IN THIS CHAPTER

- Understanding how programming works
- Learning what Rust brings to the programming table
- Exploring what you can do with Rust
- Tempering your expectations

Chapter 1
Getting to Know Rust

It's ridiculous that we computer people couldn't even make an elevator *that works without crashing!*

—GRAYDON HOARE (INVENTOR OF RUST)

Just as I was getting started writing this book, the good folks at Stack Overflow (a popular question-and-answer site for programmers) announced the results of their 2025 survey of their users. In the "most admired programming language" category, the winner was — can I get a drumroll, please? — Rust! Rust won the same category in 2024, 2023, 2022, and, well, every single year all the way back to 2016! Yes, that's right: In a world where it seems that programmers never agree on *anything*, they have agreed for ten straight years that Rust is awesome.

This near-universal love of Rust convinced me a few years ago to learn the language, and I hope it provides you with the same inspiration as you start your Rust adventure. People admire Rust because it's an elegant, powerful, and *empowering* language that enables beginners and pros alike to create amazing things.

This chapter is your gentle introduction to Rust. You discover what problems Rust solves, what makes Rust unique, and what you can do with Rust.

Programming: Making a Computer Do Your Bidding

A computer is a machine that follows instructions. Or, to put a finer point on it, a computer is a machine that does nothing until someone or something tells it what to do. That might sound surprising. After all, computers cost many hundreds, sometimes even thousands, of dollars, and are positively bristling on the inside with electronic gadgetry. Surely something so expensive and so complex must be capable of doing some useful tasks on its own.

Nope.

Computers must be told what to do, and the way you tell a computer what to do is through code. Am I talking about making a computer do anything you want? Alas, no, although that would be very useful! When you code, you're given a set of tools for the job, and the tools you work with vary depending on the language you're using.


REMEMBER

But no matter how you code — no matter what programming tools you have at your disposal — you're almost always doing one (or sometimes both) of the following:

- **Solving a problem:** One of the most common reasons why a piece of code gets written is because the coder had a pain point or an inefficiency in their life and saw a way to use code to make their life easier or more streamlined.
- **Creating something:** Another common reason to start coding is when you get a great idea and want more than anything to bring that idea to life.

No matter what you work on in your coding career, you're almost always doing one (or both) of these things — solving problems, creating stuff, or combining the two to make something that's both new *and* improved.

What is a programming language?

Rust is a programming language. Okay, fine, but what does it mean to call something a *programming language?* To understand this term, you need look no further

than the language you use to speak and write. At its most fundamental level, human language is comprised of two things — words and rules:

- Words are collections of letters that have a common meaning among all the people who speak the same language. For example, the word *book* denotes a type of object; the word *heavy* denotes a quality; and the word *read* denotes an action.
- Rules are the ways in which words can be combined to create coherent and understandable concepts. If you want to be understood by other speakers of the language, you have only a limited number of ways to throw two or more words together. "I read a heavy book" is an instantly comprehensible sentence, but "book a I read heavy" is nonsensical.

The key goal of human language is being understood by someone else who is listening to you or reading something you wrote. If you use the proper words to refer to things and actions and if you combine words according to the rules, the other person will understand you.

A programming language works in more or less the same way. That is, it, too, has words and rules:

- Words are a set of terms that refer to the specific things that your program works with or the specific ways in which those things can be manipulated. These words are known as *reserved words* or *keywords.*
- Rules are the ways in which the words can be combined to produce the desired effect. In the programming world, these rules are known as the language's *syntax.*

Programming languages, like human languages, also have the goal of being understood, but with a crucial difference: While a human listener might understand you even if you muddle your grammar ("Me want coffee"), a computer has zero tolerance for ambiguity. Every word must be correct, and every rule must be followed exactly, or the program won't work at all.

The role of programming languages

Let's say you travel to Igboland in Nigeria and want to ask a local for directions to the nearest bathroom. If that person speaks only Igbo (the native language of Igboland), one solution would be to find someone who speaks both English and Igbo and ask that person to translate your request as well as the response. Problem solved!

The person who can translate your English into Igbo is called an *interpreter*, and that task is essentially how we're able to program a computer. The problem is that a computer understands only its native language, which is called *machine language* and consists of 1s and 0s. (I won't get into this topic here, but if you're curious to know more, check out the sidebar "Memory 101: 1s and 0s.") A very simple machine-language program might look something like this:

```
10111000 00000001 00000000 00000000 00000000
10111111 00000001 00000000 00000000 00000000
01001000 10111110 00000000 01100000 01100000
00000000 00000000 00000000 00000000 00000000
10111010 00001101 00000000 00000000 00000000
00001111 00000101 10111000 00111100 00000000
00000000 00000000 00110001 11111111 00001111
00000101
```

Yikes! No sane human wants to deal with something as weird as machine language, so one of the first things that engineers did after computers were invented was come up with two remarkable inventions:

- A way of representing machine-language instructions as human-understandable English words.
- A way of converting those English words back into the machine language that the computer understands.

The first invention is called a *programming language* and consists of, in part, English (or, sometimes, English-like) words such as `if`, `loop`, and `match`. You use these generally comprehensible terms to construct *statements*, which are commands that you want the computer to carry out on your behalf.

For example, the preceding machine-language code began life, in part, as the following statement:

```
printf("Hello, world!");
```

This statement outputs the text *Hello, world!* and is written in the C programming language. C is an example of a *high-level language*, which describes any programming language that abstracts away the mind-numbing complexity of the computer's native machine language.

REMEMBER

MEMORY 101: 1S AND 0S

You might have heard someone say, with great authority, that "computers operate by processing 1s and 0s." If, upon hearing that, you were flummoxed, let me tell you that your reaction is utterly normal. It really *is* incomprehensible to us mere mortals that computers perform their wonders by slinging around just two values: 1 and 0. What's behind this mystery?

At the lowest level, a computer is a collection of billions of unimaginably teensy components called *transistors,* which operate as on/off switches for electrical current. When a transistor allows electrical current to pass through, that state is represented by a 1. When a transistor blocks electrical current from passing through, that state is represented by a 0. Each 1 or 0 is called a *binary digit,* or *bit.* One bit offers only two options: 1 or 0. Combining two bits offers four options: 00, 01, 10, or 11. Skipping ahead, I can tell you that combining eight bits offers 256 options, from 00000000 to 11111111 and every combo in between. A string of eight bits is called a *byte* and the 256 possible byte values is enough to code every letter, every number, every punctuation mark, plus a few other standard symbols that make up the American Standard Code for Information Interchange (ASCII) table. The uppercase letter *H,* for example, is 01001000 in binary. So, combine eight bits, set them so that they form the byte 01001000, and you've got the letter *H* stored on your circuit board (which might be a memory module).

Do you need to memorize the byte values for every letter, number, and symbol to code a computer? No, not even close! In fact, from a certain angle, the history of coding is the story of moving further and further away from how information is physically stored using transistors to being able to make the computer do your bidding using relatively simple English words.

The second invention from the early days of computing is called a *compiler*, the purpose of which is to take the English-like code of a programming language and convert it to something (such as machine language) that the computer can read and run. All of this happens behind the scenes, so, as a coder, you never have to lay your eyes on a single 1 or 0.

Understanding how code is written and executed

At this very early stage of your programming career, the process of coding might seem more than a little mysterious, possibly even downright puzzling. After all, from the outside a computer is a mystifying machine, so the idea that you can somehow *control* this inscrutable hunk of electronica might seem the stuff of

fantasy. Or even if you've already convinced yourself that you can make a computer do your bidding, *how* you do that might still have you scratching your head.

Perhaps the secret to being able to code a computer is having the right equipment, something like needing a loom for weaving or a lathe for woodworking.

Nope, you're way off. Maybe the most surprising thing about code is that it's nothing but text. Ever used Notepad in Windows or TextEdit on a Mac? Those bare-bones text files are essentially what you use to write your code.

REMEMBER

To describe the programming process in its most generic terms, I like to use what I call the "three-and-a-half Rs" or coding — write, run, revise, and repeat:

- **Write:** In a text file, you write your code as a series of statements, each of which is essentially an instruction to the compiler for whatever programming language you're using.
- **Run:** You invoke the programming language's compiler and tell it to process the code in the text file you wrote. Then, assuming the compile was a success, you can run the code, and the results appear, which could be the program's output or one or more error messages.
- **Revise:** Based on the results of the run (or the failure of the compile), you edit your code to fix any errors that crop up or to improve your code.
- **Repeat:** You write more code (to, say, add new functionality), run it, revise it as needed, and then repeat the cycle until your program is complete.

That's the bird's-eye view of programming. The rest of this chapter helps you get to know a little more about the programming language at the heart of this book: Rust.

Why the World Needs Rust

One evening back in 2006, a programmer named Graydon Hoare had to climb 21 flights of stairs because his apartment elevator was down due to a software crash. Hoare knew the culprit was mostly likely the elevator's system software, probably written in the language C or C++, both of which are notorious for successfully compiling programs that contain egregious errors of the kind that cause elevators to crash. Why would those programs let those error through? Because those languages expect their programmers not to make such mistakes!

Imagine you build houses for a living. For years, you've been using a particular type of pneumatic nail gun that works pretty well. Ah, but nailing is repetitive work and sometimes you don't realize your nail gun is empty. Or maybe you're rushing and you don't use enough nails for a particular wall. Or maybe you load the gun with nails that are too short. Whatever the reason, sometimes your houses have problems with loose joists or slanted walls. Not good!

Now imagine someone hands you a newfangled nail gun that's just as powerful as your old one, but it refuses to work if the gun is empty, gives you a head's up when you haven't used enough nails for a particular job, and warns you when you're using the wrong type of nail. Would you be interested? Of course you'd want this miracle nail gun!

The key point about this fictional nail gun is that it gives you a head's up when you're about to do shoddy work (or, at least, shoddy nail work). That warning is essentially what Rust offers compared to languages like C and C++. These older languages are incredibly powerful and have been used to build most of the software infrastructure we rely on today, including operating systems, databases, web servers, and embedded systems. But they also have some fundamental safety issues that lead to bugs, crashes, and security vulnerabilities. Sure, just as you could ignore the fictional nail gun's warning and build a lousy house, so, too, can you ignore Rust's warnings and write lousy software. There are no miracles here. But there are some specific problems that Rust solves:

- **Memory safety:** In languages such as C and C++, you have to manually manage your computer's memory, allocating it when you need it and freeing it when you're done. Not doing this properly is like either putting a kettle under the tap to fill it and then forgetting to turn off the tap (a memory leak), or not remembering to fill the kettle at all and turning it on (using memory with nothing in it). Rust handles the allocation and freeing of memory automatically so that these types of problems can't occur.
- **Concurrency safety:** Modern programs often need to do multiple things at once, such as a web server handling hundreds of requests simultaneously. This is called *concurrency*, and it's notoriously difficult to get right. It's like trying to coordinate a kitchen full of chefs without anyone getting in each other's way or accidentally using the same pan at the same time. Rust provides built-in protection against the most common concurrency bugs.
- **Performance:** Many "safe" programming languages achieve safety by adding overhead such as extra checking and routines that pause code execution to free unused memory (a task called *garbage collection*). It's like wearing a full suit of armor to protect yourself: very safe, sure, but you move a lot slower. Rust achieves safety checking for problems before

running a program, which means you get the safety without the performance penalty.

These aren't just theoretical advantages. Major companies such as Microsoft, Google, Facebook, and Dropbox are rewriting critical systems in Rust because they're tired of dealing with the security vulnerabilities and crashes that come with C and C++. The Linux kernel — the core of the operating system that runs most of the world's servers — is now using Rust code. Rust is the real deal.

What Makes Rust Different

Rust takes a different approach than most programming languages. How so? Here are the five main differences:

- **Ownership:** This is the beating heart of what makes Rust unique. I like to call ownership Rust's "secret sauce." Instead of "manual" memory cleanup as needed by C or C++, or "automatic" memory cleanup while your program is running as with Python or JavaScript, Rust gives every piece of data a single owner, and when that owner goes away, the memory used by the data is freed automatically.
- **Borrow checker:** This is Rust's gatekeeper for memory safety. It keeps track of how your program uses data: who owns it, who can borrow it, and when it's no longer needed.
- **Zero-cost abstractions:** This refers to Rust's ability to let you write readable, high-level code without a performance penalty (that's what "zero-cost" means). In coding, *abstraction* refers to the process of hiding complex, lower-level stuff with simpler, higher-level stuff. Usually, the layers of abstraction just slow everything down, but Rust is designed to be just as fast as languages (such as C and C++) that offer fewer abstractions.
- **Fearless concurrency:** This is Rust's system for implementing code that runs in parallel (using multiple CPU cores) in a way that prevents the serious concurrency errors that plague other languages.
- **A compiler that teaches:** This is my favorite Rust feature because, unlike other compilers that usually just berate you for doing something wrong with your code, the Rust compiler takes a gentler approach that explains not just what went wrong, but why it went wrong and how you can fix it. In other words, the compiler tries to teach you the right way to code with Rust. This is why I believe that a relatively complex language such as Rust is learnable even by new programmers. The compiler has your back.

These advantages don't come free. Rust's strict compiler can feel like an overbearing boss at first, and concepts like ownership and borrowing have a steep learning curve. But most Rust programmers will tell you the initial struggle is worth it.

Rust: Not Just for Systems Nerds

When Rust first appeared (the 1.0 version was released back in 2014), it was really a systems programming language, meaning it was a language you'd use to write operating systems, device drivers, or networking code. It was awesome at that systems stuff, and it still is. Ah, but Rust has grown far beyond that original niche:

- **Command-line tools:** Rust's speed and built-in package manager (Cargo) make it perfect for command-line utilities. Refer to Book 4 to learn how to use Rust to build your own command-line utilities.
- **WebAssembly:** Amazingly, Rust can compile to run in the web browser, opening the door for all kinds of web-based magic. Book 5 is the place to go to learn how to use Rust in the browser.
- **Web development:** There are Rust libraries that enable you write super-fast web servers, application programming interfaces, and web apps in Rust. Refer to Book 6 to learn how to build these and other web tools.

So, yep, while Rust is still a darling of the "close-to-the-metal" crowd, it's just as capable of powering your next utility, chatbot, or server.

Setting Realistic Expectations

To *accomplish* something doesn't mean to complete something easily done or to learn a skill easily acquired. It means to complete or learn something difficult, something that takes effort. Learning to program is an accomplishment. Learning to program in Rust is a *major* accomplishment. You can absolutely do these things, but I have to be honest with you now: it won't be easy, and it won't be effortless. You can't just install Rust and then hope to have it mastered by the end of the day.

So, before diving in to learn the details of installing Rust and running your first program, all of which (gulp!) happens in the next chapter, here's a list of some

useful Rust truths that will (I hope) send you on your journey with tempered expectations of how that trip might unfold:

- **There will be frustrations.** Programming requires that you do everything in an exacting, fastidious way or your code won't work. Bumping your head against that need for precision over and over can be frustrating, but it's the price for success.
- **There will be roadblocks.** It might be code that you can't get to compile or a concept that you can't get into your head. Everyone who programs runs into these kinds of barriers, but everyone also gets past them with patience and a bit of effort.
- **There will be error messages.** Rust's compiler is notoriously finicky about catching errors because it prefers that *you* deal with those errors rather than the people who will eventually use your program. The beauty of Rust is that it turns every error into a learning opportunity because the compiler tells you exactly where your code went south *and* it offers suggestions on how to fix the problem.
- **There will be wins.** Grokking a difficult Rust concept after sticking with it for a while is a win. Understanding that the Rust compiler is helping you, not berating you, is a win. Getting even a small program to compile and run is big, dopamine-inducing win. Celebrate the wins!
- **There will be a payoff for all your hard work.** If you write a ton of code, work with the compiler instead of against it, and think hard to grasp Rust's more esoteric ideas, then you'll eventually get to the point where you can build some *seriously* cool stuff. You won't get there tomorrow or next week or even next month, but you'll get there.