Fearless Concurrency

Handling concurrent programming safely and efficiently is another of Rust's major goals. *Concurrent programming*, where different parts of a program execute independently, and *parallel programming*, where different parts of a program execute at the same time, are becoming increasingly important as more computers take advantage of their multiple processors. Historically, programming in these contexts has been difficult and error prone: Rust hopes to change that.

Initially, the Rust team thought that ensuring memory safety and preventing concurrency problems were two separate challenges to be solved with different methods. Over time, the team discovered that the ownership and type systems are a powerful set of tools to help manage memory safety *and* concurrency problems! By leveraging ownership and type checking, many concurrency errors are compile-time errors in Rust rather than runtime errors. Therefore, rather than making you spend lots of time trying to reproduce the exact circumstances under which a runtime concurrency bug occurs, incorrect code will refuse to compile and present an error explaining the problem. As a result, you can fix your code while you're working on it rather than potentially after it has been shipped to production. We've nicknamed this aspect of Rust *fearless concurrency*. Fearless concurrency allows you to write code that is free of subtle bugs and is easy to refactor without introducing new bugs.

> Note: For simplicity's sake, we'll refer to many of the problems as *concurrent* rather than being more precise by saying *concurrent and/or parallel*. If this book were about concurrency and/or parallelism, we'd be more specific. For this chapter, please mentally substitute *concurrent and/or parallel* whenever we use *concurrent*.

Many languages are dogmatic about the solutions they offer for handling concurrent problems. For example, Erlang has elegant functionality for message-passing concurrency but has only obscure ways to share state between threads. Supporting only a subset of possible solutions is a reasonable strategy for higher-level languages, because a higher-level language promises benefits from giving up some control to gain abstractions. However, lower-level languages are expected to provide the solution with the best performance in any given situation and have fewer abstractions over the hardware. Therefore, Rust offers a variety of tools for modeling problems in whatever way is appropriate for your situation and requirements.

Here are the topics we'll cover in this chapter:

- How to create threads to run multiple pieces of code at the same time

- *Message-passing* concurrency, where channels send messages between threads
- *Shared-state* concurrency, where multiple threads have access to some piece of data
- The `Sync` and `Send` traits, which extend Rust's concurrency guarantees to user-defined types as well as types provided by the standard library