Document	P1662R0
Date	2019-06-10
Reply To	Lewis Baker <lbaker@fb.com>
Audience	Evolution
Targeting	C++20, C++23

Adding async RAII support to coroutines

Abstract

One of the powerful idioms used throughout C++ codebases is that of RAII (resource acquisition is initialisation). The compiler ensures that destructors are automatically called on all code-paths exiting the scope of any local variable, regardless of whether by flowing off the end of the scope, by some non-linear control flow like a `goto`, `break` or `return` statement, or when unwinding due to an exception.

This paper describes the motivation and use cases for adding support for async RAII to coroutines - i.e. the ability to have the compiler automatically await an asynchronous operation on all code-paths exiting a scope.

While this paper explores some design alternatives being considered for async RAII, it is not proposing any particular syntax or mechanism at this point in time. Rather, this paper looks at the general impacts that adding support for async RAII in a future release post C++20 will have on the existing coroutines design, regardless of the chosen syntax.

In particular, this paper identifies that the current design of the `coroutine_handle::destroy()` method means that if we were to add support for async RAII in future then it would no longer be safe in general to destroy a coroutine at suspend-points other than the initial and final suspend points as there might be some asynchronous cleanup work in scope that needs to complete first.

This paper explores the impact this limitation has on the programming model we will be able to expose for an `async_generator<T>` coroutine type and how this might also affect the programming model we expose for a synchronous `generator<T>` coroutine.

Finally, this paper recommends making some changes to the design of `coroutine_handle` before C++20 is finalized to allow the coroutines feature to be more cleanly extended to support async RAII in a future version of C++. These changes only affect people writing new types of coroutines or awaitables, they don't affect the syntax for users authoring coroutines.

A prototype of these changes have been implemented in a fork of Clang which can be used to evaluate their impacts on coroutine code.

It is important to note that these proposed changes also have overlap with the changes needed to support adding async return value optimisation [P1663R0] and heterogeneous resumption in future, should we choose to. The paper [P1745R0] contains a detailed description of the proposed changes for C++20 and a roadmap of proposed future changes. Please consider reading this paper in conjunction with those papers for the wider picture.

Motivation

One of the powerful design patterns used throughout C++ codebases is that of RAII (resource acquisition is initialisation) - where resources are acquired in a constructor and released by the destructor.

When an object is created with automatic storage duration (ie. as a local variable in a function) the compiler ensures that the destructor is called on all code-paths that exit the scope of that object (whether by normal control flow, exception or return). This in turn ensures that resources that were acquired by the object are deterministically released without the programmer having to explicitly release the resources on all code-paths.

However, there are also use-cases where we want to be able to perform an **async** cleanup operation on exit of a scope within a coroutine. For example, the cleanup operation might need to perform some network or file I/O or wait for some concurrent operation to complete on another thread before continuing the unwind, preferably without blocking the current thread.

The idea behind async RAII is to be able to have the compiler insert a `co_await` expression automatically on all code-paths that exit a scope, in much the same way that the compiler ensures calls to destructors are inserted on all code-paths that exit a scope.

For example, one hypothetical syntax for this might be via a new `co_using` statement.

```
task<int> count_matches(std::regex pattern)
{
  // co_using creates a scope that runs an async operation at scope exit.
  co_using (auto file : open_file_async())
  {
    int count = 0;
    for co_await (auto& record : read_records(file)) {
      if (std::regex_match(record.name, pattern))
        ++count;
    }
    co_return count;
  } // <-- Implicit 'co_await file.close_async();' on scope exit.
}
```

Use Cases

Waiting for concurrent operations to finish

The cppcoro library provides a type called `async_scope` that can be used to eagerly spawn off some concurrent asynchronous work and then later join that scope to wait for all of the spawned work to complete.

Example: Simple usage of `cppcoro::async_scope`

```
cppcoro::task<void> do_something(T& resource, int i);

cppcoro::task<void> example1() {
  T sharedResource;

  cppcoro::async_scope scope;

  for (int i = 0; i < 10; ++i) {
    // Launch work on the current thread.
    // Executes inline until and returns from spawn() once
    // the coroutine reaches its first suspend point.
    scope.spawn(do_something(sharedResource, i));
  }

  // Wait until all spawned work runs to completion
  // before 'sharedResource' goes out of scope.
  co_await scope.join();
}
```

However, the above code does not correctly handle the case where an exception is thrown by the call to either `do_something()` or `scope.spawn()`. e.g. due to an out of memory failure. If an exception is thrown then the `example1()` coroutine will exit without waiting for `scope.join()` to complete. This could leave the already-spawned tasks with a dangling reference to `sharedResource`.

To fix this we would need to catch the exception and manually ensure that `co_await scope.join()` is executed on all code-paths exiting the scope.

Example: Manually ensuring the async work is joined on all paths.

```
cppcoro::task<void> example2() {
  T sharedResource;
  cppcoro::async_scope scope;
  std::exception_ptr ex;
  try {
    for (int i = 0; i < 10; ++i) {
      scope.spawn(do_something(sharedResource, i));
    }
  } catch (...) {
```

```
    ex = std::current_exception();
  }
  co_await scope.join();
  if (ex) {
    std::rethrow_exception(ex);
  }
}
```

However, the above code is cumbersome to write and error-prone. Someone might later modify this code to add a `co_return` inside the try/catch block and forget to add a call to `co_await scope.join()`.

Note that the above code could be simplified somewhat if the language were extended to support using `co_await` within catch-blocks but in general it would be much better to support some kind of async RAII/scope that automatically performed the async join on scope-exit.

One potential direction is to add a declaration form of the `co_await` keyword that introduces a scope that performs asynchronous cleanup on exiting the scope (see the later section for details on this). It's usage might look something like this:

Example: One suggestion is to add the ability to declare a local variable to be an async scope by prefixing it with the `co_await` keyword

```
cppcoro::task<void> example3() {
  T sharedResource;
  co_await auto&& scope = async_scope{};
  for (int i = 0; i < 10; ++i) {
    scope.spawn(do_something(sharedResource, i));
  }
} // <-- Implicit 'co_await scope.join()' at end of async scope
```

In this example, the compiler ensures that the scope is joined on all paths that exit the scope. The code is simpler and less error-prone.

Parallels with std::thread and std::jthread

This need to join concurrent operations manually at the end of the scope is similar to the need to call `std::thread::join()` before the `std::thread` object goes out of scope.

Doing this manually is also error-prone and so many developers tend to create some kind of RAII wrapper that calls `join()` on the thread automatically in the destructor. The paper P0660R9 proposes adding a new type, `std::jthread`, to the standard library that is similar to `std::thread` except that it automatically calls `join()` in the destructor.

Unfortunately, we cannot currently take the same approach for performing asynchronous join operations. Destructors must complete synchronously and therefore cannot be coroutines and cannot `co_await` an asynchronous `join()` operation.

Composable async streams

Another key important use-case is to support performing async cleanup when an async generator is cancelled before it runs to completion.

This is different to the example above in that with the current design of generator types it is not currently possible to perform async cleanup, even writing the code manually.

First, let us cover some background on the current design of synchronous generators.

Synchronous generators and cancellation

The general design of a synchronous coroutine-based `generator<T>` type is to have the coroutine suspend at `co_yield` statements and return a reference to the argument back to the consumer.

Once the consumer is finished processing that element the consumer can perform one of two operations with regards to the generator coroutine:
It can ask for the next element by calling `iterator::operator++()`, which ends up resuming the coroutine by calling `coroutine_handle::resume()`.
Or it can destroy the `generator<T>` object, which ends up calling `coroutine_handle::destroy()`. This then calls the destructors for any in-scope objects at the most-recent suspend-point and then destroys the coroutine frame, effectively cancelling the execution of the remainder of the coroutine.

Example: Consuming a sequence of random numbers

```
generator<int> random(int min, int max, int count) {
  std::mt19937_64 gen;
  std::uniform_int_distribution<int> dist(min, max);
  while (count-- > 0) {
    co_yield dist(gen());
  }
}

void consumer() {
  for (int i : random(0, 100, 20)) {
    if (i == 42) break;
    printf("%i\n", i);
  }
}
```

The consumer in the above example might either consume the entire sequence or might exit the loop early if it finds the right value.

If the consumer exits the loop early then the generator destructor is called and the coroutine frame, which was currently suspended at the `co_yield` expression, is also destroyed along with all of the in-scope objects in coroutine.

This effectively cancels the execution of the rest of the coroutine and the only code that the coroutine gets to run in response to this cancellation are the destructors of the in-scope objects.

Asynchronous Generators

We can extend the synchronous generator to allow producing values asynchronously by mixing use of `co_yield`/`co_await`. Retrieving the next value from the stream would be an asynchronous operation and require the consumer to use `co_await` to pull the next value.

Example: A simple async generator that produces an infinite stream of values over time.

```
// Produce an integer ~100ms after the consumer asks for one.
async_generator<int> ticker()
{
  constexpr auto delay = 100ms;
  for (int i = 0;; ++i) {
    co_yield i;
    co_await async_sleep_for(delay);
  }
}

task<void> consumer()
{
  const auto start = steady_clock::now();
  auto generator = ticker();
  while (std::optional<int> x = co_await generator.next()) {
    auto elapsedMS = duration_cast<milliseconds>(steady_clock::now() - start);
    std::cout << elapsedMS.count() << "ms - tick " << x.value() << std::endl;
    if (elapsed > 10s) break;
  }
}
```

There may be cases where an async generator makes use of some kind of resource that needs to perform an async operation in order to cleanly release that resource.

For example, a generator that yields results from a database query might want to hold a database connection open while the consumer is pulling results and then cleanly disconnect once the consumer has finished pulling all the results they need. The disconnect operation may need to perform some network I/O and so should be asynchronous.

Example: An async generator with some async cleanup work to do

```
struct record {
  int id;
  std::string name;
  std::string email;
};
```

```
async_generator<record> get_records()
{
  auto connection = co_await connect_to_database();

  auto resultSet = co_await connection.execute_query("SELECT * FROM records;");
  while (!resultSet.empty()) {
    // Process a page of results at a time.
    for (auto& row : resultSet.rows()) {
      co_yield record{
        row.getInt("id"),
        row.getString("name"),
        row.getString("email")};
    }
    co_await resultSet.move_next_page();
  }

  // Cleanly disconnect once finished.
  co_await connection.disconnect();
}

task<std::string> find_bobs_email()
{
  async_generator<record> records = get_records();
  while (std::optional<record> record = co_await records.next()) {
    if (record->name == "bob")
      co_return record->email;
  }
  co_return "";
}
```

However, if the `async_generator<T>` coroutine type follows the same cancellation model as the synchronous `generator<T>` (ie. cancels by destroying the coroutine) then the `co_await connection.disconnect()` will not be run unless the consumer consumes the entire sequence. The only logic that the coroutine gets to run when it is cancelled with `destroy()` is in the destructors of in-scope objects. Destructors cannot be coroutines and therefore cannot await async operations.

This means that to be able to support async cleanup, regardless of the mechanism, we can no longer use `destroy()` to cancel the generator coroutine.

We would instead need to resume the coroutine by calling `coroutine_handle::resume()` and somehow signal to the coroutine that it has been cancelled so that it can run any async cleanup work before exiting promptly.

Under the current coroutines design, a coroutine has several options for signalling cancellation:

- It can return a status code from the `co_yield` expression that indicates whether the consumer has cancelled the coroutine or not.
- It can throw an exception from the `co_yield` expression and let that propagate out of the coroutine body.

- It can store the request to cancel internally in the promise which can later be queried using something like '`co_await is_stop_requested`' which is hooked by providing `promise_type::await_transform(is_stop_requested_t)` to extract the value from the promise.

None of these approaches is ideal.

If we were to return a value from the `co_yield` expression, e.g. a `generator_status` enum value, then this would require checking the return value for every `co_yield` expression, which can be error prone and can complicate the control flow of the generator body.

If we were to throw an exception on cancellation, eg. `operation_cancelled`, then the coroutine promise would need to catch that exception and ignore it.

Also, the use of exceptions is disabled in many environments and so having the control-flow of a `std::async_generator` type depend on exceptions would make it unusable in those environments.

The recent C++ developer survey, [2019AnnualSurvey], indicated that more than 20% of developers worked in environments that don't allow use of exceptions anywhere in the project, and a further 25% indicated that exceptions were not permitted in some parts of their project.

The performance overhead of throwing/catching exceptions is typically much higher than calling `.destroy()`, although [P1676R0] has shown that the overhead can be greatly reduced in some cases through compiler optimisations.

If we were to adopt one of these cancellation models for `async_generator<T>` then should we also try to make the programming model consistent for the synchronous `generator<T>`?

It seems reasonable to argue that we should be able to easily convert a `generator<T>` coroutine to an `async_generator<T>` by just changing the return-type and then sprinkling `co_await` where necessary. However, if the cancellation models were different then you'd need to update every `co_yield` to expression to check the return value when converting between them.

Let's revisit the previous database query example and implement correct async-cleanup using the approach where `co_yield` returns a status-code.

For example: Modified example to support async cleanup using status codes and manual calls to cleanup.

```
async_generator<record> get_records()
{
```

```
  auto connection = co_await connect_to_database();

  std::exception_ptr ex;
  try {
    auto resultSet = co_await connection.execute_query("SELECT * FROM records;");
    while (!resultSet.empty()) {
      // Process a page of results at a time.
      for (auto& row : resultSet.rows()) {
        // co_yield expression returns a status code.
        // Every co_yield result needs to be checked
        generator_status status = co_yield record{
          row.getInt("id"),
          row.getString("name"),
          row.getString("email")};
        if (status == generator_status::cancelled) goto cleanup;
      }
      co_await resultSet.move_next_page();
    }
  } catch (...) {
    ex = std::current_exception();
    goto exception;
  }

cleanup:
  // Cleanly disconnect once finished.
  co_await connection.disconnect();
  co_return;

exception:
  try {
    co_await connection.disconnect();
  } catch (...) {
    std::terminate();
  }
  std::rethrow_exception(std::move(ex));
}
```

On the consumer side, we would need to manually ensure that we awaited the cleanup operation before the async_generator object goes out of scope. e.g.

```
task<std::string> find_bobs_email()
{
  async_generator<record> records = get_records();

  std::exception_ptr ex;
  try {
    while (std::optional<record> record = co_await records.next()) {
      if (record->name == "bob") {
        // Remember to manually close the stream if we exit early.
        co_await records.cancel();
        co_return record->email;
      }
    }
  } catch (...) {
    ex = std::current_exception();
    goto exception;
  }

  // Remember to close the stream on the normal code-path.
```

```
  co_await records.cancel();

  co_return "";

exception:
  // Remember to close the stream if we exited the loop early because
  // of an exception.
  try {
    co_await records.cancel();
  } catch (...) {
    std::terminate();
  }
  std::rethrow_exception(ex);
}
```

And this code is only showing the need to handle the cleanup of a single resource with async cleanup. If we have multiple nested resources that each need to perform async cleanup operations then the control flow gets much more complicated.

If, instead, we were able to make use of an async RAII facility then the code becomes much simpler (shown here using the hypothetical `co_await` declaration syntax).

```
async_generator<record> get_records()
{
  co_await auto connection = connect_to_database();

  auto resultSet = co_await connection.execute_query("SELECT * FROM records;");
  while (!resultSet.empty()) {
    // Process a page of results at a time.
    for (auto& row : resultSet.rows()) {
      co_yield record{
        row.getInt("id"),
        row.getString("name"),
        row.getString("email")};
    }
    co_await resultSet.move_next_page();
  }
  co_return;
  // Implicit 'co_await connection.disconnect();' on scope exit
}

task<std::string> find_bobs_email()
{
  co_await auto records = get_records();

  while (std::optional<record> record = co_await records.next()) {
    if (record->name == "bob")
      co_return record->email;
  }
  co_return "";
  // Implicit 'co_await records.cancel();' on scope exit
}
```

Design Discussion

Can we just allow destructors to be coroutines/asynchronous?

No. Doing so would effectively bifurcate the C++ type-system and break a lot of existing code.

Let's imagine that we allowed some destructors to be made asynchronous. eg. the destructor returned an awaitable type that was implicitly awaited when the object was destructed.

For example: A hypothetical syntax for declaring async destructors

```
class MessageStream
{
public:
  ~MessageStream() -> task<void> {
       co_await conn.disconnect();
  }
private:
  connection conn;
};
```

These types would be viral in nature. Classes that contained non-static data-members of types that have asynchronous destructors would themselves need to have asynchronous destructors.

Such types would not be able to be deleted using the `delete` operator as that is a synchronous call.

Such types would not be allowed to be placed as local variables in a normal function since the destructor must be awaited which can only be done from a coroutine.

These types would not be able to be placed in standard containers like `std::vector` since they need to be able to call the destructors from non-async methods like `.resize()` and `.pop_back()`.
There is too much code that relies on the fact that destroying an object is a synchronous operation and making destructors potentially asynchronous would make a lot of existing generic code unable to be used with these types.

Declaring a local variable with an async destructor in a coroutine would not necessarily have any syntactic marker within the code to indicate that there is a potential suspend-point at the end-of-lifetime of the object.

One of the key benefits of the current Coroutines TS design is that suspend-points are explicitly called-out by the use of the `co_await/co_yield` keywords. This makes it easier to reason about the implications of potential thread-transitions at suspend-points when reading code.

This advantage would be lost if we were to introduce an implicit suspend-point at the end of the scope of a variable without some kind of syntactic marker at the start of the scope for that variable.

Having said that, what we essentially need to add is the equivalent to an async destructor, but it needs to be one that is layered on top of the existing synchronous destructor mechanism rather than replacing it.

User-facing syntax options

When thinking about how async RAII should be exposed to the user there are a number of things to consider.

Ideally, the object lifetime and scope rules would be as similar to existing C++ scope rules as possible.

How should we introduce the start of a new async scope?

- It would need to be something that could only be done within a coroutine, preferably involving one of the co_ keywords to indicate the presence of a suspend-point.

How should types indicate the presence of some logic to run at the end of an async scope?

- What method(s)/operator(s) should a type implement to define the logic to run at the end of an async-scope?

When should the async cleanup be run?

- At end of current block-scope?
- At semicolon?
- Ideally with the same nesting rules as for normal object lifetimes.

How should asynchronous scopes interact and nest within synchronous scopes?

- Should all async cleanups be run before all synchronous cleanups on end of a scope?
- Or should async cleanups be strictly nested and interleaved with synchronous cleanups?

Adding an `operator ~co_await()`

On of the more promising directions explored for the user-facing syntax is to build the concept of async scopes into the '`co_await`' expression itself.

The general idea is to have every `co_await` expression introduce an async scope. At the start of the scope the compiler would introduce a suspend-point for evaluating `operator co_await()`. At the end of the scope the compiler would introduce another suspend-point for evaluating a new operator, `operator ~co_await()`.

If the awaiter type returned by '`operator co_await()`' defined an '`operator ~co_await()`` then the expression '`co_await` *`async-scope-operand`*' would be lowered into the following code. Assume that the code that runs in the scope of the async scope created by this `co_await` expression is identified by *`async-scope-body`*.

```
{
  decltype(auto) __operand = async-scope-operand;
  decltype(auto) __awaitable =
    promise.await_transform((decltype(__operand)&&)__operand);

  // 'co_await __awaitable.operator co_await()';
  decltype(auto) __entryAwaiter = __awaitable.operator co_await();
  if (!__entryAwaiter.await_ready()) {
    // <suspend-coroutine>
    /*tail*/ return __entryAwaiter.await_suspend(coroutine-handle);
    // <resume-point>
  }

  std::exception_ptr __ex;
  {
    decltype(auto) __tmpResult = __entryAwaiter.await_resume();
    try {
      async-scope-body
      goto __fallthrough;
    } catch (...) {
      __ex = std::current_exception();
      goto __exception:
    }
  }

__exception:
  try {
    // 'co_await __entryAwaiter.operator ~co_await();'
    decltype(auto) __exitAwaiter = __entryAwaiter.operator ~co_await();
    if (!__exitAwaiter.await_ready()) {
      // <suspend-coroutine>
      /*tail*/ return __exitAwaiter.await_suspend(coroutine-handle);
      // <resume-point>
    }
    __exitAwaiter.await_resume();
  } catch (...) {
    std::terminate();
  }
  std::rethrow_exception(std::move(__ex));

__fallthrough:
  // 'co_await __entryAwaiter.operator ~co_await();'
  {
    decltype(auto) __exitAwaiter = __entryAwaiter.operator ~co_await();
    if (!__exitAwaiter.await_ready()) {
      // <suspend-coroutine>
```

```
      /*tail*/ return __exitAwaiter.await_suspend(coroutine-handle);
      // <resume-point>
    }
    __exitAwaiter.await_resume();
  }
}
```

Note that the nesting of the lifetimes of the various objects involved in this expression is in the following order (from outer-most to inner-most scope):

- `__operand`
  - Result of evaluating *`async-scope-operand`*
  - Calls `__operand` destructor on exit of scope.
- `__awaitable`
  - Result of calling `promise.await_transform(__operand)` if applicable, otherwise a forwarding reference to `__operand`.
  - Calls `__awaitable` destructor on exit of scope
- `__entryAwaiter`
  - Result of calling `promise.operator co_await()`
  - Calls `__entryAwaiter` destructor on exit of scope.
- Async Scope
  - Evaluates '`operator co_await()`' suspend-point on entry to scope.
  - Evaluates '`operator ~co_await()`' suspend-point on exit of scope.
    - Note that evaluation of this expression creates a new nested scope for `__exitAwaiter`.
    - The lifetime of this nested scope does not overlap with the lifetime of `__tmpResult`.
- `__tmpResult`
  - Constructed by result of evaluating `__entryAwaiter.await_resume()`.
  - Calls `__tmpResult` destructor on exit of scope.

If there was a `break`, `continue` or `goto` (including the implicit `goto final_suspend;` of a `co_return` statement) inside the *`async-scope-body`* that exited the scope then this would be translated into a jump to a new label that executed `co_await __entryAwaiter.operator ~co_await()` before then jumping to the target label.

Temporary async scopes

Note that by default, async cleanup for a `co_await` expression would be run at the end of the full-expression (ie. at the semicolon terminating the current statement).

For example: Given the following awaitable which implements the async-cleanup pattern by defining `operator ~co_await()` on the awaiter object returned from `operator co_await()`.

```
struct some_awaitable {
```

```
  struct awaiter {
    bool await_ready();
    coroutine_handle<> await_suspend(coroutine_handle<>);
    std::string await_resume();

    struct cleanup_awaiter {
      bool await_ready();
      coroutine_handle<> await_suspend(coroutine_handle<>);
      void await_resume();
    };

    cleanup_awaiter operator ~co_await();
  };

  awaiter operator co_await();
};
```

We can then write the following:

```
some_awaitable get_key();
some_awaitable get_value();

void output(std::string key, std::string value);

task<void> example() {
  output(co_await get_key(), co_await get_value());
}
```

When the semicolon terminating this statement is reached and scopes are exited, the following operations are performed in order (assuming that the compiler evaluates the 'key' parameter expression before evaluating the 'value' parameter expression).

- `std::string` destructor for '`value`' parameter.
- Evaluates '`co_await some_awaitable::awaiter::operator ~co_await()`' for the awaiter produced for the `co_await get_value()` expression.
- `some_awaitable::awaiter` destructor for '`co_await get_value()`' expression
- `some_awaitable` destructor for '`co_await get_value()`' expression
- `std::string` destructor for '`key`' parameter
- Evaluates '`co_await some_awaitable::awaiter::operator ~co_await()`' for the awaiter produced for the `co_await get_key()` expression.
- `some_awaitable::awaiter` destructor for '`co_await get_key()`' expression
- `some_awaitable` destructor for '`co_await get_key()`' expression

Lifetime extension of async scopes

Temporary async scope lifetimes, as described above, are ended at the semicolon terminating the current statement.

We can extend the lifetime of the result to the end of the enclosing block-scope by assigning the result of the co_await expression to a named variable.

For example:

```
task<void> example() {
  // Async cleanup for each operation run at the semicolon.
  std::string key = co_await get_key();
  std::string value = co_await get_value();

  process(key, value);
}
```

With this example the async cleanup for each of the `co_await` expressions still runs at their respective end-of-statement semicolons. Only the lifetime of the result is extended, not the lifetime of the awaitable and awaiter objects used in the `co_await` expression.

However, sometimes we want to be able to extend the lifetime of the whole async scope to the end of the enclosing block-scope so that the async cleanup is run at the closing curly brace rather than at the semicolon.

To support this we need to introduce some kind of new syntax for declaring a new name for the lifetime-extended scope, while still having this name refer to the result of a co_await expression.

One possible strawman syntax for this might be to prefix a variable declaration with the `co_await` keyword.

```
task<void> example() {
  // Async cleanup for each operation run at the closing curly brace.
  co_await std::string key = get_key();
  co_await std::string value = get_value();

  process(key, value);
}
```

Another alternative syntax that has been considered is adding a new `co_using` keyword which introduces a new block-scope, like a range-based for-loop.

```
task<void> example() {
  co_using (std::string key : get_key())
  co_using (std::string value : get_value())
  {
    process(key, value);
  }
}
```

The use of a block-syntax has been used by other languages, although this does not necessarily mean that it is a good fit for C++.

For example, Python has the following:

```
async with EXPR as VAR:
    BLOCK
```

And C# is considering adding the following syntax to map to their `IAsyncDisposable` interface.

```
async using (var x = expr)
{
  statements;
}
```

Although, with C# 8, the C# language is also moving towards non-block-scopes and have recently introduced the following syntax for synchronous cleanup.

```
using var x = expr;
statements;

// is syntactic sugar for

using (var x = expr)
{
  statements;
}
```

Cancellation of generator coroutines and limitations of `coroutine_handle::destroy()`

As described in the section "Composable Async Streams" above, existing designs of `generator<T>` types currently make use of the `coroutine_handle::destroy()` as a mechanism for cancelling the coroutine early if the consumer destroys the `generator<T>` object before reaching the end of the sequence.

Calling the `destroy()` method on the `coroutine_handle` when a generator is currently suspended at a `co_yield` statement will result in the destruction of any in-scope variables in the coroutine body followed by destruction of the promise, parameters and the freeing of the coroutine frame.

For synchronous generators, using `destroy()` to cancel the coroutine works reasonably well as all of the cleanup operations that need to be run are synchronous.

However, once we add in the ability to define asynchronous scopes then **cancelling a coroutine is no longer a synchronous operation and `destroy()` is therefore no longer suitable for cancelling the coroutine**.

This leaves `resume()` as the only mechanism available to cancel the coroutine cleanly. We would need to resume a coroutine with some signal that indicates it should promptly complete. eg. by returning a 'cancelled' status from `co_yield` or throwing an exception from `co_yield`.

If we want to be able to retain the ability to cancel a generator coroutine without resuming it either with an error or value then we would need some new kind of operation on a `coroutine_handle` that was not `destroy()`, as that must complete synchronously, and not `resume()`.

Generalising coroutine cancellation with the "done" signal

The suggested path forwards for solving the issues with `destroy()` identified in the previous section is to introduce an extra method on the `coroutine_handle` that allows resuming the coroutine by executing '`goto final_suspend;`'.

For the purposes of exposition am calling this method `set_done()` this name is consistent with naming used in P1341R0 in the interface of Receiver and for consistency with naming of `coroutine_handle::done()`. Other names, such as `cancel()`, `stop()`, `resume_with_done()` and `unwind()`, have been considered and can be bikeshedded if desired.

A suspended coroutine could then be resumed with the `set_done()` method and this would execute '`goto final_suspend;`' and start exiting scopes, some of which may execute some asynchronous operations. Once execution reaches `final_suspend()` the coroutine can then be safely destroyed.

If we add `set_done()` to coroutine_handle and change the `destroy()` to only be valid to call at initial and final suspend-points then the coroutine_handle class would look like:

```
template<typename Promise>
struct coroutine_handle
{
  ...
  void destroy() const;
  void resume() const;
  void set_done() const; // Resumes with 'goto final_suspend;'

  bool done() const;
  ...
};

// Usage example
void awaiter::await_suspend(coroutine_handle<> h)
{
  h.set_done();
}
```

This change has some problems, however.

Challenges with this design

While this approach can allow the coroutine to be asymmetrically resumed with the '`goto final_suspend;`' continuation, it is difficult to extend this design to support the ability to resume the coroutine with symmetric-transfer (See P0913R0).

Currently, when you return a `coroutine_handle` from the `await_suspend()` method of an awaiter it is defined to perform a tail-call to the `handle.resume()` method. However, when returning a `coroutine_handle` from `await_suspend()` there is no way to tell the compiler to generate a tail-call to the `handle.set_done()` method instead of to the `handle.resume()` method.

There are some possible approaches we could take here:

We could change set_done() to return a new coroutine_handle that represented the continuation

```
template<typename Promise>
struct coroutine_handle
{
  ...
  void destroy() const;
  void resume() const;

  // Returns a handle that can be returned from await_suspend() that symmetrically
  // transfers to the coroutine on the 'goto final_suspend;' path.
  coroutine_handle<void> set_done() const;

  bool done() const;
  ...
};

// Usage example
auto awaiter::await_suspend(coroutine_handle<> h)
{
  return h.set_done();
}
```

But this is unsatisfying as the handle returned from `set_done()` is itself a `coroutine_handle` and so you could call `set_done()` on it again. Or to be able to resume it asymmetrically you would need to call `handle.set_done().resume()`.

An alternative might be a method on the `coroutine_handle` that changes the behaviour of a subsequent call to `resume()`.

```
template<typename Promise>
struct coroutine_handle
{
  ...
  void destroy() const;
```

```
  void resume() const;

  // Modifies the state of the coroutine such that a subsequent call to
  // .resume() will resume on the 'goto final_suspend;' path.
  void set_next_resume_to_cancel() const;

  bool done() const;
  ...
};

// Usage example
auto awaiter::await_suspend(coroutine_handle<> h)
{
  h.set_next_resume_to_cancel();
  return h;
}
```

This approach is also unsatisfying. Changing the behaviour of the `resume()` method based on whether or not some state has been set modified by a prior call forces implementations to store extra state in the coroutine frame.

A conflation of responsibilities

The difficulty in extending `coroutine_handle` to support an `.set_done()` operation cleanly within its current design is because the `coroutine_handle` class is actually conflating several separate responsibilities into a single interface.

- It represents a coroutine that is suspended at a particular suspend-point.
  A suspended coroutine has multiple actions* that can be performed on it and the `coroutine_handle` interface allows the application to choose which action to perform. These actions can be thought of as continuations of the suspend-point.
- It implicitly represents the '`resume()`' continuation of the coroutine at that suspend-point when it is returned from an `await_suspend()` method.
- It represents a handle to the coroutine frame resource itself, allowing the caller to destroy the frame.

* - Technically, a coroutine has two possible continuations when it is suspended at any suspend-point (other than the final-suspend point). The program can choose to execute the `.resume()` continuation or it can choose to execute the `.destroy()` continuation. Note that while the `.resume()` continuation can either be executed asymmetrically (by calling `.resume()`) or symmetrically (by returning the handle from `await_suspend()`), the `.destroy()` continuation can only be executed asymmetrically by calling the `.destroy()` method. You cannot perform a symmetric transfer to the `.destroy()` continuation.

The conflation of the "suspend-point" and "continuation" concepts into a single entity makes it difficult to introduce additional continuation paths while still providing a uniform interface for symmetric transfer to each of those continuation types.

Separating out the suspend-point and continuation responsiblities [C++20]

The proposed solution to this is to separate the coroutine-handle concept into two distinct concepts: a **suspend-point-handle** concept which represents a coroutine suspended at a particular suspend-point, and a **continuation-handle** concept which represents a particular chosen path to execute when resuming a suspended coroutine.

A 'suspend-point' then becomes a factory for a 'continuation' and would allow a choice between resuming with the 'resume' or 'set_done' continuations.

When the compiler generates code for a `co_await` expression it would generate a call to `await_suspend()`, passing a suspend-point handle object that corresponds to the current suspend-point. The awaitable object can then call either `.resume()` or `.set_done()` on this suspend-point handle to select the desired continuation to resume with. The awaitable can then either invoke the continuation asymmetrically by calling `operator()` on the continuation-handle or can invoke the continuation symmetrically by returning it from an `await_suspend()` method.

See [P1745R0] "Coroutine changes for C++20 and beyond" for details of the proposed interfaces for these handle types.

Example: An awaitable that can cancel the awaiting coroutine if it completes with cancellation

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

  template<typename SuspendPointHandle>
  auto await_suspend(SuspendPointHandle h) {
    suspendPoint_ = h;

    return std::noop_continuation();
  }

  int await_resume() {
    return result_;
  }

private:
  // Launches the operation.
  // This will eventually call either on_cancelled() or on_complete() with
  // the result when ready.
  void start();

  void on_cancelled() {
    // Resume the coroutine with the "done" signal.
    suspendPoint_.set_done()();
  }

  void on_complete(int result) {
```

```
    // Resume the coroutine with a value.
    result_ = result;
    suspendPoint_.resume()();
  }

  // suspend_point_handle is a type-erased suspend-point handle that
  // allows the user to indicate which operations they want available
  // by those specifying operations as template arguments.
  suspend_point_handle<with_done, with_resume> suspendPoint_;
  int result_;

};
```

For other examples of usage see "Simulating a co_return statement" and "Appendix A - async_generator".

Parallels with the done/value/error signals from Sender/Receiver

The Bellevue executors meeting in September 2018 voted the Sender/Receiver design as the preferred long-term direction for executors and modelling asynchronous operations.

The paper P1341R0 describes the design of Sender/Receiver where a Receiver receives the result of an async operation via calls to one of three customisation points for a Receiver. You can think of a Receiver as a generalisation of a callback (the paper P1660R0 actually uses Callback for the name of the concept instead of Receiver).

- `set_value(receiver, values...)` - signals success and provides the result value.
- `set_done(receiver)` - signals the operation completed without a result (eg. due to cancellation)
- `set_error(receiver, error)` - signals the operation completed with an error

We can draw parallels between a `Receiver` and the `suspend_point_handle` described above:

- `handle.set_done()` now corresponds to `set_done(receiver)`
- `handle.resume()` corresponds to either
    - `set_error(receiver, error)` if `await_resume()` throws an exception
    - `set_value(receiver, value)` if `await_resume()` returns normally

There are many other correspondences between coroutines and sender/receiver concepts. The addition of a `set_done()` operation on a coroutine handle brings the design of coroutines closer to parity with the design of sender/receiver.

Callback-based Asynchrony	**Coroutine-based Asynchrony**
Sender	Awaitable
Receiver	Suspend-Point Handle

`submit(sender, receiver)`	`co_await awaitable`
	Internally this is composed of:
	`operator co_await(awaitable)` + `await_suspend(suspendPointHandle)`

`set_value(receiver, value)`	`suspendPointHandle.resume()` where `await_resume()` returns a value
`set_value(receiver)`	`suspendPointHandle.resume()` where `await_resume()` returns `void`
`set_value(receiver, A{})` `set_value(receiver, B{})`	Coroutines do not currently support resuming with one of multiple possible value types. The types would need to be coerced into a single type. eg. using std::variant
`set_value(receiver, args...)`	Coroutines do not currently support resuming with multiple values. The values would need to be coerced into a single value. eg. using std::tuple. Alternatively we would need language support for returning a pack.
`set_done(receiver)`	`suspendPointHandle.set_done()`
`set_error(receiver, error)`	`suspendPointHandle.resume()` where `await_resume()` throws

The paper P1663R0 explores a potential future evolution on the `Awaitable` concept to allow return-value optimisation for a `co_await` expression. That change would bring the `suspend_point_handle` concept closer again to the concept of a `Receiver`, replacing `handle.resume()` with two methods `handle.set_value<T>()` and `handle.set_error<E>()`.

A future paper will explore further the commonalities between Sender/Receiver and Coroutines and look at what changes would be required to bring their functionalities closer in parity by generalising a coroutine to allow resuming from a suspend-point with one of several possible types.

Forwards Compatibility Issues

Let's now look at the implications of adding support for async RAII in a future version to determine if there are any compatibility issues doing so in a future version post-C++20.

Let us assume that in C++Next we want to add support for the `operator ~co_await()` as described in the section "Adding an operator ~co_await()".

We will now have some Awaitable types authored under C++20 that do not implement `operator ~co_await()` and some Awaitable types authored under C++Next that do implement `operator ~co_await()`.

In general, any coroutine that co_awaits a type with an `operator ~co_await()` defined will be correctly compiled to ensure that async cleanup is performed. A type cannot define an `operator ~co_await()` method unless being compiled under C++Next and therefore any `co_await` expressions involving a type with async cleanup will therefore be compiled with a compiler that knows to generate calls to `operator ~co_await()` on scope exit.

However, we can run into issues when we try to use an Awaitable type with async-cleanup in a generic algorithm written against the C++20 Awaitable concept as that algorithm may make manual calls to the `operator co_await()`, `await_ready()`, `await_suspend()` and `await_resume()` methods to mimic how the compiler lowers a C++20 `co_await` expression rather than relying on the compiler to do the lowering for us.

For example, an implementation of the `transform()` algorithm for Awaitables may be implemented as follows:

```
template<typename Awaitable, typename Func>
class transform_awaitable {
  Awaitable awaitable_;
  Func func_;

  class awaiter {
    awaiter_type_t<Awaitable> awaiter_;
    Func&& func_;
  public:
    explicit awaiter(Awaitable&& awaitable, Func&& func)
    : awaiter_(static_cast<Awaitable&&>(awaitable).operator co_await())
    , func_(static_cast<Func&&>(func))
    {}

    decltype(auto) await_ready() {
      return awaiter_.await_ready();
    }

    template<typename Handle>
    decltype(auto) await_suspend(Handle h) {
      return awaiter_.await_suspend(h);
```

```
    }

    decltype(auto) await_resume() {
      if constexpr (std::is_void_v<decltype(awaiter_.await_resume())>) {
        awaiter_.await_resume();
        return std::invoke(static_cast<Func&&>(func_));
      } else {
        return std::invoke(static_cast<Func&&>(func_), awaiter_.await_resume());
      }
    }
  };

public:
  template<typename Awaitable2, typename Func2>
  transform_awaitable(Awaitable2&& awaitable, Func2&& func)
  : awaitable_(static_cast<Awaitable2&&>(awaitable))
  , func_(static_cast<Func2&&>(func))
  {}

  awaiter operator co_await() && {
    return awaiter{
      static_cast<Awaitable&&>(awaitable_),
      static_cast<Func&&>(func_)};
  }
};

template<typename Awaitable, typename Func>
auto transform(Awaitable&& awaitable, Func&& func) {
  return transform_awaitable<std::decay_t<Awaitable>, std::decay_t<Func>>{
    static_cast<Awaitable&&>(awaitable),
    static_cast<Func&&>(func)};
}
```

The current implementation of the `cppcoro::fmap()` algorithm follows this pattern.

Now, if this `transform()` algorithm from a C++20 code-base was invoked with an Awaitable type from a C++Next code-base that defined the `operator ~co_await()` then this adapter would now silently fail to run the async cleanup, likely leading to subsequent undefined behaviour of the code.

Note that if the `transform()` algorithm had been implemented in terms of a coroutine and a `co_await` expression as follows then the compiler would have done the right thing as the compiler would have been able to detect the presence of an `operator ~co_await()` and generate appropriate lowering rather than the library manually encoding this lowering.

Example: Forwards compatible algorithm that uses `co_await` expressions instead of the low-level awaitable APIs.

```
template<typename Awaitable, typename Func>
  requires std::is_void_v<await_result_t<Awaitable>>
auto transform(Awaitable awaitable, Func func)
    -> task<std::invoke_result_t<Func>> {
  co_await std::move(awaitable);
  co_return std::invoke(std::move(func));
```

```
}

template<typename Awaitable, typename Func>
  requires !std::is_void_v<await_result_t<Awaitable>>
auto transform(Awaitable awaitable, Func func)
    -> task<std::invoke_result_t<Func, await_result_t<Awaitable>>> {
  co_return std::invoke(std::move(func), co_await std::move(awaitable));
}
```

However, in general we can't assume that this will be the case. Code will be written that assumes the C++20 definition for lowering of `co_await` expressions.

So, to guard against this kind of silent failure bug we would need to introduce a new, incompatible awaitable concept that represents an awaitable type with async-cleanup, say ScopedAwaitable, and ensure that implementations of ScopedAwaitable do not also implement the C++20 Awaitable interface, or if they do, they do not assume that `operator ~co_await()` will be called.

For example, we could potentially roll addition of support for async RAII in with addition of support for async RVO as described in P1663R0, which introduces a new flavour of `operator co_await()` that accepts the suspend-point-handle as a parameter in addition to the operand to the `co_await` expression.

```
class fork_join_scope
{
  class awaiter;

  class spawner {
  public:
    spawner() : count_(1) {}

    ~spawner() {
      assert(count_.load(std::memory_order_relaxed) == 0);
    }

    template<typename Awaitable>
    void spawn(Awaitable a);

  private:
    friend class awaiter;

    std::atomic<std::size_t> count_;
    suspend_point_handle<with_done> cleanupSP_;
  };

  class awaiter {
    spawner spawner_;

  public:
    awaiter() = default;
    ~awaiter() = default;
```

```
    template<typename SuspendPointHandle>
    auto operator co_await(SuspendPointHandle sp) {
      return sp.set_value<spawner&>(spawner_);
    }

    template<typename SuspendPointHandle>
    continuation_handle operator ~co_await(SuspendPointHandle sp) {
      spawner_.cleanupSP_ = sp;
      if (spawner_.count_.fetch_sub(1, std::memory_order_acq_rel) == 1) {
        return sp.set_done();
      } else {
        return std::noop_continuation();
      }
    }
  };

public:

  template<typename SuspendPointHandle>
  awaiter operator co_await(SuspendPointHandle sp) noexcept {
    return {};
  }
};

task<void> do_work(int i);

task<void> example_usage()
{
  co_await auto& spawner = fork_join_scope{};
  for (int i = 0; i < 100; ++i) {
    spawner.spawn(do_work(i));
  }
}

task<void> ill_formed() {
  // Usage with algorithms that expect Awaitable concept is ill-formed
  co_await transform(fork_join_scope{}, [](auto& spawner) {
    spawner.spawn(do_work(1));
    spawner.spawn(do_work(2));
  });
}
```

One avenue for exploration would be to define a default implementation of the Awaitable interface in terms of ScopedAwaitable to allow existing C++20 abstractions to work with ScopedAwaitables. However, this adapter would need to ensure that async cleanup was performed before resuming the awaiting coroutine since there would be no other place it could be guaranteed to run.

While this would work for some types, it would not always be safe to do in general - if the `co_await` expression for a ScopedAwaitable returned a reference to some resource that was subsequently destroyed by the async cleanup operation then running the async cleanup before the value was used could lead to returning a dangling reference.

It is still an open question how to add support for async RAII incrementally without breaking code written against C++20 coroutines and whether or not this is possible without requiring algorithms written against the C++20 Awaitable concept are rewritten to be able to support the new Awaitable concept.

Proposed Changes for C++20

This paper proposes making the changes described in P1745R0 for C++20 to enable the ability to cleanly add support for async RAII in a future release of C++. It is not proposing that we add support for async RAII to C++20.

The key points of the changes proposed by P1745 for C++20 that are relevant here are:

- Split the `coroutine_handle` into separate "suspend-point handle" and "continuation handle" concepts.
- Replace the `coroutine_handle` type with two new type-erased handle types: `suspend_point_handle<Ops...>` and `continuation_handle`.
- Split the functionality of the `destroy()` operation into `set_done()` and `destroy()`.

These changes and their rationale are described in detail in the paper [CoroutineChanges].

The net result of these changes is to replace the use of `destroy()` for cancelling a coroutine with a new `set_done()` operation and to limit the `destroy()` method to only be valid to call at the initial and final suspend-points.

These changes also enable future evolution of the `suspend_point_handle` concept to enable other use-cases described in the paper P1663R0 (Async return-value optimisation).

Other Benefits

There are a number of other benefits to the proposed changes that are not directly related to supporting async RAII.

Simulating a 'co_return' statement

One of the benefits of separating the "cancellation" part of the `coroutine_handle::destroy()` operation out into a separate `set_done()` operation is that it enables us to write `operator co_await()` implementations that can act as if they were a '`co_return`' statement.

The `set_done()` operation resumes the coroutine by immediately executing the statement '`goto final_suspend;`'. This is exactly what a `co_return` statement does after calling `promise.return_value()` or `promise.return_void()`. This means that an awaitable can now simulate a `co_return` statement by manually calling `promise.return_value()` followed by resuming the coroutine using the `set_done()` continuation.

This can be useful for implementing an `operator co_await()` that you want to have short-circuiting behaviour that behaves as if the user had written '`co_return`'.

For example, we can write a generic `std::optional<T>::operator co_await()` that allows it to either immediately resume the coroutine with the unwrapped value of type `T` or otherwise cancel the coroutine as if by executing '`co_return std::nullopt;`'.

Example: `operator co_await()` implementation for `std::optional<T>`

```
namespace std
{
  template<typename Optional>
  struct __optional_awaiter {
    Optional&& opt_;
    bool await_ready() noexcept {
      return opt_.has_value();
    }
    template<typename Handle>
    auto await_suspend(Handle h) noexcept {
      h.promise().return_value(std::nullopt);
      return h.set_done();
    }
    decltype(auto) await_resume() noexcept {
      return static_cast<Optional&&>(opt_).value();
    }
  };

  template<typename T>
  auto operator co_await(const optional<T>& opt) {
    return __optional_awaiter<const optional<T>&>{opt};
  }
  template<typename T>
  auto operator co_await(optional<T>& opt) {
    return __optional_awaiter<optional<T>&>{opt};
  }
  template<typename T>
  auto operator co_await(const optional<T>&& opt) {
    return __optional_awaiter<const optional<T>>{std::move(opt)};
  }
  template<typename T>
  auto operator co_await(optional<T>&& opt) {
    return __optional_awaiter<optional<T>>{std::move(opt)};
  }
}
```

This implementation will now work independently of the kind of coroutine in which you try to co_await the optional value. It will now work in any coroutine context where '`co_return std::nullopt;`' is a valid statement.

This means that we can now await an `optional<T>` value inside a coroutine returning a `task<std::optional<U>>` without having to specialise `task<T>` to explicitly handle this short-circuiting behaviour.

It also means that we can now define a generic `subroutine<T>` coroutine type, which could be made the default for a coroutine that returns type `T`, such that we don't need to explicitly define a custom coroutine type for functions returning `std::optional<T>`.

Example: Defining a default coroutine type for synchronous functions

```
namespace std
{
  template<typename Ret>
  struct __subroutine_promise
  {
    __manual_lifetime<Ret> returnValue_;

    template<typename Handle>
    Ret get_return_object(Handle h) {
      scope_exit deleteFrameOnExit = [h] { h.destroy(); };
      h.resume()();
      scope_exit destroyReturnValueOnExit = [&] { returnValue_.destruct(); };
      return std::move(returnValue_).get();
    }

    void unhandled_exception() { throw; }

    void return_void() noexcept requires std::is_void_v<Ret> {
      returnValue_.construct();
    }

    template<ConvertibleTo<Ret> Value>
      requires (!std::is_void_v<Ret>)
    void return_value(Value&& v)
        noexcept(std::is_nothrow_constructibe_v<Ret, Value>) {
      returnValue_.construct((Value&&)v);
    }

    auto final_suspend() { return noop_continuation(); }
  };

  // Primary template
  template<typename Ret, typename... Args>
  struct coroutine_traits
  {
    using promise_type = __subroutine_promise<Ret>;
  };

  // Override for types that define T::promise_type
```

```
  template<typename Ret, typename... Args>
    requires requires() { typename Ret::promise_type; }
  struct coroutine_traits<Ret, Args...>
  {
    using promise_type = typename Ret::promise_type;
  };
}
```

This, combined with the definition of a `task<T>` coroutine type, such as the one defined in P1056R0, allows us to write the following without having to define custom coroutine types for `std::optional<T>`.

```
// An ordinary function
std::optional<int> try_parse(char c) {
  if (c >= '0' && c <= '9') return (int)(c - '0');
  return std::nullopt;
}

// A synchronous coroutine returning a std::optional.
std::optional<int> try_parse(const char* s) {
  int value = 0;
  do {
    value = value * 10 + co_await try_parse(*s++);
  } while (*s != '\0');
  co_return value;
}

std::task<std::string> read_string_async();

// An asynchronous coroutine returning a std::optional.
std::task<std::optional<int>> read_int_async() {
  std::string s = co_await read_string_async();
  int value = co_await try_parse(s.c_str());
  co_return value;
}
```

Further, this allows these short-circuiting types to compose without having to specialise coroutine types for each combination of types.

eg. if we implement `std::expected<T, E>::operator co_await()` to short-circuit to '`co_return std::unexpected(value.error());`' then we can do something like this:

```
std::optional<int> try_parse(const char* s); // as above
std::task<std::string> read_string_async(); // as above

std::expected<int, std::error_code> try_divide(int numerator, int divisor) {
  if (divisor == 0) return divide_by_zero_error;
  if ((numerator % divisor) != 0) return not_evenly_divisible_error;
  return numerator / divisor;
}

std::expected<std::optional<int>, std::error_code>
try_divide(const char* numeratorString, const char* divisorString) {
```

```
  int numerator = co_await try_parse(numeratorString);
  int divisor = co_await try_parse(divisorString);
  co_return co_await try_divide(numerator, divisor);
}

std::task<std::expected<std::optional<int>, std::error_code>> divide_async() {
  auto numerator = co_await read_string_async();
  auto divisor = co_await read_string_async();
  // Note we need 2 co_await here - to first to unwrap the expected and the
  // second to unwrap the optional.
  int result = co_await co_await try_divide(numerator.c_str(), divisor.c_str());
  co_return result;
}
```

Conclusion

Adding support for async RAII is something that would greatly simplify the ability to write correct, exception-safe code in coroutines that needs to perform async operations to safely release resources.

If we later add support for async RAII to the current coroutines design then this will cause it to no longer be safe in general to call `coroutine_handle::destroy()` at any suspend-point other than the initial and final suspend-points, thus potentially breaking code written against C++20 which relied on the ability to do that.

If the async RAII feature is used in conjunction with async generator coroutines we no longer have a mechanism to directly cancel a generator suspended at a `co_yield` statement without calling `resume()`. This then forces the generator to either have to manually check return-values or throw an exception from the `co_yield` expression to signal cancellation - neither of which is a desirable programming model for generators.

To keep the door open to adding support for async RAII to coroutines in future and to support adding

This paper proposes to modifying the design of `coroutine_handle` to support resuming a coroutine with a cancellation signal, `set_done()`, in addition to the existing `resume()` method.

Adding the `set_done()` signal to the current `coroutine_handle` design has implications for the ability to symmetrically transfer to a coroutine on the `set_done()` continuation. Therefore, the paper also suggests splitting the responsibilities of `coroutine_handle` into a `suspend_point_handle`, which represents a suspended coroutine, and a `continuation_handle`, which represents the chosen resumption path. This will allow representing the ability to resume the coroutine symmetrically on one of a number of possible continuation-paths with a uniform interface.

The changes propo are described in detail in P1745R0.

The proposed changes have been implemented in clang as an incremental change to the existing implementation of the Coroutines TS.

These changes should be considered for adoption prior to C++20 shipping as it will be difficult to later change the design of `coroutine_handle` to add support for `set_done()` in future.

These changes also support future extensions to support async return-value-optmisation, described in P1663R0.

Acknowledgements

Many thanks to Kirk Shoop, Eric Niebler and Gor Nishanov for reviewing and providing feedback on drafts of this paper.

References

[P0660R9] - "Stop Token and Joining Thread"
[P1745R0] - "Coroutine changes for C++20 and beyond"
[P1663R0] - "Supporting return-value-optimisation in coroutines"
[2019AnnualSurvey] - 2019 Annual C++ Developer Survey "Lite"

Appendix A - async_generator

Example implementation of an async_generator built using a modified coroutines design as described in [P1745R0] in combination with `operator ~co_await()` described in this paper.

```
template<typename T>
struct async_generator {
  struct promise_type {
    suspend_point_handle<with_set_done, with_resume> producerSP_;
    continuation_handle continuation_;
    std::add_pointer_t<T> value_ = nullptr;
    std::exception_ptr error_;

    template<typename InitialSuspendPointHandle>
    async_generator get_return_object(InitialSuspendPointHandle sp) {
      producerSP_ = sp;
      return async_generator{sp};
    }

    struct yield_awaiter {
      bool await_ready() noexcept { return false; }

      template<typename SuspendPointHandle>
      auto await_suspend(SuspendPointHandle h) noexcept {
        // Save the producer's suspend-point and resume the consumer
        h.promise().producerSP_ = h;
        return h.promise().continuation_;
      }

      void await_resume() {}
    };

    yield_awaiter yield_value(T&& x) noexcept {
      value_ = std::addressof(x);
      return yield_awaiter{};
    }

    void return_void() noexcept {
      value_ = nullptr;
    }

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

    auto final_suspend() noexcept {
      return continuation_;
    }
  };

private:
  using handle_t = suspend_point_handle<with_proimse<promise_type>, with_destroy>;
  handle_t coro_;

  explicit async_generator(handle_t coro) noexcept
  : coro_(coro) {}

public:
  async_generator(async_generator&& other) noexcept
```

```
  : coro_(std::exchange(other.coro_, {})

  ~async_generator() {
    // Should not be suspended
    assert(coro_.promise().value_ == nullptr);
    coro_.destroy();
  }

  struct async_range {
  private:
    struct next_awaiter {
      explicit next_awaiter(handle_t coro) noexcept : coro_(coro) {}

      bool await_ready() noexcept {
        return false;
      }

      template<typename SuspendPointHandle
      auto await_suspend(SuspendPointHandle h) noexcept {
        auto& promise = coro_.promise();
        promise.continuation_ = h.resume();
        return std::exchange(promise.producerSP_, {}).resume();
      }

      std::add_pointer_t<T> await_resume() noexcept {
        auto& promise = coro_.promise();
        if (!promise.producerSP_ && promise.error_) {
          std::rethrow_exception(promise.error_);
        }
        return coro_.promise().value_;
      }

    private:
      handle_t coro_;
    };

  public:
    explcit async_range(handle_t coro) noexcept
    : coro_(coro)
    {}

    auto next() noexcept {
      return next_awaiter{coro_};
    }

  private:
    handle_t coro_;
  };

private:

  struct unsubscribe_awaiter {
  public:
    explicit unsubscribe_awaiter(handle_t coro) : coro_(coro) {}

    bool await_ready() {
      // Nothing to do if already complete.
      return !coro_.promise().producerSP_;
    }

    template<typename SuspendPointHandle>
    auto await_suspend(SuspendPointHandle sp) {
      auto& promise = coro_.promise();
      promise.continuation_ = sp.resume();
      return std::exchange(promise.producerSP_, {}).set_done();
```

```
    }

    void await_resume() {}

  private:
    handle_t coro_;
  };

  struct subscribe_awaiter {
  public:
    explicit subscribe_awaiter(handle_t coro) : coro_(coro) {}

    bool await_ready() noexcept {
      return true;
    }

    void await_suspend(suspend_point_handle<>) noexcept {}

    async_range await_resume() noexcept {
      return async_range{coro_};
    }

    unsubscribe_awaiter operator ~co_await() {
      return unsubscribe_awaiter{coro_};
    }

  private:
    handle_t coro_;
  };

public:

  auto operator co_await() {
    return subscribe_awaiter{coro_};
  }
};
```

And example usage of the async_generator type

```
async_generator<int> int_ticker();

async_generator<std::string> string_ticker() {
  co_using (auto stream : int_ticker())
  {
    while (int* value = co_await stream.next()) {
      co_yield std::to_string(*value);
    }
    co_return;
  }
}

task<void> consumer() {
  co_using (auto stream : string_ticker())
  {
    while (std::string* s = co_await stream.next()) {
      std::cout << *s << std::endl;
      if (*s == "100") co_return;
    }
  }
}
```