12
Coroutines and Lazy Generators

Computing has become a world of waiting, and we need support in our programming languages to be able to express *wait*. The general idea is to suspend (temporarily pause) the current flow and hand execution over to some other flow, whenever it reaches a point where we know that we might have to wait for something. This *something* that we need to wait for could be a network request, a click from a user, a database operation, or even a memory access that is taking too long for us to block at. Instead, we say in our code that we will wait, continue some other flow, and then come back when ready. Coroutines allow us to do that.

In this chapter, we're mainly going to focus on coroutines added to C++20. You will learn what they are, how to use them, and their performance characteristics. But we will also spend some time looking at coroutines in a broader sense, since the concept is apparent in many other languages.

C++ coroutines come with very little support from the standard library. Adding standard library support for coroutines is a high-priority feature for the C++23 release. In order to use coroutines effectively in our day-to-day code, we need to implement some general abstractions. This book will show you how to implement these abstractions for the purpose of learning C++ coroutines rather than providing you with production-ready code.

It's also important to understand the various types of coroutines that exist, what coroutines can be used for, and what motivated C++ to add new language features to support coroutines.

This chapter covers a lot of ground. The next chapter is also about coroutines but with a focus on asynchronous applications. In summary, this chapter will guide you through:

- General theory about coroutines, including the difference between stackful and stackless coroutines, and how they are transformed by the compiler and executed on a computer.
- An introduction to stackless coroutines in C++. The new language support for coroutines in C++20 using `co_await`, `co_yield`, and `co_return` will be discussed and demonstrated.
- The abstractions that are needed for using C++20 coroutines as generators.
- A few real-world examples that show the benefits in terms of readability and simplicity of using coroutines and how we can write composable components that will evaluate lazily by using coroutines.

If you have been working with coroutines in other languages, you need to be prepared for two things before reading the rest of this chapter:

- Some content may feel basic to you. Although the details about how C++ coroutines work are far from trivial, the usage examples might feel trivial to you.
- Some terms we will use in this chapter (coroutines, generators, tasks, and so forth) might not align with your current view of what these are.

On the other hand, if you are completely new to coroutines, parts of this chapter may very well look like magic and take some time to grasp. I will therefore begin by showing you a few examples of how C++ code can look when using coroutines.

A few motivating examples

Coroutines are one of those features, similar to lambda expressions, that offer a way to completely change the way we write and think about C++ code. The concept is very general and can be applied in many different ways. To give you a taste of how C++ can look when using coroutines, we will here look briefly at two examples.

Yield-expressions can be used for implementing generators—objects that produce sequences of values lazily. In this example, we will use the keywords `co_yield` and `co_return` to control the flow:

```
auto iota(int start) -> Generator<int> {
  for (int i = start; i < std::numeric_limits<int>::max(); ++i) {
    co_yield i;
  }
}
auto take_until(Generator<int>& gen, int value) -> Generator<int> {
  for (auto v : gen) {
    if (v == value) {
      co_return;
    }
    co_yield v;
  }
}
int main() {
  auto i = iota(2);
  auto t = take_until(i, 5);
  for (auto v : t) {        // Pull values
    std::cout << v << ", ";
```

```
  }
  return 0;
}
// Prints: 2, 3, 4
```

In the preceding example, `iota()` and `take_until()` are coroutines. `iota()` generates a sequence of integers and `take_until()` yields values until it finds the specified value. The `Generator` template is a custom type that I will show you how to design and implement later on in this chapter.

Building generators is one common use case for coroutines, another one is implementing asynchronous tasks. The next example will demonstrate how we can use the operator `co_await` to wait for something without blocking the currently executing thread:

```
auto tcp_echo_server() -> Task<> {
  char data[1024];
  for (;;) {
    size_t n = co_await async_read(socket, buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}
```

Instead of blocking, `co_await` suspends the execution until it gets resumed and the asynchronous read and write functions have completed. The example presented here is incomplete because we don't know what `Task`, `socket`, `buffer`, and the asynchronous I/O functions are. But we will get there in the next chapter when focusing on asynchronous tasks.

Don't worry if it is not clear how these examples work at this point—we will spend a lot of time delving into the details later on in this chapter. The examples are here to give you a hint about what coroutines allow us to do if you have never encountered them before.

Before digging into C++20 coroutines, we need to discuss some terminology and common foundational ground to better understand the design and motivation for adding a rather complicated language feature to C++ in 2020.

The coroutine abstraction

We will now take a step back and talk about coroutines in general and not just focus on the coroutines added to C++20. This will give you a better understanding of why coroutines are useful but also what types of coroutines there are and how they differ. If you are already familiar with stackful and stackless coroutines and how they are executed, you can skip this section and jump right to the next section, *Coroutines in C++*.

The coroutine abstraction has been around for more than 60 years and many languages have adopted some sort of coroutines into their syntax or standard libraries. This means that coroutines can denote slightly different things in different languages and environments. Since this is a book about C++, I will use the terminology used in the C++ standard.

Coroutines are very similar to subroutines. In C++, we don't have anything explicitly called subroutines; instead, we write functions (free functions or member functions, for example) to create subroutines. I will use the terms **ordinary functions** and **subroutines** interchangeably.

Subroutines and coroutines

To understand the difference between coroutines and subroutines (ordinary functions), we will here focus on the most basic properties of subroutines and coroutines, namely, how to start, stop, pause, and resume them. A subroutine is started when some other part of our program calls it. When the subroutine returns back to the caller, the subroutine stops:

```
auto subroutine() {
 // Sequence of statements ...

 return;    // Stop and return control to caller
}
subroutine(); // Call subroutine to start it
// subroutine has finished
```

The call chain of subroutines is strictly nested. In the diagram that follows, subroutine `f()` cannot return to `main()` until subroutine `g()` has returned:

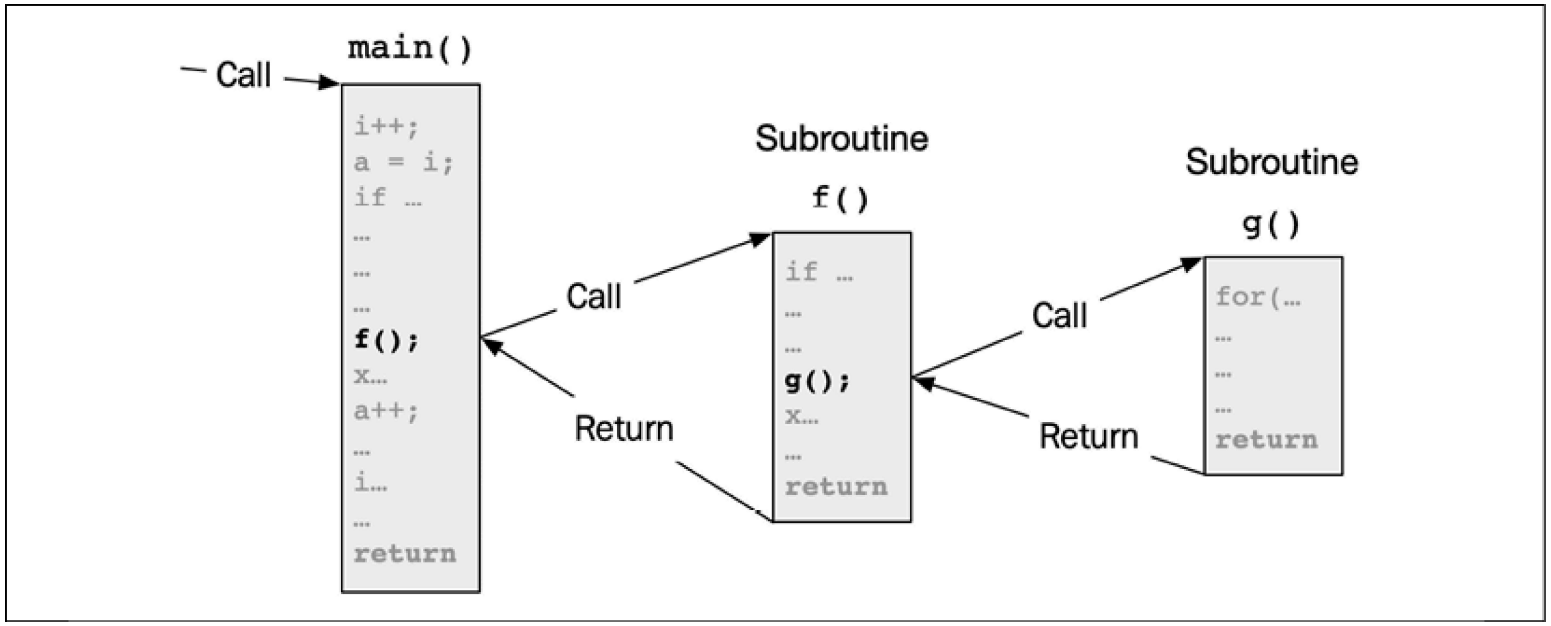

Figure 12.1: A chain of subroutine calls and returns

Coroutines can also be started and stopped just like subroutines, but they can also be **suspended** (paused) and **resumed**. If you haven't worked with coroutines before, this may seem very strange at first. The point where a coroutine is suspended and resumed is called a **suspend/resume point**. Some suspend points are implicit whereas others are explicitly marked in the code in one way or another. The following pseudo code shows three explicit suspend/resume points marked using `await` and `yield`:

```
// Pseudo code
auto coroutine() {
  value = 10;
  await something;      // Suspend/Resume point
  // ...
  yield value++;        // Suspend/Resume point
```

```
  yield value++;        // Suspend/Resume point
  // ...
  return;
}
auto res = coroutine();   // Call
res.resume();             // Resume
```

In C++, the explicit suspend points are marked using the keywords `co_await` and `co_yield`. The diagram that follows shows how a coroutine is invoked (called) from one subroutine and then later resumed from different parts of the code:

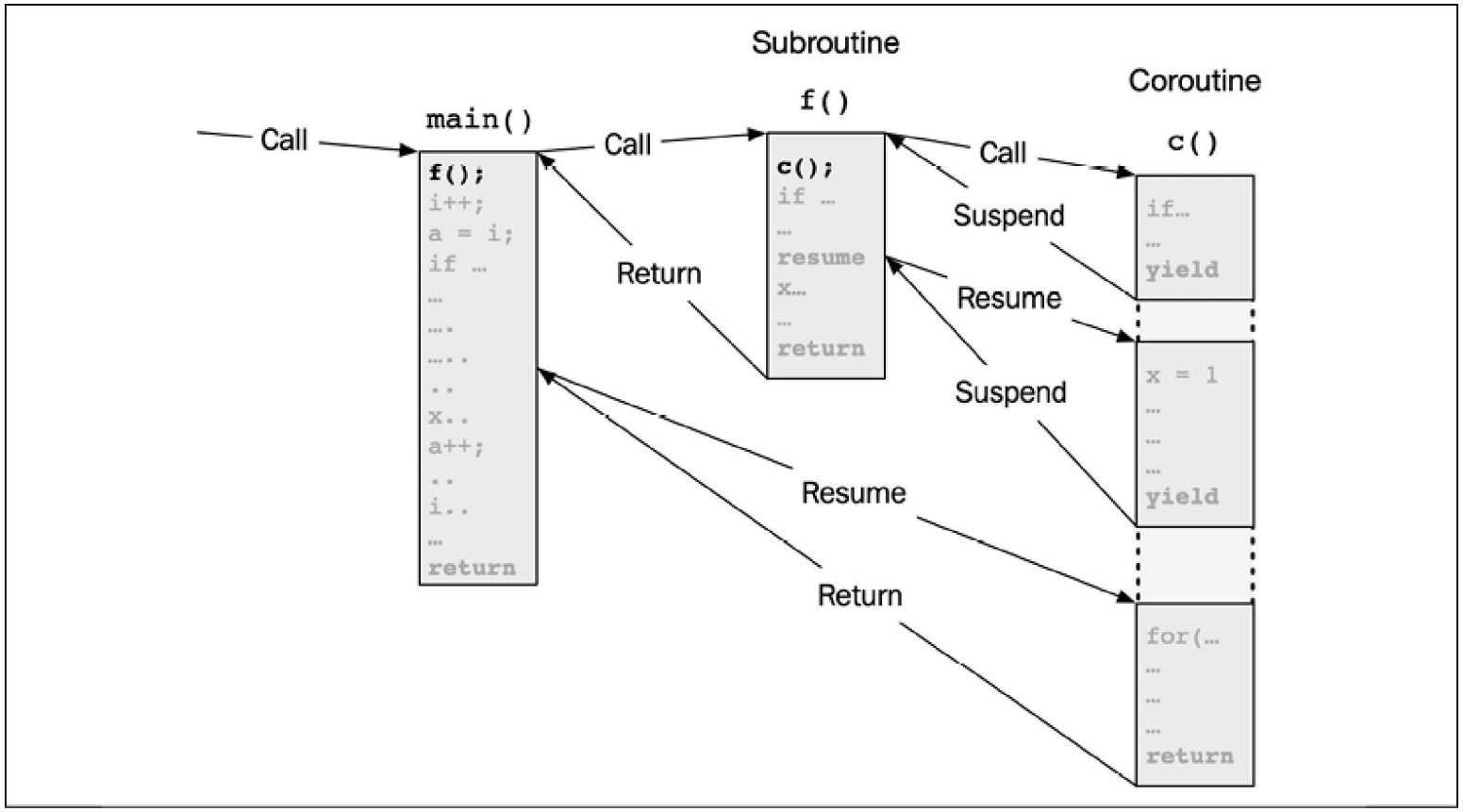

Figure 12.2: An invocation of a coroutine can suspend and resume. The coroutine invocation maintains its internal state while being suspended.

The states of local variables inside a coroutine are preserved while the coroutine is suspended. The states belong to a certain invocation of a coroutine. That is, they are not like static local variables, which are globally shared among all invocations of a function.

To summarize, coroutines are subroutines that also can be suspended and resumed. Another way to look at it is to say that subroutines are a specialization of coroutines that cannot be suspended or resumed.

From now on, I will be very strict when distinguishing between *call* and *resume*, and *suspend* and *return*. They mean completely different things. Calling a coroutine creates a new instance of a coroutine that can be suspended and resumed. Returning from a coroutine destroys the coroutine instance and it can no longer be resumed.

To really understand how coroutines can help us write efficient programs, you need to be aware of some low-level details about how functions in C++ are usually transformed to machine code and then executed.

Executing subroutines and coroutines on the CPU

We have talked about memory hierarchies, caches, virtual memory, scheduling of threads, and other hardware and operating system concepts in this book. But we haven't really talked about how instructions are being executed on the CPU using CPU registers and the stack. These concepts are important to understand when comparing subroutines with various flavors of coroutines.

CPU registers, instructions, and the stack

This section will provide a very simplified model of a CPU for the purpose of understanding context switching, function calls, and a few more details regarding the call stack. When I say CPUs in this context, I refer to some CPUs that are similar to the x86 family of CPUs equipped with multiple general-purpose registers.

A program contains a sequence of instructions that the CPU executes. The sequence of instructions is stored somewhere in the memory of the computer. The CPU keeps track of the address of the currently executing instruction in a register called a **program counter**. In that way, the CPU knows what instruction to execute next.

The CPU contains a fixed number of registers. A register is similar to a variable with a predefined name that can store a value or a memory address. Registers are the fastest data storage available on a computer and sit closest to the CPU. When the CPU manipulates data, it uses the registers. Some of the registers have a special meaning to the CPU, whereas other registers can be used more freely by the currently executing program.

Two very important registers that have a special meaning to the CPU are:

- **Program counter** (**PC**): The register that stores the memory address of the currently executing instruction. This value is automatically incremented whenever an instruction is executed. Sometimes it is also called an *instruction pointer*.
- **Stack pointer** (**SP**): It stores the address of the top of the currently used call stack. Allocating and deallocating stack memory is a matter of changing the value stored in this single register.

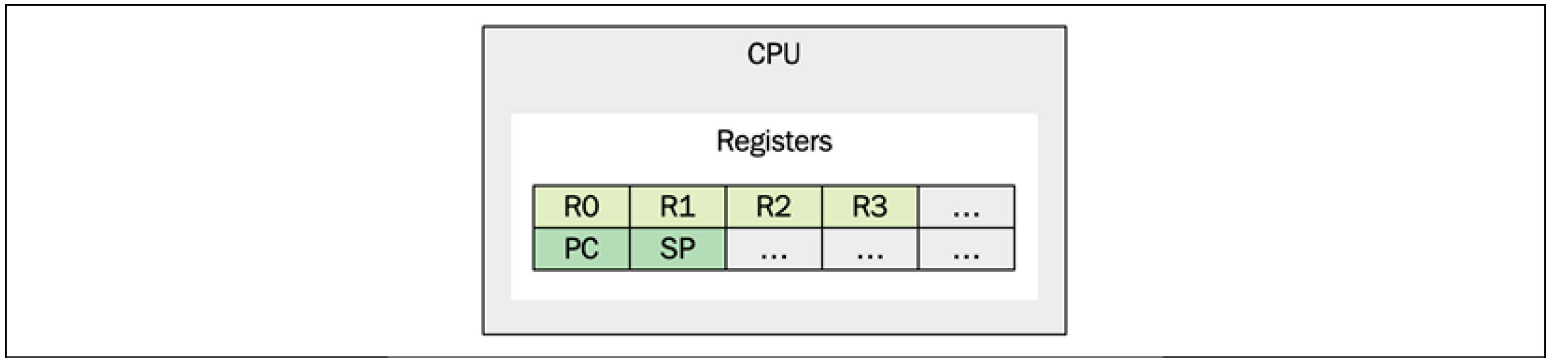

Figure 12.3: A CPU with registers

Assume that the registers are called **R0**, **R1**, **R2**, and **R3** as in the preceding diagram. A typical arithmetic instruction could then look like this:

```
add 73, R1   // Add 73 to the value stored in R1
```

Data can also be copied between registers and memory:

```
mov SP, R2   // Copy the stack pointer address to R2
mov R2, [R1] // Copy value of R2 to memory address stored in R1
```

A set of instructions refers implicitly to the call stack. The CPU knows where the top of the call stack is through the stack pointer. Allocating memory on the stack is only a matter of updating the stack pointer. The value increases or decreases depending on whether the stack grows towards higher or lower addresses.

The following instruction uses the stack:

```
push R1    // Push value of R1 to the top of the stack
```

The push instruction copies the value in the register to the place in memory pointed at by the stack pointer *and* increments (or decrements) the stack pointer.

We can also pop values from the stack by using the `pop` instruction, which also reads and updates the stack pointer:

```
pop R2     // Pop value from the stack into R2
```

Whenever an instruction is executed, the CPU automatically increments the program counter. But the program counter can also be explicitly updated through instructions, for example, the `jump` instruction:

```
jump R3    // Set the program counter to the address in R3
```

The CPU can operate in two modes: user mode or kernel mode. The CPU registers are used differently when running in user mode and kernel mode. When the CPU is executing in user mode, it runs with restricted privileges that cannot access hardware. The operating system provides system calls that run in kernel mode. A C++ library function such as `std::puts()`, which prints values to `stdout`, must therefore make a system call to complete its task, forcing the CPU to switch between user mode and kernel mode.

Transitioning between user and kernel mode is expensive. To understand why, let's think about our schematic CPU again. The CPU operates efficiently by using its registers and therefore avoids spilling values onto the stack unnecessarily. But the CPU is a shared resource among all user processes and the operating system, and whenever we need to switch between tasks (for example, when entering kernel mode), the state of the processor, including all of its registers, needs to be saved in memory so that it can be resumed later on.

Call and return

Now that you have a basic understanding of how the CPU uses registers and the stack, we can discuss subroutine invocations. There are a lot of mechanisms involved when calling and returning from a subroutine that we might take for granted. Our compilers are doing an excellent job when they transform a C++ function to highly optimized machine code.

The following list shows the aspects that need to be considered when calling, executing, and returning from a subroutine:

- Calling and returning (jumping between points in the code).
- Passing parameters—parameters can be passed through registers or on the stack, or both.
- Allocating storage for local variables on the stack.
- Returning a value—the value returned from a subroutine needs to be stored in a place where the caller can find it. Typically, this is a dedicated CPU register.
- Using registers without interfering with other functions—the registers that a subroutine uses need to be restored to the state they were in before the subroutine was called.

The exact details about how function calls are carried out are specified by something called **calling conventions**. They provide a protocol for the caller/callee to agree on who is responsible for which parts. Calling conventions differ among CPU architectures and compilers and are one of the major parts that constitutes an **application binary interface** (**ABI**).

When a function is being called, a **call frame** (or activation frame) for that function is being created. The call frame contains:

- The *parameters* passed to the function.
- The *local variables* of the function.
- A *snapshot of the registers* that we intend to use and therefore need to restore before returning.
- A *return address* that links back to the place in memory where the caller invoked the function from.
- An optional *frame pointer* that points back to the top of the caller's call frame. Frame pointers are useful for debuggers when inspecting the stack. We will not discuss frame pointers further in this book.

Thanks to the strictly nested nature of subroutines, we can save the call frames of the subroutines on the stack to support nested calls very efficiently. A call frame stored on the stack is usually called a **stack frame**.

The following diagram shows multiple call frames on a call stack and highlights the contents of a single call frame:

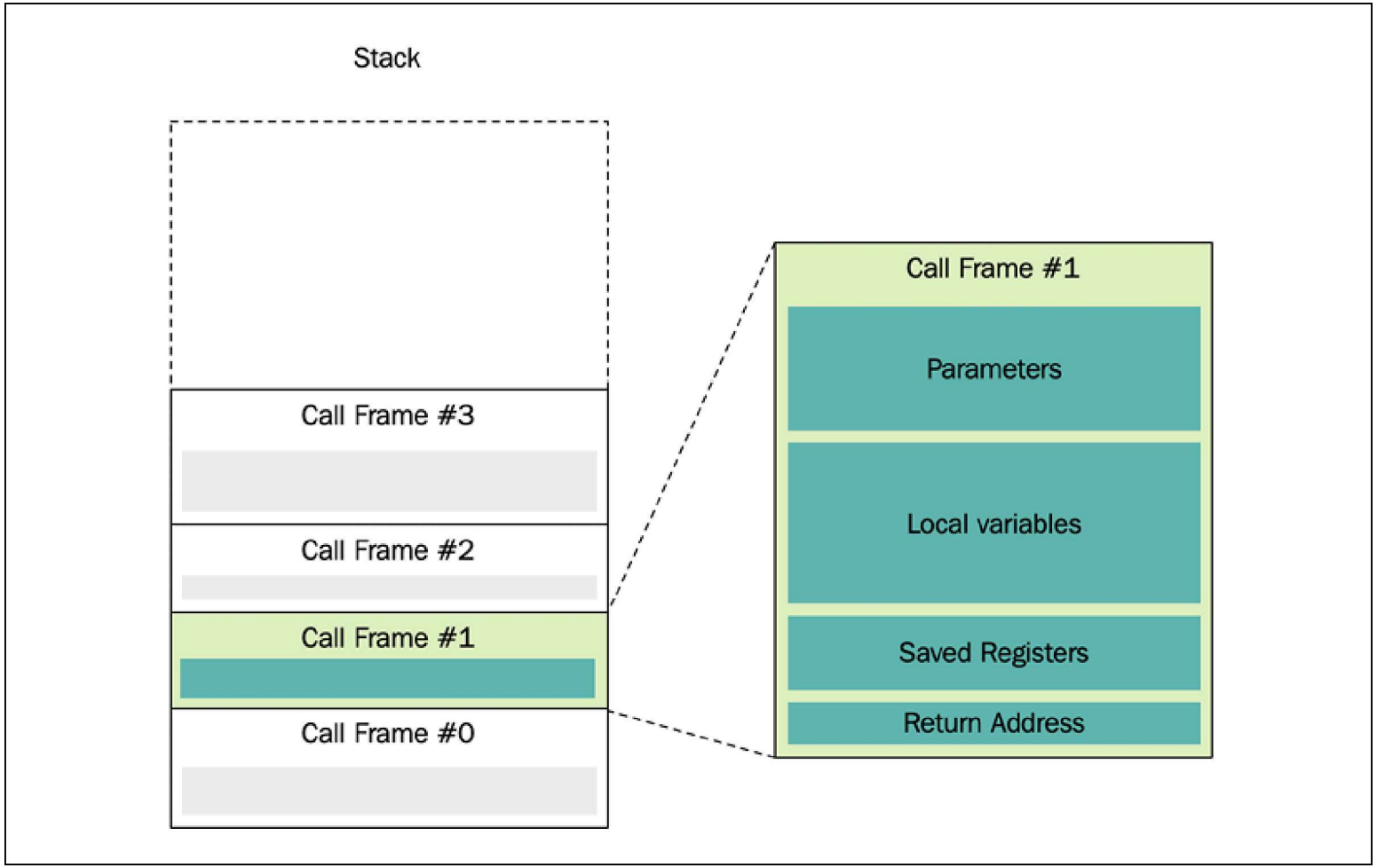

Figure 12.4: A call stack with multiple call frames. The call frame on the right-hand side is a zoomed-in version of a single call frame.

When a subroutine returns back to its caller, it uses the return address to know where to jump, restores the registers it has mutated, and pops (deallocates) the entire call frame off the stack. In this way, both the stack and the registers are restored to the states they were in before the call of the subroutine was invoked. However, there are two exceptions. Firstly, the program counter (PC) has moved to the instruction after the call. Secondly, a subroutine that returns a value back to its caller usually stores that value in a dedicated register where the caller knows where to find it.

After understanding how a subroutine is executed by temporarily using the stack and then restoring the CPU registers before returning control back to its caller, we can now start to look at how it's possible to suspend and resume coroutines.

Suspend and resume

Consider the following pseudo code that defines a coroutine with multiple suspend/resume points:

```
// Pseudo code
auto coroutine() {
  auto x = 0;
  yield x++;      // Suspend
  g();            // Call some other function
  yield x++;      // Suspend
  return;         // Return
}
auto co = coroutine(); // Call subroutine to start it
```

```
// ...              // Coroutine is suspended
auto a = resume(co);   // Resume coroutine to get
auto b = resume(co);   // next value
```

When `coroutine()` suspends, we can no longer remove the call frame as we do when a subroutine returns back to its caller. Why? Because we need to keep the current value of the variable, `x`, and also remember *where* in the coroutine we should continue executing the next time the coroutine is resumed. This information is placed into something called a **coroutine frame**. The coroutine frame contains all the information that is needed in order to resume a paused coroutine. This raises several new questions, though:

- Where is the coroutine frame stored?
- How big is the coroutine frame?
- When a coroutine calls a subroutine, it needs a stack to manage the nested call frames. What happens if we try to resume from within a nested call frame? Then we would need to restore the entire stack when the coroutine resumes.
- What is the runtime overhead of calling and returning from a coroutine?
- What is the runtime overhead of suspending and resuming a coroutine?

The short answer to these questions is that it depends on what type of coroutine we are discussing: stackless or stackful coroutines.

Stackful coroutines have a separate side stack (similar to a thread) that contains the coroutine frame and the nested call frames. This makes it possible to suspend from nested call frames:

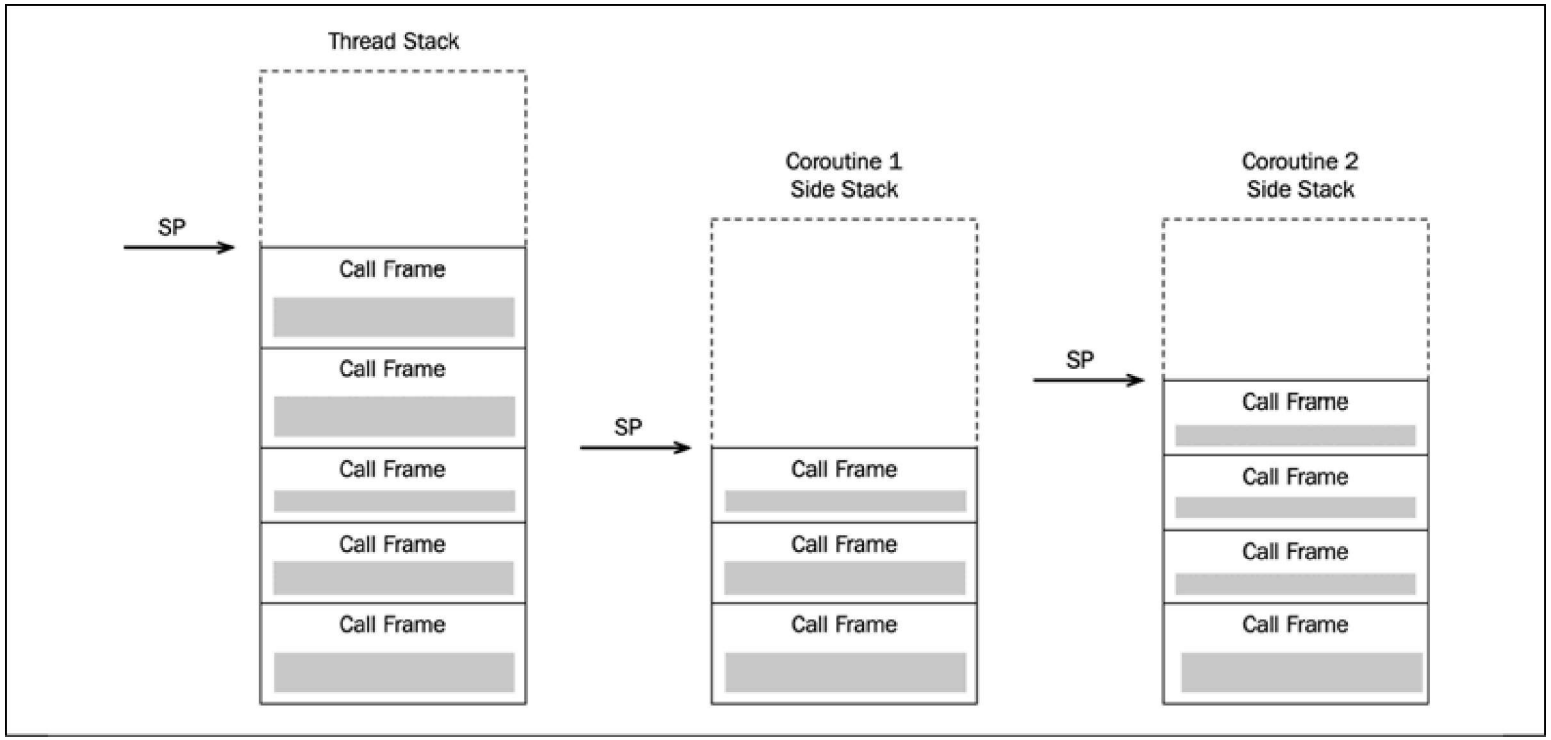

Figure 12.5: Each call to a stackful coroutine creates a separate side stack with a unique stack pointer

Suspending and resuming stackless coroutines

Stackless coroutines need to store the coroutine frame somewhere else (typically on the heap) and then use the stack of the currently executing thread to store nested call frames.

But this is not the entire truth. The caller is the one responsible for creating the call frame, saving the return address (current value of the program counter), and the parameters on the stack. The caller doesn't know that it is calling a coroutine that will suspend and resume. Therefore, the coroutine itself needs to create the coroutine frame and copy the parameters and registers from the call frame to the coroutine frame when it is called:

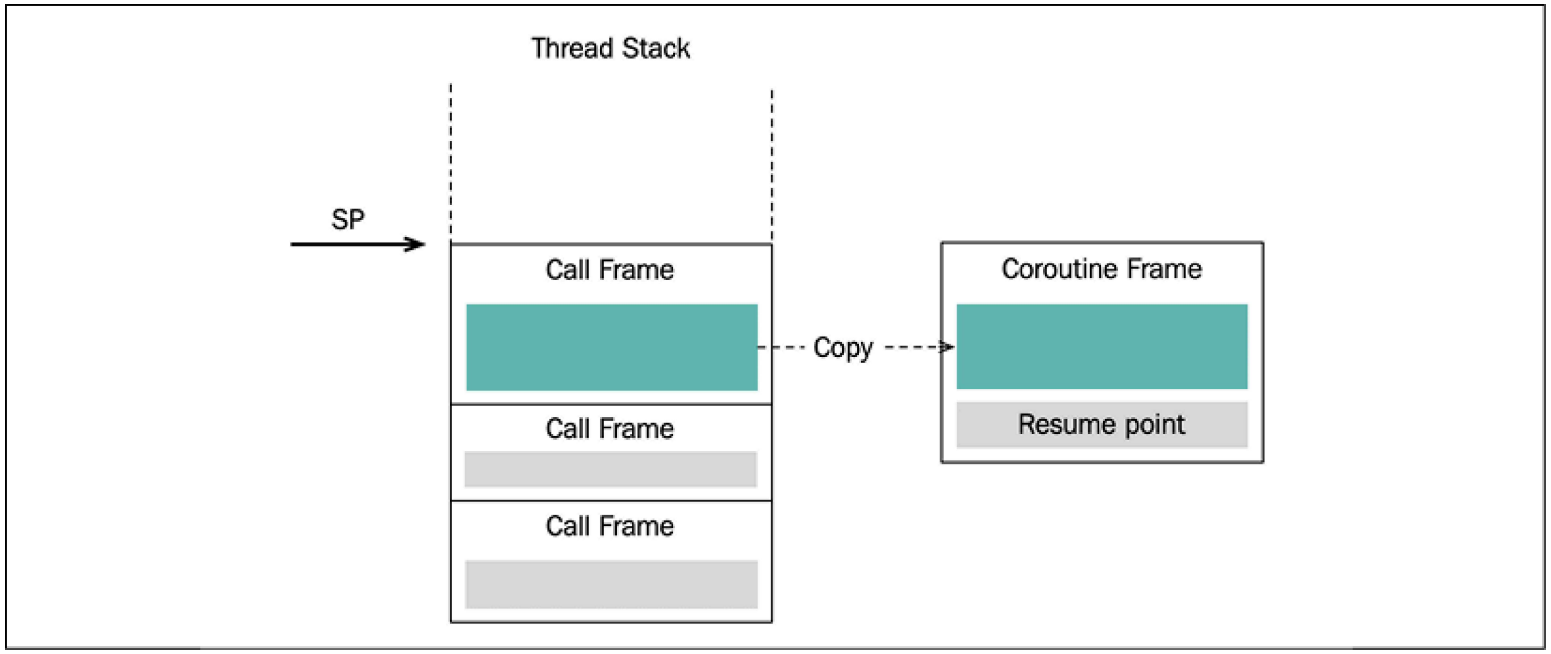

Figure 12.6: A stackless coroutine has a separate coroutine frame (usually on the heap) that contains the state necessary for resuming the coroutine

When a coroutine initially suspends, the stack frame for the coroutine is popped from the stack, but the coroutine frame continues to live on. A memory address (handle/pointer) to the coroutine frame is returned back to the caller:

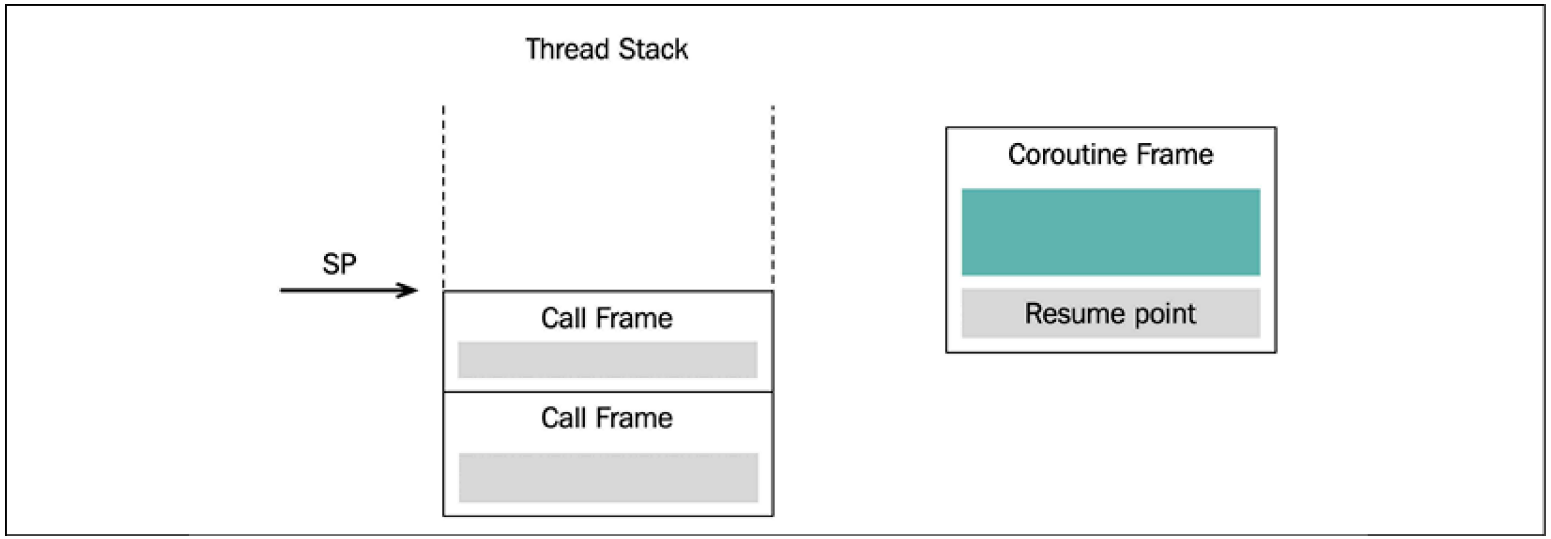

Figure 12.7: A suspended coroutine. The coroutine frame contains all the information required for resuming the coroutine.

To resume a coroutine, the caller uses the handle it received earlier and calls a resume function and passes the coroutine handle as a parameter. The resume function uses the suspend/resume point stored in the coroutine frame to continue executing the coroutine. The call to the resume function is also an ordinary function call that will generate a stack frame as illustrated in the following diagram:

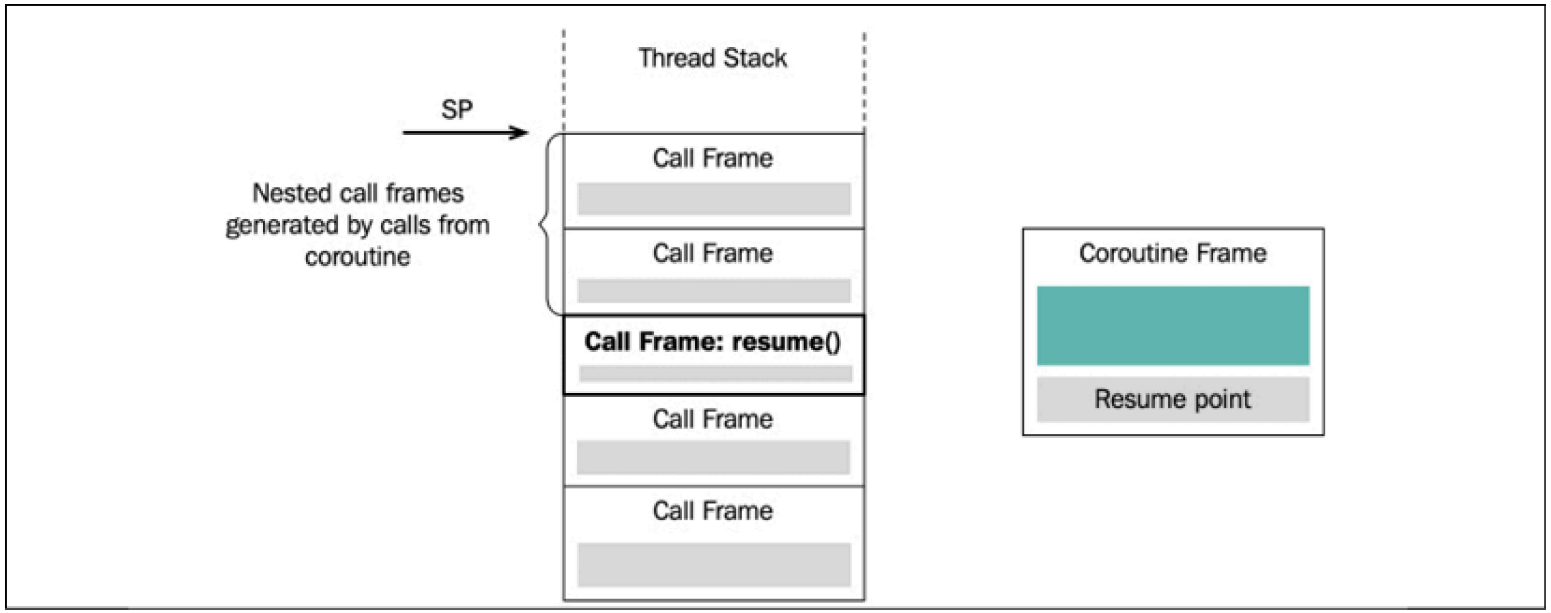

Figure 12.8: Resuming a coroutine creates a new call frame for the resume call. The resume function uses the handle to the coroutine state to resume from the right suspend point.

Finally, when a coroutine returns, it is usually suspended and eventually deallocated. The state of the stack is shown in the following diagram:

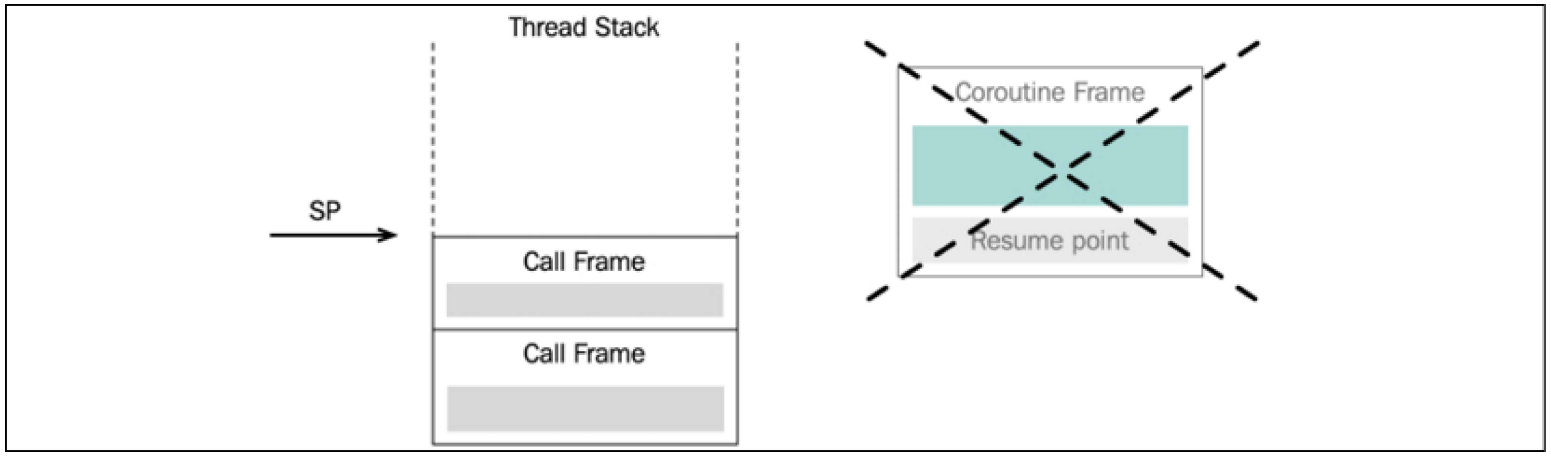

Figure 12.9: The coroutine frame is deallocated when it returns

An important consequence of not having a separate side stack per coroutine invocation is that when a stackless coroutine is suspended, it cannot have any nested call frames left on the stack. Remember, when the control is transferred back to the caller, the caller's call frame must be on the top of the stack.

As a final note, it should also be mentioned that the memory needed for the coroutine frame could be allocated *within* the call frame of the caller under some circumstances. We will discuss that in more detail when looking at C++20 coroutines.

Stackless versus stackful coroutines

As stated in the previous section, stackless coroutines use the stack of the currently running thread to handle nested function calls. The effect of this is that a stackless coroutine can never suspend from a nested call frame.

Stackful coroutines are sometimes called **fibers**, and in the programming language Go, they are called **goroutines**. Stackful coroutines remind us of threads, where each thread manages its own stack. There are two big differences between stackful coroutines (or fibers) and OS threads, though:

- OS threads are scheduled by the kernel and switching between two threads is a kernel mode operation.
- Most OSes switch OS threads **preemptively** (the thread is interrupted by the scheduler), whereas a switch between two fibers happens **cooperatively**. A running fiber keeps running until it passes control over to some manager that can then schedule another fiber.

There is also a category of threads called **user-level threads** or **green threads**. These are lightweight threads that don't involve kernel mode switching (because they run in user mode and are therefore unknown to the kernel). Fibers are one example of user-level threads. But it is also possible for user-level threads to be scheduled preemptively by a user library or by a virtual machine. Java threads are one example of preemptive user-level threads.

Stackless coroutines also allow us to write and compose multiple concurrently running tasks but without the need for an individual side stack per flow. Stackless coroutines and state machines are tightly related. It's possible to transform a state machine into a coroutine and vice versa. Why is this useful to know? Firstly, it gives you a better understanding of what stackless coroutines are. Secondly, if you are already good at identifying problems that can be solved using state machines, you can more easily see where coroutines might fit in as an appropriate solution. State machines are very general abstractions and can be applied to a great variety of problems. However, some areas where state machines are usually applied are parsing, gesture recognition, and I/O multiplexing, to mention a few. These are all areas where stackless coroutine can really shine both in terms of expressiveness and performance.

Performance cost

Coroutines are an abstraction that allow us to write lazy evaluated code and asynchronous programs in a clear and concise way. But there is a performance cost related to creating and destroying coroutines as well as suspending and resuming coroutines. When comparing the performance cost of stackless and stackful coroutines, two main aspects need to be addressed: *memory footprint* and *context switching*.

Memory footprint

Stackful coroutines need a separate call stack in order to handle suspension from within nested call frames. When calling a coroutine, we therefore need to dynamically allocate a chunk of memory for this new side stack. This immediately raises the question: how big a stack do we need to allocate? Unless we have some policy regarding how much stack a coroutine and its nested call frames can consume, we probably need to have a stack of approximately the same size as a normal call stack of a thread.

Some implementations have experimented with a segmented stack, which would allow the stack to grow if necessary. Another alternative is to start with a small contiguous stack and then copy the stack to a bigger newly allocated memory region when needed (similar to how `std::vector` grows). The coroutine implementation in Go (goroutines) has switched from using a segmented stack to a dynamically growing contiguous stack.

Stackless coroutines do not need to allocate memory for a separate side stack. Instead, they need a single allocation for storing each coroutine frame in order to support suspend and resume. This allocation happens when the coroutine is called (but not on suspend/resume). The call frame is deallocated when the coroutine returns.

In summary, stackful coroutines demand a big initial memory allocation for the coroutine frame and the side stack, or need to support a growing stack. Stackless coroutines only need to allocate memory for the coroutine frame. The memory footprint of calling a coroutine can be summarized as follows:

- Stackless: Coroutine frame
- Stackful: Coroutine frame + call stack

The next aspect of performance cost relates to suspending and resuming coroutines.

Context switching

Context switching can occur at different levels. In general, a context switch happens when we need the CPU to switch between two or many ongoing tasks. The task that is about to be paused needs to save the entire state of the CPU so that it can be restored at a later stage.

Switching between different processes and OS threads are fairly expensive operations that involve system calls, requiring the CPU to enter kernel mode. Memory caches are invalidated and, for process switching, the tables that contain the mappings between the virtual memory and physical memory need to be replaced.

Suspending and resuming coroutines is also a kind of context switch because we are switching between multiple concurrent flows. Switching between coroutines is substantially faster than switching between processes and OS threads, partly because it doesn't involve any system calls that require the CPU to run in kernel mode.

However, there is still a difference when switching between stackful coroutines and switching between stackless coroutines. The relative runtime performance of the context switches of stackful versus stackless coroutines can depend on the call patterns. But, in general, a stackful coroutine has a more expensive context switch operation since it has more information to save and restore during suspend and resume compared to a stackless coroutine. Resuming a stackless coroutine is comparable to a normal function call.

The stackless versus stackful debate has been going on in the C++ community for quite a few years now and I will do my best to stay away from the debate by concluding that they both have valid use cases—some use cases will favor stackful coroutines and other use cases will favor stackless coroutines.

This section took a little detour for the purpose of you having a better understanding of how coroutines execute and perform. Let's have a short recap of what you have learned.

What you have learned so far

Coroutines are functions that can be suspended and resumed. An ordinary function does not have this ability, which makes it possible to remove the call frame of a function that returns. However, a coroutine that is suspended needs to keep the call frame alive to be able to restore the state of the coroutine once it gets resumed. Coroutines are more powerful than subroutines and involve more bookkeeping in the generated machine code. However, thanks to the close relationship between coroutines and ordinary functions, the compilers of today are very good at optimizing stackless coroutines.

Stackful coroutines can be seen as non-preemptive user-level threads, whereas stackless coroutines offer a way to write state machines in a direct imperative fashion using the keywords `await` and `yield` to specify the suspend points.

After this introduction to the general coroutine abstraction, it's now time to understand how stackless coroutines are implemented in C++.

Coroutines in C++

The coroutines added to C++20 are stackless coroutines. There are options to use stackful coroutines in C++ as well by using third-party libraries. The most well-known cross-platform library is Boost.Fiber. C++20 stackless coroutines introduce new language constructs, while Boost.Fiber is a library that can be used with C++11 and onward. We will not discuss stackful coroutines any further in this book but will instead focus on the stackless coroutines that have been standardized in C++20.

The stackless coroutines in C++20 were designed with the following goals:

- Scalable in the sense that they add very little memory overhead. This makes it possible to have many more coroutines alive compared to the possible number of threads or stackful coroutines alive.
- Efficient context switching, which means that suspending and resuming a coroutine should be about as cheap as an ordinary function call.
- Highly flexible. C++ coroutines have more than 15 customization points, which gives application developers and library writers a lot of freedom to configure and shape coroutines as they like. Decisions about how coroutines are supposed to work can be determined by us developers rather than being hardcoded in a language specification. One example is whether a coroutine should be suspended directly after being called or continue executing to the first explicit suspend point. Such questions are usually hard-coded in other languages, but in C++ we can customize this behavior using customization points.
- Do not require C++ exceptions to handle errors. This means that you can use coroutines in environments where exceptions are turned off. Remember that coroutines are a low-level feature comparable to ordinary functions, which can be highly useful in embedded environments and systems with real-time requirements.

With these goals in mind, it's probably not a surprise that C++ coroutines can be a bit complicated to grasp at first.

What's included in standard C++ (and what's not)?

Some C++ features are pure library features (such as the Ranges library) whereas other features are pure language features (such as type inference with the help of the `auto` keyword). However, some features require additions to both the core language and the standard library. C++ coroutines are one of those features; they introduce new keywords to the language, but also add new types to the standard library.

On the language side, to recap, we have the following keywords related to coroutines:

- `co_await` : An operator that suspends the current coroutine
- `co_yield` : Returns a value to the caller and suspends the coroutine
- `co_return` : Completes the execution of a coroutine and can, optionally, return a value

On the library side, there is a new `<coroutine>` header including the following:

- `std::coroutine_handle` : A template class that refers to the coroutine state, enabling the suspending and resuming of the coroutine
- `std::suspend_never` : A trivial awaitable type that never suspends
- `std::suspend_always` : A trivial awaitable type that always suspends
- `std::coroutine_traits` : Used to define the promise type of a coroutine

The library types that comes with C++20 are the absolute minimum. For example, the infrastructure for communicating between the coroutine and the caller is not part of the C++ standard. Some of the types and functions that we need in order to use coroutines effectively in our application code have already been suggested in new C++ proposals, for example the template classes `task` and `generator` and the functions `sync_wait()` and `when_all()` . The library part of C++ coroutines will most likely be complemented in C++23.

In this book, I will provide some simplified types for filling this gap instead of using a third-party library. By implementing those types, you will get a deep understanding of how C++ coroutines work. However, designing robust library components that can be used with coroutines is hard to get right without introducing lifetime issues. So, if you are planning to use coroutines in your current project, using a third-party library may be a better alternative to implementing them from scratch. At the time of writing, the **CppCoro** library is the de facto standard for these general-purpose primitives. The library was created by Lewis Baker and is available at **https://github.com/lewissbaker/cppcoro**.

What makes a C++ function a coroutine?

A C++ function is a coroutine if it contains any of the keywords `co_await`, `co_yield`, or `co_return`. In addition, the compiler puts special requirements on the return type of a coroutine. But, nevertheless, we need to inspect the definition (the body) and not only the declaration to know whether we are facing a coroutine or an ordinary function. This means that the caller of a coroutine doesn't need to know whether it calls a coroutine or an ordinary function.

Compared to ordinary functions, a coroutine also has the following restrictions:

- A coroutine cannot use variadic arguments like `f(const char*...)`
- A coroutine cannot return `auto` or a concept type: `auto f()`
- A coroutine cannot be declared `constexpr`
- Constructors and destructors cannot be coroutines
- The `main()` function cannot be a coroutine

Once the compiler has decided that a function is a coroutine, it associates the coroutine with a number of types for making the coroutine machinery work. The following diagram highlights the different compo-

nents that are involved when a *caller* uses a *coroutine*:

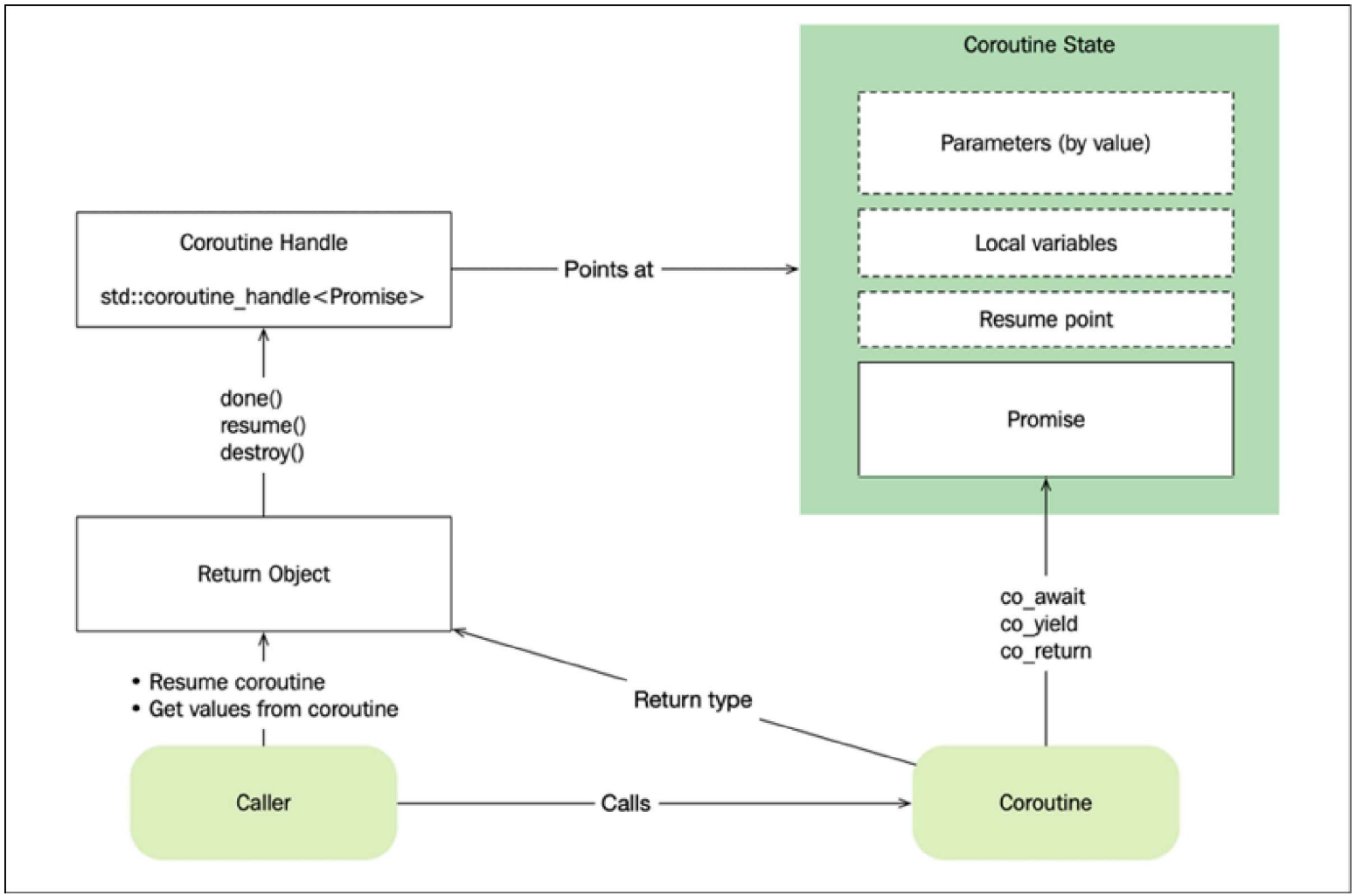

Figure 12.10: Relationship between a coroutine and its caller

The caller and the coroutine are the actual functions we will normally implement in our application code.

The **Return Object** is the type that the coroutine returns and is typically a general class template designed for some specific use case, for example, *generators* or *asynchronous tasks*. The *caller* interacts with the return object to resume the coroutine and to get values emitted from the coroutine. The return object usually delegates all its calls to the coroutine handle.

The **Coroutine Handle** is a non-owning handle to the **Coroutine State**. Through the coroutine handle we can resume and destroy the coroutine state.

The *coroutine state* is what I have previously referred to as the coroutine frame. It's an opaque object, which means that we don't know its size and we cannot access it in any other way than through the handle. The coroutine state stores everything necessary in order to resume the coroutine where it was last suspended. The coroutine state also contains the **Promise**.

The promise object is what the coroutine itself communicates with indirectly through the keywords `co_await`, `co_yield`, and `co_return`. If values or errors are submitted from the coroutine, they will first reach the promise object. The promise object acts like a channel between the coroutine and the caller, but neither of them have direct access to the promise.

Admittedly, this can look pretty dense at first sight. A complete but minimal example will help you understand the different parts a little better.

A minimal but complete example

Let's start with a minimal example for the purpose of understanding how coroutines work. Firstly, we implement a small *coroutine* that is suspended and resumed before it returns:

```
auto coroutine() -> Resumable {    // Initial suspend
  std::cout << "3 ";
  co_await std::suspend_always{};  // Suspend (explicit)
  std::cout << "5 ";
}                                  // Final suspend then return
```

Secondly, we create the *caller* of the coroutine. Pay attention to the output and the control flow of this program. Here it is:

```
int main() {
  std::cout << "1 ";
  auto resumable = coroutine(); // Create coroutine state
  std::cout << "2 ";
  resumable.resume();           // Resume
  std::cout << "4 ";
  resumable.resume();           // Resume
  std::cout << "6 ";
}                               // Destroy coroutine state
// Outputs: 1 2 3 4 5 6
```

Thirdly, the return object of the coroutine, `Resumable`, needs to be defined:

```
class Resumable {               // The return object
  struct Promise { /*...*/ };   // Nested class, see below
  std::coroutine_handle<Promise> h_;
  explicit Resumable(std::coroutine_handle<Promise> h) : h_{h} {}
```

```
public:
  using promise_type = Promise;
  Resumable(Resumable&& r) : h_{std::exchange(r.h_, {})} {}
  ~Resumable() { if (h_) { h_.destroy(); } }
  bool resume() {
    if (!h_.done()) { h_.resume(); }
    return !h_.done();
  }
};
```

Finally, the promise type is implemented as a nested class inside the `Resumable`, like this:

```
struct Promise {
  Resumable get_return_object() {
    using Handle = std::coroutine_handle<Promise>;
    return Resumable{Handle::from_promise(*this)};
  }
  auto initial_suspend() { return std::suspend_always{}; }
  auto final_suspend() noexcept { return std::suspend_always{}; }
  void return_void() {}
  void unhandled_exception() { std::terminate(); }
};
```

This example is minimal, but walks through a lot of things that are worth paying attention to and need to be understood:

- The function `coroutine()` is a coroutine because it contains the explicit suspend/resume point using `co_await`
- The coroutine doesn't yield any values but still needs to return a type (the `Resumable`) with certain constraints so that the caller can resume the coroutine
- We are using an *awaitable type* called `std::suspend_always`
- The `resume()` function of the `resumable` object resumes the coroutine from the point it was suspended
- The `Resumable` is the owner of the coroutine state. When the `Resumable` object is destructed, it destroys the coroutine using the `coroutine_handle`

The relationship between the caller, the coroutine, the coroutine handle, the promise, and the resumable is illustrated in the following diagram:

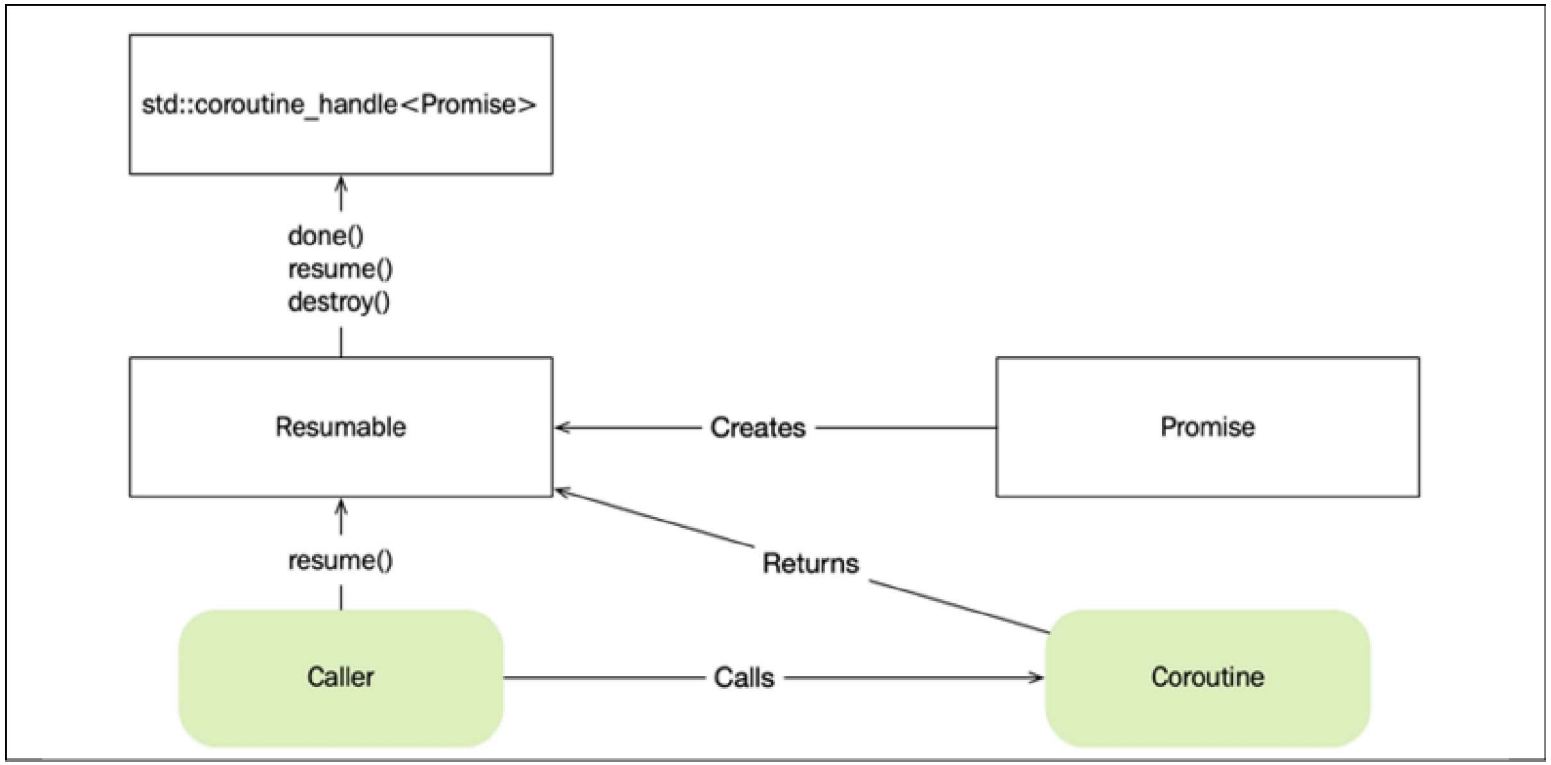

Figure 12.11: Relationship between the functions/coroutines and objects involved in the resumable example

Now it's time to look a little closer at each part. We'll begin with the `Resumable` type.

The coroutine return object

Our coroutine returns an object of type `Resumable`. This `Resumable` class is very simple. This is the object that the coroutine returns and which the caller can use in order to resume and destroy the coroutine. Here is the complete definition again for your convenience:

```
class Resumable {            // The return object
  struct Promise { /*...*/ };   // Nested class
  std::coroutine_handle<Promise> h_;
  explicit Resumable(std::coroutine_handle<Promise> h) : h_{h} {}
public:
  using promise_type = Promise;
  Resumable(Resumable&& r) : h_{std::exchange(r.h_, {})} {}
  ~Resumable() { if (h_) { h_.destroy(); } }
  bool resume() {
    if (!h_.done()) { h_.resume(); }
    return !h_.done();
  }
};
```

`Resumable` is a move-only type that is the owner of the coroutine handle (and therefore controls the lifetime of the coroutine). The move constructor ensures that the coroutine handle is cleared in the source object by using `std::exchange()`. When a `Resumable` object is destructed, it destroys the coroutine if it still owns it.

The `resume()` member function delegates the resume call to the coroutine handle if the coroutine is still alive.

Why do we need the member type alias `promise_type = Promise` inside `Resumable`? With each coroutine there is also an associated promise object. When the compiler sees a coroutine (by inspecting the body of a function), it needs to figure out the associated promise type. For that, the compiler uses the `std::coroutine_traits<T>` template, where `T` is the return type of your coroutine. You can provide a template specialization of `std::coroutine_traits<T>` or exploit the fact that the default implementation of

`std::coroutine_traits` will look for a `public` member type or alias named `promise_type` in the return type `T` of the coroutine. In our case, the `Resumable::promise_type` is an alias for `Promise`.

The promise type

The promise type controls the behavior of the coroutine. Again, here is the full definition reproduced for convenience:

```
struct Promise {
  auto get_return_object() { return Resumable{*this}; }
  auto initial_suspend() { return std::suspend_always{}; }
  auto final_suspend() noexcept { return std::suspend_always{}; }
  void return_void() {}
  void unhandled_exception() { std::terminate(); }
};
```

We should not call these functions directly; instead, the compiler inserts calls to the promise objects when it transforms a coroutine into machine code. If we don't provide these member functions, the compiler doesn't know how to generate code for us. You can think about the promise as a coroutine controller object that is responsible for:

- Producing the value returned from the invocation of the coroutine. This is handled by the function `get_return_object()`.
- Defining the behavior when the coroutine is created and before it gets destroyed by implementing the functions `initial_suspend()` and `final_supsend()`. In our `Promise` type, we say that the coroutine should be suspended at these points by returning `std::suspend_always` (see the next section).

- Customizing the behavior when the coroutine finally returns. If a coroutine uses a `co_return` with an expression that evaluates to a value of type `T`, the promise must define a member function named `return_value(T)`. Our coroutine returns no value, but the C++ standard requires us to provide the customization point called `return_void()`, which we leave empty here.
- Handling exceptions that are not handled inside the coroutine body. In the function `unhandled_exception()`, we simply call `std::terminate()`, but we will handle it more gracefully in later examples.

There are some final pieces of the code that require some more attention, namely the `co_await` expression and awaitable types.

Awaitable types

We added one explicit suspend point in our code using `co_await` and passed it an instance of the awaitable type, `std::suspend_always`. The implementation of `std::suspend_always` looks something like this:

```
struct std::suspend_always {
  constexpr bool await_ready() const noexcept { return false; }
  constexpr void await_suspend(coroutine_handle<>) const noexcept {}
  constexpr void await_resume() const noexcept {}
};
```

`std::suspend_always` is called a trivial awaitable type because it will always make a coroutine suspend by saying that it is never ready. There is another trivial awaitable type that always reports that it is ready, called `std::suspend_never`:

```
struct std::suspend_never {
  constexpr bool await_ready() const noexcept { return true; }
  constexpr void await_suspend(coroutine_handle<>) const noexcept {}
  constexpr void await_resume() const noexcept {}
};
```

We could create our own awaitable types, which we will cover in the next chapter, but for now we can manage with those two trivial standard types.

This completes the example. But we can do some more experimenting when we have the `Promise` and the `Resumable` types in place. Let's see what we can do with a started coroutine.

Passing our coroutine around

Once the `Resumable` object is created, we can pass it to other function and resume it from there. We can even pass the coroutine to another thread. The following example shows some of this flexibility:

```
auto coroutine() -> Resumable {
  std::cout << "c1 ";
  co_await std::suspend_always{};
  std::cout << "c2 ";
}
auto coro_factory() {          // Create and return a coroutine
  auto res = coroutine();
  return res;
}
int main() {
```

```
  auto r = coro_factory();
  r.resume();                // Resume from main
  auto t = std::jthread{[r = std::move(r)]() mutable {
   using namespace std::chrono_literals;
   std::this_thread::sleep_for(2s);
   r.resume();               // Resume from thread
  }};
}
```

The preceding example demonstrates that once we have called our coroutine and have got a handle to it, we can move it around just like any other moveable type. This ability to pass it to other threads is actually very useful in situations where we need to avoid the possible heap allocation of the coroutine state on a specific thread.

Allocating the coroutine state

The coroutine state, or the coroutine frame, is where the coroutine stores its state while it is suspended. The lifetime of the coroutine state starts when the coroutine is invoked by a call, and is destroyed when the coroutine executes a `co_return` statement (or the control flows off the end of the coroutine body), unless it was destroyed earlier through the coroutine handle.

The coroutine state is normally allocated on the heap. A separate heap allocation is inserted by the compiler. In some cases, though, this separate heap allocation can be elided by inlining the coroutine state into the frame of the caller (which could be an ordinary stack frame or another coroutine frame). Unfortunately, there is never any guarantee of this elision of the heap allocation.

For the compiler to be able to elide the heap allocation, the complete lifetime of the coroutine state must be strictly nested within the lifetime of the caller. In addition, the compiler needs to figure out the total size of the coroutine state and generally needs to have visibility of the body of the called coroutine so that parts of it can be inlined. Situations like virtual function calls, and calls to functions in other translation units or shared libraries, typically make this impossible. If the compiler is missing the information it needs, it will insert a heap allocation.

The heap allocation of the coroutine state is performed using `operator new`. It is possible to provide a custom class-level `operator new` on the promise type, which will then be used instead of global `operator new`. It's therefore possible to check whether the heap allocation was elided or not. And if it wasn't, we can find out how much memory is needed for the coroutine state. Here is an example using the `Promise` type we defined earlier:

```
struct Promise {
 /* Same as before ... */
 static void* operator new(std::size_t sz) {
  std::cout << "custom new for size " << sz << '\n';
  return ::operator new(sz);
 }
 static void operator delete(void* ptr) {
  std::cout << "custom delete called\n";
  ::operator delete(ptr);
 }
}
```

Another trick to verify that the heap allocations are completely elided for all coroutines using some specific promise type would be to declare `operator new` and `operator delete` but leave out their defini-

tions. If the compiler then inserts calls to these operators, the program will fail to link due to unresolved symbols.

Avoiding dangling references

The fact that a coroutine can be passed around in our code means that we need to be very careful about the lifetime of parameters we pass to a coroutine to avoid dangling references. The coroutine frame contains copies of the objects that normally live on the stack, such as local variables and parameters passed to the coroutine. If a coroutine accepts an argument by reference, the *reference* is copied, not the object. This means that we can easily end up with dangling references when following the usual guidelines for function parameters; that is, pass objects that are expensive to copy by reference to `const`.

Passing parameters to coroutines

The following coroutine uses a reference to a `const std::string`:

```
auto coroutine(const std::string& str) -> Resumable {
  std::cout << str;
  co_return;
}
```

Suppose we have a factory function that creates and returns the coroutine, like this:

```
auto coro_factory() {
  auto str = std::string{"ABC"};
  auto res = coroutine(str);
```

```
  return res;
}
```

And finally, a `main()` function that uses the coroutine:

```
int main() {
  auto coro = coro_factory();
  coro.resume();
}
```

This code exhibits undefined behavior as the `std::string` object containing the string `"ABC"` is no longer alive when the coroutine tries to access it. Hopefully, this doesn't come as a surprise to you. This problem is similar to having a lambda capture a variable by reference, and then passing the lambda to some other code without keeping the referenced object alive. A similar example can be achieved when passing around a lambda capturing variables by reference:

```
auto lambda_factory() {
  auto str = std::string{"ABC"};
  auto lambda = [&str]() {       // Capture str by reference
    std::cout << str;
  };
  return lambda;                 // Ops! str in lambda becomes
}                                // a dangling reference
int main() {
  auto f = lambda_factory();
```

```
  f();                    // Undefined behavior
}
```

As you can see, the same problem can happen with lambdas. In *Chapter 2*, *Essential C++ Techniques*, I warned you about capturing references with lambdas, and it is usually better to avoid this by capturing by value instead.

The solution to avoid dangling references with coroutines is similar: avoid passing parameters by reference when using coroutines. Instead, use pass by value, and the entire parameter object will be placed safely in the coroutine frame:

```
auto coroutine(std::string str) -> Resumable {  // OK, by value!
  std::cout << str;
  co_return;
}
auto coro_factory() {
  auto str = std::string{"ABC"};
  auto res = coroutine(str);
  return res;
}
int main() {
  auto coro = coro_factory();
  coro.resume();                  // OK!
}
```

Parameters are an important and common source of lifetime issues when using coroutines, but they are not the only source. Now we will explore some other pitfalls related to coroutines and dangling

references.

Member functions that are coroutines

A member function can also be a coroutine. For example, there is nothing that stops us from using `co_await` inside a member function, as in the following example:

```
struct Widget {
auto coroutine() -> Resumable {      // A member function
   std::cout << i_++ << " ";       // Access data member
   co_await std::suspend_always{};
   std::cout << i_++ << " ";
 }
 int i_{};
};
int main() {
 auto w = Widget{99};
 auto coro = w.coroutine();
 coro.resume();
 coro.resume();
}
// Prints: 99 100
```

It's important to understand that it's the responsibility of the caller of `coroutine()` (in this case, `main()`) to ensure that the `Widget` object, `w`, is kept alive during the entire lifetime of the coroutine. The coroutine is accessing data members from the object it belongs to, but the `Widget` object itself is *not* kept

alive by the coroutine. This can easily become a problem if we pass the coroutine to some other part of the program.

Let's say we are using some coroutine factory function as demonstrated earlier, but instead return a member function coroutine:

```
auto widget_coro_factory() {     // Create and return a coroutine
  auto w = Widget{};
  auto coro = w.coroutine();
  return coro;
}                                // Object w destructs here
int main() {
  auto r = widget_coro_factory();
  r.resume();                    // Undefined behavior
  r.resume();
}
```

This code exhibits undefined behavior because we now have a dangling reference from the coroutine to the `Widget` object created and destructed in the `widget_coro_factory()` function. In other words, we end up with two objects with distinct lifetimes, whereas one of the objects references the other but without any explicit ownership.

Lambdas that are coroutines

Not only member functions can become coroutines. It's also possible to create coroutines using lambda expressions by inserting `co_await`, `co_return`, and/or `co_yield` in the body of a lambda.

Coroutine lambdas can be a little extra tricky to deal with. One way to understand the most common life-time issue with coroutine lambdas is to think about function objects. Recall from *Chapter 2*, *Essential C++ Techniques*, that a lambda expression is transformed into a function object by the compiler. The type of this object is a class with the call operator implemented. Now, let's say we use `co_return` inside the body of a lambda; it means that the call operator `operator()()` becomes a coroutine.

Consider the following code using a lambda:

```
auto lambda = [](int i) -> Resumable {
  std::cout << i;
  co_return;            // Make it a coroutine
};
auto coro = lambda(42);   // Call, creates the coroutine frame
coro.resume();          // Outputs: 42
```

The type that the lambda corresponds to looks something like this:

```
struct LambdaType {
  auto operator()(int i) -> Resumable {  // Member function
    std::cout << i;                  // Body
    co_return;
  }
};
auto lambda = LambdaType{};
auto coro = lambda(42);
coro.resume();
```

The important thing to note here is that the actual coroutine is a *member function*, namely the call operator `operator()()`. The previous section already demonstrated the pitfalls of having coroutine member functions: we need to keep the object alive during the lifetime of the coroutine. In the preceding example, it means we need to keep the function object named `lambda` alive as long as the coroutine frame is alive.

Some usages of lambdas make it really easy to accidentally destruct the function object before the coroutine frame is destroyed. For example, by using an *immediately invoked lambda*, we can easily get into trouble:

```
auto coro = [i = 0]() mutable -> Resumable {
  std::cout << i++;
  co_await std::suspend_always{};
  std::cout << i++;
}();            // Invoke lambda immediately
coro.resume();  // Undefined behavior! Function object
coro.resume();  // already destructed
```

This code looks innocent; the lambda is not capturing anything by reference. However, the function object created by the lambda expression is a temporary object that will be destructed once it has been invoked and the coroutine captures a reference to it. When the coroutine is resumed, the program will likely crash or produce garbage.

Again, a way to understand this better is to transform the lambda to an ordinary class with `operator()` defined:

```
struct LambdaType {
  int i{0};
  auto operator()() -> Resumable {
    std::cout << i++;
    co_await std::suspend_always{};
    std::cout << i++;
  }
};
auto coro = LambdaType{}(); // Invoke operator() on temporary object
coro.resume();              // Ops! Undefined behavior
```

Now you can see that this is very similar to the case where we had a member function that was a coroutine. The function object is not kept alive by the coroutine frame.

Guidelines to prevent dangling references

Unless you have good reasons for accepting arguments by reference, choose to accept arguments by value if you are writing a coroutine. The coroutine frame will then keep a full copy of the object you pass to it, and the object is guaranteed to live as long as the coroutine frame.

If you are using lambdas or member functions that are coroutines, pay special attention to the lifetime of the object that the coroutine belongs to. Remember that the object (or function object) is *not* stored in the coroutine frame. It's the responsibility of the caller of the coroutine to keep it alive.

Handling errors

There are different ways to transfer errors from a coroutine back to the part of the code that called it or resumed it. We are not forced to use exceptions for signaling errors. Instead, we can customize error handling as we want.

A coroutine can pass an error back to the client using the coroutine by either throwing an exception or returning an error code when the client gets a value back from the coroutine (when the coroutine yields or returns).

If we are using exceptions and an exception is propagated out of the body of the coroutine, the function `unhandled_exception()` of the promise object is called. This call happens inside a catch block inserted by the compiler, so that it is possible to use `std::current_exception()` to get hold of the exception that was thrown. The result from `std::current_exception()` can then be stored in the coroutine as a `std::exception_ptr` and rethrown later on. You will see examples of this in the next chapter when using asynchronous coroutines.

Customization points

You have already seen many customization points, and I think a valid question is: why so many customization points?

- **Generality**: The customization points make it possible to use coroutines in various ways. There are very few assumptions about how to use the C++ coroutines. Library writers can customize the behavior of `co_await`, `co_yield`, and `co_return`.
- **Efficiency**: Some of the customization points are there for enabling possible optimizations depending on use cases. One example is `await_ready()`, which can return `true` to avoid an unnecessary suspension if a value is already computed.

It should also be said that we are exposed to these customization points because the C++ standard doesn't provide any types (except for the `std::coroutine_handle`) to communicate with a coroutine. Once they are in place, we can reuse those types and not worry too much about some of those customization points. Nevertheless, knowing the customization points is valuable in order to fully understand how to use C++ coroutines efficiently.

Generators

A generator is a type of coroutine that yields values back to its caller. For example, at the beginning of this chapter, I demonstrated how the generator `iota()` yielded increasing integer values. By implementing a general-purpose generator type that can act as an iterator, we can simplify the work of implementing iterators that are compatible with range-based `for`-loops, standard library algorithms, and ranges. Once we have a generator template class in place, we can reuse it.

So far in this book, you have mostly seen iterators in the context of accessing container elements and when using standard library algorithms. However, an iterator does not have to be tied to a container. It's possible to write iterators that produce values.

Implementing a generator

The generator we are about to implement is based on the generator from the CppCoro library. The generator template is intended to be used as a return type for coroutines that produces a sequence of values. It should be possible to use objects of this type together with a range-based `for` - loop and standard algorithms that accept iterators and ranges. To make this possible, we will implement three components:

- The `Generator`, which is the return object

- The `Promise`, which acts as the coroutine controller
- The `Iterator`, which is the interface between the client and the `Promise`

These three types are tightly coupled and the relationships between them and the coroutine state are presented in the following diagram:

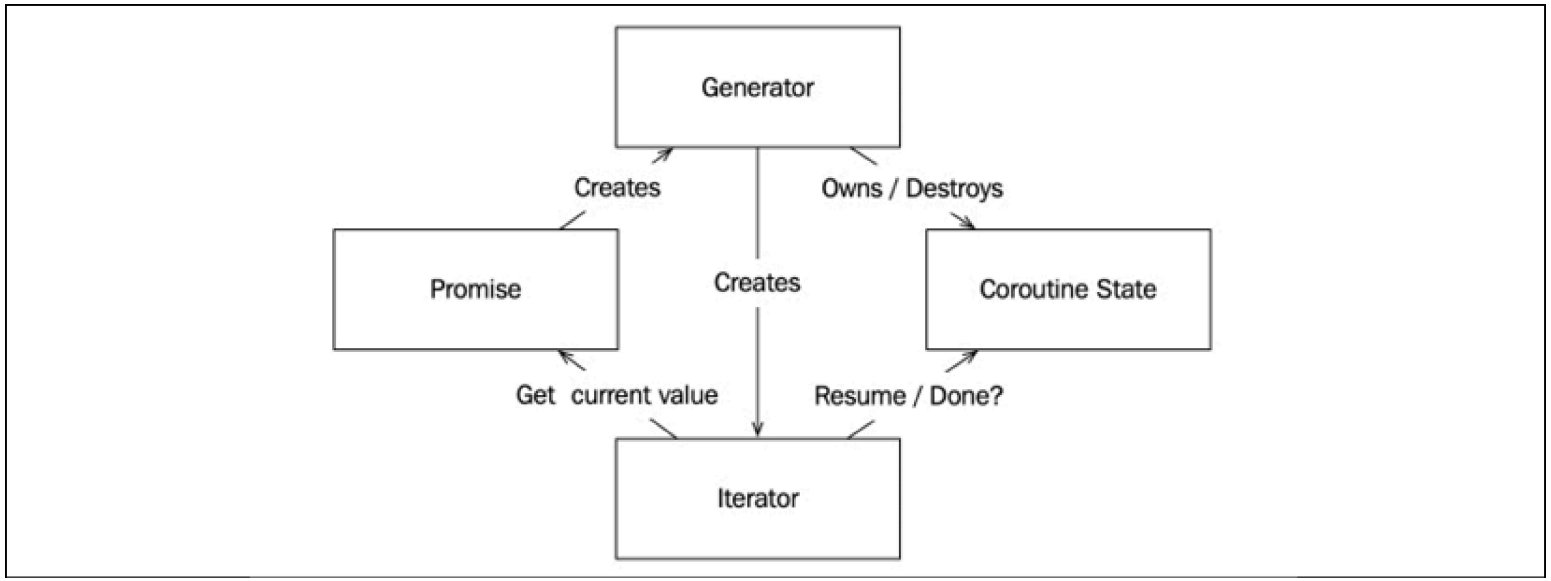

Figure 12.12: The relationships between the Iterator, Generator, Promise, and the coroutine state

The return object, in this case the `Generator` class, is tightly coupled with the `Promise` type; the `Promise` type is responsible for creating the `Generator` object, and the `Generator` type is responsible for exposing the correct `promise_type` to the compiler. Here is the implementation of `Generator`:

```
template <typename T>
class Generator {
```

```
  struct Promise { /* ... */ };   // See below
  struct Sentinel {};
  struct Iterator { /* ... */ };  // See below

  std::coroutine_handle<Promise> h_;
  explicit Generator(std::coroutine_handle<Promise> h) : h_{h} {}
public:
  using promise_type = Promise;
  Generator(Generator&& g) : h_(std::exchange(g.h_, {})) {}
  ~Generator() { if (h_) { h_.destroy();  } }
  auto begin() {
    h_.resume();
    return Iterator{h_};
  }
  auto end() { return Sentinel{}; }
};
```

The implementation of `Promise` and `Iterator` will follow soon. The `Generator` is not that different from the `Resumable` class we defined earlier. The `Generator` is the return object of the coroutine and the owner of the `std::coroutine_handle`. The generator is a moveable type. When being moved, the coroutine handle is transferred to the newly constructed `Generator` object. When a generator that owns a coroutine handle is destructed, it destroys the coroutine state by calling `destroy` on the coroutine handle.

The `begin()` and `end()` functions make it possible to use this generator in range-based `for`-loops and algorithms that accept ranges. The `Sentinel` type is empty—it's a dummy type—and the `Sentinel` instance is there to be able to pass something to the comparison operators of the `Iterator` class. The implementation of the `Iterator` looks like this:

```
struct Iterator {
  using iterator_category = std::input_iterator_tag;
  using value_type = T;
  using difference_type = ptrdiff_t;
  using pointer = T*;
  using reference = T&;

  std::coroutine_handle<Promise> h_;  // Data member

  Iterator& operator++() {
    h_.resume();
    return *this;
  }
  void operator++(int) { (void)operator++(); }
  T operator*() const { return h_.promise().value_; }
  T* operator->() const { return std::addressof(operator*()); }
  bool operator==(Sentinel) const { return h_.done(); }
};
```

The iterator needs to store the coroutine handle in a data member so that it can delegate the calls to the coroutine handle and the promise object:

- When the iterator is dereferenced, it returns the current value held by the promise
- When the iterator is incremented, it resumes the coroutine
- When the iterator is compared with the sentinel value, the iterator ignores the sentinel and delegates the call to the coroutine handle, which knows whether there are more elements to be generated

Now there is only the `Promise` type left for us to implement. The complete definition of `Promise` looks like this:

```
struct Promise {
  T value_;
  auto get_return_object() -> Generator {
    using Handle = std::coroutine_handle<Promise>;
    return Generator{Handle::from_promise(*this)};
  }
  auto initial_suspend() { return std::suspend_always{}; }
  auto final_suspend() noexcept { return std::suspend_always{}; }
  void return_void() {}
  void unhandled_exception() { throw; }
  auto yield_value(T&& value) {
    value_ = std::move(value);
    return std::suspend_always{};
  }
  auto yield_value(const T& value) {
    value_ = value;
    return std::suspend_always{};
  }
};
```

The promise object for our generator is responsible for:

- Creating the `Generator` object
- Defining the behavior when the initial and final suspend points are reached
- Keeping track of the last value that was yielded from the coroutine

- Handling exceptions thrown by the coroutine body

That's it! We now have all the pieces in place. A coroutine that returns some `Generator<T>` type can now yield values lazily using `co_yield`. The caller of the coroutine interacts with the `Generator` and `Iterator` objects to retrieve values. The interaction between the objects is illustrated next:

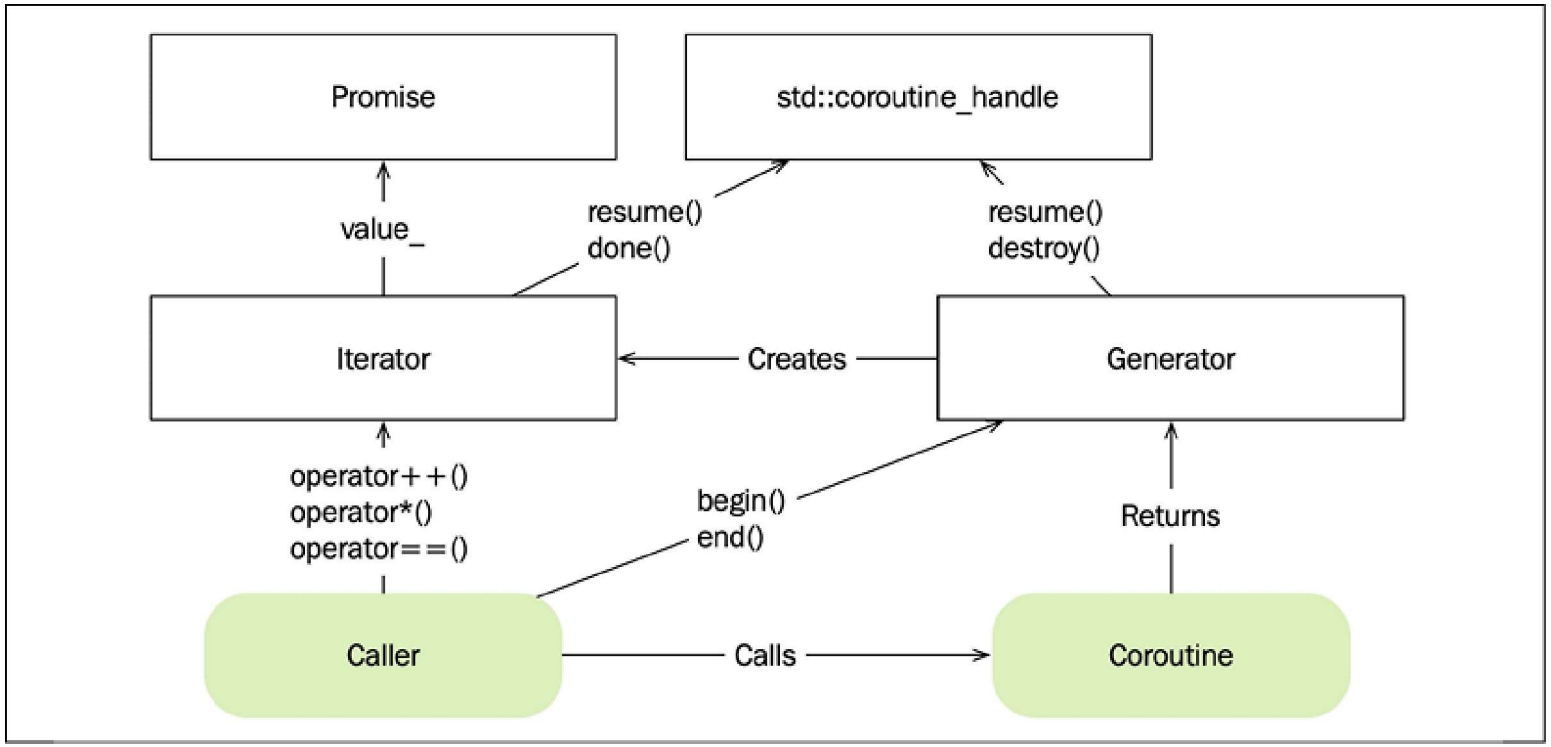

Figure 12.13: The caller communicates with the Generator and Iterator objects to retrieve values from the coroutine

Now, let's see how we can use the new `Generator` template and how it can simplify the implementation of various iterators.

Using the Generator class

This example is inspired from the talk *C++ Coroutines: Under the covers*, by Gor Nishanov at CppCon 2016 (**https://sched.co/7nKt**). It clearly demonstrates how we can benefit from the generator types we just implemented. Small composable generators can now be implemented like this:

```
template <typename T>
auto seq() -> Generator<T> {
  for (T i = {};; ++i) {
    co_yield i;
  }
}
template <typename T>
auto take_until(Generator<T>& gen, T value) -> Generator<T> {
  for (auto&& v : gen) {
    if (v == value) {
      co_return;
    }
    co_yield v;
  }
}
template <typename T>
auto add(Generator<T>& gen, T adder) -> Generator<T> {
  for (auto&& v : gen) {
    co_yield v + adder;
  }
}
```

A small usage example demonstrates that we can pass our generators to range-based `for`-loops:

```
int main() {
  auto s = seq<int>();
  auto t = take_until<int>(s, 10);
  auto a = add<int>(t, 3);
  int sum = 0;
  for (auto&& v : a) {
     sum += v;
  }
  return sum; // returns 75
}
```

The generators are lazily evaluated. No values are produced until the program reaches the `for`-loop, which pulls the values out from the chain of generators.

Another interesting aspect of this program is that when I compile it using Clang 10 with optimizations turned on, the assembly code for the *entire* program looks like this:

```
main:  # @main
mov  eax, 75
ret
```

Amazing! The program simply defines a main function that returns the value `75`. In other words, the compiler optimizer has been able to completely evaluate the chain of generators at compile time and come up with the single value `75`.

Our `Generator` class can also be used with range algorithms. In the following example we use the algorithm `includes()` to see if the sequence `{5,6,7}` is a subrange of the numbers produced by the generator:

```
int main() {
  auto s = seq<int>();                    // Same as before
  auto t = take_until<int>(s, 10);
  auto a = add<int>(t, 3);
  const auto v = std::vector{5, 6, 7};
  auto is_subrange = std::ranges::includes(a, v); // True
}
```

With the `Generator` template implemented, we can reuse it for all sorts of generator functions. We have implemented a general and highly useful library component that application code can benefit from in a great many places when building lazy generators.

Solving generator problems

I will now present a small problem and we will try to solve it using different techniques for the purpose of understanding which programming idioms we can potentially replace with generators. We are about to write a small utility for generating linearly spaced sequences between a start value and a stop value.

If you have been using MATLAB/Octave or Python NumPy, you might recognize this way of generating evenly (linearly) spaced numbers using a function called `linspace()`. It's a handy utility that can be used in various contexts with arbitrary ranges.

We will call our generator `lin_space()`. Here is a usage example of generating five equally spaced values between `2.0` and `3.0`:

```
for (auto v: lin_space(2.0f, 3.0f, 5)) {
  std::cout << v << ", ";
}
// Prints: 2.0, 2.25, 2.5, 2.75, 3.0,
```

When generating floating-point values, we have to be a little bit cautious because we cannot simply compute the size of each step (0.25 in the preceding example) and accumulate it, since the step size might not be possible to represent exactly using a floating-point data type. The possible rounding error will add up at each iteration and eventually we may end up with completely nonsensical values. What we instead need to do is to calculate a number between the start and stop value at a specific increment using linear interpolation.

C++20 added a handy utility to `<cmath>` called `std::lerp()`, which computes the linear interpolation between two values with a specified amount. In our case, the amount will be a value between 0.0 and 1.0; an amount of 0 returns the `start` value and a value of 1.0 returns the `stop` value. Here are a few examples of using `std::lerp()`:

```
auto start = -1.0;
auto stop = 1.0;
std::lerp(start, stop, 0.0);   // -1.0
std::lerp(start, stop, 0.5);   //  0.0
std::lerp(start, stop, 1.0);   //  1.0
```

The `lin_space()` functions we are about to write will all use the following small utility function template:

```cpp
template <typename T>
auto lin_value(T start, T stop, size_t index, size_t n) {
  assert(n > 1 && index < n);
  const auto amount = static_cast<T>(index) / (n - 1);
  const auto v = std::lerp(start, stop, amount);   // C++20
  return v;
}
```

The function returns a value in the linear sequence in the range [`start` , `stop`]. The `index` parameter is the current number in the sequence of the `n` total numbers we are about to generate.

With the `lin_value()` helper in place, we can now easily implement the `lin_space()` generator. Before seeing a solution using a coroutine, we will examine other common techniques. The sections to follow will explore the following different approaches when implementing `lin_space()` :

- Eagerly generate and return all values
- Using a callback (lazy)
- Using a custom iterator (lazy)
- Using the Ranges library (lazy)
- Using coroutines with our `Generator` class (lazy)

For each example, there will be a short reflection of the strengths and weaknesses of each approach.

An eager linear range

We'll begin by implementing a simple eager version that computes all the values in the range and returns a vector with all values:

```
template <typename T>
auto lin_space(T start, T stop, size_t n) {
  auto v = std::vector<T>{};
  for (auto i = 0u; i < n; ++i)
    v.push_back(lin_value(start, stop, i, n));
  return v;
}
```

Since this version returns a standard container, it's possible to use the return value with range-based `for` -loops and other standard algorithms:

```
for (auto v : lin_space(2.0, 3.0, 5)) {
  std::cout << v << ", ";
}
// Prints: 2, 2.25, 2.5, 2.75, 3,
```

This version is straightforward, and fairly easy to read. The downside is that we need to allocate a vector and fill it with *all* values, although the caller is not necessarily interested in all values. This version also lacks composability as there is no way to filter out elements in the middle without first generating all values.

Now let's try to implement a lazy version of the `lin_space()` generator.

A lazy version using a callback

In *Chapter 10, Proxy Objects and Lazy Evaluation*, we concluded that lazy evaluation can be accomplished by using callback functions. The lazy version we will implement will be based on passing a callback to `lin_space()` and invoking the callback function when emitting values:

```
template <typename T, typename F>
requires std::invocable<F&, const T&>          // C++20
void lin_space(T start, T stop, std::size_t n, F&& f) {
 for (auto i = 0u; i < n; ++i) {
  const auto y = lin_value(start, stop, i, n);
  f(y);
 }
}
```

If we want to print the values produced by the generator, we can call this function like this:

```
auto print = [](auto v) { std::cout << v << ", "; };
lin_space(-1.f, 1.f, 5, print);
// Prints: -1, -0.5, 0, 0.5, 1,
```

The iteration now take places within the `lin_space()` function. There is no way to cancel the generator, but with some changes we could have the callback function return a `bool` to indicate whether it wants more elements to be generated.

This approach works but is not very elegant. The problem with this design becomes more apparent when trying to compose generators. If we wanted to add a filter that would select some special values, we would end up having nested callback functions.

We will now move on to see how we can implement an iterator-based solution to our problem.

An iterator implementation

Another alternative is to implement a type that conforms to the range concept by exposing the `begin()` and `end()` iterators. The class template `LinSpace`, defined here, makes it possible to iterate over the linear range of values:

```
template <typename T>
struct LinSpace {
  LinSpace(T start, T stop, std::size_t n)
      : begin_{start, stop, 0, n}, end_{n} {}
  struct Iterator {
    using difference_type = void;
    using value_type = T;
    using reference = T;
    using pointer = T*;
    using iterator_category = std::forward_iterator_tag;
    void operator++() { ++i_; }
    T operator*() { return lin_value(start_, stop_, i_, n_);}
    bool operator==(std::size_t i) const { return i_ == i; }
    T start_{};
    T stop_{};
    std::size_t i_{};
    std::size_t n_{};
  };
  auto begin() { return begin_; }
  auto end() { return end_; }
 private:
```

```
  Iterator begin_{};
  std::size_t end_{};
};
template <typename T>
auto lin_space(T start, T stop, std::size_t n) {
  return LinSpace{start, stop, n};
}
```

This implementation is very efficient. However, it is afflicted with a lot of boilerplate code and the small algorithm we are trying to encapsulate is now spread out into different parts: the `LinSpace` constructor implements the initial work of setting up the start and stop values, whereas the work needed for computing the values ends up in the member functions of the `Iterator` class. This makes the implementation of the algorithm harder to understand compared with the other versions we have looked at.

A solution using the Ranges library

Yet another alternative is to compose our algorithm using building blocks from the Ranges library (C++20), as shown here:

```
template <typename T>
auto lin_space(T start, T stop, std::size_t n) {
  return std::views::iota(std::size_t{0}, n) |
    std::views::transform([=](auto i) {
      return lin_value(start, stop, i, n);
    });
}
```

Here we have the entire algorithm encapsulated inside a small function. We are using `std::views::iota` to generate the indexes for us. Converting an index to a linear value is a simple transformation that can be chained after the `iota` view.

This version is efficient and composable. The object returned from `lin_space()` is a random-access range of type `std::ranges::view`, which can be iterated over using range-based `for`-loops or passed to other algorithms.

Finally, it's time to use our `Generator` class to implement our algorithm as a coroutine.

A solution using a coroutine

After looking at no less than four versions of this very same problem, we have now reached the last solution. Here I will present a version that uses the general `Generator` class template implemented earlier:

```
template <typename T>
auto lin_space(T start, T stop, std::size_t n) -> Generator<T> {
  for (auto i = 0u; i < n; ++i) {
    co_yield lin_value(start, stop, i, n);
  }
}
```

It's compact, straightforward, and easy to understand. By using `co_yield`, we can write the code in such a way that it looks similar to the simple eager version, but without the need for collecting all the values in a container. It's possible to chain multiple generators based on coroutines, as you will see at the end of this chapter.

This version is also compatible with range-based `for` -loops and standard algorithms. However, this version exposes an input range, so it's not possible to skip ahead arbitrary number of elements, which is possible with the version using the Ranges library.

Conclusion

Obviously, there is more than one way to do it. But why did I show all these approaches?

Firstly, if you are new to coroutines, you will hopefully start to see the patterns where it can be advantageous to use coroutines.

Secondly, the `Generator` template and the use of `co_yield` allows us to implement lazy generators in a very clear and concise way. This becomes obvious when we compare the solution with other versions.

Lastly, some approaches might look very contrived for this example problem but are frequently being used in other contexts. C++ is by default an eager language, and many (myself included) have become accustomed to creating code similar to the eager version. The version using a callback might look very strange but is a commonly used pattern in asynchronous code, where coroutines can wrap or replace those callback-based APIs.

The generator type we implemented is partly based on the synchronous generator template from the CppCoro library. CppCoro also provides an `async_generator` template, which makes it possible to use the `co_await` operator within the generator coroutine. I provided the `Generator` template in this chapter for the purpose of demonstrating how a generator can be implemented and how we can interact with coroutines. But if you plan to start using generators in your code, consider using a third-party library.

A real-world example using generators

Using coroutines for simplifying iterators really shines when the examples are a little bit more advanced. Using `co_yield` with the `Generator` class allows us to implement and combine small algorithms efficiently and without the need for boilerplate code to glue it all together. This next example will try to prove that.

The problem

We will here go through an example of how we can use our `Generator` class to implement a compression algorithm that can be used in search engines to compress the search index typically stored on disk. The example is thoroughly described in the book *Introduction to Information Retrieval* by Manning et al, which is freely available at **https://nlp.stanford.edu/IR-book/**. Here follows a brief background and a short description of the problem.

Search engines use some variant of a data structure called an **inverted index**. It is like an index at the end of a book. Using the index, we can find all pages that contain the terms we are searching for.

Now imagine that we have a database full of recipes and that we build an inverted index for this database. Parts of this index might look something like this:

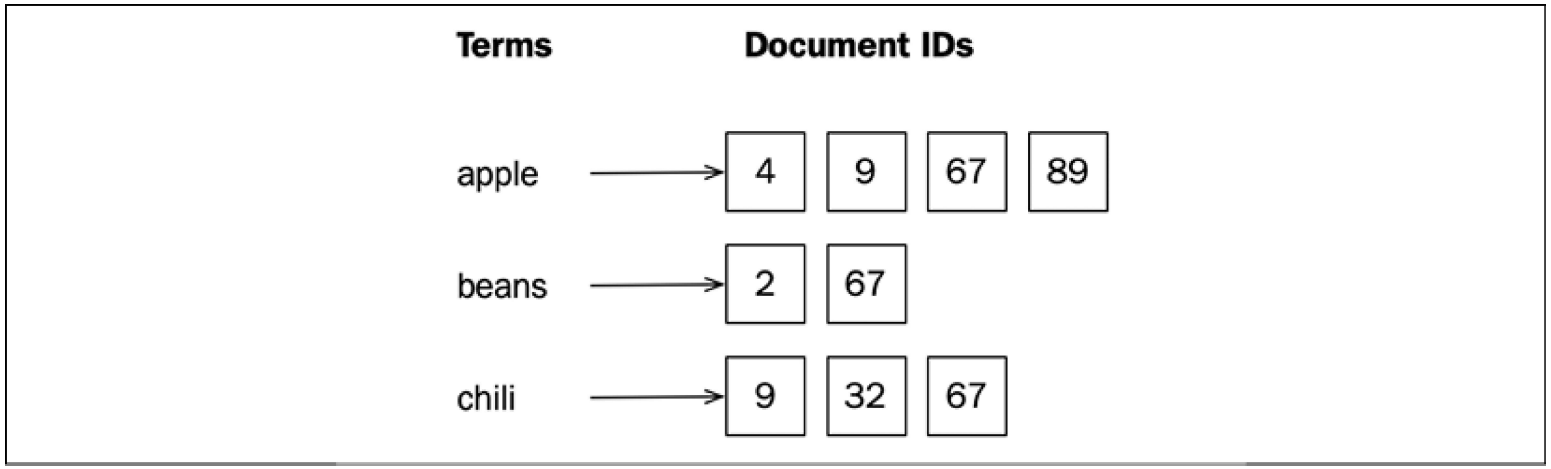

Figure 12.14: An inverted index with three terms and their corresponding lists of document references

Each term is associated with a sorted list of document identifiers. (For example, the term **apple** is included in the recipes with IDs **4**, **9**, **67**, and **89**.) If we want to find recipes that contain both **beans** *and* **chili**, we can run a merge-like algorithm to find the intersection of the lists for **beans** and **chili**:

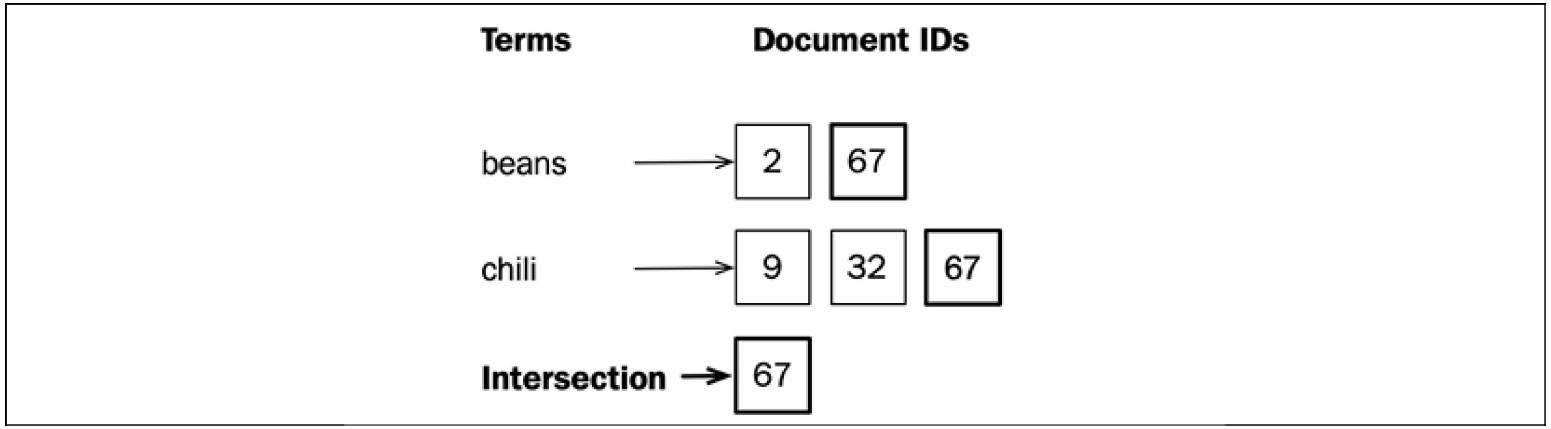

Figure 12.15 Intersection of the document lists for the terms "beans" and "chili"

Now imagine that we have a big database and we choose to represent the document identifier with a 32-bit integer. The lists of document identifiers can become very long for terms that appear in many documents and therefore we need to compress those lists. One possible way to do that is to use delta encoding combined with a variable byte encoding scheme.

Delta encoding

Since the lists are sorted, we could, instead of saving the document identifiers, store the **gap** between two adjacent elements. This technique is called **delta encoding** or **gap encoding**. The following diagram shows an example using document IDs and gaps:

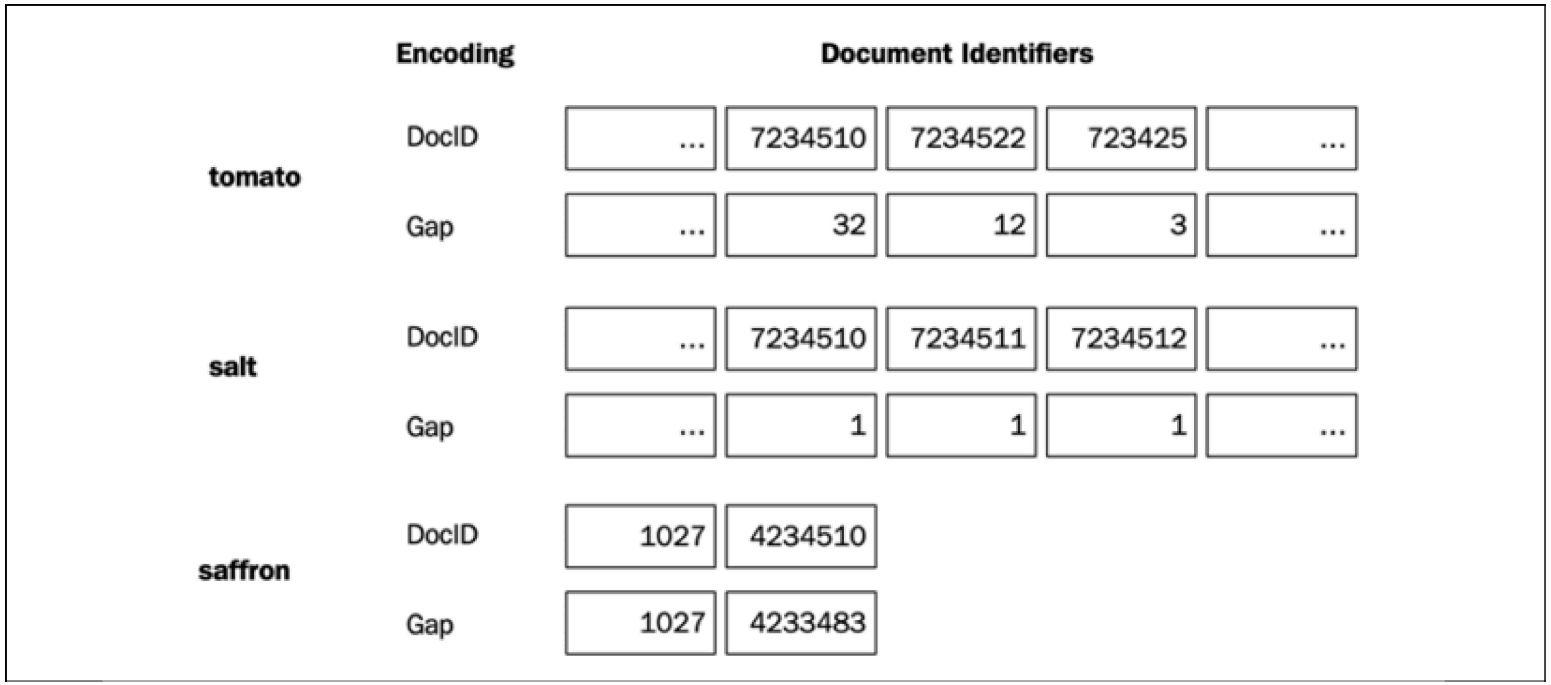

Figure 12.16: Gap encoding stores the gap between two adjacent elements in a list

Gap encoding is well-suited for this type of data; frequently used terms will consequently have many small gaps. The really long lists will only contain very small gaps. After the lists have been gap encoded, we can use a variable byte encoding scheme to actually compress the lists by using fewer bytes for smaller gaps.

But first, let's start implementing the gap encoding functionality. We will begin by writing two small coroutines that will do the gap encoding/decoding. The encoder transforms a sorted sequence of integers to a sequence of gaps:

```
template <typename Range>
auto gap_encode(Range& ids) -> Generator<int> {
  auto last_id = 0;
  for (auto id : ids) {
    const auto gap = id - last_id;
    last_id = id;
    co_yield gap;
  }
}
```

By using `co_yield`, there is no need to eagerly pass a complete list of numbers and allocate a big output list of gaps. Instead, the coroutine lazily handles one number at a time. Note how the function `gap_encode()` contains everything that there is to know about how to convert document IDs to gaps. Implementing this as a traditional iterator would be possible, but this would have logic spread out in constructors and operators on iterators.

We can build a small program to test our gap encoder:

```
int main() {
  auto ids = std::vector{10, 11, 12, 14};
  auto gaps = gap_encode();
  for (auto&& gap : gaps) {
    std::cout << gap << ", ";
  }
} // Prints: 10, 1, 1, 2,
```

The decoder does the opposite; it takes as input a range of gaps and transforms it to the list of ordered numbers:

```
template <typename Range>
auto gap_decode(Range& gaps) -> Generator<int> {
  auto last_id = 0;
  for (auto gap : gaps) {
    const auto id = gap + last_id;
    co_yield id;
    last_id = id;
  }
}
```

By using gap encoding, we will on average, store much smaller numbers. But since we are still using `int` values for storing the small gaps, we haven't really gained anything if we save these gaps to disk. Unfortunately, we cannot just use a smaller fixed-size data type, because there is still a possibility that we will encounter a really big gap that would require a full 32-bit `int`. What we want is a way to store small gaps using fewer bits, as illustrated in the following diagram:

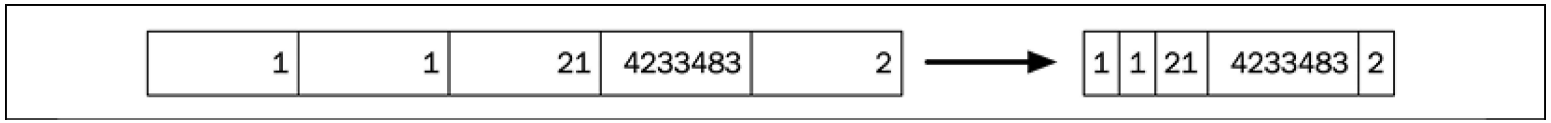

Figure 12.17: Small numbers should use fewer bytes

In order to make this list physically smaller, we can use **variable byte encoding** so that small gaps are encoded with fewer bytes than bigger gaps, as illustrated in the preceding diagram.

Variable byte encoding

Variable byte encoding is a very common compression technique. UTF-8 and MIDI message are some of the well-known encodings that uses this technique. In order to use a variable number of bytes when encoding, we use 7-bits of each byte for the actual payload. The first bit of each byte represents a **continuation bit**. It is set to `0` if there are more bytes to read, or `1` for the last byte of the encoded number. The encoding scheme is exemplified in the following diagram:

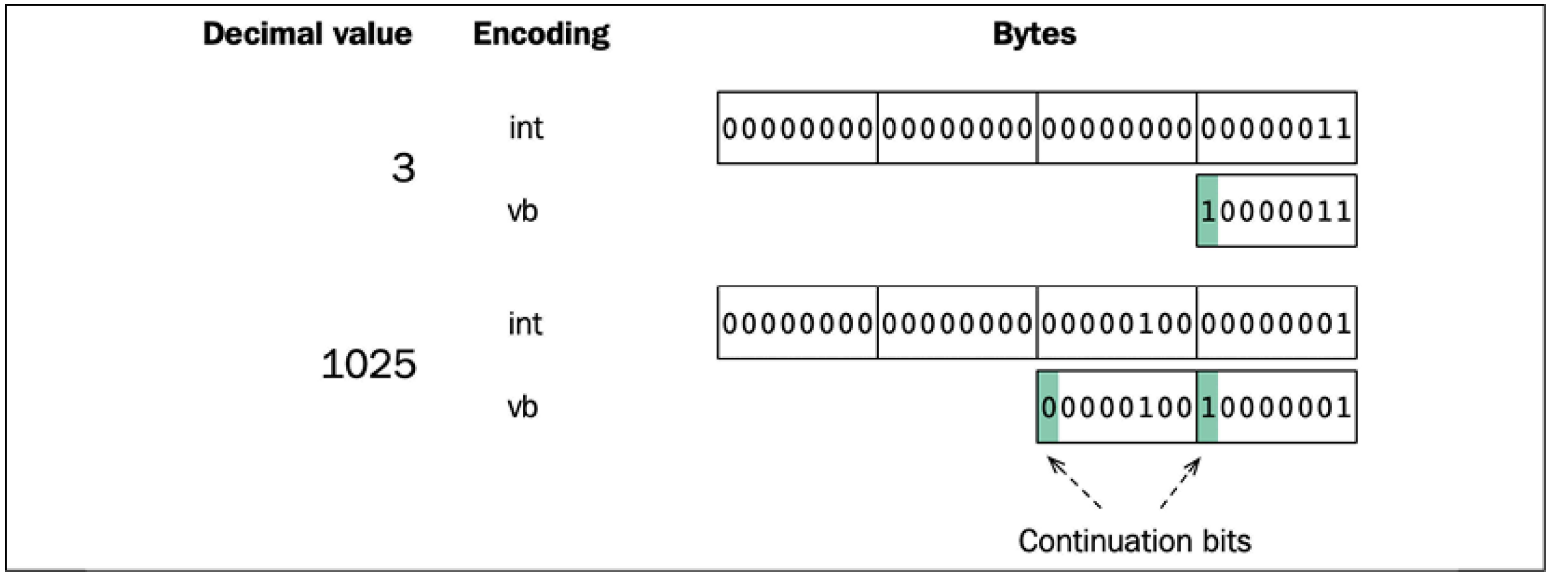

Figure 12.18: Using variable byte encoding, only one byte is required to store the decimal value 3 and two bytes for encoding the decimal value 1025

Now we are ready to implement the variable byte encoding and decoding scheme. This is a little bit more complicated than delta encoding. The encoder should transform a number into a sequence of one or multiple bytes:

```
auto vb_encode_num(int n) -> Generator<std::uint8_t> {
  for (auto cont = std::uint8_t{0}; cont == 0;) {
    auto b = static_cast<std::uint8_t>(n % 128);
    n = n / 128;
    cont = (n == 0) ? 128 : 0;
    co_yield (b + cont);
  }
}
```

The continuation bit, named `cont` in the code, is either 0 or 128, which corresponds to the bit sequence 10000000. The details in this example are not that important to understand, but to make the encoding easier, the bytes are generated in reverse order so that the least significant byte comes first. This is not a problem since we can handle that easily during the decoding.

With the number encoder in place, it's easy to encode a sequence of numbers and transform them into a sequence of bytes:

```
template <typename Range>
auto vb_encode(Range& r) -> Generator<std::uint8_t> {
  for (auto n : r) {
    auto bytes = vb_encode_num(n);
    for (auto b : bytes) {
      co_yield b;
    }
  }
}
```

The decoder is probably the most complicated part. But again, it is fully encapsulated into one single function with a clean interface:

```
template <typename Range>
auto vb_decode(Range& bytes) -> Generator<int> {
  auto n = 0;
  auto weight = 1;
  for (auto b : bytes) {
    if (b < 128) {  // Check continuation bit
```

```
      n += b * weight;
      weight *= 128;
    }
    else {
      // Process last byte and yield
      n += (b - 128) * weight;
      co_yield n;
      n = 0;      // Reset
      weight = 1;  // Reset
    }
  }
}
```

As you can see, there is very little boilerplate code needed in this code. Each coroutine encapsulates all states and describes clearly how to process one piece at a time.

The last piece we need is to combine the gap encoder with the variable byte encoder in order to compress our sorted list of document identifiers:

```
template <typename Range>
auto compress(Range& ids) -> Generator<int> {
  auto gaps = gap_encode(ids);
  auto bytes = vb_encode(gaps);
  for (auto b : bytes) {
    co_yield b;
  }
}
```

Decompress is a simple chaining of `vb_decode()` followed by `gap_decode()`:

```
template <typename Range>
auto decompress(Range& bytes) -> Generator<int> {
  auto gaps = vb_decode(bytes);
  auto ids = gap_decode(gaps);
  for (auto id : ids) {
    co_yield id;
  }
}
```

Since the `Generator` class exposes iterators, we can take this example even further and easily stream the values to and from disk using iostreams. (Although, a more realistic approach would be to use memory-mapped I/O for better performance.) Here are two small functions that writes and reads the compressed data to and from disk:

```
template <typename Range>
void write(const std::string& path, Range& bytes) {
  auto out = std::ofstream{path, std::ios::out | std::ofstream::binary};
  std::ranges::copy(bytes.begin(), bytes.end(),
                    std::ostreambuf_iterator<char>(out));
}
auto read(std::string path) -> Generator<std::uint8_t> {
  auto in = std::ifstream {path, std::ios::in | std::ofstream::binary};
  auto it = std::istreambuf_iterator<char>{in};
  const auto end = std::istreambuf_iterator<char>{};
  for (; it != end; ++it) {
    co_yield *it;
```

```
  }
}
```

A small test program will wrap this example up:

```
int main() {
 {
  auto documents = std::vector{367, 438, 439, 440};
  auto bytes = compress(documents);
  write("values.bin", bytes);
 }
 {
  auto bytes = read("values.bin");
  auto documents = decompress(bytes);
  for (auto doc : documents) {
   std::cout << doc << ", ";
  }
 }
}
// Prints: 367, 438, 439, 440,
```

This example aims to show that we can divide lazy programs into small encapsulated coroutines. The low overhead of C++ coroutines makes them suitable for building efficient generators. The `Generator` we implemented initially is a fully reusable class that helps us with minimizing the amount of boilerplate code in examples like this.

This ends the section about generators. We will now move on to discuss some general performance considerations when using coroutines.

Performance

Each time a coroutine is created (when it is first called) a coroutine frame is allocated to hold the coroutine state. The frame can be allocated on the heap, or on the stack in some circumstances. However, there are no guarantees to completely avoid the heap allocation. If you are in a situation where heap allocations are forbidden (for example, in a real-time context) the coroutine can be created and immediately suspended in a different thread, and then passed to the part of the program that needs to actually use the coroutine. Suspend and resume are guaranteed to not allocate any memory and have a cost comparable with an ordinary function call.

At the time of writing this book, compilers have experimental support for coroutines. Small experiments have shown promising results related to performance, showing that coroutines are friendly to the optimizer. However, I will not provide you with any benchmarks of coroutines in this book. Instead, I have shown you how stackless coroutines are evaluated and how it's possible for coroutines to be implemented with minimal overheads.

The generator example demonstrated that coroutines can potentially be very friendly to the compiler. The chain of generators we wrote in that example was completely evaluated at runtime. In practice, this is a very good property of C++ coroutines. They allow us to write code that is easy for both compilers and human beings to understand. C++ coroutines usually produce clean code that is easy to optimize.

Coroutines that execute on the same thread can share state without using any locking primitives and can therefore avoid the performance overhead incurred by synchronizing multiple threads. This will be

demonstrated in the next chapter.

Summary

In this chapter, you have seen how to use C++ coroutines for building generators using the keywords `co_yield` and `co_return`. To better understand how C++ stackless coroutines differ from stackful coroutines, we compared the two and also looked at the customization points that C++ coroutines offer. This gave you a deep understanding of how flexible C++ coroutines are, as well as how they can be used to achieve efficiency. Stackless coroutines are closely related to state machines. By rewriting a traditionally implemented state machine into code that uses coroutines, we explored this relationship and you saw how well compilers can transform and optimize our coroutines to machine language.

In the next chapter, we will continue to discuss coroutines by focusing on asynchronous programming and will deepen your understanding of the `co_await` keyword.