13
Asynchronous Programming with Coroutines

The generator class implemented in the previous chapter helped us to use coroutines for building lazily evaluated sequences. C++ coroutines can also be used for asynchronous programming by having a coroutine represent an asynchronous computation or an **asynchronous task**. Although asynchronous programming is the most important driver for having coroutines in C++, there is no support for asynchronous tasks based on coroutines in the standard library. If you want to use coroutines for asynchronous programming, I recommend you find and use a library that complements C++20 coroutines. I've already recommended CppCoro (https://github.com/lewissbaker/cppcoro), which at the time of writing seems like the most promising alternative. It's also possible to use asynchronous coroutines with the well-established library Boost.Asio, as you will see later on in this chapter.

This chapter will show that asynchronous programming is possible using coroutines and that there are libraries available to complement C++20 coroutines. More specifically, we will focus on:

- The `co_await` keyword and awaitable types
- The implementation of a rudimentary task type—a type that can be returned from coroutines that perform some asynchronous work
- Boost.Asio to exemplify asynchronous programming using coroutines

Before moving on, it should also be said that there are no performance-related topics in this chapter and very few guidelines and best practices are presented. Instead, this chapter serves more as an introduction to the novel feature of asynchronous coroutines in C++. We'll begin this introduction by exploring awaitable types and `co_await` statements.

Awaitable types revisited

We already talked a bit about awaitable types in the previous chapter. But now we need to get a little bit more specific about what `co_await` does and what an awaitable type is. The keyword `co_await` is a unary operator, meaning that it takes a single argument. The argument we pass to `co_await` needs to fulfill some requirements that we will explore in this section.

When we say `co_await` in our code, we express that we are *waiting* for something that may or may not be ready for us. If it's not ready, `co_await` suspends the currently executing coroutine and returns control back to its caller. When the asynchronous task has completed, it should transfer the control back to the coroutine originally waiting for the task to finish. From here on, I will typically refer to the awaiting function as the **continuation**.

Now consider the following expression:

```
co_await X{};
```

For this code to compile, `X` needs to be an awaitable type. So far we have only used the trivial awaitable types: `std::suspend_always` and `std::suspend_never`. Any type that directly implements the three mem-

ber functions listed next, or alternatively defines `operator co_wait()` to produce an object with these member functions, is an awaitable type:

- `await_ready()` returns a `bool` that indicates whether the result is ready (`true`) or whether it is necessary to suspend the current coroutine and wait for the result to become ready.
- `await_suspend(coroutine_handle)` – If `await_ready()` returned `false` , this function will be called with a handle to the coroutine that executed `co_await` . This function gives us an opportunity to start asynchronous work and subscribe for a notification that will trigger when the task has finished and thereafter resume the coroutine.
- `await_resume()` is the function responsible for unpacking the result (or error) back to the coroutine. If an error has occurred during the work initiated by `await_suspend()` , this function could rethrow the caught error or return an error code. The result of the entire `co_await` expression is whatever `await_resume()` returns.

To demonstrate the use of `operator co_await()` , here is a snippet inspired by a section from the C++20 standard that defines `operator co_await` for a time interval:

```
using namespace std::chrono;
template <class Rep, class Period>
auto operator co_await(duration<Rep, Period> d) {
 struct Awaitable {
  system_clock::duration d_;
  Awaitable(system_clock::duration d) : d_(d) {}
  bool await_ready() const { return d_.count() <= 0; }
  void await_suspend(std::coroutine_handle<> h) { /* ... */ }
  void await_resume() {}
 };
```

```
  return Awaitable{d};
}
```

With this overload in place, we can now pass a time interval to the `co_await` operator, as follows:

```
std::cout << "just about to go to sleep...\n";
co_await 10ms;              // Calls operator co_await()
std::cout << "resumed\n";
```

The example is not complete but gives you a hint about how to use the unary operator `co_await`. As you may have noticed, the three `await_*()` functions are not called directly by us; instead, they are invoked by code inserted by the compiler. Another example will clarify the transformations made by the compiler. Assume that the compiler stumbles upon the following statement in our code:

```
auto result = co_await expr;
```

Then the compiler will (very) roughly transform the code into something like this:

```
// Pseudo code
auto&& a = expr;       // Evaluate expr, a is the awaitable
if (!a.await_ready()) {  // Not ready, wait for result
  a.await_suspend(h);    // Handle to current coroutine
                         // Suspend/resume happens here
}
auto result = a.await_resume();
```

The `await_ready()` function is first called to check whether a suspension is needed. If so, `await_suspend()` is called with a handle to the coroutine that will be suspended (the coroutine with the `co_await` statement). Finally, the result of the awaitable is requested and assigned to the `result` variable.

The implicit suspend points

As you have seen in numerous examples, a coroutine defines *explicit* suspend points by using `co_await` and `co_yield`. Each coroutine also has two *implicit* suspend points:

- The **initial suspend point**, which occurs at the initial invocation of a coroutine before the coroutine body is executed
- The **final suspend point**, which occurs after the coroutine body has been executed and before the coroutine is destroyed

The promise type defines the behavior of these two points by implementing `initial_suspend()` and `final_suspend()`. Both functions return awaitable objects. Typically, we pass `std::suspend_always` from the `initial_suspend()` function so that the coroutine is started lazily rather than eagerly.

The final suspend point plays an important role for asynchronous tasks, because it makes it possible for us to tweak the behavior of `co_await`. Normally, a coroutine that has been `co_await:ed` should resume the awaiting coroutine at the final suspend point.

Next, let's get a better understanding of how the three awaitable functions are meant to be used and how they cooperate with the `co_await` operator.

Implementing a rudimentary task type

The task type we are about to implement is a type that can be returned from coroutines that represent asynchronous tasks. The task is something that a caller can wait for using `co_await`. The goal is to be able to write asynchronous application code that looks like this:

```
auto image = co_await load("image.jpg");
auto thumbnail = co_await resize(image, 100, 100);
co_await save(thumbnail, "thumbnail.jpg");
```

The standard library already provides a type that allows a function to return an object that a caller can use for waiting on a result to be computed, namely `std::future`. We could potentially wrap `std::future` into something that would conform to the awaitable interface. However, `std::future` does not support continuations, which means that whenever we try to get the value from a `std::future`, we block the current thread. In other words, there is no way to compose asynchronous operations without blocking when using `std::future`.

Another alternative would be to use `std::experimental::future` or a future type from the Boost library, which supports continuations. But these future types allocate heap memory and include synchronization primitives that are not needed in the use cases set out for our tasks. Instead, we will create a new type with minimum overhead with the responsibilities to:

- Forward return values and exceptions to the caller
- Resume the caller waiting for the result

A coroutine task type has been proposed (see P1056R0 at **http://www7.open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1056r0.html**), and the proposal gives us a good hint about what components we need. The implementation that follows is based on work presented by Gor Nishanov and source code shared by Lewis Baker, which is available in the CppCoro library.

Here is the implementation of the class template for representing an asynchronous task:

```
template <typename T>
class [[nodiscard]] Task {
 struct Promise { /* ... */ };        // See below
 std::coroutine_handle<Promise> h_;
 explicit Task(Promise & p) noexcept
    : h_{std::coroutine_handle<Promise>::from_promise(p)} {}
public:
 using promise_type = Promise;
 Task(Task&& t) noexcept : h_{std::exchange(t.h_, {})} {}
 ~Task() { if (h_) h_.destroy(); }
 // Awaitable interface
 bool await_ready() { return false; }
 auto await_suspend(std::coroutine_handle<> c) {
  h_.promise().continuation_ = c;
  return h_;
 }
 auto await_resume() -> T {
  auto& result = h_.promise().result_;
  if (result.index() == 1) {
   return std::get<1>(std::move(result));
  } else {
   std::rethrow_exception(std::get<2>(std::move(result)));
```

```
    }
  }
};
```

An explanation of each part will follow in the subsequent sections, but first we need the implementation of the promise type that uses a `std::variant` to hold a value or an error. The promise also keeps a reference to the coroutine waiting for the task to complete using the `continuation_` data member:

```
struct Promise {
  std::variant<std::monostate, T, std::exception_ptr> result_;
  std::coroutine_handle<> continuation_;  // A waiting coroutine
  auto get_return_object() noexcept { return Task{*this}; }
  void return_value(T value) {
    result_.template emplace<1>(std::move(value));
  }
  void unhandled_exception() noexcept {
    result_.template emplace<2>(std::current_exception());
  }
  auto initial_suspend() { return std::suspend_always{}; }
  auto final_suspend() noexcept {
    struct Awaitable {
      bool await_ready() noexcept { return false; }
      auto await_suspend(std::coroutine_handle<Promise> h) noexcept {
        return h.promise().continuation_;
      }
      void await_resume() noexcept {}
    };
    return Awaitable{};
```

```
  }
};
```

It's important to distinguish between the two coroutine handles we are using: the handle identifying the *current coroutine* and the handle identifying the *continuation*.

Note that this implementation doesn't support `Task<void>` due to limitations of `std::variant`, and also the limitation that we can't have both `return_value()` and `return_void()` on the same promise type. Not supporting `Task<void>` is unfortunate since not all asynchronous tasks necessarily return values. We will overcome this limitation in a while by providing a template specialization for `Task<void>`.

Since we implemented a few coroutine return types in the previous chapter (`Resumable` and `Generator`), you will already be familiar with the requirements of a type that can be returned from a coroutine. Here, we will focus on the things that are new to you, such as exception handling and the ability to resume the caller currently waiting for us. Let's start looking at how `Task` and `Promise` handle return values and exceptions.

Handling return values and exceptions

An asynchronous task can complete by returning (a value or `void`) or by throwing an exception. The value and the error need to be handed over to the caller, which has been waiting for the task to complete. As usual, this is the responsibility of the promise object.

The `Promise` class uses a `std::variant` to store the result of three possible outcomes:

- No value at all (the `std::monostate`). We use this in our variant to make it default-constructible, but without requiring the other two types to be default-constructible.
- A return value of type `T`, where `T` is the template argument of `Task`.
- A `std::exception_ptr`, which is a handle to an exception that was thrown earlier.

The exception is captured by using the `std::current_exception()` function inside the function `Promise::unhandled_exception()`. By storing a `std::exception_ptr`, we can later rethrow this exception in another context. This is also the mechanism used when exceptions are passed between threads.

A coroutine that uses `co_return value;` must have a promise type that implements `return_value()`. However, coroutines that use `co_return;`, or run off the body without returning a value, must have a promise type that implements `return_void()`. Implementing a promise type that contains both `return_void()` and `return_value()` generates a compilation error.

Resuming an awaiting coroutine

When the asynchronous task has completed, it should transfer the control back to the coroutine waiting for the task to finish. To be able to resume this continuation, the `Task` object needs the `coroutine_handle` to the continuation coroutine. This handle was passed to the `Task` object's `await_suspend()` function, and conveniently we made sure to save that handle into the promise object:

```
class Task {
 // ...
 auto await_suspend(std::coroutine_handle<> c) {
  h_.promise().continuation_ = c;      // Save handle
  return h_;
```

```
  }
  // ...
```

The `final_suspend()` function is responsible for suspending at the final suspend point of this coroutine and transferring execution to the awaiting coroutine. This is the relevant part of the `Promise` reproduced for your convenience:

```
auto Promise::final_suspend() noexcept {
 struct Awaitable {
  bool await_ready() noexcept { return false; } // Suspend
  auto await_suspend(std::coroutine_handle<Promise> h) noexcept{
   return h.promise().continuation_;  // Transfer control to
  }                                   // the waiting coroutine
  void await_resume() noexcept {}
 };
 return Awaitable{};
}
```

To begin with, returning `false` from `await_ready()` will leave the coroutine suspended at the final suspend point. The reason we do this is so that the promise is still alive and available for the continuation to have a chance to pull the result out from this promise.

Next, let's have a look at the `await_suspend()` function. This is the place where we want to resume the continuation. We could potentially call `resume()` directly on the `continuation_` handle and wait for it to finish, like this:

```
// ...
auto await_suspend(std::coroutine_handle<Promise> h) noexcept {
  h.promise().resume();       // Not recommended
}
// ...
```

However, that would run the risk of creating a long chain of nested call frames on the stack, which eventually could result in a stack overflow. Let's see how this could happen with a short example using two coroutines, `a()` and `b()`:

```
auto a() -> Task<int> {  co_return 42; }
auto b() -> Task<int> {       // The continuation
  auto sum = 0;
  for (auto i = 0; i < 1'000'000; ++i) {
    sum += co_await a();
  }
  co_return sum;
}
```

If the `Promise` object associated with coroutine `a()` directly called `resume()` on the handle to coroutine `b()`, a new call frame to resume `b()` would be created on the stack on top of the call frame for `a()`. This process would be repeated over and over again in the loop, creating new nested call frames on the stack for each iteration. This call sequence when two functions call each other is a form of recursion, sometimes called mutual recursion:

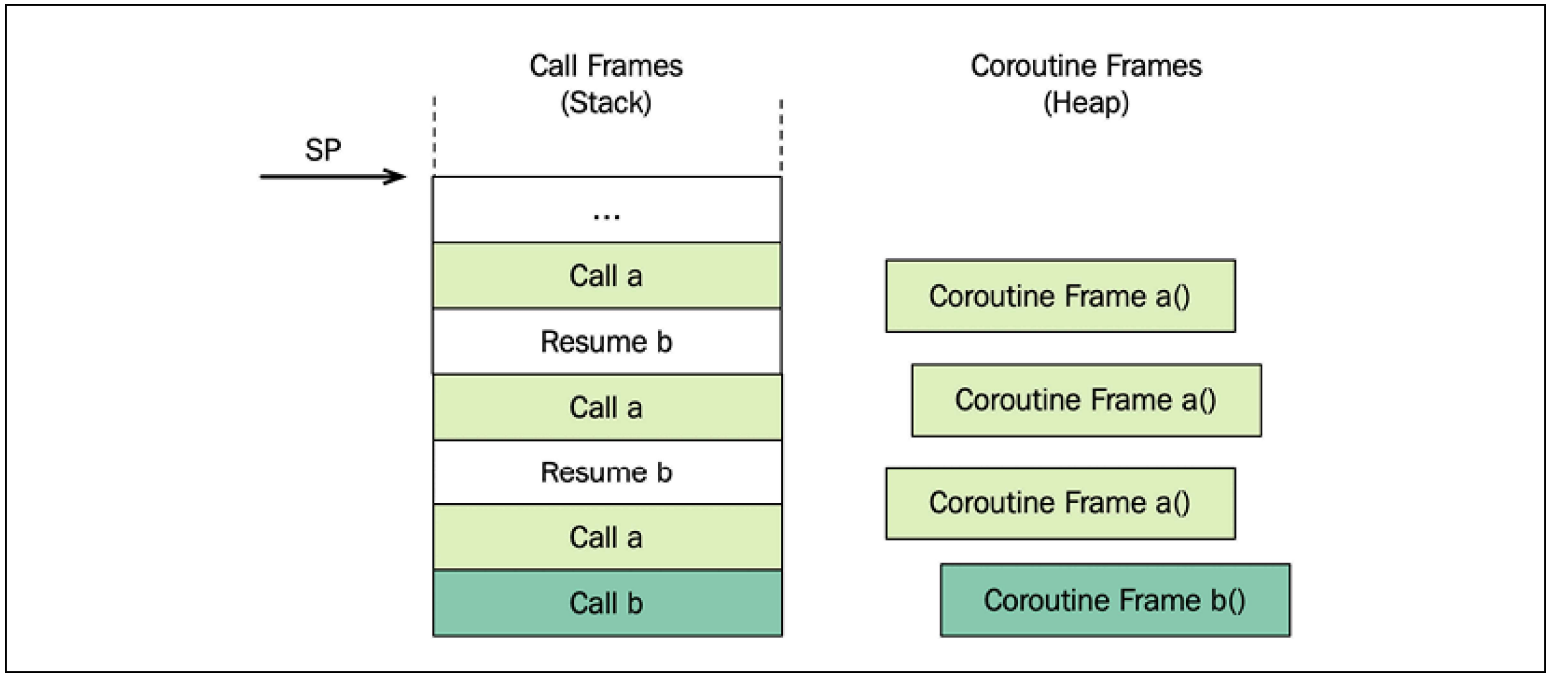

Figure 13.1: Coroutine b() calls coroutine a(), which resumes b(), which calls a(), which resumes b(), and so on

Even though there is only one coroutine frame created for `b()`, each call to `resume()` that resumes coroutine `b()` creates a new frame on the stack. The solution to avoid this problem is called **symmetric transfer**. Instead of resuming the continuation directly from the coroutine that is about to finish, the task object instead returns the `coroutine_handle` identifying the continuation from `await_suspend()`:

```
// ...
auto await_suspend(std::coroutine_handle<Promise> h) noexcept {
  return h.promise().continuation_;    // Symmetric transfer
```

```
}
// ...
```

An optimization called *tail call optimization* is then guaranteed to happen by the compiler. In our case, this means that the compiler will be able to transfer control directly to the continuation without creating a new nested call frame.

We will not spend more time on the details of symmetric transfer and tail calls, but an excellent and more in-depth explanation of these topics can be found in the article *C++ Coroutines: Understanding Symmetric Transfer* by Lewis Baker, available at **https://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer**.

As mentioned earlier, our `Task` template has the limitation of not handling a template parameter of type `void`. Now it's time to fix that.

Supporting void tasks

To overcome the limitations addressed earlier regarding the inability to handle tasks that do not produce any values, we need a template specialization for `Task<void>`. It is spelled out here for completeness, but it does not add many new insights beyond the general `Task` template defined earlier:

```
template <>
class [[nodiscard]] Task<void> {

 struct Promise {
  std::exception_ptr e_;  // No std::variant, only exception
```

```
    std::coroutine_handle<> continuation_;
    auto get_return_object() noexcept { return Task{*this}; }
    void return_void() {}   // Instead of return_value()
    void unhandled_exception() noexcept {
      e_ = std::current_exception();
    }
    auto initial_suspend() { return std::suspend_always{}; }
    auto final_suspend() noexcept {
      struct Awaitable {
        bool await_ready() noexcept { return false; }
        auto await_suspend(std::coroutine_handle<Promise> h) noexcept {
          return h.promise().continuation_;
        }
        void await_resume() noexcept {}
      };
      return Awaitable{};
    }
  };
  std::coroutine_handle<Promise> h_;
  explicit Task(Promise& p) noexcept
      : h_{std::coroutine_handle<Promise>::from_promise(p)} {}
public:
  using promise_type = Promise;

  Task(Task&& t) noexcept : h_{std::exchange(t.h_, {})} {}
  ~Task() { if (h_) h_.destroy(); }
  // Awaitable interface
  bool await_ready() { return false; }
  auto await_suspend(std::coroutine_handle<> c) {
    h_.promise().continuation_ = c;
```

```
    return h_;
  }
  void await_resume() {
    if (h_.promise().e_)
      std::rethrow_exception(h_.promise().e_);
  }
};
```

The promise type in this template specialization only keeps a reference to a potentially unhandled exception. And instead of having `return_value()` defined, the promise contains the member function `return_void()`.

We can now represent tasks that return values or `void`. But there is still some work to be done before we can actually build a standalone program to test our `Task` type.

Synchronously waiting for a task to complete

An important aspect of the `Task` type is that whatever invokes a coroutine that returns a `Task` must `co_await` on it, and is therefore also a coroutine. This creates a chain of coroutines (continuations). For example, assume we have a coroutine like this:

```
Task<void> async_func() {     // A coroutine
  co_await some_func();
}
```

Then, it's not possible to use it in the following way:

```
void f() {
  co_await async_func(); // Error: A coroutine can't return void
}
```

Once we call an asynchronous function that returns a `Task`, we need to `co_await` on it, or nothing will happen. This is also the reason why we declare `Task` to be `nodiscard`: so that it generates a compilation warning if the return value is ignored, like this:

```
void g() {
  async_func();       // Warning: Does nothing
}
```

The forced chaining of coroutines has the interesting effect that we finally get to the `main()` function of the program, which the C++ standard says is not allowed to be a coroutine. This needs to be addressed somehow, and the proposed solution is to provide at least one function that synchronously waits on the asynchronous chains to complete. For example, the CppCoro library includes the function `sync_wait()`, which has this effect of breaking the chain of coroutines, which makes it possible for an ordinary function to use coroutines.

Unfortunately, implementing `sync_wait()` is rather complicated, but in order to at least make it possible to compile and test our `Task` type, I will here provide a simplified version based on the Standard C++ Proposal P1171R0, **https://wg21.link/P1171R0**. Our goal here is to be able to write a test program like this:

```
auto some_async_func() -> Task<int> { /* ... */ }
int main() {
  auto result = sync_wait(some_async_func());
  return result;
}
```

With the aim of testing and running asynchronous tasks, let's continue with the implementation of `sync_wait()`.

Implementing sync_wait()

`sync_wait()` internally uses a custom task class specifically designed for our purpose, called `SyncWaitTask`. Its definition will be revealed in a while, but first let's have a look at the definition of the function template `sync_wait()`:

```
template<typename T>
using Result = decltype(std::declval<T&>().await_resume());
template <typename T>
Result<T> sync_wait(T&& task) {
  if constexpr (std::is_void_v<Result<T>>) {
    struct Empty {};
    auto coro = [&]() -> detail::SyncWaitTask<Empty> {
      co_await std::forward<T>(task);
      co_yield Empty{};
      assert(false);
    };
    coro().get();
  } else {
```

```
    auto coro = [&]() -> detail::SyncWaitTask<Result<T>> {
      co_yield co_await std::forward<T>(task);
      // This coroutine will be destroyed before it
      // has a chance to return.
      assert(false);
    };
    return coro().get();
  }
}
```

First, in order to specify the type that the task is returning, we use a combination of `decltype` and `declval`. The rather cumbersome `using-e`xpression gives us the type returned by `T::await_resume()`, where `T` is the type of the task passed to `sync_wait()`.

Inside `sync_wait()` we distinguish between tasks that return values and tasks that return `void`. We make a distinction here to avoid the need for implementing a template specialization of `SyncWaitTask` to handle both `void` and non-void types. Both cases are handled similarly by introducing an empty `struct`, which can be provided as the template argument to `SyncWaitTask` for handling `void` tasks.

In the case where an actual value is returned, a lambda expression is used to define a coroutine that will `co_await` on the result and then finally yield its value. It's important to note that the coroutine might resume from `co_await` on another thread, which requires us to use a synchronization primitive in the implementation of `SyncWaitTask`.

Calling `get()` on the coroutine lambda resumes the coroutine until it yields a value. The implementation of `SyncWaitTask` guarantees that the coroutine lambda will never have a chance to resume again after the `co_yield` statement.

We used `co_yield` extensively in the previous chapter, but without mentioning its relationship to `co_await`; namely that the following `co_yield` expression:

```
co_yield some_value;
```

is transformed by the compiler into:

```
co_await promise.yield_value(some_value);
```

where `promise` is the promise object associated with the currently executing coroutine. Knowing this is helpful when trying to understand the control flow between `sync_wait()` and the `SyncWaitTask` class.

Implementing SyncWaitTask

Now we are ready to inspect the `SyncWaitTask`, which is a type intended only to be used as a helper for `sync_wait()`. For that reason, we add it under a namespace called `detail` to make it clear that this class is an implementation detail:

```
namespace detail { // Implementation detail
template <typename T>
class SyncWaitTask {  // A helper class only used by sync_wait()
 struct Promise { /* ... */ }; // See below
 std::coroutine_handle<Promise> h_;
 explicit SyncWaitTask(Promise& p) noexcept
    : h_{std::coroutine_handle<Promise>::from_promise(p)} {}
public:
```

```
  using promise_type = Promise;

  SyncWaitTask(SyncWaitTask&& t) noexcept
    : h_{std::exchange(t.h_, {})} {}
  ~SyncWaitTask() { if (h_) h_.destroy();}
  // Called from sync_wait(). Will block and retrieve the
  // value or error from the task passed to sync_wait()
  T&& get() {
   auto& p = h_.promise();
   h_.resume();
   p.semaphore_.acquire();            // Block until signal
   if (p.error_)
    std::rethrow_exception(p.error_);
   return static_cast<T&&>(*p.value_);
  }
  // No awaitable interface, this class will not be co_await:ed
};
} // namespace detail
```

The most interesting part to pay attention to is the function `get()` and its blocking call to `acquire()` on a semaphore owned by the promise object. This is what makes this task type synchronously wait for a result to be ready for us. The promise type that owns the binary semaphore looks like this:

```
struct Promise {
  T* value_{nullptr};
  std::exception_ptr error_;
  std::binary_semaphore semaphore_;
  SyncWaitTask get_return_object() noexcept {
```

```
    return SyncWaitTask{*this};
  }
  void unhandled_exception() noexcept {
    error_ = std::current_exception();
  }
  auto yield_value(T&& x) noexcept {     // Result has arrived
    value_ = std::addressof(x);
    return final_suspend();
  }
  auto initial_suspend() noexcept {
    return std::suspend_always{};
  }
  auto final_suspend() noexcept {
  struct Awaitable {
      bool await_ready() noexcept { return false; }
      void await_suspend(std::coroutine_handle<Promise> h) noexcept {
        h.promise().semaphore_.release();         // Signal!
      }
      void await_resume() noexcept {}
    };
    return Awaitable{};
  }
  void return_void() noexcept { assert(false); }
};
```

There's a lot of boilerplate code here that we have already talked about. But pay special attention to `yield_value()` and `final_suspend()`, which is the interesting part of this class. Recall that the coroutine lambda inside `sync_wait()` yielded the return value like this:

```
// ...
auto coro = [&]() -> detail::SyncWaitTask<Result<T>> {
  co_yield co_await std::forward<T>(task);
  // ...
```

So, once the value is yielded, we end up in `yield_value()` of the promise object. And the fact that `yield_value()` can return an awaitable type gives us the opportunity to customize the behavior of the `co_yield` keyword. In this case, `yield_value()` returns an awaitable that will signal through the binary semaphore that a value from the original `Task` object has been produced.

The semaphore is signaled inside `await_suspend()`. We cannot signal earlier than that because the other end of the code waiting for the signal will eventually destroy the coroutine. Destroying a coroutine must only happen if the coroutine is in a suspended state.

The blocking call to `semaphore_.acquire()` from within `SyncWaitTask::get()` will return on the signal, and finally the computed value will be handed over to the client that called `sync_wait()`.

Testing asynchronous tasks with sync_wait()

Finally, a small asynchronous test program using `Task` and `sync_wait()` can be constructed like this:

```
auto height() -> Task<int> { co_return 20; }    // Dummy coroutines
auto width() -> Task<int> { co_return 30; }
auto area() -> Task<int> {
  co_return co_await height() * co_await width();
}
```

```
int main() {
  auto a = area();
  int value = sync_wait(a);
  std::cout << value;        // Outputs: 600
}
```

We have implemented the absolute minimum infrastructure for using asynchronous tasks with C++ coroutines. More infrastructure is needed, though, in order to use coroutines for asynchronous programming effectively. This is a big difference from the generator (presented in the previous chapter), which required a fairly small amount of groundwork before we could really benefit from it. To get a little bit closer to the real world, we will, in the following sections, explore some examples using Boost.Asio. The first thing we will do is to try to wrap a callback-based API inside an API compatible with C++ coroutines.

Wrapping a callback-based API

There are many asynchronous APIs based on callbacks. Typically, an asynchronous function takes a callback function provided by the caller. The asynchronous function returns immediately and then eventually invokes the callback (completion handler) when the asynchronous function has a computed value or is done waiting for something.

To show you what an asynchronous callback-based API can look like, we will take a peek at a Boost library for asynchronous I/O named **Boost.Asio**. There is a lot to learn about Boost.Asio that won't be covered here; I will only describe the absolute minimum of the Boost code and instead focus on the parts directly related to C++ coroutines.

To make the code fit the pages of the book, the examples assume that the following namespace alias has been defined whenever we use code from Boost.Asio:

```
namespace asio = boost::asio;
```

Here is a complete example of using Boost.Asio for delaying a function call but without blocking the current thread. This asynchronous example runs in a single thread:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
using namespace std::chrono;
namespace asio = boost::asio;
int main() {
  auto ctx = asio::io_context{};
  auto timer = asio::system_timer{ctx};
  timer.expires_from_now(1000ms);
  timer.async_wait([](auto error) {      // Callback
    // Ignore errors..
    std::cout << "Hello from delayed callback\n";
  });
  std::cout << "Hello from main\n";
  ctx.run();
}
```

Compiling and running this program will generate the following output:

```
Hello from main
Hello from delayed callback
```

When using Boost.Asio, we always need to create an `io_context` object that runs an event processing loop. The call to `async_wait()` is asynchronous; it returns immediately back to `main()` and invokes the callback (the lambda) when the timer expires.

The timer example does not use coroutines but instead a callback API to provide asynchronicity. Boost.Asio is also compatible with C++20 coroutines, which I will demonstrate later on. But on our path to explore awaitable types, we will take a detour and instead assume that we need to provide a coroutine-based API that returns awaitable types on top of the callback-based API of Boost.Asio. In that way, we can use a `co_await` expression to call and wait (but without blocking the current thread) for the asynchronous task to complete. Instead of using a callback, we would like to be able to write something like this:

```
std::cout << "Hello! ";
co_await async_sleep(ctx, 100ms);
std::cout << "Delayed output\n";
```

Let's see how we can implement the function `async_sleep()` so that it can be used with `co_await`. The pattern we will follow is to have `async_sleep()` return an awaitable object that will implement the three required functions: `await_ready()`, `await_suspend()`, and `await_resume()`. An explanation of the code will follow after it:

```
template <typename R, typename P>
auto async_sleep(asio::io_context& ctx,
                 std::chrono::duration<R, P> d) {
  struct Awaitable {
    asio::system_timer t_;
    std::chrono::duration<R, P> d_;
    boost::system::error_code ec_{};
    bool await_ready() { return d_.count() <= 0; }
    void await_suspend(std::coroutine_handle<> h) {
      t_.expires_from_now(d_);
      t_.async_wait([this, h](auto ec) mutable {
        this->ec_ = ec;
        h.resume();
      });
    }
    void await_resume() {
      if (ec_) throw boost::system::system_error(ec_);
    }
  };
  return Awaitable{asio::system_timer{ctx}, d};
}
```

Once again, we are creating a custom awaitable type that does all the necessary work:

- `await_ready()` will return `false` unless the timer has already reached zero.
- `await_suspend()` starts the asynchronous operation and passes a callback that will be called when the timer has expired or produced an error. The callback saves the error code (if any) and resumes the suspended coroutine.

- `await_resume()` has no result to unpack because the asynchronous function we are wrapping, `boost::asio::timer::async_wait()`, does not return any value except an optional error code.

Before we can actually test `async_sleep()` in a standalone program, we need some way to start the `io_context` run loop and break the chain of coroutines, as we did when testing the `Task` type previously. We will do that in a rather hacky way here by implementing two functions, `run_task()` and `run_task_impl()`, and a naive coroutine return type called `Detached` that ignores error handling and can be discarded by the caller:

```
// This code is here just to get our example up and running
struct Detached {
  struct promise_type {
    auto get_return_object() { return Detached{}; }
    auto initial_suspend() { return std::suspend_never{}; }
    auto final_suspend() noexcept { return std::suspend_never{};}
    void unhandled_exception() { std::terminate(); } // Ignore
    void return_void() {}
  };
};
Detached run_task_impl(asio::io_context& ctx, Task<void>&& t) {
  auto wg = asio::executor_work_guard{ctx.get_executor()};
  co_await t;
}
void run_task(asio::io_context& ctx, Task<void>&& t) {
  run_task_impl(ctx, std::move(t));
  ctx.run();
}
```

The `Detached` type makes the coroutine start immediately and runs the coroutine detached from the caller. The `executor_work_guard` prevents the `run()` call from returning until the coroutine `run_task_impl()` has completed.

Starting operations and detaching them should typically be avoided. It's similar to detached threads or allocated memory without any references. However, the purpose of this example is to demonstrate what we can use awaitable types for and how we can write asynchronous programs and run them single-threaded.

Everything is in place; the wrapper called `async_sleep()` returns a `Task` and a function `run_task()`, which can be used to execute a task. It's time to write a small coroutine to test the new code we implemented:

```
auto test_sleep(asio::io_context& ctx) -> Task<void> {
  std::cout << "Hello!  ";
  co_await async_sleep(ctx, 100ms);
  std::cout << "Delayed output\n";
}
int main() {
  auto ctx = asio::io_context{};
  auto task = test_sleep(ctx);
  run_task(ctx, std::move(task));
};
```

Executing this program will generate the following output:

```
Hello! Delayed output
```

You have seen how a callback-based API can be wrapped in a function that can be used by `co_await` and therefore allows us to use coroutines instead of callbacks for asynchronous programming. This program also provided a typical example of how the functions in the awaitable type can be used. However, as mentioned earlier, it turns out that recent versions of Boost, starting with 1.70, already provide an interface that is compatible with C++20 coroutines. In the next section, we will use this new coroutine API when building a tiny TCP server.

A concurrent server using Boost.Asio

This section will demonstrate how to write concurrent programs that have multiple threads of execution but only use a single OS thread. We are about to implement a rudimentary concurrent single-threaded TCP server that can handle multiple clients. There are no networking capabilities in the C++ standard library, but fortunately Boost.Asio provides us with a platform-agnostic interface for handling socket communication.

Instead of wrapping the callback-based Boost.Asio API, I will demonstrate how to use the `boost::asio::awaitable` class for the purpose of showing a more realistic example of how asynchronous application programming using coroutines can look. The class template `boost::asio::awaitable` corresponds to the `Task` template we created earlier; it's used as a return type for coroutines that represent asynchronous computations.

Implementing the server

The server is very simple; once a client connects, it starts updating a numeric counter and writes back the value whenever it is updated. This time we will follow the code from top to bottom, starting with the `main()` function:

```
#include <boost/asio.hpp>
#include <boost/asio/awaitable.hpp>
#include <boost/asio/use_awaitable.hpp>
using namespace std::chrono;
namespace asio = boost::asio;
using boost::asio::ip::tcp;
int main() {
  auto server = [] {
    auto endpoint = tcp::endpoint{tcp::v4(), 37259};
    auto awaitable = listen(endpoint);
    return awaitable;
  };
  auto ctx = asio::io_context{};
  asio::co_spawn(ctx, server, asio::detached);
  ctx.run(); // Run event loop from main thread
}
```

The mandatory `io_context` runs the event processing loop. It's possible to invoke `run()` from multiple threads as well, if we want our server to execute multiple OS threads. In our case we only use one thread but with multiple concurrent flows. The function `boost::asio::co_spawn()` starts a detached concurrent flow. The server is implemented using a lambda; it defines a TCP endpoint (with port 37259) and starts listening for incoming client connections on the endpoint.

The coroutine `listen()` is fairly simple and looks like this:

```
auto listen(tcp::endpoint endpoint) -> asio::awaitable<void> {
  auto ex = co_await asio::this_coro::executor;
  auto a = tcp::acceptor{ex, endpoint};
```

```
  while (true) {
    auto socket = co_await a.async_accept(asio::use_awaitable);
    auto session = [s = std::move(socket)]() mutable {
      auto awaitable = serve_client(std::move(s));
      return awaitable;
    };
    asio::co_spawn(ex, std::move(session), asio::detached);
  }
}
```

The executor is the object responsible for actually executing our asynchronous functions. An executor may represent a thread pool or a single system thread, for example. We will most likely see some form of executors in upcoming versions of C++ to give us programmers more control and flexibility over when and where our code executes (including GPUs).

Next, the coroutine runs an infinite loop and waits for TCP clients to connect. The first `co_await` expression returns a socket when a new client successfully connects to our server. The socket object is then moved to the coroutine `serve_client()`, which will serve the newly connected client until the client disconnects.

The main application logic of the server happens in the coroutine that handles each client. Here is how it looks:

```
auto serve_client(tcp::socket socket) -> asio::awaitable<void> {
  std::cout << "New client connected\n";
  auto ex = co_await asio::this_coro::executor;
  auto timer = asio::system_timer{ex};
```

```
  auto counter = 0;
  while (true) {
   try {
    auto s = std::to_string(counter) + "\n";
    auto buf = asio::buffer(s.data(), s.size());
    auto n = co_await async_write(socket, buf, asio::use_awaitable);
    std::cout << "Wrote " << n << " byte(s)\n";
    ++counter;
    timer.expires_from_now(100ms);
    co_await timer.async_wait(asio::use_awaitable);
   } catch (...) {
    // Error or client disconnected
    break;
   }
  }
 }
```

Each coroutine invocation serves one unique client during the entire client session; it runs until the client disconnects from the server. The coroutine updates a counter at regular intervals (every 100 ms) and writes the value asynchronously back to the client using `async_write()`. Note how we can write the function `serve_client()` in a linear fashion although it invokes two asynchronous operations: `async_write()` and `async_wait()`.

Running and connecting to the server

Once we have started this server, we can connect clients on port 37259. To try this out, I'm using a tool called `nc` (netcat), which can be used for communicating over TCP and UDP. Here is an example of a short session where a client connects to the server running on localhost:

```
[client] $ nc localhost 37259
0
1
2
3
```

We can start multiple clients and they will all be served by a dedicated `serve_client()` coroutine invocation and have their own copy of the incrementing counter variable, as shown in the screenshot below:

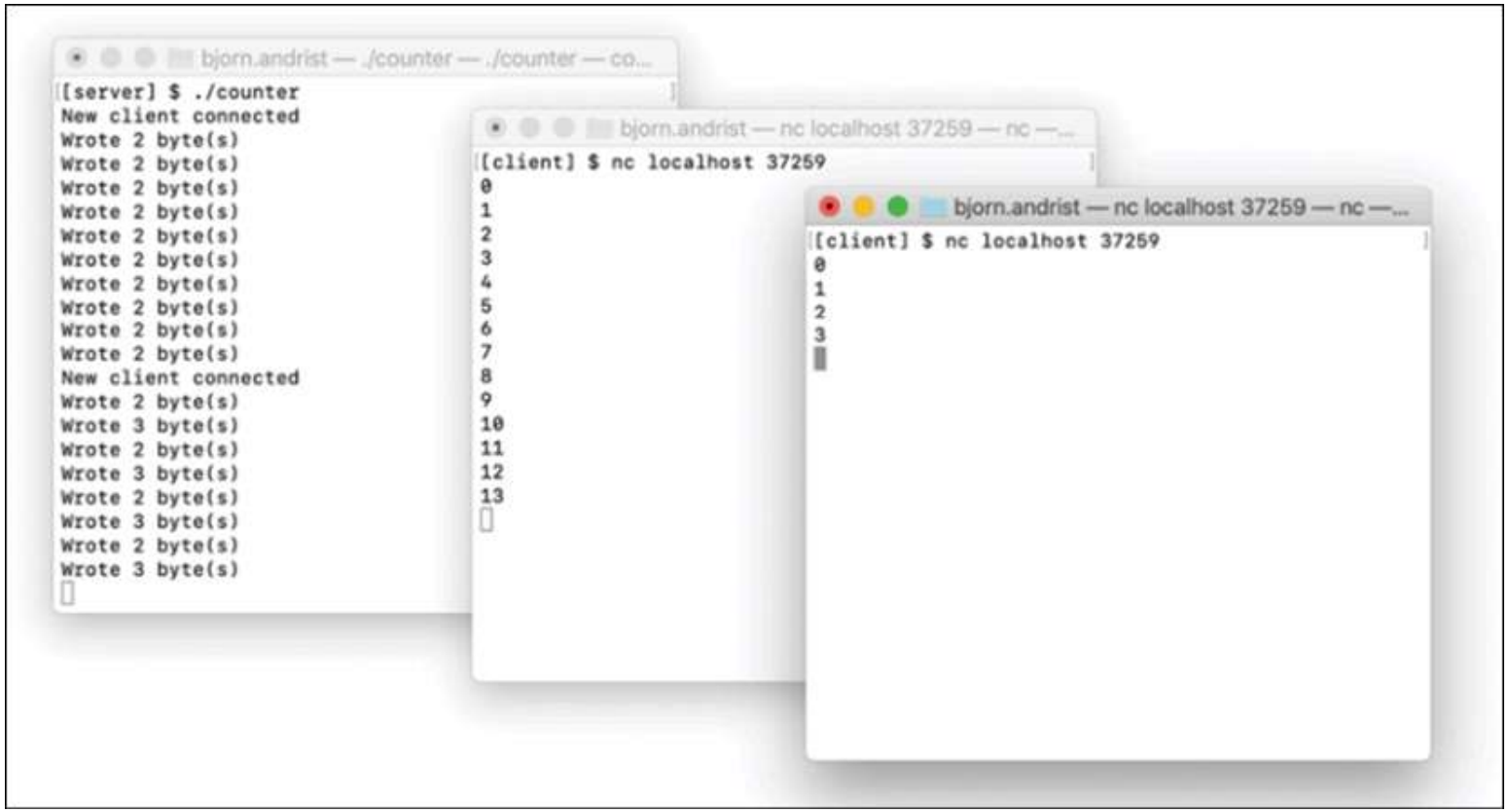

Figure 13.2: A running server with two connected clients

Another way to create an application serving multiple sessions concurrently would be to create one thread for each new client that connects. However, the memory overhead of threads would set the limit of the number of sessions substantially lower compared to this model using coroutines.

The coroutines in this example are all executed on the same thread, which makes the locking of shared resources unnecessary. Imagine we had a global counter that each session updated. If we used multiple threads, the access to the global counter would need some kind of synchronization (using a mutex or an atomic data type). This is not necessary for coroutines that execute on the same thread. In other words, coroutines that execute on the same thread can share state without using any locking primitives.

What we have achieved with the server (and what we haven't)

The example application using Boost.Asio demonstrates that coroutines can be used for asynchronous programming. Instead of implementing continuations with nested callbacks, we can write code in a linear fashion using `co_await` statements. However, this example is minimal and avoids some really important aspects of asynchronous programming, such as:

- Asynchronous read and write operations. The server only writes data to its clients and ignores the challenge of synchronizing read and write operations.
- Canceling asynchronous tasks and graceful shutdown. The server runs in an infinite loop, completely ignoring the challenge of a clean shutdown.
- Error handling and exception safety when using multiple `co_await` statements.

These topics are immensely important but are out of scope for this book. I already mentioned that detached operations are best avoided. Creating detached tasks using `boost::asio::co_spawn()`, as shown in

the example, should be done with utmost caution. A fairly new programming paradigm for avoiding detached work is called **structured concurrency**. It aims to solve exception safety and the cancellation of multiple asynchronous tasks by encapsulating concurrency into general and reusable algorithms such as `when_all()` and `stop_when()`. The key idea is to never allow some child task to exceed the lifetime of its parent. This makes it possible to pass local variables by reference to asynchronous child operations safely and with better performance. Strictly nested lifetimes of concurrent tasks also make the code easier to reason about.

Another important aspect is that asynchronous tasks should always be lazy (immediately suspended), so that continuations can be attached before any exceptions can be thrown. This is also a requirement if you want to be able to cancel a task in a safe manner.

There will most likely be a lot of talks, libraries, and articles related to this important subject in the years to come. Two talks from CppCon 2019 addressed this topic:

- *A Unifying Abstraction for Async in C++*, Eric Neibler and D. S. Hollman, **https://sched.co/SfrC**
- *Structured Concurrency: Writing Safer Concurrent Code with Coroutines and Algorithms*, Lewis Baker, **https://sched.co/SfsU**

Summary

In this chapter, you've seen how to use C++ coroutines for writing asynchronous tasks. To be able to implement the infrastructure in the form of a `Task` type and a `sync_wait()` function, you needed to fully understand the concept of awaitable types and how they can be used to customize the behavior of coroutines in C++.

By using Boost.Asio, we could build a truly minimal but fully functional concurrent server application executing on a single thread while handling multiple client sessions.

Lastly, I briefly introduced a methodology called structured concurrency and gave some directions for where you can find more information about this topic.

In the next chapter, we will move on to explore parallel algorithms, which are a way to speed up concurrent programs by utilizing multiple cores.