

NOTE: this guide is currently undergoing a rewrite after a long time without much work. It is work in progress, much is missing, and what exists is a bit rough.

Introduction

This book is a guide to asynchronous programming in Rust. It is designed to help you take your first steps and to discover more about advanced topics. We don't assume any experience with asynchronous programming (in Rust or another language), but we do assume you're familiar with Rust already. If you want to learn about Rust, you could start with [The Rust Programming Language](#).

This book has two main parts: **part one** is a beginners guide, it is designed to be read in-order and to take you from total beginner to intermediate level. Part two is a collection of stand-alone chapters on more advanced topics. It should be useful once you've worked through part one or if you already have some experience with async Rust.

You can navigate this book in multiple ways:

- You can read it front to back, in order. This is the recommend path for newcomers to async Rust, at least for **part one** of the book.
- There is a summary contents on the left-hand side of the webpage.
- If you want information about a broad topic, you could start with the topic index.
- If you want to find all discussion about a specific topic, you could start with the detailed index.
- You could see if your question is answered in the FAQs.

What is Async Programming and why would you do it?

In concurrent programming, the program does multiple things at the same time (or at least appears to). Programming with threads is one form of concurrent programming. Code within a thread is written in sequential style and the operating system executes threads concurrently. With async programming, concurrency happens entirely within your program (the operating system is not involved). An async runtime (which is just another crate in Rust) manages async tasks in conjunction with the programmer explicitly yielding control by using the `await` keyword.

Because the operating system is not involved, *context switching* in the async world is very fast. Furthermore, async tasks have much lower memory overhead than operating system threads. This makes async programming a good fit for systems which need to handle very many

concurrent tasks and where those tasks spend a lot of time waiting (for example, for client responses or for IO).

Async programming also offers the programmer fine-grained control over how tasks are executed (levels of parallelism and concurrency, control flow, scheduling, and so forth). This means that async programming can be expressive as well as ergonomic for many uses. In particular, async programming in Rust has a powerful concept of cancellation and supports many different flavours of concurrency (expressed using constructs including `spawn` and its variations, `join`, `select`, `for_each_concurrent`, etc.). These allow composable and reusable implementations of concepts like timeouts, pausing, and throttling.

Hello, world!

Just to give you a taste of what async Rust looks like, here is a 'hello, world' example. There is no concurrency, and it doesn't really take advantage of being async. It does define and use an async function, and it does print "hello, world!":

```
// Define an async function.
async fn say_hello() {
    println!("hello, world!");
}

#[tokio::main] // Boilerplate which lets us write `async fn main`, we'll explain
it later.
async fn main() {
    // Call an async function and await its result.
    say_hello().await;
}
```

We'll explain everything in detail later. For now, note how we define an asynchronous function using `async fn` and call it using `.await` - an async function in Rust doesn't do anything unless it is `await ed`¹.

Like all examples in this book, if you want to see the full example (including `Cargo.toml`, for example) or to run it yourself locally, you can find them in the book's GitHub repo: e.g., [examples/hello-world](https://github.com/async-rs/async-book/blob/main/examples/hello-world).

Development of Async Rust

The async features of Rust have been in development for a while, but it is not a 'finished' part of the language. Async Rust (at least the parts available in the stable compiler and standard

libraries) is reliable and performant. It is used in production in some of the most demanding situations at the largest tech companies. However, there are some missing parts and rough edges (rough in the sense of ergonomics rather than reliability). You are likely to stumble upon some of these parts during your journey with async Rust. For most missing parts, there are workarounds and these are covered in this book.

Currently, working with async iterators (also known as streams) is where most users find some rough parts. Some uses of async in traits are not yet well-supported. Async closures don't exist yet, and there is not a good solution for async destruction.

Async Rust is being actively worked on. If you want to follow development, you can check out the Async Working Group's [home page](#) which includes their [roadmap](#). Or you could read the async [project goal](#) within the Rust Project.

Rust is an open source project. If you'd like to contribute to development of async Rust, start at the [contributing docs](#) in the main Rust repo.

¹ This is actually a bad example because `println` is *blocking IO* and it is generally a bad idea to do blocking IO in async functions. We'll explain what blocking IO is in [chapter TODO](#) and why you shouldn't do blocking IO in an async function in [chapter TODO](#).

Navigation

TODO Intro to navigation

- By topic
- [FAQs](#)
- Index

Topic index

Concurrency and parallelism

- Introduction
- [Running async tasks in parallel using `spawn`](#)

Correctness and safety

- Cancellation
 - [Introduction](#)

Performance

- Blocking
 - [Introduction](#)

Testing

- [Unit test syntax](#)

Index

- Async/ async
 - blocks
 - closures
 - functions
 - traits
 - c.f., threads
- await
- Blocking
 - IO
- Cancellation
 - CancellationToken
- Concurrency
 - c.f., parallelism
- Executor
- Futures
 - Future trait
- IO
 - Blocking
- Joining tasks
- JoinHandle
 - abort
- Multitasking
 - Cooperative
 - Pre-emptive
- Parallelism

- c.f., concurrency
- Reactor
- Runtimes
- Scheduler
- Spawning tasks
- Tasks
 - Spawning
- Testing
 - Unit tests
- Threads
- Tokio
- Traits
 - `async`
 - `Future`

Part 1: A guide to asynchronous programming in Rust

This part of the book is a tutorial-style guide to async Rust. It is aimed at newcomers to async programming in Rust. It should be useful whether or not you've done async programming in other languages. If you have, you might skip the first section or skim it as a refresher. You might also want to read this [comparison to async in other languages](#) sooner rather than later.

We'll start by discussing different models of [concurrent programming](#), using processes, threads, or async tasks. This chapter will cover the essential parts of Rust's async model before we get into the nitty-gritty of programming in the second chapter where we introduce the `async` and `await` syntax.

Concurrent programming

The goal of this chapter is to give you a high-level idea of how async concurrency works and how it is different from concurrency with threads. I think it is important to have a good mental model of what is going on before getting in to the practicalities, but if you're the kind of person who likes to see some real code first, you might like to read the next chapter or two and then come back to this one.

We'll start with some motivation, then cover [sequential programming](#), [programming with threads or processes](#), and then [async programming](#). The chapter finishes with a section on [concurrency and parallelism](#).

Users want their computers to do multiple things. Sometimes users want to do those things at the same time (e.g., be listening to a music app at the same time as typing in their editor). Sometimes doing multiple tasks at the same time is more efficient (e.g., getting some work done in the editor while a large file downloads). Sometimes there are multiple users wanting to use a single computer at the same time (e.g., multiple clients connected to a server).

To give a lower-level example, a music program might need to keep playing music while the user interacts with the user interface (UI). To 'keep playing music', it might need to stream music data from the server, process that data from one format to another, and send the processed data to the computer's audio system via the operating system (OS). For the user, it might need to send and receive data or commands to the server in response to the user instructions, it might need to send signals to the subsystem playing music (e.g., if the user changes track or pauses), it might need to update the graphical display (e.g., highlighting a button or changing the track name), and it must keep the mouse cursor or text inputs responsive while doing all of the above.

Doing multiple things at once (or appearing to do so) is called concurrency. Programs (in conjunction with the OS) must manage their concurrency and there are many ways to do that. We'll describe some of those ways in this chapter, but we'll start with purely sequential code, i.e., no concurrency at all.

Sequential execution

The default mode of execution in most programming languages (including Rust) is sequential execution.

```
do_a_thing();  
println!("hello!");  
do_another_thing();
```

Each statement is completed before the next one starts¹. Nothing happens in between those statements². This might sound trivial but it is a really useful property for reasoning about our code. However, it also means we waste a lot of time. In the above example, while we're waiting for `println!("hello!")` to happen, we could have executed `do_another_thing()`. Perhaps we could even have executed all three statements at the same time.

Even where IO³ happens (printing using `println!` is IO - it is outputting text to the console via a call to the OS), the program will wait for the IO to complete⁴ before executing the next statement. Waiting for IO to complete before continuing with execution *blocks* the program from making other progress. Blocking IO is the easiest kind of IO to use, implement, and reason about, but it is also the least efficient - in a sequential world, the program can do nothing while it waits for the IO to complete.

¹ This isn't really true: modern compilers and CPUs will reorganize your code and run it any order they like. Sequential statements are likely to overlap in many different ways. However, this should never be *observable* to the program itself or its users.

² This isn't true either: even when one program is purely sequential, other programs might be running at the same time; more on this in the next section.

³ IO is an acronym of input/output. It means any communication from the program to the world outside the program. That might be reading or writing to disk or the network, writing to the terminal, getting user input from a keyboard or mouse, or communicating with the OS or another program running in the system. IO is interesting in the context of concurrency because it takes several orders of magnitude longer to happen than nearly any task a program might do internally. That typically means lots of waiting, and that waiting time is an opportunity to do other work.

⁴ Exactly when IO is complete is actually rather complicated. From the program's perspective a single IO call is complete when control is returned from the OS. This usually indicates that data has been sent to some hardware or other program, but it doesn't necessarily mean that the data has actually been written to disk or displayed to the user, etc. That might require more work in the hardware or periodic flushing of caches, or for another program to read the data. Mostly we don't need to worry about this, but it's good to be aware of.

Processes and threads

Processes and threads are concepts which are provided by the operating system to provide concurrency. There is one process per executable, so supporting multiple processes means a computer can run multiple programs⁵ concurrently; there can be multiple threads per process, which means there can also be concurrency *within* a process.

There are many small differences in the way that processes and threads are handled. The most important difference is that memory is shared between threads but not between processes⁶. That means that communication between processes happens by some kind of message passing, similar to communicating between programs running on different computers. From a program's perspective, the single process is their whole world; creating new processes means running new programs. Creating new threads, however, is just part of the program's regular execution.

Because of these distinctions between processes and threads, they feel very different to a programmer. But from the OS's perspective they are very similar and we'll discuss their properties as if they were a single concept. We'll talk about threads, but unless we note otherwise, you should understand that to mean 'threads or processes'.

The OS is responsible for *scheduling* threads, which means it decides when threads run and for how long. Most modern computers have multiple cores, so they can run multiple threads at literally the same time. However, it is common to have many more threads than cores, so the OS will run each thread for a small amount of time and then pause it and run a different thread for some time⁷. When multiple threads are run on a single core in this fashion, it is called *interleaving* or *time-slicing*. Since the OS chooses when to pause a thread's execution, it is called *pre-emptive multitasking* (multitasking here just means running multiple threads at the same time); the OS *pre-empts* execution of a thread (or more verbosely, the OS pre-emptively pauses execution. It is pre-emptive because the OS is pausing the thread to make time for another thread, before the first thread would otherwise pause, to ensure that the second thread can execute before it becomes a problem that it can't).

Let's look at IO again. What happens when a thread blocks waiting for IO? In a system with threads, then the OS will pause the thread (it's just going to be waiting in any case) and wake it up again when the IO is complete⁸. Depending on the scheduling algorithm, it might take some time after the IO completes until the OS wakes up the thread waiting for IO, since the OS might wait for other threads to get some work done. So now things are much more efficient: while one thread waits for IO, another thread (or more likely, many threads due to multitasking) can make progress. But, from the perspective of the thread doing IO, things are still sequential - it waits for the IO to finish before starting the next operation.

A thread can also choose to pause itself by calling a `sleep` function, usually with a timeout. In this case the OS pauses the thread at the thread's own request. Similar to pausing due to pre-emption or IO, the OS will wake the thread up again later (after the timeout) to continue execution.

When an OS pauses one thread and starts another (for any reason), it is called *context switching*. The context being switched includes the registers, operating system records, and the contents of many caches. That's a non-trivial amount of work. Together with the transfer of control to

the OS and back to a thread, and the costs of working with stale caches, context switching is an expensive operation.

Finally, note that some hardware or OSs do not support processes or threads, this is more likely in the embedded world.

⁵ from the user's perspective, a single program may include multiple processes, but from the OS's perspective each process is a separate program.

⁶ Some OSs do support sharing memory between processes, but using it requires special treatment and most memory is not shared.

⁷ Exactly how the OS chooses which thread to run and for how long (and on which core), is a key part of scheduling. There are many options, both high-level strategies and options to configure those strategies. Making good choices here is crucial for good performance, but it is complicated and we won't dig into it here.

⁸ There's another option which is that the thread can *busy wait* by just spinning in a loop until the IO is finished. This is not very efficient since other threads won't get to run and is uncommon in most modern systems. You may come across it in the implementations of locks or in very simple embedded systems.

Async programming

Async programming is a kind of concurrency with the same high-level goals as concurrency with threads (do many things at the same time), but a different implementation. The two big differences between async concurrency and concurrency with threads, is that async concurrency is managed entirely within the program with no help from the OS⁹, and that multitasking is cooperative rather than pre-emptive¹⁰ (we'll explain that in a minute). There are many different models of async concurrency, we'll compare them later on in the guide, but for now we'll focus only on Rust's model.

To distinguish them from threads, we'll call a sequence of executions in async concurrency a task (they're also called *green threads*, but this sometimes has connotations of pre-emptive scheduling and implementation details like one stack per task). The way a task is executed, scheduled, and represented in memory is very different to a thread, but for a high-level intuition, it can be useful to think of tasks as just like threads, but managed entirely within the program, rather than by the OS.

In an async system, there is still a scheduler which decides which task to run next (it's part of the program, not part of the OS). However, the scheduler cannot pre-empt a task. Instead a task must voluntarily give up control and allow another task to be scheduled. Because tasks must cooperate (by giving up control), this is called cooperative multitasking.

Using cooperative rather than pre-emptive multitasking has many implications:

- between points where control might be yielded, you can guarantee that code will be executed sequentially - you'll never be unexpectedly paused,
- if a task takes a long time between yield points (e.g., by doing blocking IO or performing long-running computation), other tasks will not be able to make progress,
- implementing a scheduler is much simpler and scheduling (and context switching) has fewer overheads.

Async concurrency is much more efficient than concurrency with threads. The memory overheads are much lower and context switching is a much cheaper operation - it doesn't require handing control to the OS and back to the program and there is much less data to switch. However, there can still be some cache effects - although the OS's caches such as the [TLB](#) don't need to be changed, tasks are likely to operate on different parts of memory, so data required by the newly scheduled task may not be in a memory cache.

Asynchronous *IO* is an alternative to blocking IO (it's sometimes called non-blocking IO). Async IO is not directly tied to async concurrency, but the two are often used together. In async IO, a program initiates IO with one system call and then can either check or be notified when the IO completes. That means the program is free to get other work done while the IO takes place. In Rust, the mechanics of async IO are handled by the async runtime (the scheduler is also part of the runtime, we'll discuss runtimes in more detail later in this book, but essentially the runtime is just a library which takes care of some of the fundamental async stuff).

From the perspective of the whole system, blocking IO in a concurrent system with threads and non-blocking IO in an async concurrent system are similar. In both cases, IO takes time and other work gets done while the IO is happening:

- With threads, the thread doing IO requests IO from the OS, the thread is paused by the OS, other threads get work done, and when the IO is done, the OS wakes up the thread so it can continue execution with the result of the IO.
- With async, the task doing IO requests IO from the runtime, the runtime requests IO from the OS but the OS returns control to the runtime. The runtime pauses the IO task and schedules other tasks to get work done. When the IO is done, the runtime wakes up the IO task so it can continue execution with the result of the IO.

The advantage of using async IO, is that the overheads are much lower so a system can support orders of magnitude more tasks than threads. That makes async concurrency particularly well-suited for tasks with lots of users which spend a lot of time waiting for IO (if they don't spend a lot of time waiting and instead do lots of CPU-bound work, then there is not so much advantage to the low-overheads because the bottleneck will be CPU and memory resources).

Threads and async are not mutually exclusive: many programs use both. Some programs have parts which are better implemented using threads and parts which are better implemented using async. For example, a database server may use async techniques to manage network

communication with clients, but use OS threads for computation on data. Alternatively, a program may be written only using async concurrency, but the runtime will execute tasks on multiple threads. This is necessary for a program to make use of multiple CPU cores. We'll cover the intersection of threads and async tasks in a number of places later in the book.

⁹ We'll start our explanation assuming a program only has a single thread, but expand on that later. There will probably be other processes running on the system, but they don't really affect how async concurrency works.

¹⁰ There are some programming languages (or even libraries) which have concurrency which is managed within the program (without the OS), but with a pre-emptive scheduler rather than relying on cooperation between threads. Go is a well-known example. These systems don't require `async` and `await` notation, but have other downsides including making interop with other languages or the OS much more difficult, and having a heavyweight runtime. Very early versions of Rust had such a system, but no traces of it remained by 1.0.

Concurrency and Parallelism

So far we've been talking about concurrency (doing, or appearing to do, many things at the same time), and we've hinted at parallelism (the presence of multiple CPU cores which facilitates literally doing many things at the same time). These terms are sometimes used interchangeably, but they are distinct concepts. In this section, we'll try to precisely define these terms and the difference between them. I'll use simple pseudo-code to illustrate things.

Imagine a single task broken into a bunch of sub-tasks:

```
task1 {  
    subTask1-1()  
    subTask1-2()  
    ...  
    subTask1-100()  
}
```

Let's pretend to be a processor which executes such pseudocode. The obvious way to do so is to first do `subTask1-1` then do `subTask1-2` and so on until we've completed all sub-tasks. This is sequential execution.

Now consider multiple tasks. How might we execute them? We might start one task, do all the sub-tasks until the whole task is complete, then start on the next. The two tasks are being executed sequentially (and the sub-tasks within each task are also being executed sequentially). Looking at just the sub-tasks, you'd execute them like this:

```

subTask1-1()
subTask1-2()
...
subTask1-100()
subTask2-1()
subTask2-2()
...
subTask2-100()

```

Alternatively, you could do `subTask1`, then put `task1` aside (remembering how far you got) and pick up the next task and do the first sub-task from that one, then go back to `task1` to do a sub-task. The two tasks would be interleaved, we call this concurrent execution of the two tasks. It might look like:

```

subTask1-1()
subTask2-1()
subTask1-2()
subTask2-2()
...
subTask1-100()
subTask2-100()

```

Unless one task can observe the results or side-effects of a different task, then from the task's perspective, the sub-tasks are still being executed sequentially.

There's no reason we have to limit ourselves to two tasks, we could interleave any number and do so in any order.

Note that no matter how much concurrency we add, the whole job takes the same amount of time to complete (in fact it might take longer with more concurrency due to the overheads of context switching between them). However, for a given sub-task, we might get it finished earlier than in the purely sequential execution (for a user, this might feel more responsive).

Now, imagine it's not just you processing the tasks, you've got some processor friends to help you out. You can work on tasks at the same time and get the work done faster! This is *parallel* execution (which is also concurrent). You might execute the sub-tasks like:

Processor 1	Processor 2
=====	=====
subTask1-1()	subTask2-1()
subTask1-2()	subTask2-2()
...	...
subTask1-100()	subTask2-100()

If there are more than two processors, we can process even more tasks in parallel. We could also do some interleaving of tasks on each processor or sharing of tasks between processors.

In real code, things are a bit more complicated. Some sub-tasks (e.g., IO) don't require a processor to actively participate, they just need starting and some time later collecting the results. And some sub-tasks might require the results (or side-effects) of a sub-task from a different task in order to make progress (synchronization). Both these scenarios limit the effective ways that tasks can be concurrently executed and that, together with ensuring some concept of fairness, is why scheduling is important.

Enough silly examples, let's try to define things properly

Concurrency is about ordering of computations and parallelism is about the mode of execution.

Given two computations, we say they are sequential (i.e., not concurrent) if we can observe that one happens before the other, or that they are concurrent if we cannot observe (or alternatively, it does not matter) that one happens before the other.

Two computations happen in parallel if they are literally happening at the same time. We can think of parallelism as a resource: the more parallelism is available, the more computations can happen in a fixed period of time (assuming that computation happens at the same speed). Increasing the concurrency of a system without increasing parallelism can never make it faster (although it can make the system more responsive and it may make it feasible to implement optimizations which would otherwise be impractical).

To restate, two computations may happen one after the other (neither concurrent nor parallel), their execution may be interleaved on a single CPU core (concurrent, but not parallel), or they may be executed at the same time on two cores (concurrent and parallel)¹¹.

Another useful framing¹² is that concurrency is a way of organizing code and parallelism is a resource. This is a powerful statement! That concurrency is about organising code rather than executing code is important because from the perspective of the processor, concurrency without parallelism simply doesn't exist. It's particularly relevant for async concurrency because that is implemented entirely in user-side code - not only is it 'just' about organizing code, but you can easily prove that to yourself by just reading the source code. That parallelism is a resource is also useful because it reminds us that for parallelism and performance, only the number of processor cores is important, not how the code is organized with respect to concurrency (e.g., how many threads there are).

Both threaded and async systems can offer both concurrency and parallelism. In both cases, concurrency is controlled by code (spawning threads or tasks) and parallelism is controlled by the scheduler, which is part of the OS for threads (configured by the OS's API), and part of the

runtime library for `async` (configured by choice of runtime, how the runtime is implemented, and options that the runtime provides to client code). There is however, a practical difference due to convention and common defaults. In threaded systems, each concurrent thread is executed in parallel using as much parallelism as possible. In `async` systems, there is no strong default: a system may run all tasks in a single thread, it may assign multiple tasks to a single thread and lock that thread to a core (so groups of tasks execute in parallel, but within a group each task executes concurrently, but never in parallel with other tasks within the group), or tasks may be run in parallel with or without limits. For the first part of this guide, we will use the Tokio runtime which primarily supports the last model. I.e., the behavior regarding parallelism is similar to concurrency with threads. Furthermore, we'll see features in `async Rust` which explicitly support concurrency but not parallelism, independent of the runtime.

¹¹ Can computation be parallel but not concurrent? Sort of but not really. Imagine two tasks (a and b) which consist of one sub-task each (1 and 2 belonging to a and b, respectively). By the use of synchronisation, we can't start sub-task 2 until sub-task 1 is complete and task a has to wait for sub-task 2 to complete until it is complete. Now a and b run on different processors. If we look at the tasks as black boxes, we can say they are running in parallel, but in a sense they are not concurrent because their ordering is fully determined. However, if we look at the sub-tasks we can see that they are neither parallel or concurrent.

¹² Which I think is due to Aaron Turon and is reflected in some of the design of Rust's standard library, e.g., in the [available_parallelism](#) function.

Summary

- There are many models of execution. We described sequential execution, threads and processes, and asynchronous programming.
 - Threads are an abstraction provided (and scheduled) by the OS. They usually involve pre-emptive multitasking, are parallel by default, and have fairly high overheads of management and context switching.
 - Asynchronous programming is managed by a user-space runtime. Multi-tasking is cooperative. It has lower overheads than threads, but feels a bit different to programming with threads since it uses different programming primitives (`async` and `await` , and futures, rather than first-class threads).
- Concurrency and parallelism are different but closely related concepts.
 - Concurrency is about ordering of computation (operations are concurrent if their order of execution cannot be observed).
 - Parallelism is about computing on multiple processors (operations are parallel if they are literally happening at the same time).
- Both OS threads and `async` programming provide concurrency and parallelism; `async` programming can also offer constructs for flexible or fine-grained concurrency which are not part of most operating systems' threads API.

Async and Await

In this chapter we'll get started doing some async programming in Rust and we'll introduce the `async` and `await` keywords.

`async` is an annotation on functions (and other items, such as traits, which we'll get to later); `await` is an operator used in expressions. But before we jump into those keywords, we need to cover a few core concepts of async programming in Rust, this follows from the discussion in the previous chapter, here we'll relate things directly to Rust programming.

Rust async concepts

The runtime

Async tasks must be managed and scheduled. There are typically more tasks than cores available so they can't all be run at once. When one stops executing another must be picked to execute. If a task is waiting on IO or some other event, it should not be scheduled, but when that completes, it should be scheduled. That requires interacting with the OS and managing IO work.

Many programming languages provide a runtime. Commonly, this runtime does a lot more than manage async tasks - it might manage memory (including garbage collection), have a role in exception handling, provide an abstraction layer over the OS, or even be a full virtual machine. Rust is a low-level language and strives towards minimal runtime overhead. The async runtime therefore has a much more limited scope than many other languages' runtimes. There are also many ways to design and implement an async runtime, so Rust lets you choose one depending on your requirements, rather than providing one. This does mean that getting started with async programming requires an extra step.

As well as running and scheduling tasks, a runtime must interact with the OS to manage async IO. It must also provide timer functionality to tasks (which intersects with IO management). There are no strong rules about how a runtime must be structured, but some terms and division of responsibilities are common:

- *reactor* or *event loop* or *driver* (equivalent terms): dispatches IO and timer events, interacts with the OS, and does the lowest-level driving forward of execution,
- *scheduler*: determines when tasks can execute and on which OS threads,

- *executor* or *runtime*: combines the reactor and scheduler, and is the user-facing API for running async tasks; *runtime* is also used to mean the whole library of functionality (e.g., everything in the Tokio crate, not just the Tokio executor which is represented by the `Runtime` type).

As well as the executor as described above, a runtime crate typically includes many utility traits and functions. These might include traits (e.g., `AsyncRead`) and implementations for IO, functionality for common IO tasks such as networking or accessing the file system, locks, channels, and other synchronisation primitives, utilities for timing, utilities for working with the OS (e.g., signal handling), utility functions for working with futures and streams (async iterators), or monitoring and observation tools. We'll cover many of those in this guide.

There are many async runtimes to choose from. Some have very different scheduling policies, or are optimised for a specific task or domain. For most of this guide we'll use the `Tokio` runtime. It's a general purpose runtime and is the most popular runtime in the ecosystem. It's a great choice for getting started and for production work. In some circumstances, you might get better performance or be able to write simpler code with a different runtime. Later in this guide we'll discuss some of the other available runtimes and why you might choose one or another, or even write your own.

To get up and running as quickly as possible, you need just a little boilerplate. You'll need to include the Tokio crate as a dependency in your `Cargo.toml` (just like any other crate):

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

And you'll use the `tokio::main` annotation on your `main` function so that it can be an async function (which is otherwise not permitted in Rust):

```
#[tokio::main]
async fn main() { ... }
```

That's it! You're ready to write some asynchronous code!

The `#[tokio::main]` annotation initializes the Tokio runtime and starts an async task for running the code in `main`. Later in this guide we'll explain in more detail what that annotation is doing and how to use async code without it (which will give you more flexibility).

Futures and tasks

The basic unit of async concurrency in Rust is the *future*. A future is just a regular old Rust object (a struct or enum, usually) which implements the `'Future'` trait. A future represents a

deferred computation. That is, a computation that will be ready at some point in the future.

We'll talk a lot about futures in this guide, but it's easiest to get started without worrying too much about them. We'll mention them quite a bit in the next few sections, but we won't really define them or use them directly until later. One important aspect of futures is that they can be combined to make new, 'bigger' futures (we'll talk a lot more about *how* they can be combined later).

I've used the term 'async task' quite a bit in an informal way in the previous chapter and this one. I've used the term to mean a logical sequence of execution; analogous to a thread but managed within a program rather than externally by the OS. It is often useful to think in terms of tasks, however, Rust itself has no concept of a task and the term is used to mean different things! It is confusing! To make it worse, runtimes do have a concept of a task and different runtimes have slightly different concepts of tasks.

From here on in, I'm going to try to be precise about the terminology around tasks. When I use just 'task' I mean the abstract concept of a sequence of computation that may occur concurrently with other tasks. I'll use 'async task' to mean exactly the same thing, but in contrast to a task which is implemented as an OS thread. I'll use 'runtime's task' to mean whatever kind of task a runtime imagines, and 'tokio task' (or some other specific runtime) to mean Tokio's idea of a task.

An async task in Rust is just a future (usually a 'big' future made by combining many others). In other words, a task is a future which is executed. However, there are times when a future is 'executed' without being a runtime's task. This kind of a future is intuitively a *task* but not a *runtime's task*. I'll spell this out more when we get to an example of it.

Async functions

The `async` keyword is a modifier on function declarations. E.g., we can write `pub async fn send_to_server(...)`. An async function is simply a function declared using the `async` keyword, and what that means is that it is a function which can be executed asynchronously, in other words the caller *can choose not to* wait for the function to complete before doing something else.

In more mechanical terms, when an async function is called, the body is not executed as it would be for a regular function. Instead the function body and its arguments are packaged into a future which is returned in lieu of a real result. The caller can then decide what to do with that future (if the caller wants the result 'straight away', then it will `await` the future, see the next section).

Within an `async` function, code is executed in the usual, sequential way¹, being `async` makes no difference. You can call synchronous functions from `async` functions, and execution proceeds as usual. One extra thing you can do within an `async` function is use `await` to await other `async` functions (or futures), which *may* cause yielding of control so that another task can execute.

¹ like any other thread, the thread the `async` function is running on may be pre-empted by the operating system and paused so another thread can get some work done. However, from the function's point of view this is not observable without inspecting data which may have been modified by other threads (and which could have been modified by another thread executing in parallel without the current thread being paused).

await

We stated above that a future is a computation that will be ready at some point in the future. To get the result of that computation, we use the `await` keyword. If the result is ready immediately or can be computed without waiting, then `await` simply does that computation to produce the result. However, if the result is not ready, then `await` hands control over to the scheduler so that another task can proceed (this is cooperative multitasking mentioned in the previous chapter).

The syntax for using `await` is `some_future.await`, i.e., it is a postfix keyword used with the `.` operator. That means it can be used ergonomically in chains of method calls and field accesses.

Consider the following functions:

```
// An async function, but it doesn't need to wait for anything.
async fn add(a: u32, b: u32) -> u32 {
    a + b
}

async fn wait_to_add(a: u32, b: u32) -> u32 {
    sleep(1000).await;
    a + b
}
```

If we call `add(15, 3).await` then it will return immediately with the result `18`. If we call `wait_to_add(15, 3).await`, we will eventually get the same answer, but while we wait another task will get an opportunity to run.

In this silly example, the call to `sleep` is a stand-in for doing some long-running task where we have to wait for the result. This is usually an IO operation where the result is data read from an external source or confirmation that writing to an external destination succeeded. Reading

looks something like `let data = read(...).await?`. In this case `await` will cause the current task to wait while the read happens. The task will resume once reading is completed (other tasks could get some work done while the reading task waits). The result of reading could be data successfully read or an error (handled by the `?`).

Note that if we call `add` or `wait_to_add` or `read` without using `.await` we won't get any answer!

What?

Calling an async function returns a future, it doesn't immediately execute the code in the function. Furthermore, a future does not do any work until it is awaited². This is in contrast to some other languages where an async function returns a future which begins executing immediately.

This is an important point about async programming in Rust. After a while it will be second nature, but it often trips up beginners, especially those who have experience with async programming in other languages.

An important intuition about futures in Rust is that they are inert objects. To get any work done they must be driven forward by an external force (usually an async runtime).

We've described `await` quite operationally (it runs a future, producing a result), but we talked in the previous chapter about async tasks and concurrency, how does `await` fit into that mental model? First, let's consider pure sequential code: logically, calling a function simply executes the code in the function (with some assignment of variables). In other words, the current task continues executing the next 'chunk' of code which is defined by the function. Similarly, in an async context, calling a non-async function simply continues execution with that function. Calling an async function finds the code to run, but doesn't run it. `await` is an operator which continues execution of the current task, or if the current task can't continue right now, gives another task an opportunity to continue.

`await` can only be used inside an async context, for now that means inside an async function (we'll see more kinds of async contexts later). To understand why, remember that `await` might hand over control to the runtime so that another task can execute. There is only a runtime to hand control to in an async context. For now, you can imagine the runtime like a global variable which is only accessible in async functions, we'll explain later how it really works.

Finally, for one more perspective on `await`: we mentioned earlier that futures can be combined together to make 'bigger' futures. `async` functions are one way to define a future, and `await` is one way to combine futures. Using `await` on a future combines that future into the future produced by the async function it's used inside. We'll talk in more detail about this perspective and other ways to combine futures later.

² Or `poll`, which is a lower-level operation than `await` and happens behind the scenes when using `await`. We'll talk about polling later when we talk about futures in detail.

Some `async/await` examples

Let's start by revisiting our 'hello, world!' example:

```
// Define an async function.
async fn say_hello() {
    println!("hello, world!");
}

#[tokio::main] // Boilerplate which lets us write `async fn main`, we'll explain
it later.
async fn main() {
    // Call an async function and await its result.
    say_hello().await;
}
```

You should now recognise the boilerplate around `main`. It's for initializing the Tokio runtime and creating an initial task to run the `async main` function.

`say_hello` is an `async` function, when we call it, we have to follow the call with `.await` to run it as part of the current task. Note that if you remove the `.await`, then running the program does nothing! Calling `say_hello` returns a future, but it is never executed so `println` is never called (the compiler will warn you, at least).

Here's a slightly more realistic example, taken from the [Tokio tutorial](#).

```
#[tokio::main]
async fn main() -> Result<(),> {
    // Open a connection to the mini-redis address.
    let mut client = client::connect("127.0.0.1:6379").await?;

    // Set the key "hello" with value "world"
    client.set("hello", "world".into()).await?;

    // Get key "hello"
    let result = client.get("hello").await?;

    println!("got value from the server; result={:?}", result);

    Ok(())
}
```

The code is a bit more interesting, but we're essentially doing the same thing - calling async functions and then awaiting to execute the result. This time we're using `?` for error handling - it works just like in synchronous Rust.

For all the talk so far about concurrency, parallelism, and asynchrony, both these examples are 100% sequential. Just calling and awaiting async functions does not introduce any concurrency unless there are other tasks to schedule while the awaiting task is waiting. To prove this to ourselves, let's look at another simple (but contrived) example:

```
use std::io::{stdout, Write};
use tokio::time::{sleep, Duration};

async fn say_hello() {
    print!("hello, ");
    // Flush stdout so we see the effect of the above `print` immediately.
    stdout().flush().unwrap();
}

async fn say_world() {
    println!("world!");
}

#[tokio::main]
async fn main() {
    say_hello().await;
    // An async sleep function, puts the current task to sleep for 1s.
    sleep(Duration::from_millis(1000)).await;
    say_world().await;
}
```

Between printing "hello" and "world", we put the current task to sleep³ for one second. Observe what happens when we run the program: it prints "hello", does nothing for one second, then prints "world". That is because executing a single task is purely sequential. If we had some concurrency, then that one second nap would be an excellent opportunity to get some other work done, like printing "world". We'll see how to do that in the next section.

³ Note that we're using an async sleep function here, if we were to use `sleep` from `std` we'd put the whole thread to sleep. That wouldn't make any difference in this toy example but in a real program it would mean other tasks could not be scheduled on that thread during that time. That is very bad.

Spawning tasks

We've talked about `async` and `await` as a way to run code in an async task. And we've said that `await` can put the current task to sleep while it waits for IO or some other event. When that happens, another task can run, but how do those other tasks come about? Just like we use

`std::thread::spawn` to spawn a new task, we can use `tokio::spawn` to spawn a new async task. Note that `spawn` is a function of Tokio, the runtime, not from Rust's standard library, because tasks are purely a runtime concept.

Here's a tiny example of running an async function on a separate task by using `spawn`:

```
use tokio::{spawn, time::{sleep, Duration}};

async fn say_hello() {
    // Wait for a while before printing to make it a more interesting race.
    sleep(Duration::from_millis(100)).await;
    println!("hello");
}

async fn say_world() {
    sleep(Duration::from_millis(100)).await;
    println!("world!");
}

#[tokio::main]
async fn main() {
    spawn(say_hello());
    spawn(say_world());
    // Wait for a while to give the tasks time to run.
    sleep(Duration::from_millis(1000)).await;
}
```

Similar to the last example, we have two functions printing "hello" and "world!". But this time we run them concurrently (and in parallel) rather than sequentially. If you run the program a few times you should see the strings printing in both orders - sometimes "hello" first, sometimes "world!" first. A classic concurrent race!

Let's dive into what is happening here. There are three concepts in play: futures, tasks, and threads. The `spawn` function takes a future (which remember can be made up of many smaller futures) and runs it as a new Tokio task. Tasks are the the concept which the Tokio runtime schedules and manages (not individual futures). Tokio (in its default configuration) is a multi-threaded runtime which means that when we spawn a new task, that task may be run on a different OS thread from the task it was spawned from (it may be run on the same thread, or it may start on one thread and then be moved to another later on).

So, when a future is spawned as a task it runs *concurrently* with the task it was spawned from and any other tasks. It may also run in parallel to those tasks if it is scheduled on a different thread.

To summarise, when we write two statements following each other in Rust, they are executed sequentially (whether in async code or not). When we write `await`, that does not change the concurrency of sequential statements. E.g., `foo(); bar();` is strictly sequential - `foo` is called

and afterwards, `bar` is called. That is true whether `foo` and `bar` are async functions or not. `foo().await; bar().await;` is also strictly sequential, `foo` is fully evaluated and then `bar` is fully evaluated. In both cases another thread might be interleaved with the sequential execution and in the second case, another async task might be interleaved at the await points, but the two statements are executed sequentially *with respect to each other* in both cases.

If we use either `thread::spawn` or `tokio::spawn` we introduce concurrency and potentially parallelism, in the first case between threads and in the second between tasks.

Later in the guide we'll see cases where we execute futures concurrently, but never in parallel.

Joining tasks

If we want to get the result of executing a spawned task, then the spawning task can wait for it to finish and use the result, this is called *joining* the tasks (analogous to [joining](#) threads, and the APIs for joining are similar).

When a task is spawned, the spawn function returns a `JoinHandle`. If you just want the task to do its own thing executing, the `JoinHandle` can be discarded (dropping the `JoinHandle` does not affect the spawned task). But if you want the spawning task to wait for the spawned task to complete and then use the result, you can `await` the `JoinHandle` to do so.

For example, let's revisit our 'Hello, world!' example one more time:

```

use tokio::{spawn, time::{sleep, Duration}};

async fn say_hello() {
    // Wait for a while before printing to make it a more interesting race.
    sleep(Duration::from_millis(100)).await;
    println!("hello");
}

async fn say_world() {
    sleep(Duration::from_millis(100)).await;
    println!("world");
}

#[tokio::main]
async fn main() {
    let handle1 = spawn(say_hello());
    let handle2 = spawn(say_world());

    let _ = handle1.await;
    let _ = handle2.await;

    println!("!");
}

```

The code is similar to last time, but instead of just calling `spawn`, we save the returned `JoinHandle`s and later `await` them. Since we're waiting for those tasks to complete before we exit the `main` function, we no longer need the `sleep` in `main`.

The two spawned tasks are still executing concurrently. If you run the program a few times you should see both orderings. However, the `await`ed join handles are a limit on the concurrency: the final exclamation mark (!) will *always* be printed last (you could experiment with moving `println!("!");` relative to the `await`s. You'll probably need to change with the sleep times too to get observable effects).

If we immediately `await`ed the `JoinHandle` of the first `spawn` rather than saved it and later `await`ed (i.e., written `spawn(say_hello()).await;`), then we'd have spawned another task to run the 'hello' future, but the spawning task would have waited for it to finish before doing anything else. In other words, there is no possible concurrency! You almost never want to do this (because why bother with the `spawn`? Just write the sequential code).

JoinHandle

We'll quickly look at `JoinHandle` in a little more depth. The fact that we can `await` a `JoinHandle` is a clue that a `JoinHandle` is itself a future. `spawn` is not an `async` function, it's a regular function that returns a future (`JoinHandle`). It does some work (to schedule the task) before returning the future (unlike an `async` future), which is why we don't *need* to `await`

`spawn`. Awaiting a `JoinHandle` waits for the spawned task to complete and then returns the result. In the above example, there was no result, we just waited for the task to complete.

`JoinHandle` is a generic type and its type parameter is the type returned by the spawned task. In the above example, the type would be `JoinHandle<()>`, a future that results in a `String` would produce a `JoinHandle` with type `JoinHandle<String>`.

awaiting a `JoinHandle` returns a `Result` (which is why we used `let _ = ...` in the above example, it avoids a warning about an unused `Result`). If the spawned task completed successfully, then the task's result will be in the `Ok` variant. If the task panicked or was aborted (a form of cancellation, see [TODO](#)), then the result will be an `Err` containing a [JoinError docs](#). If you are not using cancellation via `abort` in your project, then unwrapping the result of `JoinHandle.await` is a reasonable approach, since that is effectively propagating a panic from the spawned task to the spawning task.

More async/await topics

Unit tests

How to unit test async code? The issue is that you can only await from inside an async context, and unit tests in Rust are not async. Luckily, most runtimes provide a convenience attribute for tests similar to the one for `async main`. Using Tokio, it looks like this:

```
#[tokio::test]
async fn test_something() {
    // Write a test here, including all the `await`s you like.
}
```

There are many ways to configure the test, see the [docs](#) for details.

There are some more advanced topics in testing async code (e.g., testing for race conditions, deadlock, etc.), and we'll cover some of those [later](#) in this guide.

Blocking and cancellation

Blocking and cancellation are important to keep in mind when programming with async Rust. These concepts are not localised to any particular feature or function, but are ubiquitous properties of the system which you must understand to write correct code.

Blocking IO

We say a thread (note we're talking about OS threads here, not async tasks) is blocked when it can't make any progress. That's usually because it is waiting for the OS to complete a task on its behalf (usually I/O). Importantly, while a thread is blocked, the OS knows not to schedule it so that other threads can make progress. This is fine in a multithreaded program because it lets other threads make progress while the blocked thread is waiting. However, in an async program, there are other tasks which should be scheduled on the same OS thread, but the OS doesn't know about those and keeps the whole thread waiting. This means that rather than the single task waiting for its I/O to complete (which is fine), many tasks have to wait (which is not fine).

We'll talk soon about non-blocking/async I/O. For now, just know that non-blocking I/O is I/O which the async runtime knows about and so will only the current task will wait, the thread will not be blocked. It is very important to only use non-blocking I/O from an async task, never blocking I/O (which is the only kind provided in Rust's standard library).

Blocking computation

You can also block the thread by doing computation (this is not quite the same as blocking I/O, since the OS is not involved, but the effect is similar). If you have a long-running computation (with or without blocking I/O) without yielding control to the runtime, then that task will never give the runtime's scheduler a chance to schedule other tasks. Remember that async programming uses cooperative multitasking. Here a task is not cooperating, so other tasks won't get a chance to get work done. We'll discuss ways to mitigate this later.

There are many other ways to block a whole thread, and we'll come back to blocking several times in this guide.

Cancellation

Cancellation means stopping a future (or task) from executing. Since in Rust (and in contrast to many other async/await systems), futures must be driven forward by an external force (like the async runtime), if a future is no longer driven forward then it will not execute any more. If a future is dropped (remember, a future is just a plain old Rust object), then it can never make any more progress and is canceled.

Cancellation can be initiated in a few ways:

- By simply dropping a future (if you own it).
- Calling `abort` on a task's 'JoinHandle' (or an `AbortHandle`).
- Via a `CancellationToken` (which requires the future being canceled to notice the token and cooperatively cancel itself).
- Implicitly, by a function or macro like `select`.

The middle two are specific to Tokio, though most runtimes provide similar facilities. Using a `CancellationToken` requires cooperation of the future being canceled, but the others do not. In these other cases, the canceled future will get no notification of cancellation and no opportunity to clean up (besides its destructor). Note that even if a future has a cancellation token, it can still be canceled via the other methods which won't trigger the cancellation token.

From the perspective of writing async code (in async functions, blocks, futures, etc.), the code might stop executing at any `await` (including hidden ones in macros) and never start again. In

order for your code to be correct (specifically to be *cancellation safe*), it must work correctly whether it completes normally or whether it terminates at any await point¹.

```

async fn some_function(input: Option<Input>) {
    let Some(input) = input else {
        return;           // Might terminate here (`return`).
    };

    let x = foo(input)?;   // Might terminate here (`?`).

    let y = bar(x).await;  // Might terminate here (`await`).

    // ...

    //                Might terminate here (implicit return).
}

```

An example of how this can go wrong is if an async function reads data into an internal buffer, then awaits the next datum. If reading the data is destructive (i.e., cannot be re-read from the original source) and the async function is canceled, then the internal buffer will be dropped, and the data in it will be lost. It is important to consider how a future and any data it touches will be impacted by canceling the future, restarting the future, or starting a new future which touches the same data.

We'll be coming back to cancellation and cancellation safety a few times in this guide, and there is a whole [chapter](#) on the topic in the reference section.

¹ It is interesting to compare cancellation in async programming with canceling threads. Canceling a thread is possible (e.g., using `pthread_cancel` in C, there is no direct way to do this in Rust), but it is almost always a very, very bad idea since the thread being canceled can terminate anywhere. In contrast, canceling an async task can only happen at an await point. As a consequence, it is very rare to cancel an OS thread without terminating the whole process and so as a programmer, you generally don't worry about this happening. In async Rust however, cancellation is definitely something which *can* happen. We'll be discussing how to deal with that as we go along.

Async blocks

A regular block (`{ ... }`) groups code together in the source and creates a scope of encapsulation for names. At runtime, the block is executed in order and evaluates to the value of its last expression (or the unit type (`()`) if there is no trailing expression).

Similarly to async functions, an async block is a deferred version of a regular block. An async block scopes code and names together, but at runtime it is not immediately executed and evaluates to a future. To execute the block and obtain the result, it must be `await` ed. E.g.:

```
let s1 = {
    let a = 42;
    format!("The answer is {a}")
};

let s2 = async {
    let q = question().await;
    format!("The question is {q}")
};
```

If we were to execute this snippet, `s1` would be a string which could be printed, but `s2` would be a future; `question()` would not have been called. To print `s2`, we first have to `s2.await`.

An `async` block is the simplest way to start an `async` context and create a future. It is commonly used to create small futures which are only used in one place.

Unfortunately, control flow with `async` blocks is a little quirky. Because an `async` block creates a future rather than straightforwardly executing, it behaves more like a function than a regular block with respect to control flow. `break` and `continue` cannot go 'through' an `async` block like they can with regular blocks; instead you have to use `return`:

```
loop {
    {
        if ... {
            // ok
            continue;
        }
    }

    async {
        if ... {
            // not ok
            // continue;

            // ok - continues with the next execution of the `loop`, though note
            that if there was
            // code in the loop after the async block that would be executed.
            return;
        }
    }.await
}
```

To implement `break` you would need to test the value of the block (a common idiom is to use `ControlFlow` for the value of the block, which also allows use of `?`).

Likewise, `?` inside an `async` block will terminate execution of the future in the presence of an error, causing the `await`ed block to take the value of the error, but won't exit the surrounding function (like `?` in a regular block would). You'll need another `?` after `await` for that:


```

async {
    let x = foo()?;    // This `?` only exits the async block, not the surrounding
function.
    consume(x);
    Ok(())
}.await?

```

Annoyingly, this often confuses the compiler since (unlike functions) the 'return' type of an async block is not explicitly stated. You'll probably need to add some type annotations on variables or use turbofished types to make this work, e.g., `Ok::<_, MyError>()` instead of `Ok()` in the above example.

A function which returns an async block is pretty similar to an async function. Writing `async fn foo() -> ... { ... }` is roughly equivalent to `fn foo() -> ... { async { ... } }`. In fact, from the caller's perspective they are equivalent, and changing from one form to the other is not a breaking change. Furthermore, you can override one with the other when implementing an async trait (see below). However, you do have to adjust the type, making the `Future` explicit in the async block version: `async fn foo() -> Foo` becomes `fn foo() -> impl Future<Output = Foo>` (you might also need to make other bounds explicit, e.g., `Send` and `'static`).

You would usually prefer the async function version since it is simpler and clearer. However, the async block version is more flexible since you can execute some code when the function is called (by writing it outside the async block) and some code when the result is awaited (the code inside the async block).

Async closures

- closures
 - coming soon (<https://github.com/rust-lang/rust/pull/132706>, <https://blog.rust-lang.org/inside-rust/2024/08/09/async-closures-call-for-testing.html>)
 - async blocks in closures vs async closures

Lifetimes and borrowing

- Mentioned the static lifetime above
- Lifetime bounds on futures (`Future + '_` , etc.)
- Borrowing across await points
- I don't know, I'm sure there are more lifetime issues with async functions ...

Send + 'static bounds on futures

- Why they're there, multi-threaded runtimes
- spawn local to avoid them
- What makes an `async fn` `Send + 'static` and how to fix bugs with it

Async traits

- syntax
 - The `Send + 'static` issue and working around it
 - `trait_variant`
 - explicit future
 - return type notation (<https://blog.rust-lang.org/inside-rust/2024/09/26/rtn-call-for-testing.html>)
- overriding
 - future vs async notation for methods
- object safety
- capture rules (<https://blog.rust-lang.org/2024/09/05/impl-trait-capture-rules.html>)
- history and `async-trait` crate

Recursion

- Allowed (relatively new), but requires some explicit boxing
 - forward reference to futures, pinning
 - https://rust-lang.github.io/async-book/07_workarounds/04_recursion.html
 - <https://blog.rust-lang.org/2024/03/21/Rust-1.77.0.html#support-for-recursion-in-async-fn>
 - `async-recursion` macro (https://docs.rs/async-recursion/latest/async_recursion/)

IO and issues with blocking

Blocking and non-blocking IO

- High level view
- How async IO fits with async concurrency
- Why blocking IO is bad
- forward ref to streams for streams/sinks

Read and Write

- async Read and Write traits
 - part of the runtime
- how to use
- specific implementations
 - network vs disk
 - tcp, udp
 - file system is not really async, but io_uring (ref to that chapter)
 - practical examples
 - stdout, etc.
 - pipe, fd, etc.

Memory management

- Issues with buffer management and async IO
- Different solutions and pros and cons
 - zero-copy approach
 - shared buffer approach
- Utility crates to help with this, Bytes, etc.

Advanced topics on IO

- buf read/write

- Read + Write, split, join
- copy
- simplex and duplex
- cancelation

The OS view of IO

- Different kinds of IO and mechanisms, completion IO, reference to completion IO chapter in adv section
 - different runtimes can facilitate this
 - mio for low-level interface

Other blocking operations

- Why this is bad
- Long running CPU work
 - Using Tokio for just CPU work: <https://thenewstack.io/using-rustlangs-async-tokio-runtime-for-cpu-bound-tasks/>
- Solutions
 - spawn blocking
 - thread pool
 - etc.
- yielding to the runtime
 - not the same as Rust's yield keyword
 - await doesn't yield
 - implicit yields in Tokio

Concurrency primitives

- concurrent composition of futures
 - c.f., sequential composition with `await`, composition of tasks with `spawn`
 - `concurrent/task` behaviour
 - behaviour on error
- streams as alternative, forward ref
- different versions in different runtimes/other crates
 - focus on the Tokio versions

From [comment](#): A framing I've started using is that tasks are not the `async/await` form of threads; it's more accurate to think of them as parallelizable futures. This framing does not match Tokio and `async-std`'s current task design; but both also have trouble propagating cancellation. See `parallel_future` and `tasks` are the wrong abstraction for more.

Join

- Tokio/futures-rs join macro
- c.f., joining tasks
- join in futures-concurrency
- `FuturesUnordered`
 - like a dynamic version of join
 - forward ref to stream

Race/select

- Tokio select macro
- cancellation issues
- different behaviour of futures-rs version
- race in futures-concurrency

Channels, locking, and synchronization

note on runtime specificness of sync primitives

Why we need async primitives rather than use the sync ones

Channels

- basically same as the std ones, but await
 - communicate between tasks (same thread or different)
- one shot
- mpsc
- other channels
- bounded and unbounded channels

Locks

- async Mutex
 - c.f., `std::Mutex` - can be held across await points (borrowing the mutex in the guard, guard is Send, scheduler-aware? or just because lock is async?), lock is async (will not block the thread waiting for lock to be available)
 - even a clippy lint for holding the guard across await (https://rust-lang.github.io/rust-clippy/master/index.html#await_holding_lock)
 - more expensive because it can be held across await
 - use `std::Mutex` if you can
 - can use `try_lock` or mutex is expected to not be under contention
 - lock is not magically dropped when yield (that's kind of the point of a lock!)
 - deadlock by holding mutex over await
 - tasks deadlocked, but other tasks can make progress so might not look like a deadlock in process stats/tools/OS
 - usual advice - limit scope, minimise locks, order locks, prefer alternatives
 - no mutex poisoning
 - `lock_owned`
 - `blocking_lock`
 - cannot use in async
 - applies to other locks (should the above be moved before discussion of mutex specifically? Probably yes)

- RWLock
- Semaphore
- yielding

Other synchronization primitives

- notify, barrier
- OnceCell
- atomics

Tools for async programming

- Why we need specialist tools for async
- Are there other tools to cover
 - loom

Monitoring

- [Tokio console](#)

Tracing and logging

- issues with async tracing
- tracing crate (<https://github.com/tokio-rs/tracing>)

Debugging

- Understanding async backtraces (RUST_BACKTRACE and in a debugger)
- Techniques for debugging async code
- Using Tokio console for debugging
- Debugger support (WinDbg?)

Profiling

- How async messes up flamegraphs
- How to profile async IO
- Getting insight into the runtime
 - Tokio metrics

Destruction and clean-up

- Object destruction and recap of Drop
- General clean up requirements in software
- Async issues
 - Might want to do stuff async during clean up, e.g., send a final message
 - Might need to clean up stuff which is still being used async-ly
 - Might want to clean up when an async task completes or cancels and there is no way to catch that
 - State of the runtime during clean-up phase (esp if we're panicking or whatever)
 - No async Drop
 - WIP
 - forward ref to completion io topic

Cancellation

- How it happens (recap of more-async-await.md)
 - drop a future
 - cancellation token
 - abort functions
- What we can do about 'catching' cancellation
 - logging or monitoring cancellation
- How cancellation affects other futures tasks (forward ref to cancellation safety chapter, this should just be a heads-up)

Panicking and async

- Propagation of panics across tasks (spawn result)
- Panics leaving data inconsistent (tokio mutexes)
- Calling async code when panicking (make sure you don't)

Patterns for clean-up

- Avoid needing clean up (abort/restart)

- Don't use async for cleanup and don't worry too much
- async clean up method + dtor bomb (i.e., separate clean-up from destruction)
- centralise/out-source clean-up in a separate task or thread or supervisor object/process

Why no async Drop (yet)

- Note this is advanced section and not necessary to read
- Why async Drop is hard
- Possible solutions and there issues
- Current status

Futures

We've talked a lot about futures in the preceding chapters; they're a key part of Rust's async programming story! In this chapter we're going to get into some of the details of what futures are and how they work, and some libraries for working directly with futures.

The Future and IntoFuture traits

- Future
 - Output assoc type
 - No real detail here, polling is in the next section, reference adv sections on Pin, executors/wakers
- IntoFuture
 - Usage - general, in await, async builder pattern (pros and cons in using)
- Boxing futures, `Box<dyn Future>` and how it used to be common and necessary but mostly isn't now, except for recursion, etc.

Polling

- what it is and who does it, Poll type
 - ready is final state
- how it connects with await
- drop = cancel
 - for futures and thus tasks
 - implications for async programming in general
 - reference to chapter on cancellation safety

Fusing

futures-rs crate

- History and purpose
 - see streams chapter

- helpers for writing executors or other low-level futures stuff
 - pinning and boxing
- executor as a partial runtime (see alternate runtimes in reference)
- TryFuture
- convenience futures: pending, ready, ok/err, etc.
- combinator functions on FutureExt
- alternative to Tokio stuff
 - functions
 - IO traits

futures-concurrency crate

https://docs.rs/futures-concurrency/latest/futures_concurrency/

Runtimes and runtime issues

Running async code

- Explicit startup vs async main
- tokio context concept
- `block_on`
- runtime as reflected in the code (Runtime, Handle)
- runtime shutdown

Threads and tasks

- default work stealing, multi-threaded
 - revisit `Send` + 'static bounds
- `yield`
- `spawn-local`
- `spawn-blocking` (recap), `block-in-place`
- tokio-specific stuff on yielding to other threads, local vs global queues, etc

Configuration options

- thread pool size
- single threaded, thread per core etc.

Alternate runtimes

- Why you'd want to use a different runtime or implement your own
- What kind of variations exist in the high-level design
- Forward ref to adv chapters

Timers and Signal handling

Time and Timers

- runtime integration, don't use `thread::sleep`, etc.
- std Instant and Duration
- sleep
- interval
- timeout

Signal handling

- what is signal handling and why is it an async issue?
- very OS specific
- see Tokio docs

Async iterators (FKA streams)

- Stream as an async iterator or as many futures
- WIP
 - current status
 - futures and Tokio Stream traits
 - nightly trait
- lazy like sync iterators
- pinning and streams (forward ref to pinning chapter)
- fused streams

Consuming an async iterator

- while let with async next
- for_each, for_each_concurrent
- collect
- into_future, buffered

Stream combinators

- Taking a future instead of a closure
- Some example combinators
- unordered variations
- StreamGroup

Implementing an async iterator

- Implementing the trait
- Practicalities and util functions
- `async_iter` stream macro

Sinks

- <https://docs.rs/futures/latest/futures/sink/index.html>

Future work

- current status
 - <https://rust-lang.github.io/rfcs/2996-async-iterator.html>
- async next vs poll
- async iteration syntax
- (async) generators
- lending iterators

Getting Started

Welcome to Asynchronous Programming in Rust! If you're looking to start writing asynchronous Rust code, you've come to the right place. Whether you're building a web server, a database, or an operating system, this book will show you how to use Rust's asynchronous programming tools to get the most out of your hardware.

What This Book Covers

This book aims to be a comprehensive, up-to-date guide to using Rust's async language features and libraries, appropriate for beginners and old hands alike.

- The early chapters provide an introduction to async programming in general, and to Rust's particular take on it.
- The middle chapters discuss key utilities and control-flow tools you can use when writing async code, and describe best-practices for structuring libraries and applications to maximize performance and reusability.
- The last section of the book covers the broader async ecosystem, and provides a number of examples of how to accomplish common tasks.

With that out of the way, let's explore the exciting world of Asynchronous Programming in Rust!

Why Async?

We all love how Rust empowers us to write fast, safe software. But how does asynchronous programming fit into this vision?

Asynchronous programming, or *async* for short, is a *concurrent programming model* supported by an increasing number of programming languages. It lets you run a large number of concurrent tasks on a small number of OS threads, while preserving much of the look and feel of ordinary synchronous programming, through the `async/await` syntax.

Async vs other concurrency models

Concurrent programming is less mature and "standardized" than regular, sequential programming. As a result, we express concurrency differently depending on which concurrent programming model the language is supporting. A brief overview of the most popular concurrency models can help you understand how asynchronous programming fits within the broader field of concurrent programming:

- **OS threads** don't require any changes to the programming model, which makes it very easy to express concurrency. However, synchronizing between threads can be difficult, and the performance overhead is large. Thread pools can mitigate some of these costs, but not enough to support massive IO-bound workloads.
- **Event-driven programming**, in conjunction with *callbacks*, can be very performant, but tends to result in a verbose, "non-linear" control flow. Data flow and error propagation is often hard to follow.
- **Coroutines**, like threads, don't require changes to the programming model, which makes them easy to use. Like *async*, they can also support a large number of tasks. However, they abstract away low-level details that are important for systems programming and custom runtime implementors.
- **The actor model** divides all concurrent computation into units called actors, which communicate through fallible message passing, much like in distributed systems. The actor model can be efficiently implemented, but it leaves many practical issues unanswered, such as flow control and retry logic.

In summary, asynchronous programming allows highly performant implementations that are suitable for low-level languages like Rust, while providing most of the ergonomic benefits of threads and coroutines.

Async in Rust vs other languages

Although asynchronous programming is supported in many languages, some details vary across implementations. Rust's implementation of async differs from most languages in a few ways:

- **Futures are inert** in Rust and make progress only when polled. Dropping a future stops it from making further progress.
- **Async is zero-cost** in Rust, which means that you only pay for what you use. Specifically, you can use async without heap allocations and dynamic dispatch, which is great for performance! This also lets you use async in constrained environments, such as embedded systems.
- **No built-in runtime** is provided by Rust. Instead, runtimes are provided by community maintained crates.
- **Both single- and multithreaded** runtimes are available in Rust, which have different strengths and weaknesses.

Async vs threads in Rust

The primary alternative to async in Rust is using OS threads, either directly through `std::thread` or indirectly through a thread pool. Migrating from threads to async or vice versa typically requires major refactoring work, both in terms of implementation and (if you are building a library) any exposed public interfaces. As such, picking the model that suits your needs early can save a lot of development time.

OS threads are suitable for a small number of tasks, since threads come with CPU and memory overhead. Spawning and switching between threads is quite expensive as even idle threads consume system resources. A thread pool library can help mitigate some of these costs, but not all. However, threads let you reuse existing synchronous code without significant code changes—no particular programming model is required. In some operating systems, you can also change the priority of a thread, which is useful for drivers and other latency sensitive applications.

Async provides significantly reduced CPU and memory overhead, especially for workloads with a large amount of IO-bound tasks, such as servers and databases. All else equal, you can have orders of magnitude more tasks than OS threads, because an async runtime uses a small amount of (expensive) threads to handle a large amount of (cheap) tasks. However, async Rust results in larger binary blobs due to the state machines generated from async functions and since each executable bundles an async runtime.

On a last note, asynchronous programming is not *better* than threads, but different. If you don't need async for performance reasons, threads can often be the simpler alternative.

Example: Concurrent downloading

In this example our goal is to download two web pages concurrently. In a typical threaded application we need to spawn threads to achieve concurrency:

```
fn get_two_sites() {  
    // Spawn two threads to do work.  
    let thread_one = thread::spawn(|| download("https://www.foo.com"));  
    let thread_two = thread::spawn(|| download("https://www.bar.com"));  
  
    // Wait for both threads to complete.  
    thread_one.join().expect("thread one panicked");  
    thread_two.join().expect("thread two panicked");  
}
```

However, downloading a web page is a small task; creating a thread for such a small amount of work is quite wasteful. For a larger application, it can easily become a bottleneck. In async Rust, we can run these tasks concurrently without extra threads:

```
async fn get_two_sites_async() {  
    // Create two different "futures" which, when run to completion,  
    // will asynchronously download the webpages.  
    let future_one = download_async("https://www.foo.com");  
    let future_two = download_async("https://www.bar.com");  
  
    // Run both futures to completion at the same time.  
    join!(future_one, future_two);  
}
```

Here, no extra threads are created. Additionally, all function calls are statically dispatched, and there are no heap allocations! However, we need to write the code to be asynchronous in the first place, which this book will help you achieve.

Custom concurrency models in Rust

On a last note, Rust doesn't force you to choose between threads and async. You can use both models within the same application, which can be useful when you have mixed threaded and async dependencies. In fact, you can even use a different concurrency model altogether, such as event-driven programming, as long as you find a library that implements it.

The State of Asynchronous Rust

Parts of async Rust are supported with the same stability guarantees as synchronous Rust. Other parts are still maturing and will change over time. With async Rust, you can expect:

- Outstanding runtime performance for typical concurrent workloads.
- More frequent interaction with advanced language features, such as lifetimes and pinning.
- Some compatibility constraints, both between sync and async code, and between different async runtimes.
- Higher maintenance burden, due to the ongoing evolution of async runtimes and language support.

In short, async Rust is more difficult to use and can result in a higher maintenance burden than synchronous Rust, but gives you best-in-class performance in return. All areas of async Rust are constantly improving, so the impact of these issues will wear off over time.

Language and library support

While asynchronous programming is supported by Rust itself, most async applications depend on functionality provided by community crates. As such, you need to rely on a mixture of language features and library support:

- The most fundamental traits, types and functions, such as the `Future` trait are provided by the standard library.
- The `async/await` syntax is supported directly by the Rust compiler.
- Many utility types, macros and functions are provided by the `futures` crate. They can be used in any async Rust application.
- Execution of async code, IO and task spawning are provided by "async runtimes", such as Tokio and `async-std`. Most async applications, and some async crates, depend on a specific runtime. See "[The Async Ecosystem](#)" section for more details.

Some language features you may be used to from synchronous Rust are not yet available in async Rust. Notably, Rust did not let you declare async functions in traits until 1.75.0 stable (and still has limitations on dynamic dispatch for those traits). Instead, you need to use workarounds to achieve the same result, which can be more verbose.

Compiling and debugging

For the most part, compiler- and runtime errors in async Rust work the same way as they have always done in Rust. There are a few noteworthy differences:

Compilation errors

Compilation errors in async Rust conform to the same high standards as synchronous Rust, but since async Rust often depends on more complex language features, such as lifetimes and pinning, you may encounter these types of errors more frequently.

Runtime errors

Whenever the compiler encounters an async function, it generates a state machine under the hood. Stack traces in async Rust typically contain details from these state machines, as well as function calls from the runtime. As such, interpreting stack traces can be a bit more involved than it would be in synchronous Rust.

New failure modes

A few novel failure modes are possible in async Rust, for instance if you call a blocking function from an async context or if you implement the `Future` trait incorrectly. Such errors can silently pass both the compiler and sometimes even unit tests. Having a firm understanding of the underlying concepts, which this book aims to give you, can help you avoid these pitfalls.

Compatibility considerations

Asynchronous and synchronous code cannot always be combined freely. For instance, you can't directly call an async function from a sync function. Sync and async code also tend to promote different design patterns, which can make it difficult to compose code intended for the different environments.

Even async code cannot always be combined freely. Some crates depend on a specific async runtime to function. If so, it is usually specified in the crate's dependency list.

These compatibility issues can limit your options, so make sure to research which async runtime and what crates you may need early. Once you have settled in with a runtime, you won't have to worry much about compatibility.

Performance characteristics

The performance of async Rust depends on the implementation of the async runtime you're using. Even though the runtimes that power async Rust applications are relatively new, they perform exceptionally well for most practical workloads.

That said, most of the async ecosystem assumes a *multi-threaded* runtime. This makes it difficult to enjoy the theoretical performance benefits of single-threaded async applications, namely cheaper synchronization. Another overlooked use-case is *latency sensitive tasks*, which are important for drivers, GUI applications and so on. Such tasks depend on runtime and/or OS support in order to be scheduled appropriately. You can expect better library support for these use cases in the future.

async/.await Primer

`async / .await` is Rust's built-in tool for writing asynchronous functions that look like synchronous code. `async` transforms a block of code into a state machine that implements a trait called `Future`. Whereas calling a blocking function in a synchronous method would block the whole thread, blocked `Future`s will yield control of the thread, allowing other `Future`s to run.

Let's add some dependencies to the `Cargo.toml` file:

```
[dependencies]
futures = "0.3"
```

To create an asynchronous function, you can use the `async fn` syntax:

```
async fn do_something() { /* ... */ }
```

The value returned by `async fn` is a `Future`. For anything to happen, the `Future` needs to be run on an executor.

```
// `block_on` blocks the current thread until the provided future has run to
// completion. Other executors provide more complex behavior, like scheduling
// multiple futures onto the same thread.
use futures::executor::block_on;

async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // Nothing is printed
    block_on(future); // `future` is run and "hello, world!" is printed
}
```

Inside an `async fn`, you can use `.await` to wait for the completion of another type that implements the `Future` trait, such as the output of another `async fn`. Unlike `block_on`, `.await` doesn't block the current thread, but instead asynchronously waits for the future to complete, allowing other tasks to run if the future is currently unable to make progress.

For example, imagine that we have three `async fn`: `learn_song`, `sing_song`, and `dance`:


```

async fn learn_song() -> Song { /* ... */ }
async fn sing_song(song: Song) { /* ... */ }
async fn dance() { /* ... */ }

```

One way to do learn, sing, and dance would be to block on each of these individually:

```

fn main() {
    let song = block_on(learn_song());
    block_on(sing_song(song));
    block_on(dance());
}

```

However, we're not giving the best performance possible this way—we're only ever doing one thing at once! Clearly we have to learn the song before we can sing it, but it's possible to dance at the same time as learning and singing the song. To do this, we can create two separate `async fn` which can be run concurrently:

```

async fn learn_and_sing() {
    // Wait until the song has been learned before singing it.
    // We use `.await` here rather than `block_on` to prevent blocking the
    // thread, which makes it possible to `dance` at the same time.
    let song = learn_song().await;
    sing_song(song).await;
}

async fn async_main() {
    let f1 = learn_and_sing();
    let f2 = dance();

    // `join!` is like `.await` but can wait for multiple futures concurrently.
    // If we're temporarily blocked in the `learn_and_sing` future, the `dance`
    // future will take over the current thread. If `dance` becomes blocked,
    // `learn_and_sing` can take back over. If both futures are blocked, then
    // `async_main` is blocked and will yield to the executor.
    futures::join!(f1, f2);
}

fn main() {
    block_on(async_main());
}

```

In this example, learning the song must happen before singing the song, but both learning and singing can happen at the same time as dancing. If we used `block_on(learn_song())` rather than `learn_song().await` in `learn_and_sing`, the thread wouldn't be able to do anything else while `learn_song` was running. This would make it impossible to dance at the same time. By `.await`-ing the `learn_song` future, we allow other tasks to take over the current thread if

`learn_song` is blocked. This makes it possible to run multiple futures to completion concurrently on the same thread.

Under the Hood: Executing Futures and Tasks

In this section, we'll cover the underlying structure of how `Future`s and asynchronous tasks are scheduled. If you're only interested in learning how to write higher-level code that uses existing `Future` types and aren't interested in the details of how `Future` types work, you can skip ahead to the `async / await` chapter. However, several of the topics discussed in this chapter are useful for understanding how `async / await` code works, understanding the runtime and performance properties of `async / await` code, and building new asynchronous primitives. If you decide to skip this section now, you may want to bookmark it to revisit in the future.

Now, with that out of the way, let's talk about the `Future` trait.

The Future Trait

The `Future` trait is at the center of asynchronous programming in Rust. A `Future` is an asynchronous computation that can produce a value (although that value may be empty, e.g. `()`). A *simplified* version of the future trait might look something like this:

```
trait SimpleFuture {
    type Output;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

Futures can be advanced by calling the `poll` function, which will drive the future as far towards completion as possible. If the future completes, it returns `Poll::Ready(result)`. If the future is not able to complete yet, it returns `Poll::Pending` and arranges for the `wake()` function to be called when the `Future` is ready to make more progress. When `wake()` is called, the executor driving the `Future` will call `poll` again so that the `Future` can make more progress.

Without `wake()`, the executor would have no way of knowing when a particular future could make progress, and would have to be constantly polling every future. With `wake()`, the executor knows exactly which futures are ready to be `poll`ed.

For example, consider the case where we want to read from a socket that may or may not have data available already. If there is data, we can read it in and return `Poll::Ready(data)`, but if no data is ready, our future is blocked and can no longer make progress. When no data is available, we must register `wake` to be called when data becomes ready on the socket, which will tell the executor that our future is ready to make progress. A simple `SocketRead` future might look something like this:

```

pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            // The socket has data -- read it into a buffer and return it.
            Poll::Ready(self.socket.read_buf())
        } else {
            // The socket does not yet have data.
            //
            // Arrange for `wake` to be called once data is available.
            // When data becomes available, `wake` will be called, and the
            // user of this `Future` will know to call `poll` again and
            // receive data.
            self.socket.set_readable_callback(wake);
            Poll::Pending
        }
    }
}

```

This model of `Future`s allows for composing together multiple asynchronous operations without needing intermediate allocations. Running multiple futures at once or chaining futures together can be implemented via allocation-free state machines, like this:

```

/// A SimpleFuture that runs two other futures to completion concurrently.
///
/// Concurrency is achieved via the fact that calls to `poll` each future
/// may be interleaved, allowing each future to advance itself at its own pace.
pub struct Join<FutureA, FutureB> {
    // Each field may contain a future that should be run to completion.
    // If the future has already completed, the field is set to `None`.
    // This prevents us from polling a future after it has completed, which
    // would violate the contract of the `Future` trait.
    a: Option<FutureA>,
    b: Option<FutureB>,
}

impl<FutureA, FutureB> SimpleFuture for Join<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        // Attempt to complete future `a`.
        if let Some(a) = &mut self.a {
            if let Poll::Ready(()) = a.poll(wake) {
                self.a.take();
            }
        }

        // Attempt to complete future `b`.
        if let Some(b) = &mut self.b {
            if let Poll::Ready(()) = b.poll(wake) {
                self.b.take();
            }
        }

        if self.a.is_none() && self.b.is_none() {
            // Both futures have completed -- we can return successfully
            Poll::Ready(())
        } else {
            // One or both futures returned `Poll::Pending` and still have
            // work to do. They will call `wake()` when progress can be made.
            Poll::Pending
        }
    }
}

```

This shows how multiple futures can be run simultaneously without needing separate allocations, allowing for more efficient asynchronous programs. Similarly, multiple sequential futures can be run one after another, like this:

```

/// A SimpleFuture that runs two futures to completion, one after another.
//
// Note: for the purposes of this simple example, `AndThenFut` assumes both
// the first and second futures are available at creation-time. The real
// `AndThen` combinator allows creating the second future based on the output
// of the first future, like `get_breakfast.and_then(|food| eat(food))`.
pub struct AndThenFut<FutureA, FutureB> {
    first: Option<FutureA>,
    second: FutureB,
}

impl<FutureA, FutureB> SimpleFuture for AndThenFut<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if let Some(first) = &mut self.first {
            match first.poll(wake) {
                // We've completed the first future -- remove it and start on
                // the second!
                Poll::Ready(()) => self.first.take(),
                // We couldn't yet complete the first future.
                // Notice that we disrupt the flow of the `poll` function with the
                `return` statement.
                Poll::Pending => return Poll::Pending,
            };
        }
        // Now that the first future is done, attempt to complete the second.
        self.second.poll(wake)
    }
}

```

These examples show how the `Future` trait can be used to express asynchronous control flow without requiring multiple allocated objects and deeply nested callbacks. With the basic control-flow out of the way, let's talk about the real `Future` trait and how it is different.

```

trait Future {
    type Output;
    fn poll(
        // Note the change from `&mut self` to `Pin<&mut Self>`:
        self: Pin<&mut Self>,
        // and the change from `wake: fn()` to `cx: &mut Context<'_>`:
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}

```

The first change you'll notice is that our `self` type is no longer `&mut self`, but has changed to `Pin<&mut Self>`. We'll talk more about pinning in [a later section](#), but for now know that it

allows us to create futures that are immovable. Immovable objects can store pointers between their fields, e.g. `struct MyFut { a: i32, ptr_to_a: *const i32 }`. Pinning is necessary to enable `async/await`.

Secondly, `wake: fn()` has changed to `&mut Context<'_>`. In `SimpleFuture`, we used a call to a function pointer (`fn()`) to tell the future executor that the future in question should be polled. However, since `fn()` is just a function pointer, it can't store any data about *which* `Future` called `wake`.

In a real-world scenario, a complex application like a web server may have thousands of different connections whose wakeups should all be managed separately. The `Context` type solves this by providing access to a value of type `Waker`, which can be used to wake up a specific task.

Task Wakeups with Waker

It's common that futures aren't able to complete the first time they are `poll`ed. When this happens, the future needs to ensure that it is polled again once it is ready to make more progress. This is done with the `Waker` type.

Each time a future is polled, it is polled as part of a "task". Tasks are the top-level futures that have been submitted to an executor.

`Waker` provides a `wake()` method that can be used to tell the executor that the associated task should be awoken. When `wake()` is called, the executor knows that the task associated with the `Waker` is ready to make progress, and its future should be polled again.

`Waker` also implements `clone()` so that it can be copied around and stored.

Let's try implementing a simple timer future using `Waker`.

Applied: Build a Timer

For the sake of the example, we'll just spin up a new thread when the timer is created, sleep for the required time, and then signal the timer future when the time window has elapsed.

First, start a new project with `cargo new --lib timer_future` and add the imports we'll need to get started to `src/lib.rs`:

```
use std::{
    future::Future,
    pin::Pin,
    sync::{Arc, Mutex},
    task::{Context, Poll, Waker},
    thread,
    time::Duration,
};
```

Let's start by defining the future type itself. Our future needs a way for the thread to communicate that the timer has elapsed and the future should complete. We'll use a shared `Arc<Mutex<..>>` value to communicate between the thread and the future.

```
pub struct TimerFuture {
    shared_state: Arc<Mutex<SharedState>>,
}

/// Shared state between the future and the waiting thread
struct SharedState {
    /// Whether or not the sleep time has elapsed
    completed: bool,

    /// The waker for the task that `TimerFuture` is running on.
    /// The thread can use this after setting `completed = true` to tell
    /// `TimerFuture`'s task to wake up, see that `completed = true`, and
    /// move forward.
    waker: Option<Waker>,
}
```

Now, let's actually write the `Future` implementation!

```
impl Future for TimerFuture {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        // Look at the shared state to see if the timer has already completed.
        let mut shared_state = self.shared_state.lock().unwrap();
        if shared_state.completed {
            Poll::Ready(())
        } else {
            // Set waker so that the thread can wake up the current task
            // when the timer has completed, ensuring that the future is polled
            // again and sees that `completed = true`.
            //
            // It's tempting to do this once rather than repeatedly cloning
            // the waker each time. However, the `TimerFuture` can move between
            // tasks on the executor, which could cause a stale waker pointing
            // to the wrong task, preventing `TimerFuture` from waking up
            // correctly.
            //
            // N.B. it's possible to check for this using the `Waker::will_wake`
            // function, but we omit that here to keep things simple.
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

Pretty simple, right? If the thread has set `shared_state.completed = true`, we're done! Otherwise, we clone the `waker` for the current task and pass it to `shared_state.waker` so that the thread can wake the task back up.

Importantly, we have to update the `waker` every time the future is polled because the future may have moved to a different task with a different `waker`. This will happen when futures are

passed around between tasks after being polled.

Finally, we need the API to actually construct the timer and start the thread:

```
impl TimerFuture {
    /// Create a new `TimerFuture` which will complete after the provided
    /// timeout.
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        // Spawn the new thread
        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            // Signal that the timer has completed and wake up the last
            // task on which the future was polled, if one exists.
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Woot! That's all we need to build a simple timer future. Now, if only we had an executor to run the future on...

Applied: Build an Executor

Rust's `Future`s are lazy: they won't do anything unless actively driven to completion. One way to drive a future to completion is to `.await` it inside an `async` function, but that just pushes the problem one level up: who will run the futures returned from the top-level `async` functions? The answer is that we need a `Future` executor.

`Future` executors take a set of top-level `Future`s and run them to completion by calling `poll` whenever the `Future` can make progress. Typically, an executor will `poll` a future once to start off. When `Future`s indicate that they are ready to make progress by calling `wake()`, they are placed back onto a queue and `poll` is called again, repeating until the `Future` has completed.

In this section, we'll write our own simple executor capable of running a large number of top-level futures to completion concurrently.

For this example, we depend on the `futures` crate for the `ArcWake` trait, which provides an easy way to construct a `Waker`. Edit `Cargo.toml` to add a new dependency:

```
[package]
name = "timer_future"
version = "0.1.0"
authors = ["XYZ Author"]
edition = "2021"

[dependencies]
futures = "0.3"
```

Next, we need the following imports at the top of `src/main.rs`:

```
use futures::{
    future::{BoxFuture, FutureExt},
    task::{waker_ref, ArcWake},
};
use std::{
    future::Future,
    sync::mpsc::{sync_channel, Receiver, SyncSender},
    sync::{Arc, Mutex},
    task::Context,
    time::Duration,
};
// The timer we wrote in the previous section:
use timer_future::TimerFuture;
```

Our executor will work by sending tasks to run over a channel. The executor will pull events off of the channel and run them. When a task is ready to do more work (is awoken), it can schedule itself to be polled again by putting itself back onto the channel.

In this design, the executor itself just needs the receiving end of the task channel. The user will get a sending end so that they can spawn new futures. Tasks themselves are just futures that can reschedule themselves, so we'll store them as a future paired with a sender that the task can use to requeue itself.

```

/// Task executor that receives tasks off of a channel and runs them.
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

/// `Spawner` spawns new futures onto the task channel.
#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

/// A future that can reschedule itself to be polled by an `Executor`.
struct Task {
    /// In-progress future that should be pushed to completion.
    ///
    /// The `Mutex` is not necessary for correctness, since we only have
    /// one thread executing tasks at once. However, Rust isn't smart
    /// enough to know that `future` is only mutated from one thread,
    /// so we need to use the `Mutex` to prove thread-safety. A production
    /// executor would not need this, and could use `UnsafeCell` instead.
    future: Mutex<Option<BoxFuture<'static, ()>>>,

    /// Handle to place the task itself back onto the task queue.
    task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() -> (Executor, Spawner) {
    // Maximum number of tasks to allow queueing in the channel at once.
    // This is just to make `sync_channel` happy, and wouldn't be present in
    // a real executor.
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender })
}

```

Let's also add a method to spawner to make it easy to spawn new futures. This method will take a future type, box it, and create a new `Arc<Task>` with it inside which can be enqueued onto the executor.

```
impl Spawner {
    fn spawn(&self, future: impl Future<Output = ()> + 'static + Send) {
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender.try_send(task).expect("too many tasks queued");
    }
}
```

To poll futures, we'll need to create a `Waker`. As discussed in the [task wakeups section](#), `Waker`s are responsible for scheduling a task to be polled again once `wake` is called. Remember that `Waker`s tell the executor exactly which task has become ready, allowing them to poll just the futures that are ready to make progress. The easiest way to create a new `Waker` is by implementing the `ArcWake` trait and then using the `waker_ref` or `.into_waker()` functions to turn an `Arc<impl ArcWake>` into a `Waker`. Let's implement `ArcWake` for our tasks to allow them to be turned into `Waker`s and awoken:

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        // Implement `wake` by sending this task back onto the task channel
        // so that it will be polled again by the executor.
        let cloned = arc_self.clone();
        arc_self
            .task_sender
            .try_send(cloned)
            .expect("too many tasks queued");
    }
}
```

When a `Waker` is created from an `Arc<Task>`, calling `wake()` on it will cause a copy of the `Arc` to be sent onto the task channel. Our executor then needs to pick up the task and poll it. Let's implement that:

```

impl Executor {
    fn run(&self) {
        while let Ok(task) = self.ready_queue.recv() {
            // Take the future, and if it has not yet completed (is still Some),
            // poll it in an attempt to complete it.
            let mut future_slot = task.future.lock().unwrap();
            if let Some(mut future) = future_slot.take() {
                // Create a `LocalWaker` from the task itself
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&waker);
                // `BoxFuture<T>` is a type alias for
                // `Pin<Box<dyn Future<Output = T> + Send + 'static>>`.
                // We can get a `Pin<&mut dyn Future + Send + 'static>`
                // from it by calling the `Pin::as_mut` method.
                if future.as_mut().poll(context).is_pending() {
                    // We're not done processing the future, so put it
                    // back in its task to be run again in the future.
                    *future_slot = Some(future);
                }
            }
        }
    }
}

```

Congratulations! We now have a working futures executor. We can even use it to run `async/.await` code and custom futures, such as the `TimerFuture` we wrote earlier:

```

fn main() {
    let (executor, spawner) = new_executor_and_spawner();

    // Spawn a task to print before and after waiting on a timer.
    spawner.spawn(async {
        println!("howdy!");
        // Wait for our timer future to complete after two seconds.
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("done!");
    });

    // Drop the spawner so that our executor knows it is finished and won't
    // receive more incoming tasks to run.
    drop(spawner);

    // Run the executor until the task queue is empty.
    // This will print "howdy!", pause, and then print "done!".
    executor.run();
}

```

Executors and System IO

In the previous section on [The Future Trait](#), we discussed this example of a future that performed an asynchronous read on a socket:

```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            // The socket has data -- read it into a buffer and return it.
            Poll::Ready(self.socket.read_buf())
        } else {
            // The socket does not yet have data.
            //
            // Arrange for `wake` to be called once data is available.
            // When data becomes available, `wake` will be called, and the
            // user of this `Future` will know to call `poll` again and
            // receive data.
            self.socket.set_readable_callback(wake);
            Poll::Pending
        }
    }
}
```

This future will read available data on a socket, and if no data is available, it will yield to the executor, requesting that its task be awoken when the socket becomes readable again. However, it's not clear from this example how the `Socket` type is implemented, and in particular it isn't obvious how the `set_readable_callback` function works. How can we arrange for `wake()` to be called once the socket becomes readable? One option would be to have a thread that continually checks whether `socket` is readable, calling `wake()` when appropriate. However, this would be quite inefficient, requiring a separate thread for each blocked IO future. This would greatly reduce the efficiency of our async code.

In practice, this problem is solved through integration with an IO-aware system blocking primitive, such as `epoll` on Linux, `kqueue` on FreeBSD and Mac OS, IOCP on Windows, and `ports` on Fuchsia (all of which are exposed through the cross-platform Rust crate [mio](#)). These primitives all allow a thread to block on multiple asynchronous IO events, returning once one of the events completes. In practice, these APIs usually look something like this:


```

struct IoBlocker {
    /* ... */
}

struct Event {
    // An ID uniquely identifying the event that occurred and was listened for.
    id: usize,

    // A set of signals to wait for, or which occurred.
    signals: Signals,
}

impl IoBlocker {
    /// Create a new collection of asynchronous IO events to block on.
    fn new() -> Self { /* ... */ }

    /// Express an interest in a particular IO event.
    fn add_io_event_interest(
        &self,

        /// The object on which the event will occur
        io_object: &IoObject,

        /// A set of signals that may appear on the `io_object` for
        /// which an event should be triggered, paired with
        /// an ID to give to events that result from this interest.
        event: Event,
    ) { /* ... */ }

    /// Block until one of the events occurs.
    fn block(&self) -> Event { /* ... */ }
}

let mut io_blocker = IoBlocker::new();
io_blocker.add_io_event_interest(
    &socket_1,
    Event { id: 1, signals: READABLE },
);
io_blocker.add_io_event_interest(
    &socket_2,
    Event { id: 2, signals: READABLE | WRITABLE },
);
let event = io_blocker.block();

// prints e.g. "Socket 1 is now READABLE" if socket one became readable.
println!("Socket {:?} is now {:?}", event.id, event.signals);

```

Futures executors can use these primitives to provide asynchronous IO objects such as sockets that can configure callbacks to be run when a particular IO event occurs. In the case of our `SocketRead` example above, the `Socket::set_readable_callback` function might look like the following pseudocode:

```

impl Socket {
    fn set_readable_callback(&self, waker: Waker) {
        // `local_executor` is a reference to the local executor.
        // This could be provided at creation of the socket, but in practice
        // many executor implementations pass it down through thread local
        // storage for convenience.
        let local_executor = self.local_executor;

        // Unique ID for this IO object.
        let id = self.id;

        // Store the local waker in the executor's map so that it can be called
        // once the IO event arrives.
        local_executor.event_map.insert(id, waker);
        local_executor.add_io_event_interest(
            &self.socket_file_descriptor,
            Event { id, signals: READABLE },
        );
    }
}

```

We can now have just one executor thread which can receive and dispatch any IO event to the appropriate `waker`, which will wake up the corresponding task, allowing the executor to drive more tasks to completion before returning to check for more IO events (and the cycle continues...).

async/.await

In [the first chapter](#), we took a brief look at `async / .await`. This chapter will discuss `async / .await` in greater detail, explaining how it works and how `async` code differs from traditional Rust programs.

`async / .await` are special pieces of Rust syntax that make it possible to yield control of the current thread rather than blocking, allowing other code to make progress while waiting on an operation to complete.

There are two main ways to use `async`: `async fn` and `async` blocks. Each returns a value that implements the `Future` trait:

```
// `foo()` returns a type that implements `Future<Output = u8>`.
// `foo().await` will result in a value of type `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type that implements
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

As we saw in the first chapter, `async` bodies and other futures are lazy: they do nothing until they are run. The most common way to run a `Future` is to `.await` it. When `.await` is called on a `Future`, it will attempt to run it to completion. If the `Future` is blocked, it will yield control of the current thread. When more progress can be made, the `Future` will be picked up by the executor and will resume running, allowing the `.await` to resolve.

async Lifetimes

Unlike traditional functions, `async fn`s which take references or other non-`'static` arguments return a `Future` which is bounded by the lifetime of the arguments:

```
// This function:
async fn foo(x: &u8) -> u8 { *x }

// Is equivalent to this function:
fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {
    async move { *x }
}
```

This means that the future returned from an `async fn` must be `.await`ed while its non-`'static` arguments are still valid. In the common case of `.await`ing the future immediately after calling the function (as in `foo(&x).await`) this is not an issue. However, if storing the future or sending it over to another task or thread, this may be an issue.

One common workaround for turning an `async fn` with references-as-arguments into a `'static` future is to bundle the arguments with the call to the `async fn` inside an `async` block:

```
fn bad() -> impl Future<Output = u8> {
    let x = 5;
    borrow_x(&x) // ERROR: `x` does not live long enough
}

fn good() -> impl Future<Output = u8> {
    async {
        let x = 5;
        borrow_x(&x).await
    }
}
```

By moving the argument into the `async` block, we extend its lifetime to match that of the `Future` returned from the call to `good`.

async move

`async` blocks and closures allow the `move` keyword, much like normal closures. An `async move` block will take ownership of the variables it references, allowing it to outlive the current scope, but giving up the ability to share those variables with other code:

```

/// `async` block:
///
/// Multiple different `async` blocks can access the same local variable
/// so long as they're executed within the variable's scope
async fn blocks() {
    let my_string = "foo".to_string();

    let future_one = async {
        // ...
        println!("{my_string}");
    };

    let future_two = async {
        // ...
        println!("{my_string}");
    };

    // Run both futures to completion, printing "foo" twice:
    let ((), ()) = futures::join!(future_one, future_two);
}

/// `async move` block:
///
/// Only one `async move` block can access the same captured variable, since
/// captures are moved into the `Future` generated by the `async move` block.
/// However, this allows the `Future` to outlive the original scope of the
/// variable:
fn move_block() -> impl Future<Output = ()> {
    let my_string = "foo".to_string();
    async move {
        // ...
        println!("{my_string}");
    }
}

```

.awaiting on a Multithreaded Executor

Note that, when using a multithreaded `Future` executor, a `Future` may move between threads, so any variables used in `async` bodies must be able to travel between threads, as any `.await` can potentially result in a switch to a new thread.

This means that it is not safe to use `Rc`, `&RefCell` or any other types that don't implement the `Send` trait, including references to types that don't implement the `Sync` trait.

(Caveat: it is possible to use these types as long as they aren't in scope during a call to `.await`.)

Similarly, it isn't a good idea to hold a traditional non-futures-aware lock across an `.await`, as it can cause the threadpool to lock up: one task could take out a lock, `.await` and yield to the executor, allowing another task to attempt to take the lock and cause a deadlock. To avoid this, use the `Mutex` in `futures::lock` rather than the one from `std::sync`.

Pinning

To poll futures, they must be pinned using a special type called `Pin<T>`. If you read the explanation of [the `Future` trait](#) in the previous section "[Executing Futures and Tasks](#)", you'll recognize `Pin` from the `self: Pin<&mut Self>` in the `Future::poll` method's definition. But what does it mean, and why do we need it?

Why Pinning

`Pin` works in tandem with the `Unpin` marker. Pinning makes it possible to guarantee that an object implementing `!Unpin` won't ever be moved. To understand why this is necessary, we need to remember how `async / .await` works. Consider the following code:

```
let fut_one = /* ... */;  
let fut_two = /* ... */;  
async move {  
    fut_one.await;  
    fut_two.await;  
}
```

Under the hood, this creates an anonymous type that implements `Future`, providing a `poll` method that looks something like this:

```
// The `Future` type generated by our `async { ... }` block
struct AsyncFuture {
    fut_one: FutOne,
    fut_two: FutTwo,
    state: State,
}

// List of states our `async` block can be in
enum State {
    AwaitingFutOne,
    AwaitingFutTwo,
    Done,
}

impl Future for AsyncFuture {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        loop {
            match self.state {
                State::AwaitingFutOne => match self.fut_one.poll(..) {
                    Poll::Ready(()) => self.state = State::AwaitingFutTwo,
                    Poll::Pending => return Poll::Pending,
                }
                State::AwaitingFutTwo => match self.fut_two.poll(..) {
                    Poll::Ready(()) => self.state = State::Done,
                    Poll::Pending => return Poll::Pending,
                }
                State::Done => return Poll::Ready(()),
            }
        }
    }
}
```

When `poll` is first called, it will poll `fut_one`. If `fut_one` can't complete, `AsyncFuture::poll` will return. Future calls to `poll` will pick up where the previous one left off. This process continues until the future is able to successfully complete.

However, what happens if we have an `async` block that uses references? For example:

```
async {
    let mut x = [0; 128];
    let read_into_buf_fut = read_into_buf(&mut x);
    read_into_buf_fut.await;
    println!("{:?}", x);
}
```

What struct does this compile down to?


```
struct ReadIntoBuf<'a> {  
    buf: &'a mut [u8], // points to `x` below  
}  
  
struct AsyncFuture {  
    x: [u8; 128],  
    read_into_buf_fut: ReadIntoBuf<'what_lifetime?>,  
}
```

Here, the `ReadIntoBuf` future holds a reference into the other field of our structure, `x`. However, if `AsyncFuture` is moved, the location of `x` will move as well, invalidating the pointer stored in `read_into_buf_fut.buf`.

Pinning futures to a particular spot in memory prevents this problem, making it safe to create references to values inside an `async` block.

Pinning in Detail

Let's try to understand pinning by using a slightly simpler example. The problem we encounter above is a problem that ultimately boils down to how we handle references in self-referential types in Rust.

For now our example will look like this:

```

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
}

impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
        }
    }

    fn init(&mut self) {
        let self_ref: *const String = &self.a;
        self.b = self_ref;
    }

    fn a(&self) -> &str {
        &self.a
    }

    fn b(&self) -> &String {
        assert!(!self.b.is_null(), "Test::b called without Test::init being called first");
        unsafe { &*(self.b) }
    }
}

```

`Test` provides methods to get a reference to the value of the fields `a` and `b`. Since `b` is a reference to `a` we store it as a pointer since the borrowing rules of Rust don't allow us to define this lifetime. We now have what we call a self-referential struct.

Our example works fine if we don't move any of our data around as you can observe by running this example:

```

fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    println!("a: {}, b: {}", test2.a(), test2.b());
}

```

We get what we'd expect:

```
a: test1, b: test1
a: test2, b: test2
```

Let's see what happens if we swap `test1` with `test2` and thereby move the data:

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    std::mem::swap(&mut test1, &mut test2);
    println!("a: {}, b: {}", test2.a(), test2.b());
}
```

Naively, we could think that what we should get a debug print of `test1` two times like this:

```
a: test1, b: test1
a: test1, b: test1
```

But instead we get:

```
a: test1, b: test1
a: test1, b: test2
```

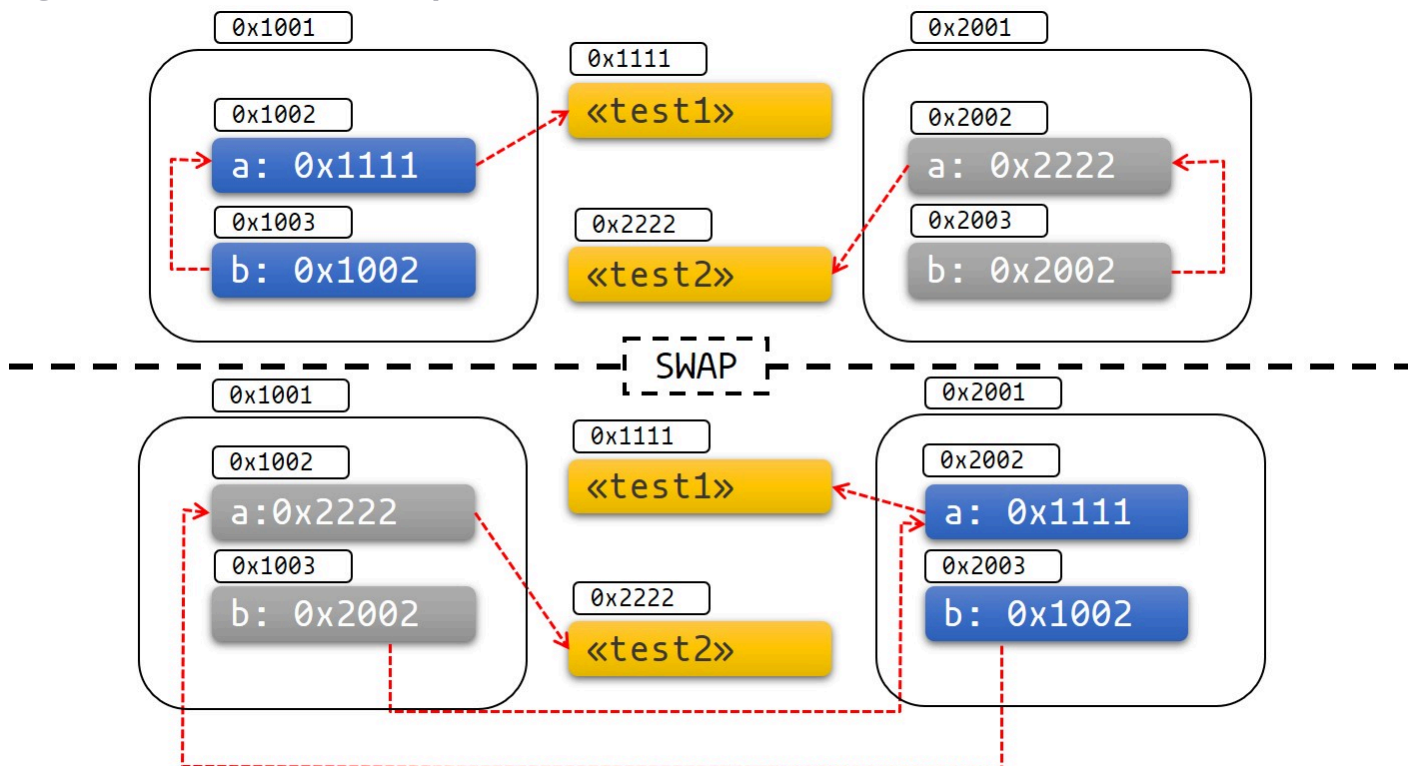
The pointer to `test2.b` still points to the old location which is inside `test1` now. The struct is not self-referential anymore, it holds a pointer to a field in a different object. That means we can't rely on the lifetime of `test2.b` to be tied to the lifetime of `test2` anymore.

If you're still not convinced, this should at least convince you:

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    std::mem::swap(&mut test1, &mut test2);
    test1.a = "I've totally changed now!".to_string();
    println!("a: {}, b: {}", test2.a(), test2.b());
}
```

The diagram below can help visualize what's going on:

Fig 1: Before and after swap

It's easy to get this to show undefined behavior and fail in other spectacular ways as well.

Pinning in Practice

Let's see how pinning and the `Pin` type can help us solve this problem.

The `Pin` type wraps pointer types, guaranteeing that the values behind the pointer won't be moved if it is not implementing `Unpin`. For example, `Pin<&mut T>`, `Pin<&T>`, `Pin<Box<T>>` all guarantee that `T` won't be moved if `T: !Unpin`.

Most types don't have a problem being moved. These types implement a trait called `Unpin`. Pointers to `Unpin` types can be freely placed into or taken out of `Pin`. For example, `u8` is `Unpin`, so `Pin<&mut u8>` behaves just like a normal `&mut u8`.

However, types that can't be moved after they're pinned have a marker called `!Unpin`. Futures created by `async/await` are an example of this.

Pinning to the Stack

Back to our example. We can solve our problem by using `Pin`. Let's take a look at what our example would look like if we required a pinned pointer instead:

```

use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}

impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned, // This makes our type `!Unpin`
        }
    }

    fn init(self: Pin<&mut Self>) {
        let self_ptr: *const String = &self.a;
        let this = unsafe { self.get_unchecked_mut() };
        this.b = self_ptr;
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        assert!(!self.b.is_null(), "Test::b called without Test::init being called first");
        unsafe { &*(self.b) }
    }
}

```

Pinning an object to the stack will always be `unsafe` if our type implements `!Unpin`. You can use a crate like [pin_utils](#) to avoid writing our own `unsafe` code when pinning to the stack.

Below, we pin the objects `test1` and `test2` to the stack:

```
pub fn main() {
    // test1 is safe to move before we initialize it
    let mut test1 = Test::new("test1");
    // Notice how we shadow `test1` to prevent it from being accessed again
    let mut test1 = unsafe { Pin::new_unchecked(&mut test1) };
    Test::init(test1.as_mut());

    let mut test2 = Test::new("test2");
    let mut test2 = unsafe { Pin::new_unchecked(&mut test2) };
    Test::init(test2.as_mut());

    println!("a: {}, b: {}", Test::a(test1.as_ref()), Test::b(test1.as_ref()));
    println!("a: {}, b: {}", Test::a(test2.as_ref()), Test::b(test2.as_ref()));
}
```

Now, if we try to move our data now we get a compilation error:

```
pub fn main() {
    let mut test1 = Test::new("test1");
    let mut test1 = unsafe { Pin::new_unchecked(&mut test1) };
    Test::init(test1.as_mut());

    let mut test2 = Test::new("test2");
    let mut test2 = unsafe { Pin::new_unchecked(&mut test2) };
    Test::init(test2.as_mut());

    println!("a: {}, b: {}", Test::a(test1.as_ref()), Test::b(test1.as_ref()));
    std::mem::swap(test1.get_mut(), test2.get_mut());
    println!("a: {}, b: {}", Test::a(test2.as_ref()), Test::b(test2.as_ref()));
}
```

The type system prevents us from moving the data, as follows:

```

error[E0277]: `PhantomPinned` cannot be unpinned
  --> src\test.rs:56:30
   |
56 |         std::mem::swap(test1.get_mut(), test2.get_mut());
   |                                ^^^^^^^^^ within `test1::Test`, the trait `Unpin`
is not implemented for `PhantomPinned`
   |
   = note: consider using `Box::pin`
note: required because it appears within the type `test1::Test`
  --> src\test.rs:7:8
   |
 7 |     struct Test {
   |           ^^^^^
note: required by a bound in `std::pin::Pin:::<&'a mut T>::get_mut`
  --> <...>rustlib/src/rust\library\core\src\pin.rs:748:12
   |
748 |         T: Unpin,
   |           ^^^^^^ required by this bound in `std::pin::Pin:::<&'a mut
T>::get_mut`

```

It's important to note that stack pinning will always rely on guarantees you give when writing `unsafe`. While we know that the *pointee* of `&'a mut T` is pinned for the lifetime of `'a` we can't know if the data `&'a mut T` points to isn't moved after `'a` ends. If it does it will violate the Pin contract.

A mistake that is easy to make is forgetting to shadow the original variable since you could drop the `pin` and move the data after `&'a mut T` like shown below (which violates the Pin contract):

```

fn main() {
    let mut test1 = Test::new("test1");
    let mut test1_pin = unsafe { Pin::new_unchecked(&mut test1) };
    Test::init(test1_pin.as_mut());

    drop(test1_pin);
    println!(r#"test1.b points to "test1": {:?}..."#, test1.b);

    let mut test2 = Test::new("test2");
    mem::swap(&mut test1, &mut test2);
    println!("... and now it points nowhere: {:?}", test1.b);
}

```

Pinning to the Heap

Pinning an `!Unpin` type to the heap gives our data a stable address so we know that the data we point to can't move after it's pinned. In contrast to stack pinning, we know that the data will be pinned for the lifetime of the object.

```
use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}

impl Test {
    fn new(txt: &str) -> Pin<Box<Self>> {
        let t = Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned,
        };
        let mut boxed = Box::pin(t);
        let self_ptr: *const String = &boxed.a;
        unsafe { boxed.as_mut().get_unchecked_mut().b = self_ptr };

        boxed
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        unsafe { &*(self.b) }
    }
}

pub fn main() {
    let test1 = Test::new("test1");
    let test2 = Test::new("test2");

    println!("a: {}, b: {}", test1.as_ref().a(), test1.as_ref().b());
    println!("a: {}, b: {}", test2.as_ref().a(), test2.as_ref().b());
}
```

Some functions require the futures they work with to be `Unpin`. To use a `Future` or `Stream` that isn't `Unpin` with a function that requires `Unpin` types, you'll first have to pin the value using either `Box::pin` (to create a `Pin<Box<T>>`) or the `pin_utils::pin_mut!` macro (to

create a `Pin<&mut T>`). `Pin<Box<Fut>>` and `Pin<&mut Fut>` can both be used as futures, and both implement `Unpin`.

For example:

```
use pin_utils::pin_mut; // `pin_utils` is a handy crate available on crates.io

// A function which takes a `Future` that implements `Unpin`.
fn execute_unpin_future(x: impl Future<Output = ()> + Unpin) { /* ... */ }

let fut = async { /* ... */ };
execute_unpin_future(fut); // Error: `fut` does not implement `Unpin` trait

// Pinning with `Box`:
let fut = async { /* ... */ };
let fut = Box::pin(fut);
execute_unpin_future(fut); // OK

// Pinning with `pin_mut!`:
let fut = async { /* ... */ };
pin_mut!(fut);
execute_unpin_future(fut); // OK
```

Summary

1. If `T: Unpin` (which is the default), then `Pin<'a, T>` is entirely equivalent to `&'a mut T`. In other words: `Unpin` means it's OK for this type to be moved even when pinned, so `Pin` will have no effect on such a type.
2. Getting a `&mut T` to a pinned `T` requires `unsafe` if `T: !Unpin`.
3. Most standard library types implement `Unpin`. The same goes for most "normal" types you encounter in Rust. A `Future` generated by `async/await` is an exception to this rule.
4. You can add a `!Unpin` bound on a type on nightly with a feature flag, or by adding `std::marker::PhantomPinned` to your type on stable.
5. You can either pin data to the stack or to the heap.
6. Pinning a `!Unpin` object to the stack requires `unsafe`.
7. Pinning a `!Unpin` object to the heap does not require `unsafe`. There is a shortcut for doing this using `Box::pin`.

8. For pinned data where $\tau: !\text{Unpin}$ you have to maintain the invariant that its memory will not get invalidated or repurposed *from the moment it gets pinned until when `drop` is called*. This is an important part of the *pin contract*.

The Stream Trait

The `Stream` trait is similar to `Future` but can yield multiple values before completing, similar to the `Iterator` trait from the standard library:

```
trait Stream {
    /// The type of the value yielded by the stream.
    type Item;

    /// Attempt to resolve the next item in the stream.
    /// Returns `Poll::Pending` if not ready, `Poll::Ready(Some(x))` if a value
    /// is ready, and `Poll::Ready(None)` if the stream has completed.
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}
```

One common example of a `Stream` is the `Receiver` for the channel type from the `futures` crate. It will yield `Some(val)` every time a value is sent from the `Sender` end, and will yield `None` once the `Sender` has been dropped and all pending messages have been received:

```
async fn send_recv() {
    const BUFFER_SIZE: usize = 10;
    let (mut tx, mut rx) = mpsc::channel::<i32>(BUFFER_SIZE);

    tx.send(1).await.unwrap();
    tx.send(2).await.unwrap();
    drop(tx);

    // `StreamExt::next` is similar to `Iterator::next`, but returns a
    // type that implements `Future<Output = Option<T>>`.
    assert_eq!(Some(1), rx.next().await);
    assert_eq!(Some(2), rx.next().await);
    assert_eq!(None, rx.next().await);
}
```

Iteration and Concurrency

Similar to synchronous `Iterator`s, there are many different ways to iterate over and process the values in a `Stream`. There are combinator-style methods such as `map`, `filter`, and `fold`, and their early-exit-on-error cousins `try_map`, `try_filter`, and `try_fold`.

Unfortunately, `for` loops are not usable with `Stream`s, but for imperative-style code, `while let` and the `next / try_next` functions can be used:

```
async fn sum_with_next(mut stream: Pin<&mut dyn Stream<Item = i32>>>) -> i32 {
    use futures::stream::StreamExt; // for `next`
    let mut sum = 0;
    while let Some(item) = stream.next().await {
        sum += item;
    }
    sum
}

async fn sum_with_try_next(
    mut stream: Pin<&mut dyn Stream<Item = Result<i32, io::Error>>>,
) -> Result<i32, io::Error> {
    use futures::stream::TryStreamExt; // for `try_next`
    let mut sum = 0;
    while let Some(item) = stream.try_next().await? {
        sum += item;
    }
    Ok(sum)
}
```

However, if we're just processing one element at a time, we're potentially leaving behind opportunity for concurrency, which is, after all, why we're writing `async` code in the first place. To process multiple items from a stream concurrently, use the `for_each_concurrent` and `try_for_each_concurrent` methods:

```
async fn jump_around(
    mut stream: Pin<&mut dyn Stream<Item = Result<u8, io::Error>>>,
) -> Result<(), io::Error> {
    use futures::stream::TryStreamExt; // for `try_for_each_concurrent`
    const MAX_CONCURRENT_JUMPERS: usize = 100;

    stream.try_for_each_concurrent(MAX_CONCURRENT_JUMPERS, |num| async move {
        jump_n_times(num).await?;
        report_n_jumps(num).await?;
        Ok(())
    }).await?;

    Ok(())
}
```

Executing Multiple Futures at a Time

Up until now, we've mostly executed futures by using `.await`, which blocks the current task until a particular `Future` completes. However, real asynchronous applications often need to execute several different operations concurrently.

In this chapter, we'll cover some ways to execute multiple asynchronous operations at the same time:

- `join!` : waits for futures to all complete
- `select!` : waits for one of several futures to complete
- Spawning: creates a top-level task which ambiently runs a future to completion
- `FuturesUnordered` : a group of futures which yields the result of each subfuture

join!

The `futures::join` macro makes it possible to wait for multiple different futures to complete while executing them all concurrently.

join!

When performing multiple asynchronous operations, it's tempting to simply `.await` them in a series:

```
async fn get_book_and_music() -> (Book, Music) {
    let book = get_book().await;
    let music = get_music().await;
    (book, music)
}
```

However, this will be slower than necessary, since it won't start trying to `get_music` until after `get_book` has completed. In some other languages, futures are ambiently run to completion, so two operations can be run concurrently by first calling each `async fn` to start the futures, and then awaiting them both:

```
// WRONG -- don't do this
async fn get_book_and_music() -> (Book, Music) {
    let book_future = get_book();
    let music_future = get_music();
    (book_future.await, music_future.await)
}
```

However, Rust futures won't do any work until they're actively `.await` ed. This means that the two code snippets above will both run `book_future` and `music_future` in series rather than running them concurrently. To correctly run the two futures concurrently, use `futures::join!`:

```
use futures::join;

async fn get_book_and_music() -> (Book, Music) {
    let book_fut = get_book();
    let music_fut = get_music();
    join!(book_fut, music_fut)
}
```

The value returned by `join!` is a tuple containing the output of each `Future` passed in.

try_join!

For futures which return `Result`, consider using `try_join!` rather than `join!`. Since `join!` only completes once all subfutures have completed, it'll continue processing other futures even after one of its subfutures has returned an `Err`.

Unlike `join!`, `try_join!` will complete immediately if one of the subfutures returns an error.

```
use futures::try_join;

async fn get_book() -> Result<Book, String> { /* ... */ Ok(Book) }
async fn get_music() -> Result<Music, String> { /* ... */ Ok(Music) }

async fn get_book_and_music() -> Result<(Book, Music), String> {
    let book_fut = get_book();
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}
```

Note that the futures passed to `try_join!` must all have the same error type. Consider using the `.map_err(|e| ...)` and `.err_into()` functions from `futures::future::TryFutureExt` to consolidate the error types:

```
use futures::{
    future::TryFutureExt,
    try_join,
};

async fn get_book() -> Result<Book, ()> { /* ... */ Ok(Book) }
async fn get_music() -> Result<Music, String> { /* ... */ Ok(Music) }

async fn get_book_and_music() -> Result<(Book, Music), String> {
    let book_fut = get_book().map_err(|()| "Unable to get book".to_string());
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}
```


select!

The `futures::select` macro runs multiple futures simultaneously, allowing the user to respond as soon as any future completes.

```
use futures::{
    future::FutureExt, // for `.fuse()`
    pin_mut,
    select,
};

async fn task_one() { /* ... */ }
async fn task_two() { /* ... */ }

async fn race_tasks() {
    let t1 = task_one().fuse();
    let t2 = task_two().fuse();

    pin_mut!(t1, t2);

    select! {
        () = t1 => println!("task one completed first"),
        () = t2 => println!("task two completed first"),
    }
}
```

The function above will run both `t1` and `t2` concurrently. When either `t1` or `t2` finishes, the corresponding handler will call `println!`, and the function will end without completing the remaining task.

The basic syntax for `select` is `<pattern> = <expression> => <code>, ,` repeated for as many futures as you would like to `select` over.

default => ... and complete => ...

`select` also supports `default` and `complete` branches.

A `default` branch will run if none of the futures being `select` ed over are yet complete. A `select` with a `default` branch will therefore always return immediately, since `default` will be run if none of the other futures are ready.

`complete` branches can be used to handle the case where all futures being `select` ed over have completed and will no longer make progress. This is often handy when looping over a

```

select! .

use futures::{future, select};

async fn count() {
    let mut a_fut = future::ready(4);
    let mut b_fut = future::ready(6);
    let mut total = 0;

    loop {
        select! {
            a = a_fut => total += a,
            b = b_fut => total += b,
            complete => break,
            default => unreachable!(), // never runs (futures are ready, then
complete)
        };
    }
    assert_eq!(total, 10);
}

```

Interaction with Unpin and FusedFuture

One thing you may have noticed in the first example above is that we had to call `.fuse()` on the futures returned by the two `async fn`s, as well as pinning them with `pin_mut`. Both of these calls are necessary because the futures used in `select` must implement both the `Unpin` trait and the `FusedFuture` trait.

`Unpin` is necessary because the futures used by `select` are not taken by value, but by mutable reference. By not taking ownership of the future, uncompleted futures can be used again after the call to `select`.

Similarly, the `FusedFuture` trait is required because `select` must not poll a future after it has completed. `FusedFuture` is implemented by futures which track whether or not they have completed. This makes it possible to use `select` in a loop, only polling the futures which still have yet to complete. This can be seen in the example above, where `a_fut` or `b_fut` will have completed the second time through the loop. Because the future returned by `future::ready` implements `FusedFuture`, it's able to tell `select` not to poll it again.

Note that streams have a corresponding `FusedStream` trait. Streams which implement this trait or have been wrapped using `.fuse()` will yield `FusedFuture` futures from their `.next()` / `.try_next()` combinators.

```

use futures::{
    stream::{Stream, StreamExt, FusedStream},
    select,
};

async fn add_two_streams(
    mut s1: impl Stream<Item = u8> + FusedStream + Unpin,
    mut s2: impl Stream<Item = u8> + FusedStream + Unpin,
) -> u8 {
    let mut total = 0;

    loop {
        let item = select! {
            x = s1.next() => x,
            x = s2.next() => x,
            complete => break,
        };
        if let Some(next_num) = item {
            total += next_num;
        }
    }

    total
}

```

Concurrent tasks in a `select` loop with `Fuse` and `FuturesUnordered`

One somewhat hard-to-discover but handy function is `Fuse::terminated()`, which allows constructing an empty future which is already terminated, and can later be filled in with a future that needs to be run.

This can be handy when there's a task that needs to be run during a `select` loop but which is created inside the `select` loop itself.

Note the use of the `.select_next_some()` function. This can be used with `select` to only run the branch for `Some(_)` values returned from the stream, ignoring `None`s.

```

use futures::{
    future::{Fuse, FusedFuture, FutureExt},
    stream::{FusedStream, Stream, StreamExt},
    pin_mut,
    select,
};

async fn get_new_num() -> u8 { /* ... */ }

async fn run_on_new_num(_: u8) { /* ... */ }

async fn run_loop(
    mut interval_timer: impl Stream<Item = ()> + FusedStream + Unpin,
    starting_num: u8,
) {
    let run_on_new_num_fut = run_on_new_num(starting_num).fuse();
    let get_new_num_fut = Fuse::terminated();
    pin_mut!(run_on_new_num_fut, get_new_num_fut);
    loop {
        select! {
            () = interval_timer.select_next_some() => {
                // The timer has elapsed. Start a new `get_new_num_fut`
                // if one was not already running.
                if get_new_num_fut.is_terminated() {
                    get_new_num_fut.set(get_new_num().fuse());
                }
            },
            new_num = get_new_num_fut => {
                // A new number has arrived -- start a new `run_on_new_num_fut`,
                // dropping the old one.
                run_on_new_num_fut.set(run_on_new_num(new_num).fuse());
            },
        }
        // Run the `run_on_new_num_fut`
        () = run_on_new_num_fut => {},
        // panic if everything completed, since the `interval_timer` should
        // keep yielding values indefinitely.
        complete => panic!("`interval_timer` completed unexpectedly"),
    }
}

```

When many copies of the same future need to be run simultaneously, use the `FuturesUnordered` type. The following example is similar to the one above, but will run each copy of `run_on_new_num_fut` to completion, rather than aborting them when a new one is created. It will also print out a value returned by `run_on_new_num_fut`.

```

use futures::{
    future::{Fuse, FusedFuture, FutureExt},
    stream::{FusedStream, FuturesUnordered, Stream, StreamExt},
    pin_mut,
    select,
};

async fn get_new_num() -> u8 { /* ... */ 5 }

async fn run_on_new_num(_: u8) -> u8 { /* ... */ 5 }

async fn run_loop(
    mut interval_timer: impl Stream<Item = ()> + FusedStream + Unpin,
    starting_num: u8,
) {
    let mut run_on_new_num_futs = FuturesUnordered::new();
    run_on_new_num_futs.push(run_on_new_num(starting_num));
    let get_new_num_fut = Fuse::terminated();
    pin_mut!(get_new_num_fut);
    loop {
        select! {
            () = interval_timer.select_next_some() => {
                // The timer has elapsed. Start a new `get_new_num_fut`
                // if one was not already running.
                if get_new_num_fut.is_terminated() {
                    get_new_num_fut.set(get_new_num().fuse());
                }
            },
            new_num = get_new_num_fut => {
                // A new number has arrived -- start a new `run_on_new_num_fut`.
                run_on_new_num_futs.push(run_on_new_num(new_num));
            },
        }
        // Run the `run_on_new_num_futs` and check if any have completed
        res = run_on_new_num_futs.select_next_some() => {
            println!("run_on_new_num_fut returned {:?}", res);
        },
        // panic if everything completed, since the `interval_timer` should
        // keep yielding values indefinitely.
        complete => panic!("`interval_timer` completed unexpectedly"),
    }
}

```

Spawning

Spawning allows you to run a new asynchronous task in the background. This allows us to continue executing other code while it runs.

Say we have a web server that wants to accept connections without blocking the main thread. To achieve this, we can use the `async_std::task::spawn` function to create and run a new task that handles the connections. This function takes a future and returns a `JoinHandle`, which can be used to wait for the result of the task once it's completed.

```
use async_std::{task, net::TcpListener, net::TcpStream};
use futures::AsyncWriteExt;

async fn process_request(stream: &mut TcpStream) -> Result<(), std::io::Error>{
    stream.write_all(b"HTTP/1.1 200 OK\r\n\r\n").await?;
    stream.write_all(b"Hello World").await?;
    Ok(())
}

async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").await.unwrap();
    loop {
        // Accept a new connection
        let (mut stream, _) = listener.accept().await.unwrap();
        // Now process this request without blocking the main loop
        task::spawn(async move {process_request(&mut stream).await});
    }
}
```

The `JoinHandle` returned by `spawn` implements the `Future` trait, so we can `.await` it to get the result of the task. This will block the current task until the spawned task completes. If the task is not awaited, your program will continue executing without waiting for the task, cancelling it if the function is completed before the task is finished.

```
use futures::future::join_all;
async fn task_spawner(){
    let tasks = vec![
        task::spawn(my_task(Duration::from_secs(1))),
        task::spawn(my_task(Duration::from_secs(2))),
        task::spawn(my_task(Duration::from_secs(3))),
    ];
    // If we do not await these tasks and the function finishes, they will be
    dropped
    join_all(tasks).await;
}
```

To communicate between the main task and the spawned task, we can use channels provided by the async runtime used.

Workarounds to Know and Love

Rust's `async` support is still fairly new, and there are a handful of highly-requested features still under active development, as well as some subpar diagnostics. This chapter will discuss some common pain points and explain how to work around them.

Send Approximation

Some `async fn` state machines are safe to be sent across threads, while others are not. Whether or not an `async fn Future` is `Send` is determined by whether a non-`Send` type is held across an `.await` point. The compiler does its best to approximate when values may be held across an `.await` point, but this analysis is too conservative in a number of places today.

For example, consider a simple non-`Send` type, perhaps a type which contains an `Rc`:

```
use std::rc::Rc;

#[derive(Default)]
struct NotSend(Rc<()>);
```

Variables of type `NotSend` can briefly appear as temporaries in `async fn`s even when the resulting `Future` type returned by the `async fn` must be `Send`:

```
async fn bar() {}
async fn foo() {
    NotSend::default();
    bar().await;
}

fn require_send(_: impl Send) {}

fn main() {
    require_send(foo());
}
```

However, if we change `foo` to store `NotSend` in a variable, this example no longer compiles:

```
async fn foo() {
    let x = NotSend::default();
    bar().await;
}
```

```

error[E0277]: `std::rc::Rc<()>` cannot be sent between threads safely
--> src/main.rs:15:5
   |
15 |     require_send(foo());
   |     ^^^^^^^^^^^^^^^^^ `std::rc::Rc<()>` cannot be sent between threads safely
   |
   = help: within `impl std::future::Future`, the trait `std::marker::Send` is not
implemented for `std::rc::Rc<()>`
   = note: required because it appears within the type `NotSend`
   = note: required because it appears within the type `{NotSend, impl
std::future::Future, ()}`
   = note: required because it appears within the type `[static
generator@src/main.rs:7:16: 10:2 {NotSend, impl std::future::Future, ()}]`
   = note: required because it appears within the type
`std::future::GenFuture<[static generator@src/main.rs:7:16: 10:2 {NotSend, impl
std::future::Future, ()}]>`
   = note: required because it appears within the type `impl std::future::Future`
   = note: required because it appears within the type `impl std::future::Future`
note: required by `require_send`
--> src/main.rs:12:1
   |
12 | fn require_send(_: impl Send) {}
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

error: aborting due to previous error

For more information about this error, try `rustc --explain E0277`.

This error is correct. If we store `x` into a variable, it won't be dropped until after the `.await`, at which point the `async fn` may be running on a different thread. Since `Rc` is not `Send`, allowing it to travel across threads would be unsound. One simple solution to this would be to drop the `Rc` before the `.await`, but unfortunately that does not work today.

In order to successfully work around this issue, you may have to introduce a block scope encapsulating any non-`Send` variables. This makes it easier for the compiler to tell that these variables do not live across an `.await` point.

```

async fn foo() {
    {
        let x = NotSend::default();
    }
    bar().await;
}

```

Recursion

Internally, `async fn` creates a state machine type containing each sub-`Future` being `.await`ed. This makes recursive `async fn`s a little tricky, since the resulting state machine type has to contain itself:

```
// This function:
async fn foo() {
    step_one().await;
    step_two().await;
}

// generates a type like this:
enum Foo {
    First(StepOne),
    Second(StepTwo),
}

// So this function:
async fn recursive() {
    recursive().await;
    recursive().await;
}

// generates a type like this:
enum Recursive {
    First(Recursive),
    Second(Recursive),
}
```

This won't work—we've created an infinitely-sized type! The compiler will complain:

```
error[E0733]: recursion in an async fn requires boxing
  --> src/lib.rs:1:1
   |
1  | async fn recursive() {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: a recursive `async fn` call must introduce indirection such as
`Box::pin` to avoid an infinitely sized future
```

In order to allow this, we have to introduce an indirection using `Box`.

Prior to Rust 1.77, due to compiler limitations, just wrapping the calls to `recursive()` in `Box::pin` isn't enough. To make this work, we have to make `recursive` into a non-`async` function which returns a `.boxed()` `async` block:

```
use futures::future::{BoxFuture, FutureExt};

fn recursive() -> BoxFuture<'static, ()> {
    async move {
        recursive().await;
        recursive().await;
    }.boxed()
}
```

In newer version of Rust, [that compiler limitation has been lifted](#).

Since Rust 1.77, support for recursion in `async fn` with allocation indirection [becomes stable](#), so recursive calls are permitted so long as they use some form of indirection to avoid an infinite size for the state of the function.

This means that code like this now works:

```
async fn recursive_pinned() {
    Box::pin(recursive_pinned()).await;
    Box::pin(recursive_pinned()).await;
}
```

async in Traits

Currently, `async fn` cannot be used in traits on the stable release of Rust. Since the 17th November 2022, an MVP of `async-fn-in-trait` is available on the nightly version of the compiler tool chain, [see here for details](#).

In the meantime, there is a work around for the stable tool chain using the [async-trait crate](#) from [crates.io](#).

Note that using these trait methods will result in a heap allocation per-function-call. This is not a significant cost for the vast majority of applications, but should be considered when deciding whether to use this functionality in the public API of a low-level function that is expected to be called millions of times a second.

Last updates: <https://blog.rust-lang.org/2023/12/21/async-fn-rpit-in-traits.html>

The Async Ecosystem

Rust currently provides only the bare essentials for writing async code. Importantly, executors, tasks, reactors, combinators, and low-level I/O futures and traits are not yet provided in the standard library. In the meantime, community-provided async ecosystems fill in these gaps.

The Async Foundations Team is interested in extending examples in the Async Book to cover multiple runtimes. If you're interested in contributing to this project, please reach out to us on [Zulip](#).

Async Runtimes

Async runtimes are libraries used for executing async applications. Runtimes usually bundle together a *reactor* with one or more *executors*. Reactors provide subscription mechanisms for external events, like async I/O, interprocess communication, and timers. In an async runtime, subscribers are typically futures representing low-level I/O operations. Executors handle the scheduling and execution of tasks. They keep track of running and suspended tasks, poll futures to completion, and wake tasks when they can make progress. The word "executor" is frequently used interchangeably with "runtime". Here, we use the word "ecosystem" to describe a runtime bundled with compatible traits and features.

Community-Provided Async Crates

The Futures Crate

The [futures crate](#) contains traits and functions useful for writing async code. This includes the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits, and utilities such as combinators. These utilities and traits may eventually become part of the standard library.

`futures` has its own executor, but not its own reactor, so it does not support execution of async I/O or timer futures. For this reason, it's not considered a full runtime. A common choice is to use utilities from `futures` with an executor from another crate.

Popular Async Runtimes

There is no asynchronous runtime in the standard library, and none are officially recommended. The following crates provide popular runtimes.

- [Tokio](#): A popular async ecosystem with HTTP, gRPC, and tracing frameworks.
- [async-std](#): A crate that provides asynchronous counterparts to standard library components.
- [smol](#): A small, simplified async runtime. Provides the `Async` trait that can be used to wrap structs like `UnixStream` or `TcpListener`.
- [fuchsia-async](#): An executor for use in the Fuchsia OS.

Determining Ecosystem Compatibility

Not all async applications, frameworks, and libraries are compatible with each other, or with every OS or platform. Most async code can be used with any ecosystem, but some frameworks and libraries require the use of a specific ecosystem. Ecosystem constraints are not always documented, but there are several rules of thumb to determine whether a library, trait, or function depends on a specific ecosystem.

Any async code that interacts with async I/O, timers, interprocess communication, or tasks generally depends on a specific async executor or reactor. All other async code, such as async expressions, combinators, synchronization types, and streams are usually ecosystem independent, provided that any nested futures are also ecosystem independent. Before beginning a project, it's recommended to research relevant async frameworks and libraries to ensure compatibility with your chosen runtime and with each other.

Notably, `Tokio` uses the `mio` reactor and defines its own versions of async I/O traits, including `AsyncRead` and `AsyncWrite`. On its own, it's not compatible with `async-std` and `smol`, which rely on the [async-executor crate](#), and the `AsyncRead` and `AsyncWrite` traits defined in `futures`.

Conflicting runtime requirements can sometimes be resolved by compatibility layers that allow you to call code written for one runtime within another. For example, the [async_compat crate](#) provides a compatibility layer between `Tokio` and other runtimes.

Libraries exposing async APIs should not depend on a specific executor or reactor, unless they need to spawn tasks or define their own async I/O or timer futures. Ideally, only binaries should be responsible for scheduling and running tasks.

Single Threaded vs Multi-Threaded Executors

Async executors can be single-threaded or multi-threaded. For example, the `async-executor` crate has both a single-threaded `LocalExecutor` and a multi-threaded `Executor`.

A multi-threaded executor makes progress on several tasks simultaneously. It can speed up the execution greatly for workloads with many tasks, but synchronizing data between tasks is usually more expensive. It is recommended to measure performance for your application when you are choosing between a single- and a multi-threaded runtime.

Tasks can either be run on the thread that created them or on a separate thread. Async runtimes often provide functionality for spawning tasks onto separate threads. Even if tasks are executed on separate threads, they should still be non-blocking. In order to schedule tasks on a multi-threaded executor, they must also be `Send`. Some runtimes provide functions for spawning non-`Send` tasks, which ensures every task is executed on the thread that spawned it. They may also provide functions for spawning blocking tasks onto dedicated threads, which is useful for running blocking synchronous code from other libraries.

Final Project: Building a Concurrent Web Server with Async Rust

In this chapter, we'll use asynchronous Rust to modify the Rust book's [single-threaded web server](#) to serve requests concurrently.

Recap

Here's what the code looked like at the end of the lesson.

```
src/main.rs:
```

```

use std::fs;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;

fn main() {
    // Listen for incoming TCP connections on localhost port 7878
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    // Block forever, handling each request that arrives at this IP address
    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    // Read the first 1024 bytes of data from the stream
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    // Respond with greetings or a 404,
    // depending on the data in the request
    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };
    let contents = fs::read_to_string(filename).unwrap();

    // Write response back to the stream,
    // and flush the stream to ensure the response is sent back to the client
    let response = format!("{status_line}{contents}");
    stream.write_all(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}

```

hello.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

404.html :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

If you run the server with `cargo run` and visit `127.0.0.1:7878` in your browser, you'll be greeted with a friendly message from Ferris!

Running Asynchronous Code

An HTTP server should be able to serve multiple clients concurrently; that is, it should not wait for previous requests to complete before handling the current request. The book [solves this problem](#) by creating a thread pool where each connection is handled on its own thread. Here, instead of improving throughput by adding threads, we'll achieve the same effect using asynchronous code.

Let's modify `handle_connection` to return a future by declaring it an `async fn`:

```
async fn handle_connection(mut stream: TcpStream) {
    //<-- snip -->
}
```

Adding `async` to the function declaration changes its return type from the unit type `()` to a type that implements `Future<Output=()>`.

If we try to compile this, the compiler warns us that it will not work:

```
$ cargo check
    Checking async-rust v0.1.0 (file:///projects/async-rust)
warning: unused implementer of `std::future::Future` that must be used
--> src/main.rs:12:9
12 |         handle_connection(stream);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    = note: `#[warn(unused_must_use)]` on by default
    = note: futures do nothing unless you `.await` or poll them
```

Because we haven't `await`ed or `poll`ed the result of `handle_connection`, it'll never run. If you run the server and visit `127.0.0.1:7878` in a browser, you'll see that the connection is refused; our server is not handling requests.

We can't `await` or `poll` futures within synchronous code by itself. We'll need an asynchronous runtime to handle scheduling and running futures to completion. Please consult the [section on choosing a runtime](#) for more information on asynchronous runtimes, executors, and reactors. Any of the runtimes listed will work for this project, but for these examples, we've chosen to use the `async-std` crate.

Adding an Async Runtime

The following example will demonstrate refactoring synchronous code to use an async runtime; here, `async-std`. The `#[async_std::main]` attribute from `async-std` allows us to write an asynchronous main function. To use it, enable the `attributes` feature of `async-std` in `Cargo.toml`:

```
[dependencies.async-std]
version = "1.6"
features = ["attributes"]
```

As a first step, we'll switch to an asynchronous main function, and `await` the future returned by the async version of `handle_connection`. Then, we'll test how the server responds. Here's what that would look like:

```
#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    for stream in listener.incoming() {
        let stream = stream.unwrap();
        // Warning: This is not concurrent!
        handle_connection(stream).await;
    }
}
```

Now, let's test to see if our server can handle connections concurrently. Simply making `handle_connection` asynchronous doesn't mean that the server can handle multiple connections at the same time, and we'll soon see why.

To illustrate this, let's simulate a slow request. When a client makes a request to `127.0.0.1:7878/sleep`, our server will sleep for 5 seconds:

```

use std::time::Duration;
use async_std::task;

async fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        task::sleep(Duration::from_secs(5)).await;
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };
    let contents = fs::read_to_string(filename).unwrap();

    let response = format!("{status_line}{contents}");
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}

```

This is very similar to the [simulation of a slow request](#) from the Book, but with one important difference: we're using the non-blocking function `async_std::task::sleep` instead of the blocking function `std::thread::sleep`. It's important to remember that even if a piece of code is run within an `async fn` and `await`ed, it may still block. To test whether our server handles connections concurrently, we'll need to ensure that `handle_connection` is non-blocking.

If you run the server, you'll see that a request to `127.0.0.1:7878/sleep` will block any other incoming requests for 5 seconds! This is because there are no other concurrent tasks that can make progress while we are `await`ing the result of `handle_connection`. In the next section, we'll see how to use `async` code to handle connections concurrently.

Handling Connections Concurrently

The problem with our code so far is that `listener.incoming()` is a blocking iterator. The executor can't run other futures while `listener` waits on incoming connections, and we can't handle a new connection until we're done with the previous one.

In order to fix this, we'll transform `listener.incoming()` from a blocking iterator to a non-blocking Stream. Streams are similar to iterators, but can be consumed asynchronously. For more information, see the [chapter on Streams](#).

Let's replace our blocking `std::net::TcpListener` with the non-blocking `async_std::net::TcpListener`, and update our connection handler to accept an `async_std::net::TcpStream`:

```
use async_std::prelude::*;

async fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).await.unwrap();

    //<-- snip -->
    stream.write(response.as_bytes()).await.unwrap();
    stream.flush().await.unwrap();
}
```

The asynchronous version of `TcpListener` implements the `Stream` trait for `listener.incoming()`, a change which provides two benefits. The first is that `listener.incoming()` no longer blocks the executor. The executor can now yield to other pending futures while there are no incoming TCP connections to be processed.

The second benefit is that elements from the Stream can optionally be processed concurrently, using a Stream's `for_each_concurrent` method. Here, we'll take advantage of this method to handle each incoming request concurrently. We'll need to import the `stream` trait from the `futures` crate, so our `Cargo.toml` now looks like this:

```
+ [dependencies]
+ futures = "0.3"

[dependencies.async-std]
version = "1.6"
features = ["attributes"]
```

Now, we can handle each connection concurrently by passing `handle_connection` in through a closure function. The closure function takes ownership of each `TcpStream`, and is run as soon

as a new `TcpStream` becomes available. As long as `handle_connection` does not block, a slow request will no longer prevent other requests from completing.

```
use async_std::net::TcpListener;
use async_std::net::TcpStream;
use futures::stream::StreamExt;

#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").await.unwrap();
    listener
        .incoming()
        .for_each_concurrent(/* limit */ None, |tcpstream| async move {
            let tcpstream = tcpstream.unwrap();
            handle_connection(tcpstream).await;
        })
        .await;
}
```

Serving Requests in Parallel

Our example so far has largely presented cooperative multitasking concurrency (using `async` code) as an alternative to preemptive multitasking (using threads). However, `async` code and threads are not mutually exclusive. In our example, `for_each_concurrent` processes each connection concurrently, but on the same thread. The `async-std` crate allows us to spawn tasks onto separate threads as well. Because `handle_connection` is both `Send` and non-blocking, it's safe to use with `async_std::task::spawn`. Here's what that would look like:

```
use async_std::task::spawn;

#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").await.unwrap();
    listener
        .incoming()
        .for_each_concurrent(/* limit */ None, |stream| async move {
            let stream = stream.unwrap();
            spawn(handle_connection(stream));
        })
        .await;
}
```

Now we are using both cooperative multitasking concurrency and preemptive multitasking to handle multiple requests at the same time! See the [section on multithreaded executors](#) for more information.

Testing the TCP Server

Let's move on to testing our `handle_connection` function.

First, we need a `TcpStream` to work with. In an end-to-end or integration test, we might want to make a real TCP connection to test our code. One strategy for doing this is to start a listener on `localhost` port 0. Port 0 isn't a valid UNIX port, but it'll work for testing. The operating system will pick an open TCP port for us.

Instead, in this example we'll write a unit test for the connection handler, to check that the correct responses are returned for the respective inputs. To keep our unit test isolated and deterministic, we'll replace the `TcpStream` with a mock.

First, we'll change the signature of `handle_connection` to make it easier to test.

`handle_connection` doesn't actually require an `async_std::net::TcpStream`; it requires any struct that implements `async_std::io::Read`, `async_std::io::Write`, and `marker::Unpin`. Changing the type signature to reflect this allows us to pass a mock for testing.

```
use async_std::io::{Read, Write};

async fn handle_connection(mut stream: impl Read + Write + Unpin) {
```

Next, let's build a mock `TcpStream` that implements these traits. First, let's implement the `Read` trait, with one method, `poll_read`. Our mock `TcpStream` will contain some data that is copied into the read buffer, and we'll return `Poll::Ready` to signify that the read is complete.

```

use super::*;
use futures::io::Error;
use futures::task::{Context, Poll};

use std::cmp::min;
use std::pin::Pin;

struct MockTcpStream {
    read_data: Vec<u8>,
    write_data: Vec<u8>,
}

impl Read for MockTcpStream {
    fn poll_read(
        self: Pin<&mut Self>,
        _: &mut Context,
        buf: &mut [u8],
    ) -> Poll<Result<usize, Error>> {
        let size: usize = min(self.read_data.len(), buf.len());
        buf[..size].copy_from_slice(&self.read_data[..size]);
        Poll::Ready(Ok(size))
    }
}

```

Our implementation of `Write` is very similar, although we'll need to write three methods: `poll_write`, `poll_flush`, and `poll_close`. `poll_write` will copy any input data into the mock `TcpStream`, and return `Poll::Ready` when complete. No work needs to be done to flush or close the mock `TcpStream`, so `poll_flush` and `poll_close` can just return `Poll::Ready`.

```

impl Write for MockTcpStream {
    fn poll_write(
        mut self: Pin<&mut Self>,
        _: &mut Context,
        buf: &[u8],
    ) -> Poll<Result<usize, Error>> {
        self.write_data = Vec::from(buf);

        Poll::Ready(Ok(buf.len()))
    }

    fn poll_flush(self: Pin<&mut Self>, _: &mut Context) -> Poll<Result<(),
Error>> {
        Poll::Ready(Ok(()))
    }

    fn poll_close(self: Pin<&mut Self>, _: &mut Context) -> Poll<Result<(),
Error>> {
        Poll::Ready(Ok(()))
    }
}

```

Lastly, our mock will need to implement `Unpin`, signifying that its location in memory can safely be moved. For more information on pinning and the `Unpin` trait, see the [section on pinning](#).

```
impl Unpin for MockTcpStream {}
```

Now we're ready to test the `handle_connection` function. After setting up the `MockTcpStream` containing some initial data, we can run `handle_connection` using the attribute `#[async_std::test]`, similarly to how we used `#[async_std::main]`. To ensure that `handle_connection` works as intended, we'll check that the correct data was written to the `MockTcpStream` based on its initial contents.

```
use std::fs;

#[async_std::test]
async fn test_handle_connection() {
    let input_bytes = b"GET / HTTP/1.1\r\n";
    let mut contents = vec![0u8; 1024];
    contents[..input_bytes.len()].clone_from_slice(input_bytes);
    let mut stream = MockTcpStream {
        read_data: contents,
        write_data: Vec::new(),
    };

    handle_connection(&mut stream).await;

    let expected_contents = fs::read_to_string("hello.html").unwrap();
    let expected_response = format!("HTTP/1.1 200 OK\r\n\r\n{}",
expected_contents);
    assert!(stream.write_data.starts_with(expected_response.as_bytes()));
}
```

Appendix : Translations of the Book

For resources in languages other than English.

- [Русский](#)
- [Français](#)
- [فارسی](#)