy()` method synchronously calls the destructors of any in-scope objects before freeing the coroutine frame memory. The `destroy()` method is required to complete synchronously and optimisations like HALO (See P0981R0) rely on `destroy()` completing synchronously so that it knows the memory used by the coroutine frame is no longer in-use.

This need for `destroy()` to complete synchronously does not generalise well to supporting cancellation of a coroutine once support for async RAII has been added to the language (see P1662R0 for a discussion of async RAII). Once we have async RAII support then exiting the scopes of a suspended coroutine may involve executing some async operations and thus cancelling a coroutine cleanly needs to be an asynchronous operation.

By introducing a separate `set_done()` method which resumes the coroutine and executes '`goto done;`' it allows the coroutine to subsequently suspend again while executing async cleanup. It will then eventually reach the '`done:`' label and execute the '`promise.done()`' method at which point the library can safely call `.destroy()` to destroy the coroutine frame since it knows that all async cleanup has completed.

For more details see the paper P1662R0 - "Adding async RAII support to coroutines".

Note that the addition of the `set_done()` continuation path can also potentially be used to `break` out of a `'for co_await'`. See the later section "Add support for async range-based for loops".

Rename `std::noop_coroutine()` to `std::noop_continuation()`

This step performs the following changes:

- Adds a new `std::noop_continuation_handle` type

- Removes `std::noop_coroutine_promise` type
- Renames `std::noop_coroutine()` to `std::noop_continuation()` and changes the return-type to `std::noop_continuation_handle`

The `std::noop_coroutine()` method in the current coroutines design returns a `coroutine_handle` that has a no-op `resume()` and `destroy()` method.

The representation of `std::noop_coroutine()` as a `coroutine_handle` was always a bit of stretched analogy as it does not actually refer to a coroutine. It was only ever meaningful to call `.resume()` on the handle - the semantics of a call to `.destroy()` or `.done()` was well-defined but not meaningful.

Logically, the returned handle represented a no-op continuation that could be returned from `await_suspend()` to allow it to resume the caller of the top-most call to `coroutine_handle::resume()` on the stack.

Now that we have split the concept of a suspend-point from that of a continuation we can now just have this function return a continuation-handle without needing to implement the other parts of the `coroutine_handle` interface that were not relevant because the handle was not referring to a coroutine.

Limit suspend-point handles to only allow resuming a coroutine once

The design of `coroutine_handle` in the Coroutines TS permitted using the any `coroutine_handle` that referred to the coroutine to resume the coroutine regardless of what suspend-point the coroutine was currently suspended at.

This step restricts a `suspend_point_handle` to only being valid to resume the coroutine from that particular suspend-point. Once the coroutine has been resumed from a given suspend point it is no longer valid to use that suspend-point handle to resume the coroutine again. You must use the suspend-point handle passed to `await_suspend()` when the coroutine next suspends.

Note that it is still valid to use a suspend-point handle's `promise()` method to access the promise for that coroutine, provided that the coroutine has not yet been destroyed.

Motivation for this change:

- **This restriction enables support for more efficient non-type-erased suspend-point handles.** If a coroutine can be resumed by any suspend-point handle to the coroutine then this forces the compiler to store the resumption state in a shared location in the coroutine frame so that the coroutine can be resumed by any of the handles, even if the coroutine is ultimately resumed by a non-type-erased handle.
  If we remove this capability then we allow implementations to store the resumption state in the handle itself, either encoded in the handle's type or stored in a type-erased form as a data-member of the handle, rather than in the coroutine frame. This can avoid stores to the coroutine state.

- **This change opens the door to supporting async return-value-optimisation in future.**
  To support return-value optimisation a `suspend_point_handle` will need to carry information about the result-type and the address of the result of the `co_await`/`co_yield` expression so that it can construct the result in-place.
  This will potentially require a different suspend-point handle type for each suspend-point as the handle will need to know about the return-type and return-value address of that particular suspend-point, which could be different for each suspend-point in a coroutine.
- **This change also supports using different kinds of suspend-point handle for different kinds of suspend-points in a coroutine.**
  eg. to support a coroutine where some suspend-points use async RVO-style awaitables and a corresponding async RVO-style suspend-point handle and other suspend-points use Coroutines TS-style awaitables with a resume()-style suspend-point handle.
  In general, the only way to know what kind of handle is valid to resume a coroutine from its current suspend point is to use the handle that was provided by that suspend-point's expression.

Modify the compiler to pass compiler-generated, per-suspend-point handle types to `get_return_object()` and `await_suspend()`

With this change we make the compiler generate a unique, anonymous handle type for each suspend-point in each coroutine and have the compiler invoke the `get_return_object()` or `await_suspend()` method with an instance of that suspend-point type instead of invoking with an instance of a `suspend_point_handle` type.

The per-suspend-point handle type can encode information about which suspend-point in the coroutine it represents in the type itself, allowing the compiler to avoid indirect jumps when resuming the coroutine via this handle.

This change also adds a number of named concepts to the library that allow `await_suspend()` methods to constrain their arguments to requiring particular operations to be present on the passed suspend-point-handle type: `SuspendPointWithDone`, `SuspendPointWithResume`, `SuspendPointWithPromise`, `SuspendPointWithDestroy`.

Motivation for this change:

Needed to support type-erasing to different flavours of `suspend_point_handle` with different sets of operations as template arguments.

The per-suspend-point handle type can be cast to any `suspend_point_handle` type as long as the suspend-point supports all of the operations listed in the template arguments of the `suspend_point_handle` type.

See the `generator<T>` example in Appendix A which type erases the handle passed to `get_return_object()` as both a `suspend_point_handle<with_resume, with_destroy, with_promise<promise_type>>` and `suspend_point_handle<with_resume, with_set_done>`.

However, a `suspend_point_handle` cannot be cast to a `suspend_point_handle` with a different subset of operations. If the compiler were to pass an already-type-erased `suspend_point_handle` type then we wouldn't be able to obtain a type-erased handle that contained a subset of the set of operations, eg. so that the handle could be stored in a variable that also needs to store a handle from a different kind of supsend-point.

Adding support for passing non-type-erased handles later could be a breaking change.

Let's say that, initially, we decided to specify that the compiler will pass a particular type-erased `suspend_point_handle` for a given `co_await` expression. eg.

```
template<typename Promise = void>
using await_suspend_point_handle =
  suspend_point_handle<with_resume, with_set_done, with_promise<Promise>>;
```

And let's say that we also define the `suspend_point_handle` type so it is possible to implicitly cast from `await_suspend_point_handle<Promise>` to `await_suspend_point_handle<void>`.

Then it would be possible (and indeed likely) that someone could write an Awaitable type with the following interface:

```
struct my_awaitable {
  bool await_ready();

  template<typename Promise>
  continuation_handle await_suspend(await_suspend_point_handle<Promise> h);

  void await_resume();
};
```

If coroutines in C++20 were to be specified such that the `await_suspend()` method were to be invoked with an instance of type `await_suspend_point_handle<concrete_promise_type>` then the compiler's invocation of this `await_suspend()` method would compile fine - the `Promise` template parameter would be deduced to `concrete_promise_type`.

However, if in C++Next we wanted to change the rule to allow invocation of the `await_suspend()` method with a compiler-generated per-suspend-point handle type which was convertible to `await_suspend_point_handle<concrete_promise_type>` then the compiler would no longer be able to deduce the `Promise` template parameter and the code would stop compiling.

This particular problem could be alleviated by having the per-suspend-point handle type inherit from `await_suspend_point_handle<concrete_promise_type>`.
However, this would not solve a more subtle problem that could lead to a silent change in behaviour.

Consider an Awaitable that changes its behaviour based on whether or not the promise of the awaiting coroutine supports some capability:

```
struct my_awaitable {
  bool await_ready();

  template<typename Promise>
    requires requires (Promise& promise) {
      { promise.get_stop_token() } -> std::stop_token;
    }
  continuation_handle await_suspend(await_suspend_point_handle<Promise> h) {
    // Start operation with cancellation support
    // ...
  }

  template<typename Handle>
  continuation_handle await_suspend(Handle h) {
    // Start operation without cancellation support
    // ...
  }

  void await_resume();
};
```

Now, if in C++20 the compiler were to invoke the `await_suspend()` method with `await_suspend_point_handle<concrete_promise_type>` and `concrete_promise_type` supported the `get_stop_token()` method then the compiler would end up calling the first overload with cancellation support.

However, if in C++Next the compiler started calling the `await_suspend()` method with a compiler-generated per-suspend-point handle type which inherited from `await_suspend_point_handle<concrete_promise_type>` then all of a sudden the second overload becomes a better match than the first overload and the compiler starts calling the second overload, silently changing the semantics of the program.

It is possible for users to write this code differently to allow for the possibility of this change so that it does not change behaviour. However, we cannot guarantee that users will not write code like this.

Thus, It will not be possible to safely add support for passing non-type-erased suspend-point-handles in a future version without requiring a change in the method name or arity of the call to `await_suspend()`.

Roadmap of changes for post-C++20

The prior section described the set of changes proposed for C++20. To help give some context to those changes, this section describes some of the future changes that are being considered. the proposed changes would enable or greatly simplify adding in future versions of the standard.

Add support for async RAII

The paper P1662R0 "Adding async RAII support to coroutines" describes some use-cases and motivation for adding support for async RAII in future.

The general idea is to add the ability to define an async operation that is awaited at the end of an async scope to allow performing async cleanup operations or asynchronously waiting for concurrent operations to complete before exiting a scope and destroying a resource they were using.

This is expected to be an important feature to add for coroutines to simplify the ability to write correct concurrent code in the presence of exceptions or other non-trivial control-flow.

However, adding support for async RAII means that we could no longer use `coroutine_handle::destroy()` to cancel an async generator as there may be some async cleanup operations that need to complete before the coroutine can be safely destroyed.

If we do not adopt the changes proposed in this paper for C++20 then if we later add support for async RAII we will have to find some other mechanism for cancelling async generators currently suspended at a `co_yield` expression. The two possibilities is either cancel it by throwing an exception from the `co_yield` expression or return a status-code from the `co_yield` expression which needs to be manually checked.

Adding the `set_done()` signal proposed in this paper provides an async cancellation resumption path for cancelling async generators, allowing us to keep the same cancellation model as synchronous generators, even in the presence of async RAII.

Add support for async return-value optimisation

The paper P1663R0 "Supporting return-value optimisation in coroutines" describes the use-cases and motivation for adding support for return-value optimisation in coroutines.

One of the key capabilities needed to be able to support return-value optimisation for coroutines is to allow awaitable types to directly construct the result of a `co_await` or `co_yield` expression rather than having to store the value somewhere, resume the coroutine and return a copy from `await_resume()`.

To support this capability, the suspend-point handle passed to a co_await expression that uses RVO would have `.set_value<T>(ctorArgs...)` and `.set_exception(eptr)` methods instead of `.resume()`. The suspend-point handle would have knowledge of the return-address of the `co_await` expression and calling `set_value()` would directly construct the result of the `co_await` expression at that address.

This would require a new Awaitable concept that had a different interface - it would not need the `await_ready()` and `await_resume()` methods of the current Awaitable concept.

For example: A simple awaitable type with current awaitable interface vs async RVO awaitable interface

```
struct simple_awaitable {
  coroutine_handle<> coro_;
  std::variant<std::monostate,
               T,
               exception_ptr> result_;

  bool await_ready() { return false; }

  void await_suspend(coroutine_handle<> h) {
    coro_ = h;
    start();
  }

  T await_resume() {
    // Branch required after result
    // to handle value/error case
    if (result_.index() == 2) {
      std::rethrow_exception(
        std::get<2>(result_));
    }

    // Move-construct resulting value.
    return std::get<1>(std::move(result_));
  }

private:
  void start();

  void on_complete(const X& x) {
    // Construct result
    result_.emplace<1>(x);
    coro_.resume();
  }

  void on_error(std::exception_ptr e) {
    result_.emplace<2>(e);
    coro_.resume();
  }

};
```

```
struct simple_awaitable {
  std::suspend_point_handle<
    with_value<T>, with_exception> sp_;

  // Typedef to indicate what the result type
  // will be.
  using await_result_type = T;

  template<typename SuspendPoint>
  auto operator co_await(SuspendPoint sp) {
    sp_ = sp;
    start();
    return std::noop_continuation();
  }

private:
  void on_complete(const X& x) {
    // Directly construct result (no copy)
    auto continuation = sp_.set_value<T>(x);

    // Directly resume value path (no branch)
    continuation();
  }

  void on_error(std::exception_ptr e) {
    // Copies exception_ptr to coroutine frame
    auto continuation = sp_.set_exception(e);

    // Directly resume error path (no branch)
    continuation();
  }
};
```

Adding this capability requires the ability to support multiple possible resumption paths and also requires allowing different suspend-points to use different handle types for each operation.

Support for this style of suspend-point handle, with `set_value()` and `set_exception()` methods, is also a pre-requisite for supporting heterogeneous resume (see later section).

Add support for async range-based for-loops

The Coroutines TS originally included the definition of a 'for co_await' range-based for-loop that was based on a similar pattern to that of synchronous ranges using iterators but where the advancement of an iterator was an async operation invoked with co_await.

However, this capability was removed from the draft pending further investigation into the design of async-ranges as it was not clear that an iterator-based lowering was necessarily going to be the most appropriate representation for async ranges.

There are some interesting possibilities for the lowering of a 'for co_await' loop if we have a co_await expression able to have set_done(), set_error() and set_value() continuations.

If the loop variable were initialised with the result of a co_await expression then the set_done() continuation would map to end-of-stream and exit the loop, the set_value() continuation would construct the value for the loop variable for the next iteration of the loop and the set_error() continuation would exit the loop with an exception.

More research is required in this area.

Add support for co_return as a suspend-point

The current behaviour of `co_return` is to immediately execute '`goto final_suspend;`' after the call to `promise.return_value/return_void` returns. We cannot currently suspend the coroutine at the `co_return` statement.

Adding support for turning `co_return` into a suspend-point, similar to `co_yield` except without the ability to resume with a value, would be useful for some use-cases.

For example, the ability to suspend at a `co_return` suspend-point would allow an `async_generator` to resume its consumer with the 'end-of-stream' value prior to running cleanup of the in-scope objects. When this is combined with async RAII, which means that cleaning up in-scope objects might not complete synchronously, this can reduce the latency involved in the consumer receiving and processing the end-of-stream signal.

With the introduction of the `set_done()` operation we could expose the `co_return` keyword as a suspend-point where `set_done()` is the only continuation path supported by that kind of suspend-point.

This would make `co_return` logically and semantically similar to `co_yield`, except the compiler can take advantage of the fact that the `co_return` suspend-point has no 'value' continuation to do things like dead-code elimination.

Add support for non-type-erased coroutines

Supporting non-type-erased coroutines is an important step to being able to defer creation of the coroutine frame, defer choice of the promise type and expose coroutine-frames as types.

The paper P1342R0 "Unifying Coroutines TS and Core Coroutines" discusses a potential direction for incrementally adopting these changes to the current design. This same incremental approach would also be applicable on top of the changes proposed in this paper.

Add support for delaying choice of promise-type

The ability to delay the choice of the promise_type to use for a coroutine until some point after the coroutine was invoked, for example when the returned task is subsequently awaited, will allow many important use-cases.

For example, it will allow context to be injected automatically from the awaiting coroutine into the callee. This can be used to propagate things like allocators, executors and stop tokens automatically to child coroutines without needing to explicitly pass them as parameters to the function and without needing to type-erase them.

The paper [P1342R0] discusses a potential direction for incrementally adding support for this in future.

Add support for exposing coroutine-frames as types

There has been concern raised [P0973R0] about the potential overhead of the implicit heap allocation that is required by the current design of coroutines. While the heap-allocation elison optimisation can often eliminate this overhead, it is not guaranteed.

The ability to represent a coroutine-frame as a type in the C++ type system would allow a program to statically guarantee that coroutine frames were not heap allocated. However, representing the coroutine frame as a type has some challenges which have been discussed in [P1492R0].

The paper [P1342R0] discusses a potential direction for incrementally adding support for this if and when these challenges are solved.

Add support for heterogeneous resume

You can think of a coroutine as being a function the compiler transforms into a state-machine and where `co_await` expressions are a kind of async `operator()` that invokes the awaitable object passed as the

operand with the rest of the function as a lambda/callback that captures a reference to the coroutine state and that has a body that represents the rest of the function body after the co_await expression.

Much of the benefit of coroutines comes from it simplifying callback-based code that you would previously have to had to manually break up into individual callbacks.

Ideally, coroutines could be used to replace and simplify many existing uses of callbacks.

However, some callback-based APIs allow the caller to pass a generic callback that has an overloaded or generic `operator()` and where the callback is invoked with one of several possible types.

For example, the paper [P1341R0] describes the Sender/Receiver design for representing callback-based asynchronous operations and describes the impedance mismatch between what is possible with coroutines and Awaitable types today and what is possible with the general callback-based model of Sender/Receiver.

We have been exploring the potential for evolving coroutines to also be able to support operations invoking the continuation with one of several possible types, something we have been calling "heterogeneous resume".

Example: A hypothetical coroutine design that matches the semantics of callbacks

<table>
<tr><th>Callback-based code</th><th>Equivalent coroutine-based code</th></tr>
<tr><td><pre>
template<typename T>
concept Logger = ...;

template<typename Callback>
void with_logger(std::string_view key,
                 Callback cb)
{
  if (key == “syslogd”) {
       cb(syslog_logger{});
  } else if (key.starts_with(“file:”)) {
       cb(file_logger{key.substr(5)});
  } else {
       cb(noop_logger{});
  }
}

void example(int count)
{
  with_logger(
    getenv(“LOGGER”),
    [&](Logger auto logger)
    {
      for (int i = 0; i < count; ++i)
      {
        logger.log(“work start {0}”, i);
        do_work(i);
        logger.log(“work end {0}”, i);
</pre></td><td><pre>
template<typename T>
concept Logger = ...;

auto get_logger(std::string_view key)
  [->] lazy<Logger>
{
  if (key == “syslogd”) {
    co_return syslog_logger{};
  } else if (key.starts_with(“file:”)) {
    co_return file_logger{key.substr(5)};
  } else {
    co_return noop_logger{};
  }
}

void example(int count)
{
  Logger auto logger =
    co_await get_logger(getenv(“LOGGER”));

  for (int i = 0; i < count; ++i)
  {
    logger.log(“work start {0}”, i);
    do_work(i);
    logger.log(“work end {0}”, i);
  }
}
</pre></td></tr>
</table>

``
        }
    });
}
``	

In this example, the continuation of the `co_await` expression is instantiated multiple times, each with a different type, just like with the callback example. There are multiple resumption paths that can be taken and each `co_return` statement in `get_logger()` will dispatch to the corresponding resumption path.

There is no type-erasure going on here with either of these implementations.
The goal is to make the RHS simply syntactic sugar for what appears on the LHS.

The ability to resume a coroutine with multiple possible types could be used to replace `std::visit()`, iterate over elements of a `std::tuple`, or other variable-length heterogeneous sequences using a '`for co_await`' loop.

This capability is part of a long-term vision we have for unifying callbacks with coroutines. This capability should be able to be built incrementally on top of changes described in [P1342R0], [P1662R0] and this paper. More research is required to resolve some implementation challenges, however.

Implementation Experience

A prototype of the changes proposed by this paper for C++20 has been implemented in Clang for the purposes of evaluating the implementability and library impacts of the revised design.

The implementation can be found at: https://github.com/lewissbaker/llvm/tree/P1745

Note: This implementation also incorporates the changes proposed by P1713R0 to allow both `return_value()` and `return_void()` to be defined on the same `promise_type`.

The compiler changes required in the Clang front-end were straight forward to implement as incremental changes on top of the existing coroutines implementation.

The changes to merge `get_return_object()`/`initial_suspend()` and simplify `final_suspend()` simplified the logic required to implement codegen for the corresponding parts of the coroutine.

Rather than have the compiler procedurally generating the per-suspend-point handle type definitions, the prototype implementation simply instantiates templated implementations of the per-suspend-point handle types that are provided by a standard library, passing an anonymous tag struct created for each suspend-point as one of the template arguments to ensure these types were unique.

A production implementation could exercise more control over the codegen for the handle types if it were to procedurally generate the type definitions; for example to avoid generating code for unused resumption-paths if it detected that the corresponding method on the per-suspend-point handle type was never instantiated.

No changes were required to the LLVM intrinsics for coroutines at the IR level to implement the prototype. However, some changes to the coroutine intrinsics would be necessary to generalise it to support more than two resumption paths for a given suspend-point (the current intrinsics have a hard-coded 'destroy' and 'resume' resumption-path).

Further changes to the LLVM intrinsics would be required to take advantage of the extra optimisation possibilities per-suspend-point handle types and the ability to store coroutine suspension state in the handles offers.

The <experimental/coroutine> header in libc++ was updated to define the `suspend_point_handle` and `continuation_handle` types.

Implementations of some basic coroutine types, task<T> and generator<T>, were modified to incorporate the proposed changes. These types required relatively small and straight-forward changes, mostly consisting of changing some type-names, removing methods from the promise_type and adding an extra method call to create/invoke the continuation. See Appendix A for some examples.

A subsequent revision of this paper will report on further implementation experience gained once the conversions of existing open-source coroutine libraries to use the revised changes have been completed.

Bikeshedding

Bikeshedding coroutine resumption method names

The proposal as it stands uses the name resume() for the method that returns a `continuation_handle` which resumes the coroutine with a result for consistency with the naming of the existing `coroutine_handle::resume()` function. However, this naming may be confusing since the method does not actually resume the coroutine but returns a handle that needs to be invoked to resume the coroutine.

Some alternative names for the coroutine resumption methods have been considered and can be bikeshedded if necessary:

Operation	**Current proposed name**	**Potential alternative names**
Resume coroutine with result of calling	`.resume()`	`.continue_with_result()` `.set_result()`

await_resume()		
Resume coroutine and immediately execute 'goto done;'	.set_done()	.continue_with_done() .continue_with_break() .stop() .cancel()
Resume coroutine with an RVO-constructed value of type T	.set_value<T>(args...)	.continue_with_value<T>(args...) .
Resume coroutine by calling std::rethrow_exception().	.set_exception(eptr)	.continue_with_exception(eptr)
Resume coroutine by executing 'throw err;'	.set_error<E>(err)	.continue_with_error<E>(err)

Notes:

- Using the name 'result' instead of 'resume' may tie in with proposed naming change in P1610R0 which proposes renaming await_resume() to await_result().
- The naming with the 'continue_' prefix is intended to convey an association with the term 'continuation_handle'.
- The term 'break' would tie in with the use of this continuation path as a way of breaking out of a 'for co_await' loop.

References

[P0913R0] - "Add symmetric coroutine control transfer" - Gor Nishanov
[P0973R0] - "Coroutines TS Use Cases and Design Issues" - Geoff Romer, James Dennett
[P0981R0] - "Halo: coroutine Heap Allocation eLision Optimisation: the joint response" - Richard Smith, Gor Nishanov
[P1341R0] - "Unifying Asynchronous APIs in the Standard Library" - Lewis Baker
[P1342R0] - "Unifying Coroutines TS and Core Coroutines" - Lewis Baker
[P1477R1] - "Coroutines TS Simplifications" - Lewis Baker
[P1492R0] - "Coroutines: Language and Implementation Impact"
[P1493R0] - "Coroutines: Use-cases and Trade-offs"
[P1610R0] - "Rename await_resume() to await_result()" - Mathias Stearn
[P1660R0] - "A compromise executor design sketch"
[P1662R0] - "Adding async RAII support to coroutines" - Lewis Baker
[P1663R0] - "Supporting return-value-optimisation in coroutines" - Lewis Baker
[P1667R0] - "Cancellation is not an error" - Kirk Shoop

[P1713R0] - "Allowing both co_return; and co_return value; in the same coroutine" - Lewis Baker

Acknowledgements

Many thanks for the comments and feedback from Kirk Shoop, Eric Niebler, Lee Howes and Gor Nishanov.

Appendix A - Examples

Example: A simple async event type that shows usage of type-erased handles

```
class async_event {
  class awaiter {
  public:

    bool await_ready() noexcept { return event_.ready(); }

    template<AwaitSuspendPointHandle SP>
    std::continuation_handle await_suspend(SP sp) noexcept {
      /
      sp_ = sp;
      void* oldState = nullptr;
      if (event_.state_.compare_exchange_strong(oldState, this)) {
        return std::noop_continuation();
      } else {
        // Use the non-type-erased suspend-point handle.
        return sp.resume();
      }
    }

    void await_resume() noexcept {}

  private:
    friend class async_event;

    explicit awaiter(async_event& e) noexcept : event_(e) {}

    async_event& event_;
    std::suspend_point_handle<with_resume> sp_;
  };

  void set() {
    void* oldState = state_.exchange(this);
    if (oldState != nullptr && oldState != this) {
      awaiter* a = static_cast<awaiter*>(oldState);
      a->sp_.resume()();
    }
  }

  bool ready() const noexcept { return state_.load() == this; }

  awaiter operator co_await() noexcept {
    return awaiter{*this};
  }

private:
  std::atomic<void*> state_ = nullptr;
};
```

Example: An async task<T> coroutine type before and after the proposed changes.

```
template<typename T>
struct task {
  struct promise_type {
    std::optional<T> value_;
    std::exception_ptr error_;
    std::coroutine_handle<> consumer_;


    task get_return_object() {
      return task{
        std::coroutine_handle<promise_type>::
          from_promise(*this)};
    }

    suspend_always initial_suspend() { return {};
}

    auto final_suspend() {
      struct awaiter {
        bool await_ready() { return false; }
        auto await_suspend(
            std::coroutine_handle<promise_type> h)
{
          return h.promise().consumer_;
        }
        void await_resume() {}
      };
      return awaiter{};
    }

    template<ConvertibleTo<T> U>
    void return_value(U&& value) {
      value_.emplace((U&&)value);
    }

    void unhandled_exception() {
      error_ = std::current_exception();
    }
  };

  using handle_t =
    std::coroutine_handle<promise_type>;



  handle_t coro_;

  explicit task(handle_t h) : coro_(h) {}

  task(task&& t)
  : coro_(std::exchange(t.coro_, {})) {}

  ~task() {
    if (coro_) coro_.destroy();
  }

  struct awaiter {
    handle_t coro_;

    bool await_ready() { return false; }

    auto await_suspend(auto h) {
```

```
template<typename t>
struct task {
  struct promise_type {
    std::optional<T> value_;
    std::exception_ptr error_;
    std::suspend_point_handle<
      std::with_resume, std::with_done> consumer_;

    template<typename Handle>
    task get_return_object(Handle h) {
      return task{h};
    }



    auto done() {
      if (value_ || error_) {
        return consumer_.resume();
      } else {
        return consumer_.set_done();
      }
    }




    template<ConvertibleTo<T> U>
    void return_value(U&& value) {
      value_.emplace((U&&)value);
    }

    void unhandled_exception() {
      error_ = std::current_exception();
    }
  };

  using handle_t =
    std::suspend_point_handle<
      std::with_resume, std::with_destroy,
      std::with_promise<promise_type>>;

  handle_t coro_;

  explicit task(handle_t h) : coro_(h) {}

  task(task&& t)
  : coro_(std::exchange(t.coro_, {})) {}

  ~task() {
    if (coro_) coro_.destroy();
  }

  struct awaiter {
    handle_t coro_;

    bool await_ready() { return false; }

    auto await_suspend(auto h) {
      coro_.promise().consumer_ = h;
      return coro_.resume();
```

```
      coro_.promise().consumer_ = h;
      return coro_;
    }

    T await_resume() {
      auto& error = coro_.promise().error_;
      if (error) std::rethrow_exception(error);
      return *std::move(coro_.promise().value_);
    }
  };

  auto operator co_await() && {
    return awaiter{coro_};
  }
};
```

```
    }

    T await_resume() {
      auto& error = coro_.promise().error_;
      if (error) std::rethrow_exception(error);
      return *std::move(coro_.promise().value_);
    }
  };

  auto operator co_await() && {
    return awaiter{coro_};
  }
};
```

Example: A simple generator<T> coroutine type before and after the proposed changes

```
template<typename T>
struct generator {
  struct promise_type {
    std::add_pointer_t<T> value_;

    generator get_return_object() {
      return generator{
        std::coroutine_handle<promise_type>::
          from_promise(*this)};
    }

    std::suspend_always initial_suspend() {
      return {};
    }
    std::suspend_always final_suspend() {
      return {};
    }

    void return_void() {}

    void unhandled_exception() {
      throw;
    }

    std::suspend_always yield_value(T&& value) {
      value_ = std::addressof(value);
      return {};
    }
  };
```

```
template<typename T>
struct generator {
  struct promise_type {
    std::add_pointer_t<T> value_;
    std::suspend_point_handle<
      with_set_done, with_resume> sp_;

    template<typename Handle>
    generator get_return_object(Handle h) {
      sp_ = h;
      return generator{h};
    }

    auto done() {
      sp_ = {};
      return std::noop_continuation();
    }

    void return_void() {}

    void unhandled_exception() {
      sp_ = {};
      throw;
    }

    struct yield_awaiter {
      bool await_ready() { return false; }
      template<typename Handle>
      auto await_suspend(Handle h) {
        h.promise().sp_ = h;
        return std::noop_continuation();
      }
      void await_resume() {}
    };

    yield_awaiter yield_value(T&& value) {
      value_ = std::addressof(value);
      return {};
    }
  };
```

```
  using handle_t =
    std::coroutine_handle<promise_type>;


  handle_t coro_;

  explicit generator(handle_t h) : coro_(h) {}

  generator(generator&& g)
  : coro_(std::exchange(g.coro_, {})) {}

  ~generator() {
    if (coro_) {
      coro_.destroy();
    }
  }




  struct sentinel {};

  struct iterator {
    handle_t coro_;

    iterator& operator++() {
      coro_.resume();
      return *this;
    }

    T&& operator*() const {
      return (T&&)*coro_.promise().value_;
    }

    auto* operator->() const {
      return coro_.promise().value_;
    }

    bool operator==(sentinel) const {
      return coro_.done();
    }

    bool operator!=(sentinel s) const {
      return !operator==(s);
    }
  };

  iterator begin() {
    coro_.resume();
    return iterator{coro_};
  }

  sentinel end() { return {}; }
};
```

```
  using handle_t = std::suspend_point_handle<
    std::with_resume, std::with_destroy,
    std::with_promise<promise_type>>;

  handle_t coro_;

  explicit generator(handle_t h) : coro_(h) {}

  generator(generator&& g)
  : coro_(std::exchange(g.coro_, {})) {}

  ~generator() {
    if (coro_) {
      if (coro_.promise().sp_) {
        coro_.promise().sp_.set_done()();
      }
      coro_.destroy();
    }
  }

  struct sentinel {};

  struct iterator {
    handle_t coro_;

    iterator& operator++() {
      coro_.promise().sp_.resume()();
      return *this;
    }

    T&& operator*() const {
      return (T&&)*coro_.promise().value_;
    }

    auto* operator->() const {
      return coro_.promise().value_;
    }

    bool operator==(sentinel) const {
      return !!coro_.promise().sp_;
    }

    bool operator!=(sentinel) const {
      return !operator==(s);
    }
  };

  iterator begin() {
    coro_.resume()();
    return iterator{coro_};
  }

  sentinel end() { return {}; }
};
```

Example: A detached-task that launches immediately and self-destructs when it runs to completion

<table>
<tr><th>Coroutines TS</th><th>With proposed changes</th></tr>
<tr><td>

```
struct detached_task {
  struct promise_type {



    detached_task get_return_object() {

      return {};
    }

    std::suspend_never initial_suspend() {
      return {};
    }

    std::suspend_never final_suspend() {
      return {};
    }


    void unhandled_exception() { std::terminate();
}
    void return_void() {}
  };
};
```

</td><td>

```
struct detached_task {
  struct promise_type {
    std::suspend_point_handle<with_destroy> sp_;

    template<typename Handle>
    detached_task get_return_object(Handle handle)
{
      sp_ = handle;
      return {};
    }






    auto done() {
      sp_.destroy();
      return std::noop_continuation();
    }

    void unhandled_exception() { std::terminate();
}
    void return_void() {}
  };
};
```

</td></tr>
</table>