

LEARNING MADE EASY



Rust

ALL-IN-ONE

for
dummies[®]
A Wiley Brand



6
Books
in one!

Paul McFedries

Author of *HTML, CSS, & JavaScript*
All-in-One For Dummies

Rust All-in-One For Dummies®

To view this book's Cheat Sheet, simply go to www.dummies.com and search for “Rust All-in-One For Dummies Cheat Sheet” in the Search box.

Table of Contents

[Cover](#)

[Table of Contents](#)

[Title Page](#)

[Copyright](#)

[Introduction](#)

[About This Book](#)

[Foolish Assumptions](#)

[Icons Used in This Book](#)

[Beyond the Book](#)

[Book 1: Welcome to the Rust Side](#)

[Chapter 1: Getting to Know Rust](#)

[Programming: Making a Computer Do Your Bidding](#)

[Why the World Needs Rust](#)

[What Makes Rust Different](#)

[Rust: Not Just for Systems Nerds](#)

[Setting Realistic Expectations](#)

[Chapter 2: Getting Your Hands Rusty](#)

[Getting to the Command Line](#)

[Installing Rust: Easier Done Than Said](#)

[Meet Cargo: Your New Coding Friend](#)

[Setting Up Your Development Environment](#)

[Creating Your First Rust Project](#)

[Understanding the Project Structure](#)

[Compiling and Running Your First Program](#)

[Learning Rust: A Roadmap](#)

Chapter 3: Speaking Rust: The Basics

[main\(\): A Home for Your Code](#)

[Variables: Storing Your Digital Stuff](#)

[Make Mine a Mutable](#)

[Annotating Your Code with Comments](#)

[Outputting Text](#)

[Talking about Types](#)

[Working with the Standard Library](#)

Chapter 4: Crafting Expressions

[Understanding Expression Structure](#)

[Building Numeric Expressions](#)

[Building Comparison Expressions](#)

[Building Logical Expressions](#)

[Understanding Operator Precedence](#)

[Corralling Code into Expression Blocks](#)

Chapter 5: Controlling the Flow

[Making Decisions with if](#)

[Handling Repetitive Code with Loops](#)

Chapter 6: Pattern Matching: The Swiss Army Knife of Rust

[Becoming a match Master](#)

[Introducing Block Scope](#)

[Project: Escape the Dungeon \(a Tiny Text Adventure\)](#)

Chapter 7: Functions: Teaching Rust New Tricks

[What Is a Function?](#)

[Why Functions Matter](#)

[Writing Your First Function](#)

[Arguments: How to Pass 'Em, How to Use 'Em](#)

[Return Values: Bringing Data Back Alive](#)

[Digging Deeper into Variable Scope](#)

[Functions That Call Other Functions](#)

[Project: Temperature Converter](#)

Book 2: Learning the Rust Way of Doing Things

Chapter 1: Ownership: Rust's Secret Sauce

[Understanding Ownership without Losing Your Mind](#)

[Moving: How Rust Transfers Ownership](#)

[Borrowing: Sharing Your Toys](#)

[References: Pointing to Things Safely](#)

[Move? Borrow? Clone? Which One Should You Use?](#)

[Project: Creating a Text Analyzer](#)

Chapter 2: Structs: Rolling Your Own Types

[You Need More Struct in Your Life](#)

[Meet the Struct](#)

[A Couple of Useful Struct Shortcuts](#)

[Methods: Teaching Structs New Behaviors](#)

[Implementing Associated Functions](#)

[Project: Building a Simple Bank Account Manager](#)

Chapter 3: Enums: Embracing Possibilities

[Meet the Enum](#)

[Option: It's Always Something \(or Is It?\)](#)

[Result: Dealing with Errors Gracefully](#)

[Project: Modeling an Authentication Flow](#)

Chapter 4: Storing Data in Rust

[Tuples: Groups of Things](#)

[Arrays: Fixed-Sized Storage](#)

[Slices: Versions of Your Stuff](#)

[Vectors: Dynamic Arrays That Grow](#)

[Hash Maps: Key-Value Storage Made Easy](#)

[Project: The Prime Explorer](#)

Chapter 5: The String Section: Text in Rust

[String Instruments: String and &str](#)

[Making String Music: Creating Strings](#)

[String Works: The Greatest Hits](#)

[String Pieces: Slicing Strings](#)

[String Movements: Common String Patterns](#)

[Project: Slugifying a String](#)

Chapter 6: Functional Programming with Closures and Iterators

[What on Earth Is Functional Programming?](#)

[Closures: Anonymous Functions with Attitude](#)

[Meet the Iterator Pattern](#)

[Creating an Iterator from a Collection](#)

[Consuming Iterators](#)

[Adapting Iterators](#)

[Project: Word Wrangler](#)

Chapter 7: Organizing Your Code

[How Rust Treats Code Organization](#)

[Starting Small: Organizing Code in One File](#)

[Growing Up: Moving to Multiple Files](#)

[Controlling Access: Public versus Private](#)

[Importing and Using Code](#)

[Working with External Dependencies](#)

[Real-World Organization Patterns](#)

Book 3: Deeper Rust Concepts

Chapter 1: Traits: Shared Behaviors

[Traits Are Promises](#)

[When Your Types Won't Behave](#)

[Deriving: Traits in the Fast Lane](#)

[Behave Yourself: Implementing Traits for Your Own Types](#)

[Printing Prettier with the Display Trait](#)

[Setting Sensible Defaults with the Default Trait](#)

[Project: Building a Media Library with Traits](#)

Chapter 2: Generics: Code That Works for Almost Anything

[Why Write the Same Code Twice?](#)

[Introducing Generics](#)

[Type Bounds and Constraints](#)

[Traits and Generics = Chef's Kiss](#)

[Project: Build a Generic Logger That Works with Any Type](#)

Chapter 3: Lifetimes: How Long Things Live

[Introducing Lifetimes](#)

[When Rust Demands Lifetime Annotations](#)

[Adding Lifetime Annotations to Functions](#)

[Lifetimes in Structs: Keeping References Alive](#)

[Lifetime Elision: When the Compiler Does the Work](#)

[The 'static Lifetime: Data That Lives \(Sort of\) Forever](#)

[Project: A Zero-Copy Config Parser](#)

Chapter 4: Handling Errors Like a Pro

[Recoverable versus Unrecoverable: Know When to Panic!](#)

[Result<T, E>: Your Go-to Type for Errors](#)

[Saving Time with Helper Methods](#)

[Propagating Errors with the ? Operator](#)

[Grokking Rust's Error Types](#)

[Magically Creating a Result from an Option](#)

[Creating Your Own Error Types](#)

Chapter 5: Macros: When Code Writes Code

[Useful Macro Magic: println! and vec!](#)

[Using Macros in Your Own Code](#)

[Macro Syntax: Learning to Read the Hieroglyphics](#)

[Writing Macros](#)

[Project: Building a Timer Macro](#)

Chapter 6: Parallelism and Concurrency: Rust Multitasking

[Parallelism versus Concurrency: What's the Difference?](#)

[Spawning Threads: Your First Parallel Adventure](#)

[Passing Messages through Channels](#)

[Sharing State Safely with Arc and Mutex](#)

[Borrowing Data with Scoped Threads](#)

[Iterating in Parallel with Rayon](#)

[Async/Await: Concurrency without the Thread Overhead](#)

[Project: Collatz Conjecture Explorer](#)

Chapter 7: Testing, Debugging, and Documenting

[Testing Your Code](#)

[Debugging Your Code](#)

[Cleaning Your Code](#)

[Documenting Your Code](#)

Book 4: Building Command-Line Tools

Chapter 1: Coding Your First Tools

[Understanding What Makes a Good CLI Tool](#)

[Touring a Typical CLI](#)

[Parsing Command-Line Arguments](#)

[Reading from Standard Input](#)

[Basic File Operations for CLI Tools](#)

[Project: A Word Replacer Tool](#)

Chapter 2: File Processing and Configuration

[Learning File System Foundations](#)

[Reading and Writing Files Safely](#)

[Serializing and Deserializing Data](#)

[Project: Creating a File Organizer](#)

Chapter 3: Polishing Your Command-Line Tools

[Organizing Your Tools with Subcommands](#)

[Fancifying Output with Colors](#)

[Implementing Progress Bars](#)

[Project: Building a Multicommand File Organizer](#)

Book 5: WebAssembly: Rust in Your Browser

Chapter 1: WebAssembly Basics and Setup

[Web *What* Now? Understanding WebAssembly](#)

[Why Rust and WebAssembly Are Best Friends](#)

[Setting Up Your Rust-to-WASM Toolchain](#)

[Running Your First Rust Function in the Browser](#)

[Running JavaScript Functions in Rust](#)

[Common WASM Build Commands](#)

[Project: Building a Rust-Powered Password Analyzer](#)

Chapter 2: Exchanging Data between Rust and JavaScript

[Understanding the wasm-bindgen Bridge](#)

[Exchanging Data through the Wall](#)

[Swapping Strings](#)

[Passing Arrays, Vectors, and Other List-Like Things](#)

[Project: Building a Remote Markdown Renderer](#)

Chapter 3: Building Interactive WebAssembly Apps

[Controlling DOM from Rust](#)

[Handling Browser Events](#)

[Managing Application State](#)

[Project: Populating DOM from Rust](#)

Book 6: Networking with Rust

Chapter 1: Building a Web Server

[Understanding HTTP from the Ground Up](#)

[Getting a Minimal Web Server Off the Ground](#)

[Implementing Routing and Handler Functions](#)

[Organizing Your Web Server Code](#)

[Serving Static Files](#)

[Logging and Debugging](#)

Chapter 2: Building a REST API

[What's an API?](#)

[Okay, So What's a REST API?](#)
[From HTML to JSON: Changing Your Mindset](#)
[Getting Started on the API](#)
[POST, PUT, DELETE: The REST of the HTTP Verbs](#)
[Building a Resource: The Book Library API](#)
[Implementing CRUD Operations](#)
[Validating Data and Handling Errors](#)
[Migrating to a Real Database with SQLx](#)
[Testing Your API](#)
[Deployment Considerations](#)

Chapter 3: Building a Server-Side Web App

[Setting Up Your Web App Foundation](#)
[Templates: HTML with Superpowers](#)
[Serving Static Files \(CSS and Friends\)](#)
[Connecting to the Database](#)
[Authentication: Who Are You?](#)
[Managing Bookmarks](#)
[Extending the App](#)

Index

About the Author

Dedication

Author's Acknowledgments

Connect with Dummies

End User License Agreement

List of Tables

Book 1 Chapter 2

[TABLE 2-1 Terminal Keyboard Shortcuts](#)

Book 1 Chapter 3

[TABLE 3-1 Rust's Integer Data Types](#)

Book 1 Chapter 4

[TABLE 4-1 Basic Arithmetic Operators](#)

[TABLE 4-2 Arithmetic Assignment Operators](#)

[TABLE 4-3 Some Floating-Point Constants](#)

[TABLE 4-4 Some Floating-Point Methods](#)

[TABLE 4-5 Comparison Operators](#)

[TABLE 4-6 Logical Operators](#)

[TABLE 4-7 Rust Order of Precedence for Operators](#)

Book 5 Chapter 1

[TABLE 1-1 Common JavaScript Namespaces](#)

[TABLE 1-2 Common Global Browser Functions](#)

List of Illustrations

Book 1 Chapter 2

[FIGURE 2-1: In VS Code, use the Extensions pane to search for and install the r...](#)

[FIGURE 2-2: Using VS Code to view the `main.rs` file created by Cargo.](#)

[FIGURE 2-3: You can use VS Code's Terminal panel to run shell commands. No more...](#)

Book 2 Chapter 7

[FIGURE 7-1: A module tree for a hypothetical app.](#)

[FIGURE 7-2: A module tree for the Dog Survey code.](#)

[FIGURE 7-3: A typical directory structure for a medium-sized Rust app.](#)

[FIGURE 7-4: A larger Rust project with multiple crates.](#)

Book 3 Chapter 7

[FIGURE 7-1: With `rust-analyzer` installed, you can click Run Test in VS Code to ...](#)

[FIGURE 7-2: Set a breakpoint before starting your debugging session.](#)

[FIGURE 7-3: A Rust program in break mode.](#)

[FIGURE 7-4: In VS Code, the `rust-analyzer` extension uses your documentation com...](#)

[FIGURE 7-5: Cargo uses your doc comments to build a rather nice website for you...](#)

Book 5 Chapter 1

[FIGURE 1-1: Honest-to-goodness Rust code running in a web browser!](#)

[FIGURE 1-2: The password analyzer doing its thing.](#)

Book 5 Chapter 2

[FIGURE 2-1: The remote markdown renderer running in the browser.](#)

Book 5 Chapter 3

[FIGURE 3-1: The status `div` updated from Rust.](#)

[FIGURE 3-2: A web page with its application state managed by Rust.](#)

[FIGURE 3-3: A web page with DOM \(mostly\) populated by Rust.](#)

Book 6 Chapter 1

[FIGURE 1-1: Your minimal web server is up and serving.](#)



Rust

ALL-IN-ONE

by Paul McFedries

for
dummies[®]
A Wiley Brand

Rust All-in-One For Dummies®

Published by: **John Wiley & Sons, Inc.**, 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Copyright © 2026 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial technologies or similar technologies.

Media and software compilation copyright © 2026 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial technologies or similar technologies.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

The manufacturer's authorized representative according to the EU General Product Safety Regulation is Wiley-VCH GmbH, Boschstr. 12, 69469 Weinheim, Germany, e-mail: Product_Safety@wiley.com.

Trademarks: Wiley, For Dummies, the Dummies Man logo, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and may not be used without written permission. Rust is a trademark or registered trademark of Rust Foundation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit

<https://hub.wiley.com/community/support/dummies>.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this

book may not be included in e-books or in print-on-demand. If this book refers to media that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2026937721.

ISBN 978-1-394-40190-1 (pbk); ISBN 978-1-394-40191-8 (ebk); ISBN 978-1-394-40192-5 (ebk)

Introduction

When you visit the official Rust language website at www.rust-lang.org/, the very first thing you see is the following tagline:

Rust: A language empowering everyone to build reliable and efficient software.

Note that it doesn't say "A language empowering systems programmers to build reliable and efficient software." Nor does it say "A language empowering network engineers to build reliable and efficient software." It doesn't specify computer science PhDs, hackers, or uber-nerds. No, it says *everyone*.

And, so, that means *you*.

I don't know what you've heard about Rust, but there's a good chance that you've read or been told that it's too hard, it's too complex, it's for experienced programmers only. I don't know what you've heard, but hear this: That's mostly nonsense, but there *is* a grain of truth in there. Rust *does* ask more of you than some languages. Its safety features mean the compiler will push back on code that other languages would happily accept, which can be frustrating at first. But those compiler complaints are catching bugs that would otherwise bite you later. And once you learn to work with the compiler instead of against it, Rust becomes a genuinely enjoyable language to write.

Rust has a learning curve, but does that make it *too* hard? No. Rust is a mature, feature-rich language, but does that make it *too* complex? Nope. Rust is definitely easier to learn if you already know how to code, but does that mean people new to programming should look elsewhere? Definitely not.

These are the short answers to some of the typical negative ideas that people associate with Rust. This book is the long answer.

About This Book

Welcome, then, to *Rust All-in-One For Dummies*. This book teaches you everything you need to know about coding with Rust, from what it is, to how to get started, to writing and running your first programs. My goal is to show you that going from being clueless about coding to being comfortable with one of the most powerful programming languages out there is a lot more doable than you probably think. This book shows that even the greenest rookie coder can learn how to use Rust to build programs that will amaze their family and friends (and themselves).

Note that I didn't say that this book teaches you everything there is to know about Rust. If you're looking for lots of Rust history, computer science theory, and long-winded explanations of coding concepts, I'm sorry, but you won't find any of those things here. My philosophy throughout this book comes from Linus Torvalds, the creator of the Linux operating system (which is now using lots of Rust code): "Talk is cheap. Show me the code." I explain what needs to be explained and then I move on without further ado (or, most of the time, without any ado at all) to examples and programs that do more to illuminate a concept than any verbose explanations I could muster. (And believe me, I can muster verbosity with the best of them.)

How you approach this book depends on your current level of programming expertise (or lack thereof):

- » If you're just starting out, begin at the beginning with the five chapters in [Book 1](#). They will give you a solid programming foundation. Then tackle at least the first five chapters in [Book 2](#) (but all seven chapters would be best). They will give you a solid Rust foundation.
- » If you're coming here from having read my book *Coding For Dummies*, hey, great to see you again! Since you're familiar with coding basics, you can skim through [Book 1](#) to see how Rust does things, and then tackle [Books 2](#) and [3](#). This approach will give you

all the knowledge you need to pick and choose what you want to learn throughout the rest of the book.

- » If you're an experienced programmer, take a quick tour through [Book 1](#) to get up to speed on how Rust handles things like variables, loops, conditionals, and functions. Then spend your time using [Books 2](#) and [3](#) to learn the unique features of Rust, including ownership and borrowing, data types, closures, iterators, traits, generics, lifetimes, and error handling. You can then pick and choose what project-oriented chapters interest you in [Books 4](#) through [6](#).

Foolish Assumptions

This book is not a primer on using a computer or editing text. This is a book on learning to code in Rust, pure and simple, so I assume only the following:

- » You know how to operate a basic text editor and how to get around the operating system and file system on your computer.
- » You have an internet connection.
- » You know how to use your web browser.

Yep, that's it. And, just to be clear, what I'm saying here is that I do *not* assume you have any experience programming. Sure, the greater your coding background, the easier Rust is to learn, but you can still learn Rust even if you've never coded before.

Icons Used in This Book



REMEMBER This icon points out interesting tidbits that are likely to be repeatedly useful to you in your Rust coding career, so it pays to remember them.



TIP

This icon lets you in on secret and not-so-secret ways to do things easier, faster, or more efficiently. Please save your applause to the end.



WARNING

This icon appears when I'm trying to help you avoid mistakes that can cost you time or money or cause embarrassment.



**TECHNICAL
STUFF**

This icon signifies material that isn't critical to understand (or even to read, for that matter) but will definitely satisfy the curious. Think "inquiring minds want to know" when you see this icon.

Beyond the Book

You can access the examples used in the book in the following places:

» **My website:** <https://paulmcfedries.com/books/rust-aio-fd>

» **GitHub:** <https://github.com/paulmcfe/rust-aio-fd>

The examples are organized by minibook and by chapter. For each example, you can view the code, download it, and copy it to your computer's clipboard. However, please resist the temptation to just copy and paste the code. The best way to learn Rust is to get what I call a "fingertip feeling" for it by typing the code yourself. Sure, it takes more effort, but the Rustiness will seep into your brain faster this way.

Book 1

Welcome to the Rust Side

Contents at a Glance

Chapter 1: Getting to Know Rust

[Programming: Making a Computer Do Your Bidding](#)

[Why the World Needs Rust](#)

[What Makes Rust Different](#)

[Rust: Not Just for Systems Nerds](#)

[Setting Realistic Expectations](#)

Chapter 2: Getting Your Hands Rusty

[Getting to the Command Line](#)

[Installing Rust: Easier Done Than Said](#)

[Meet Cargo: Your New Coding Friend](#)

[Setting Up Your Development Environment](#)

[Creating Your First Rust Project](#)

[Understanding the Project Structure](#)

[Compiling and Running Your First Program](#)

[Learning Rust: A Roadmap](#)

Chapter 3: Speaking Rust: The Basics

[main\(\): A Home for Your Code](#)

[Variables: Storing Your Digital Stuff](#)

[Make Mine a Mutable](#)

[Annotating Your Code with Comments](#)

[Outputting Text](#)

[Talking about Types](#)

[Working with the Standard Library](#)

Chapter 4: Crafting Expressions

[Understanding Expression Structure](#)

[Building Numeric Expressions](#)

[Building Comparison Expressions](#)

[Building Logical Expressions](#)

[Understanding Operator Precedence](#)

[Corralling Code into Expression Blocks](#)

Chapter 5: Controlling the Flow

[Making Decisions with if](#)

[Handling Repetitive Code with Loops](#)

Chapter 6: Pattern Matching: The Swiss Army Knife of Rust

[Becoming a match Master](#)

[Introducing Block Scope](#)

[Project: Escape the Dungeon \(a Tiny Text Adventure\)](#)

Chapter 7: Functions: Teaching Rust New Tricks

[What Is a Function?](#)

[Why Functions Matter](#)

[Writing Your First Function](#)

[Arguments: How to Pass 'Em, How to Use 'Em](#)

[Return Values: Bringing Data Back Alive](#)

[Digging Deeper into Variable Scope](#)

[Functions That Call Other Functions](#)

[Project: Temperature Converter](#)

Chapter 1

Getting to Know Rust

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
-

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.

Chapter 2

Getting Your Hands Rusty

IN THIS CHAPTER

- » Getting comfy with the command line
 - » Installing Rust and friends
 - » Getting ready to code
 - » Creating and running your first Rust program
-

The Rust programming language is fundamentally about empowerment: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program with confidence in a wider variety of domains than you did before.

—NICHOLAS MATSAKIS AND AARON TURON

Welcome to the part where you stop reading about Rust and actually start using it! Think of this chapter as both your personal Rust installation manual and your first-day orientation tour. By the end of this chapter, you'll be a veritable (or, at least, budding) command-line pro, you'll have Rust installed on your computer, you'll understand why Cargo is about to become your favorite development tool, and you'll have created, compiled, and run your first Rust program. That's a pretty good start!

Don't worry if you've never installed a programming language before and if you're not even sure what "compiled" means. This chapter takes everything step by step. And if you're an experienced developer, I'm willing to bet that you'll find Rust's approach to managing code refreshingly straightforward compared to some other languages you might have wrestled with.

Getting to the Command Line

Whether you use Windows, macOS, or even a Linux distro such as Ubuntu, you're no doubt accustomed to making things happen by using your mouse or trackpad to click or double-click icons, buttons, and other visual knickknacks. But every Windows PC, Mac, and Linux box has an alternative tool for doing things: the command line. The *command line* is a text-based interface that enables you to interact with the computer by typing commands rather than clicking and dragging graphical elements. The program where the command line lives is called the *terminal*, and the software that interprets your commands is called the *shell*.

Although you'll likely code your Rust apps using a mouse-friendly text editor, at least some Rust action will take place at your computer's command line, and how you get there depends on whether you're using Windows, macOS, or Linux.

Launching Terminal on Windows

Windows offers a Terminal app that you can use to run your Rust-related commands. To launch the app, click Start, type **term**, and then click Terminal in the search results. (Alternatively, right-click Start and then click Terminal in the shortcut menu that pops up.)



REMEMBER If you're running an older version of Windows that doesn't include the new Terminal app, you can download the app from the Microsoft Store.

Terminal is home to multiple shells. PowerShell is the default, so it appears when you launch Terminal, and it's the one you should use for running Rust commands.

Launching Terminal on macOS

macOS offers the Terminal app for all your Mac-related command-line frolics. You have three main ways to get there:

- » **Spotlight Search:** Click the Spotlight Search icon in the menu bar or press $\text{⌘} + \text{Spacebar}$, type **term**, and then click Terminal in the search results.
- » **Launchpad:** Click Launchpad in the dock, click Other, and then click Terminal.
- » **Finder:** Click Finder on the dock, and then click Applications $\Rightarrow$ Utilities $\Rightarrow$ Terminal.



REMEMBER As a way of differentiating command-line statements and Rust language statements, I precede the former with a percentage sign ($\%$), which represents the command-line prompt in macOS Terminal. (The prompt is $>$ in Windows Terminal and $\$$ in Linux.) So, when you read something like $\% \text{ pwd}$, it means “type `pwd` and press Enter or Return.” Don’t type the $\%$ or you’ll get an error.

Launching Terminal on Linux

If you’re using Linux, you probably already know your way around a terminal. Most Linux distributions come with a default terminal application, and you can usually open it by pressing $\text{Ctrl} + \text{Alt} + \text{T}$, by right-clicking the desktop and then clicking Open Terminal, or by searching for *Terminal* in your application menu.

Some useful command-line shortcuts

You’ll likely spend a nontrivial portion of your Rust career hanging around the command line, so you might as well learn a few handy keyboard shortcuts to make your terminal work easier and more efficient. To that worthy end, [Table 2-1](#) presents a bunch of keyboard techniques and their Windows, macOS, and Linux shortcuts.

TABLE 2-1 Terminal Keyboard Shortcuts

<i>Action</i>	<i>Windows Shortcut</i>	<i>macOS Shortcut</i>	<i>Linux Shortcut</i>

Action	Windows Shortcut	macOS Shortcut	Linux Shortcut
Interrupt or stop current command	Ctrl+C	Control+C	Ctrl+C
Clear terminal screen	Ctrl+L	Control+L or Cmd+K	Ctrl+L
Cancel typed text or line	Esc	Control+U	Ctrl+U
Autocomplete file or command	Tab	Tab	Tab
Cycle command history	Up arrow or down arrow	Up arrow or down arrow	Up arrow or down arrow
Search command history	F8 or Ctrl+R	Control+R	Ctrl+R
Delete word (backward)	Ctrl+Backspace	Control+W	Ctrl+W
Delete line (cursor to start)	Ctrl+U	Control+U	Ctrl+U
Delete line (cursor to end)	Ctrl+K	Control+K	Ctrl+K
Move cursor word by word	Ctrl+left arrow or Ctrl+right arrow	Option+left arrow or Option+right arrow	Alt+left arrow or Alt+right arrow
Move to start or end of line	Home or End	Control+A or Control+E	Ctrl+A or Ctrl+E
Paste from clipboard	Right-click or Ctrl+V	Cmd+V or Control+Y	Ctrl+Shift+V
Copy selected text	Right-click or Ctrl+C	Cmd+C	Ctrl+Shift+C
Scroll up or down terminal output	Ctrl+Up or Ctrl+Down	Shift+Page Up or Shift+Page Down	Shift+Page Up or Shift+Page Down

Basic command-line survival skills

Once you have your command line open, here are a few essential commands that work on all platforms:

» **cd *path* (change directory):** Moves to the directory specified by *path*. In particular:

- `cd ..` moves up to the parent directory of the current directory.
- `cd ../sibling` moves up to the parent directory of the current directory, and then into the subdirectory specified by *sibling*.

» **pwd (print working directory):** Displays the full path name of the current directory.

» **ls (list) on Mac and Linux or dir (directory) on Windows:**
Displays a list of the files and subdirectories in the current directory.

» **mkdir *name* (make directory):** Creates a directory named *name*.

Don't sweat memorizing these commands now. You'll be using them a ton as you work with your Rust projects, so they'll become embedded in your brain with practice.

Installing Rust: Easier Done Than Said

Okay, now it's time for the main event: getting Rust installed on your computer. If you're expecting a convoluted, hours-long, headache-inducing process, I'm happy to report that the Rust installation doesn't tick any of those boxes. Instead, getting Rust running is refreshingly painless thanks to a tool called `rustup`, which handles (and hides) all the messy installation details.



TIP

Technically, you don't have to install Rust on your computer if you just want to mess around with it. Rust's overlords have created a great little app called Rust Playground, where you can type and

run Rust code without installing anything. To give Playground a whirl, send your nearest web browser to <https://play.rust-lang.org/>.

Installing Rust on Windows

Before getting to the Rust portion of the show, you need to install the Visual Studio build tools (not the full Visual Studio IDE, just the build tools) because Rust on Windows requires certain compiler components to do its job:

1. **Point your browser to**
<https://visualstudio.microsoft.com/downloads/>.
2. **Click the Free Download button under the Community version, and then run the installer.**
3. **When the Visual Studio Installer window appears, click Modify next to the installed version of Visual Studio.**
4. **Select the Desktop Development with C++ check box, and then click Modify.**

It will take a few minutes for the installation to complete, but it's best to wait it out before proceeding.

For Rust, Windows users get a choice of installation paths (but if you're running Windows Subsystem for Linux, skip to the "[Installing Rust on macOS and Linux](#)" section):

- » **Graphical:** This is the easiest way to go, especially if you're not used to the command line. Send your favorite web browser to the Install Rust page at www.rust-lang.org/tools/install, click the button to download the 64-bit version, run the downloaded file, and then follow the prompts.
- » **Command line:** If you prefer some command-line practice, open a PowerShell terminal, type the following mouthful of a command, and then press Enter:

```
Invoke-WebRequest -Uri https://win.rustup.rs/x86_64 -OutFile rustup-init.exe; .\rustup-init.exe
```

Either way, you'll be prompted to install all the Rust bits and pieces. When asked, be sure to choose the standard installation.

Installing Rust on macOS and Linux

For Mac and Linux users, the installation is a command-line-only affair, so open a terminal and run the following long-winded command (remembering, of course, not to type the % prompt):

```
% curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```



TIP

If you don't feel like typing all that, well, who would? You can make your life easier and, more importantly, you can ensure that you enter the command correctly, by pointing your browser at <https://rustup.rs>, which displays the `curl` command in a box with a handy Copy button next to it. Click that button, return to the command line, and then paste the command. Easier than pie!

When the list of installation options appears, be sure to choose the standard installation (usually by just pressing Return or Enter).

What happens during installation

When you run the installer, several things happen behind the scenes:

- » **Rustup gets installed:** Rustup is your Rust version manager and updater.
- » **The latest stable Rust gets downloaded:** This version of Rust includes the compiler (`rustc`) and Rust's standard library (which is essentially what you're learning in this book).
- » **Cargo gets installed:** Cargo is Rust's package manager and build tool (a build tool automates the process of converting your source code into a runnable program; more on Cargo in a moment).

- » **Your PATH gets updated:** PATH tells your computer where to find the Rust tools.

After the installation is complete, you need to restart your terminal session so that it can find the new Rust tools.

Verifying the installation

Before moving on, it pays to take a few seconds to verify that everything installed correctly. Here's the first command to run at the command line:

```
% rustc --version
```

Something like this should appear:

```
rustc 1.89.0 (6b00bc388 2025-08-04)
```

Now run this command:

```
% cargo --version
```

Something along these lines should appear onscreen:

```
cargo 1.89.0 (873a06493 2025-08-04)
```

If, instead, you get an error such as `rustc: command not found` or `cargo: command not found`, here are some things to try:

- » If you installed via the command line, the installation command is long and complex, so double-check that you entered it correctly. How? Return to the terminal window, press the up-arrow key until the command reappears, and then put on your proofreading hat.
- » Close and re-open your terminal window, if you didn't do that already, and then try the verification commands again.
- » If you're still getting a `not found` error, the best thing to do at this point is just run the installation again.
- » If you're still having a problem, make sure your computer is connected to the internet. It's also probably a good idea to log off and then back on (or restart your computer) to get a fresh take on things.

Updating Rust

Head back to the preceding section and examine the results of both the `rustc --version` and `cargo --version` commands. Each includes, in parentheses, a nine-character string, followed by a date, such as 2025-08-04. That date is when the version of Rust shown earlier in the results, such as 1.89.0, was finalized (but not yet released). Here's how to read a version number such as 1.89.0:

- » **1:** The major version number. If this number changes, it would indicate a dramatic shift in Rust's foundations. The fact that Rust has been at major version 1 since May 2015 means they've committed to not breaking your existing code in fundamental ways.
- » **89:** The minor version number. When this number goes up, it means Rust has added new capabilities, but your existing code should still work perfectly.
- » **0:** The patch version number, which refers to small fixes or improvements that are released as needed. (0 means no patches have been released yet for the current minor version.)

The good news is that you don't have to worry about version numbers all that much because the Rust folks are careful about releasing only stable versions of the language.

You're probably thinking that new minor versions of Rust are released only rarely. Not so! One of the most surprising things about Rust is that a new minor version is released, clockwork-like, every six weeks! So, as I write this, I'm on version 1.89.0, released August 7, 2025. If I was nerdy enough, I could write in my calendar (in pen, if I was an old-school nerd) that version 1.90 will be released on September 18, 2025.

Ah, I hear you saying, now he's going to give me a thousand-character-long command for updating Rust. Nope. The command to get the latest release of Rust installed on your system could hardly be simpler or shorter:

```
% rustup update
```

It really is as easy as that. Rust’s tooling makes common tasks like this refreshingly simple.



REMEMBER If, for whatever reason, Rust just isn’t doing it for you, you can get Rust off your computer and move on with your life by running the command `rustup self uninstall`.

Meet Cargo: Your New Coding Friend

In programming lingo, a *package manager* is a tool that helps a developer install, configure, manage, and update the various software packages that a coding project depends on. If you’re coming to Rust from another programming language, you might already be familiar with your language’s package manager, such as npm (Node.js), pip (Python), or Maven (Java). Cargo is similar to these tools but generally more integrated and easier to use. That’s quite an understatement because most package managers seem designed from the ground up to make a developer’s life more complicated than it needs to be, but Cargo is designed to simplify, simplify, simplify.

What Cargo can do

Check out this list of Cargo’s main capabilities:

- » **Creating projects:** Sets up new Rust projects with the right structure (refer to “Creating Your First Rust Project,” coming right up).
- » **Managing dependencies:** Downloads and manages external code libraries (called “crates” in Rust) that your code requires (a required library is called a *dependency*). Refer to Book 2, [Chapter 7](#) to learn about crates and dependencies.
- » **Compiling code:** Manages the compilation of your code and all its dependencies into an executable file. Refer to the section “[Compiling](#)

[and Running Your First Program](#)” for more.

- » **Testing programs:** Runs tests to make sure your code works the way it’s supposed to. You learn how to set up tests in Book 3, [Chapter 7](#).
- » **Documenting code:** Generates documentation from your code. Book 3, [Chapter 7](#) is also the place to be to learn how code documentation works.
- » **Publishing projects:** Helps you share your code with others.

If you’re new to coding, believe me when I tell you that this is a *very* impressive list. In most other languages, you need a bunch of separate tools to accomplish all these tasks. Cargo does it all — and with little fuss or fanfare.

Cargo also makes sure that every Rust project follows the same set of helpful conventions. Every Rust project you create — heck, every Rust project *everyone* creates — has the same basic structure, which means that once you understand one Rust project’s foundation, you have a head start on understanding any other Cargo-based project. This consistency is a gift that keeps on giving as you work on different Rust codebases.

Cargo’s ease of use and consistency might not sound important to you right now, but the wonderful thing about Cargo is that it not only makes developing with Rust smoother and easier but also makes the development process *enjoyable*. For most common tasks, Cargo handles the details automatically, letting you focus on writing code rather than wrestling with configuration files and build scripts.

Essential Cargo commands

For the sake of having a handy reference, following is a list of the Cargo commands you’ll use most often (don’t worry if this list doesn’t make sense to you now; all will be explained in due course):

- » **cargo new *project*:** Creates a new Rust project with the name *project*.
- » **cargo build:** Builds the current project without running it.

- » **cargo run:** Builds the current project and then launches the resulting program.
- » **cargo check:** Checks your code for errors without creating an executable. This method is faster than a full build and useful for quick error checking.
- » **cargo test:** Runs all the tests in your project (refer to Book 3, [Chapter 7](#)).
- » **cargo doc:** Generates documentation for your project (refer to Book 3, [Chapter 7](#)).
- » **cargo clean:** Deletes all build artifacts. This command is useful if something seems wrong with your build.

Setting Up Your Development Environment

Another wonderful aspect of Rust is that, once it's installed, you need only one more thing to start code: a text editor. Yep, that's it. All Rust code resides in plain text files that use the `.rs` file extension. So, will any garden-variety text editor do? Technically, yes, but Rust is a serious programming language, so you should be using a serious editor. That might sound like an expensive proposition, but two popular options for coding your Rust projects are completely, blissfully, wallet-bypassingly free:

- » **Visual Studio Code:** Known in the trade as VS Code or simply Code, this Microsoft editor combined with the rust-analyzer extension is hard to beat. Versions are available for Windows, macOS, and Linux. Refer to the installation and configuration instructions that follow.
- » **Rust Rover:** This editor was built with Rust in mind, so it's a great choice for doing all things Rust. You can download Rust Rover from www.jetbrains.com/rust/.

If you're an experienced programmer with an emotional attachment to an existing text editor, such as Vim, Emacs, or Sublime, feel free to keep using your favorite app. Rust plug-ins are available for most big-time editors, so just be sure to install the appropriate one before continuing.

If you're new to coding, VS Code is the best way to go (and it's the app I use throughout this book). Here are the steps to install VS Code and the rust-analyzer extension, which provides top-notch Rust support:

1. Download and install VS Code from

<https://code.visualstudio.com/>.

2. Run VS Code.

3. Click the Extensions icon in the sidebar (pointed out in [Figure 2-1](#)).

Alternatively, press Ctrl+Shift+X (Windows or Linux) or ⌘+Shift+X (macOS).

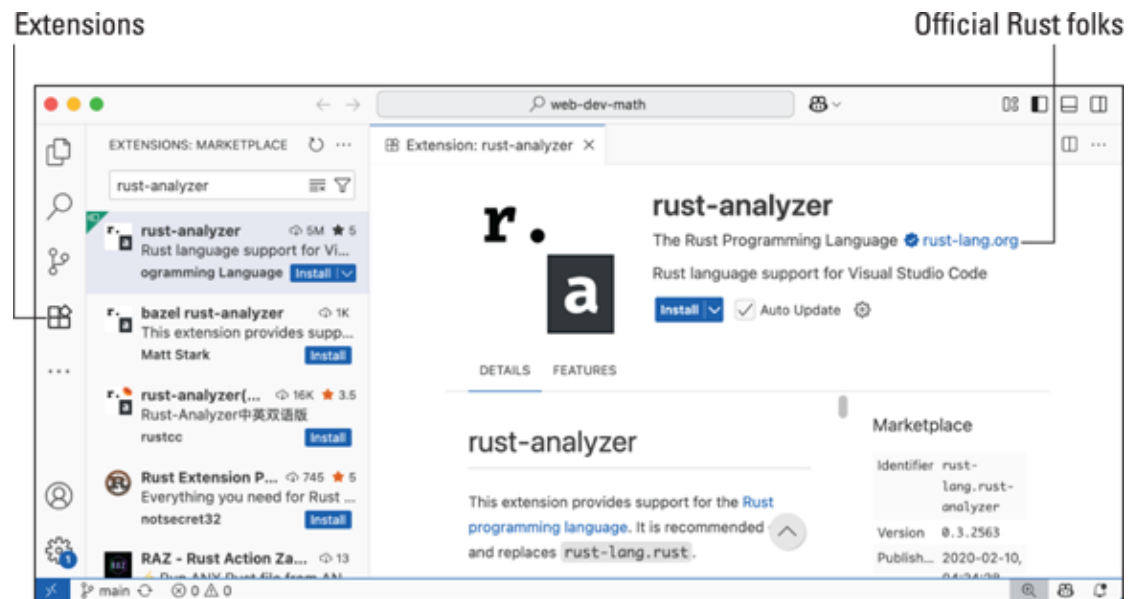


FIGURE 2-1: In VS Code, use the Extensions pane to search for and install the rust-analyzer extension.

4. In the Search box at the top of the Extensions pane, type rust-analyzer (as shown in the figure).

5. In the search results, click rust-analyzer.

Multiple results that include the phrase *rust-analyzer* are likely to show up. Before moving on to Step 6, make sure you're on the extension provided by rust-lang.org (refer to [Figure 2-1](#)), which is the official Rust language source.

6. Click Install.

If VS Code mumbles something about trusting the publisher “The Rust Programming Language,” say “But of course!” and click Trust Publisher & Install.

VS Code installs the rust-analyzer extension.

That's it! VS Code will now highlight Rust syntax, show you errors as you type, and provide helpful suggestions. Sweet.

Creating Your First Rust Project

Now comes the fun part: creating your first honest-to-goodness Rust project. As you learn shortly, Cargo makes this process almost embarrassingly simple.

First, though, you need somewhere to store your stuff. That is, you need to create a directory that will act as the home base for all your Rust projects. I suggest that you create a new subfolder with a name such as `rust_projects` (or whatever name you prefer) in your user account's main folder. You could do this using your operating system's file app (such as File Manager in Windows or Finder in macOS), but if you want more command-line practice, open a terminal (if you're using Windows, make sure it's PowerShell) and enter the following two commands:

```
% mkdir ~/rust_projects
% cd ~/rust_projects
```

In the Windows command prompt, do this, instead:

```
mkdir %USERPROFILE%\rust_projects
cd %USERPROFILE%\rust_projects
```

In all these commands, the tilde (~) represents your user account directory, so the first command makes a subdirectory named

`rust_projects` and the second command moves into that subdirectory.

Now you're ready to make things happen! At the command line, enter the following:

```
% cargo new hello-world
```

This command says, in effect, “Hey, Cargo! Create a new Rust project named `hello-world` for me, would you?” You immediately get the following in response:

```
Creating binary (application) 'hello-world' package
note: see more 'Cargo.toml' keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html
```

This seems like gibberish, but all it really means is that Cargo, based on nothing more than a simple command, created a complete, ready-to-run Rust project for you!

Understanding the Project Structure

Every Rust project that Cargo creates follows the same foundational structure. This consistency means that understanding even the simplest Rust project gives you a useful starting point for navigating larger projects.

Exploring the directory structure

The first thing Cargo did was create a subdirectory to store your project. That subdirectory — known as your project's *root* directory — has the same name as whatever you specify as the project name. That name is `hello-world` in the example, so navigate to that new subdirectory and ask for a listing of its contents:

```
% cd hello-world
% ls
```

The listing includes at least a file named `Cargo.toml` (which I discuss shortly) and a subdirectory named `src` (short for *source*; refer to the next section), which is where Cargo stashed the Rust source code that it generated automatically for you.

Checking out the *src* directory

To check out what's inside `src`, run the following command:

```
% ls src
```

(At the Windows command prompt, use `dir src` instead.) A single filename appears: `main.rs`. This is the file that stores the Rust code. To take a peek at that code, you have a couple of options:

- » At the command line, type `cat src/main.rs` and press Enter or Return.
- » In VS Code, follow these steps:
 1. **Choose File ⇒ Open Folder to display the Open Folder dialog.**
 2. **Navigate to and open your user profile's `rust_projects` folder (or whatever you named it).**
 3. **Click Select Folder.**
 4. **When VS Code asks if you trust the authors of this folder, click Yes, I Trust the Authors.**
 5. **In the Explorer pane, click `hello-world`, click `src`, and then, at long last, click `main.rs`.**

The code appears, as shown in [Figure 2-2](#).

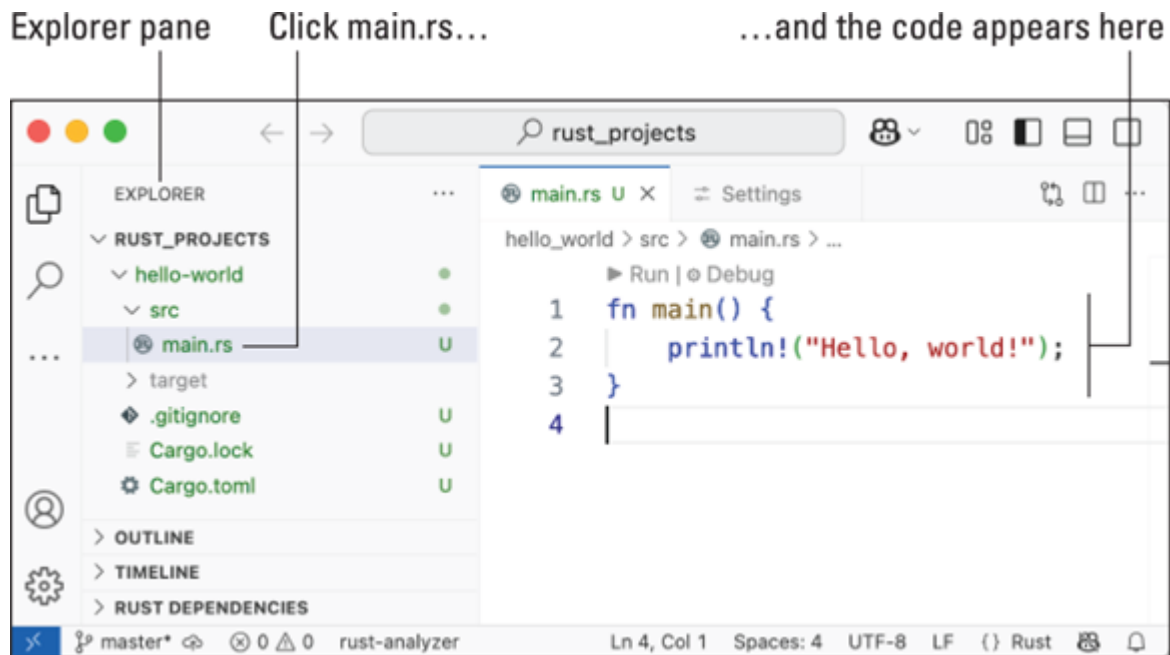


FIGURE 2-2: Using VS Code to view the `main.rs` file created by Cargo.

Cargo automatically generated a complete “Hello, world!” program for you (which is, by long-standing tradition, everyone’s first program). You don’t need to understand exactly what’s going on with this code (although you will soon enough), but here are two things to note for now about this little program:

- » The `fn main()` part is the main entry point for every Rust program. When you run any Rust app, it runs whatever code resides between the braces (`{` and `}`).
- » The `println!` statement outputs to the terminal window whatever appears between the quotation marks inside the parentheses.

It’s not much, but it’s a fully functional Rust program that (as you learn in just a sec) will compile and run flawlessly.



REMEMBER You might think that naming this first project `hello-world` had something to do with getting a “Hello, world!” program, but that’s not the case. This little Rust app is what Cargo always forges when

you create a Rust project. You could have run `cargo new rumpelstiltskin` and the contents of `main.rs` would have been the same.

Examining the Cargo.toml file

The Cargo.toml file is your project’s configuration file. TOML, which stands for “Tom’s Obvious, Minimal Language” (yes, really), is designed to be easy for humans to read and write.

Let’s look at what Cargo generated for you:

```
[package]
name = "hello-world"
version = "0.1.0"
edition = "2024"

[dependencies]
```

This might look sparse, but it contains everything Cargo needs to know about your project:

- » **name:** The name of your project. This becomes the name of the executable file that gets created when you build your project (as I describe in the “[Compiling and Running Your First Program](#)” section).
- » **version:** The current version of your project. Cargo uses semantic versioning, such as 1.2.3, where the numbers represent, respectively, your program’s major, minor, and patch versions.
- » **edition:** Which edition of Rust your project uses. Rust releases new editions every few years that can include small changes to the language. Don’t worry about this for now; the latest edition is always a good choice.
- » **dependencies:** This section lists external libraries (crates) that your project uses. It’s empty now, but you’ll add some dependencies to your projects as you work through the book.

Other files and directories you might encounter

Most Rust projects also include the following items:

- » **cargo.lock:** This file appears after you build your project or add dependencies. It locks in the exact versions of all dependencies to ensure reproducible builds. You generally don't need to edit this file manually or even think about it ever again.
- » **target:** This directory is created when you build your project. It contains all the compiled code and other build artifacts. Your compiled code (refer to the next section) is stored in this directory.
- » **.gitignore:** This Git-related file specifies which files or directories or both to ignore. Cargo automatically initializes your project as a Git repository, assuming you'll want version control. If you don't know what Git is, don't worry about it because it won't affect your ability to write Rust code.

Compiling and Running Your First Program

Okay, you're very close to running your first Rust program. The next step is to compile it, where *compiling* means that the plain text that resides in `main.rs` gets converted into code that your computer can execute. The result is an *executable* file that you (or anyone else, for that matter) can run.

To compile your project, head for a command-line window, make sure you're in the project's main directory (such as the `hello-world` directory in our sample app), and then run the following command:

```
% cargo build
```

What just happened? Cargo checked your code for errors and, if it didn't find any gaffes, compiled your code into an executable file, which it added to the `target/debug` folder. So now you're ready to run your code. At the command line, enter the following:


```
% ./target/debug/hello-world      # macOS or Linux
> .\target\debug\hello-world.exe # Windows
```

(The dot [.] at the start of the command is a reference to the current directory.) A split second later, the result appears:

```
Hello, world!
```

Congratulations, you just created, compiled, and ran your first Rust program!



REMEMBER

MEMORY 101: PROGRAMS LIVE AND BREATHE MEMORY

In documentation, user forums, and books related to languages such as Python and JavaScript, you'll rarely come across the word *memory*. But in the Rust documentation, Rust user hangouts, and very much in this book, talk of memory is everywhere. What's going on here?

First, know that memory is a huge topic for languages such as Python and JavaScript. It's just that all the memory-related stuff (such as the garbage collection I mention in Book 1, [Chapter 1](#)) happens way behind the scenes, so most programmers never have to think about it.

In other words, *every* programming language must manage memory at some level. Why? Because memory is where programs live. Sure, the program's executable file and its data (if it uses any) reside on some physical medium such as a hard drive. But once you run the program and it reads its data, everything is loaded into the computer's memory. But here are three important facts about memory:

- **Memory is a finite resource.** Depending on the device on which the program is running, there is only so much memory for a program and its data, so not using more memory than is strictly necessary is a solid programming goal.
- **Some memory operations are faster than others.** Allocating data on the stack is much faster than allocating on the heap because the stack uses a simple pointer adjustment, while heap allocation requires finding and tracking free memory blocks.
- **Some program operations are faster than others.** A big part of Rust programming is manipulating things in memory, but some of those manipulations take longer and use more resources than others. Your job as a Rust programmer is to not waste precious memory and resources by using an inefficient operation when a more efficient one exists.

This focus on memory efficiency is exactly why Rust doesn't use a garbage collector like Python or JavaScript or rely on you to control memory allocation directly. Instead, Rust uses strict rules to handle memory usage while keeping you safe from common memory mishaps.

Compiling and running in one fell swoop

In the preceding section, you compiled and ran your program with two commands:

```
% cargo build
% ./target/debug/hello-world
```



TIP

There's nothing onerous about this two-step procedure, but why do two things when you can get away with doing just one? When you need to run both steps — the compile and the execution — you can combine them into a single command:

```
% cargo run
```

Nice!



REMEMBER

Once your Rust programs get beyond a certain size (and you'll be building some larger apps in this book), it can take quite a while to compile the Rust code into an executable file. If all you really want to know is whether your code contains any errors, it's best to run `cargo check` instead of `cargo build` or `cargo run`.

Compiling for release

When you run `cargo build` or `cargo run`, the Rust compiler creates a directory called `target` because it's where Cargo places the target of your build: the compiled executable. It's actually a *debug* version of the executable file, which includes extra information called debug symbols that help you track down problems by mapping the compiled code back

to your original source code. That's why the resulting executable gets stored in the `/target/debug` directory.

However, the debug version of the executable file is larger than necessary and not optimized for speed. Make no mistake: the debug version is absolutely fine while you're coding and testing your program. But when you're finally ready to send your program out into the world, you need to create the *release* version, which doesn't include the debugging info and does include a number of behind-the-scenes efficiencies (called *optimizations* in the programming trade) that noticeably speed up your program.

To compile the release version, run the following command from your project's root directory in any shell:

```
cargo build --release
```

If you also want to run the release version, use this command:

```
cargo run --release
```

Either way, the resulting executable is stored in your project's `target/release` directory.

Compiling and running from VS Code

In this chapter, you didn't write any code, but that's going to change quickly because the rest of the book is all about forging code with your bare hands. That coding happens in your text editor, and the usual workflow is the following:

1. Modify whatever `.rs` file you're working on.
2. Save your changes.
3. Switch to your terminal program.
4. Enter `cargo run` to compile and run your program, and then check the results.
5. Switch back to your text editor.
6. Repeat Steps 1 through 5 until your program is done or sleep beckons.

That's a lot of switching back and forth between text editor and shell. But here's the good news: If you're running VS Code, that program has a built-in terminal app! Having a shell right inside VS Code means you can run all your Rust-related commands without leaving the friendly confines of VS Code.

To open the Terminal panel in your Rust project's root directory, right-click that directory in the Explorer pane, and then click Open in Integrated Terminal. Alternatively, either select View ⇒ Terminal or press Ctrl+` (back tick), and then use the Terminal panel prompt to run the `cd project` command, where *project* is the path to your project's root directory.

The Terminal panel appears at the bottom of the VS Code window and you're now free to use that panel to enter the same commands as you would in your regular shell. In [Figure 2-3](#), for example, I entered the `cargo run` command in the Terminal panel and the result is the same as if I'd run the command in my terminal app.



TIP If you're running VS Code with the rust-analyzer extension installed, here's a handy trick you can use to run your compiled Rust code directly. Just above the code, click the Run command (pointed out in [Figure 2-3](#)). Clicking Run opens the Terminal panel (if it's not open already) and automatically enters the `cargo run` command. You're welcome!

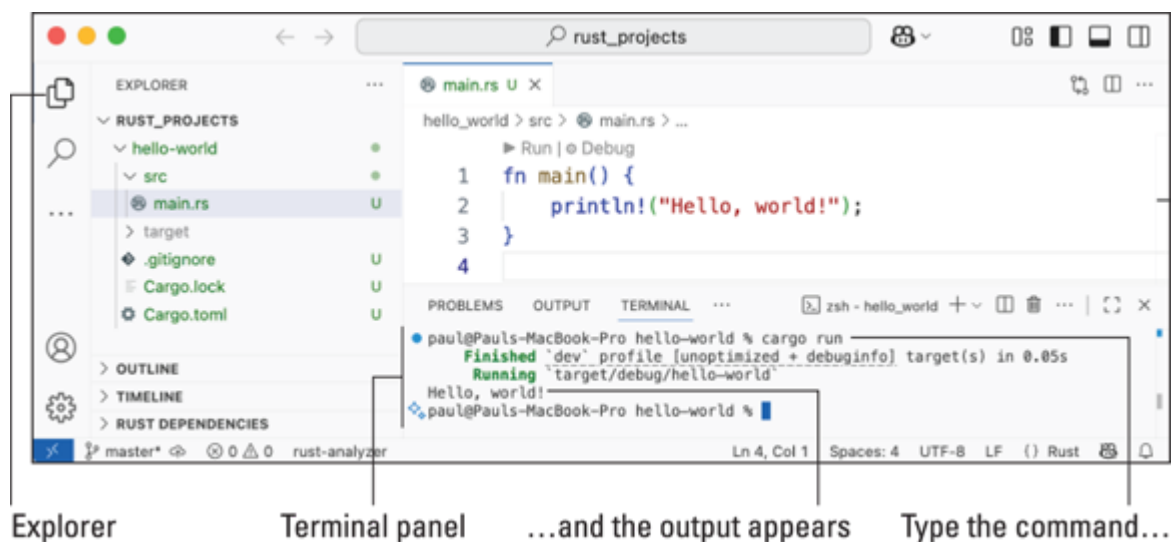


FIGURE 2-3: You can use VS Code’s Terminal panel to run shell commands. No more switching!

Okay, nice work out there! With Rust installed, you’re now ready to start exploring the basic knickknacks of the language, which is what the rest of the chapters here in [Book 1](#) are all about.

Learning Rust: A Roadmap

Rust is a big, sprawling language and learning it can often feel like you’re adrift on a vast ocean, picking up bits of programming flotsam and computer science jetsam here and there. But that ocean has a few major islands that represent the foundational concepts that you need to grasp to truly understand Rust. These concepts won’t mean much to you now, but I present them here so that when you come across them as you work through the book, you’ll know these are places you need to spend time exploring. There are eight foundational concepts in all:

- » **Understanding how memory works:** Most programming languages hide the difference between stack memory (fast, automatic, limited) and heap memory (slower, manually managed, vast). Rust makes this distinction visible and important. Throughout this book, you’ll find sidebars labeled “[Memory 101](#).” Don’t skip these because I use them to teach you everything you need to know about memory from a Rust angle.

- » **Immutability by default:** Everything in Rust is unchangeable (the technical term is *immutable*) unless you explicitly say otherwise. This concept is a major safety feature because things that are immutable can't be changed in a way that creates security or reliability problems. It's also a different way of thinking about data. Most languages assume that you want to change things and make you work to prevent it. Rust assumes that you don't want to change things and makes you work to allow it. You learn about immutability in Book 1, [Chapter 3](#).
- » **The strict type system:** Although most of the time Rust is smart enough to infer what type of data you're defining, when it can't, Rust refuses to guess what you mean. If you have an integer and try to use it as text, Rust won't mindlessly convert it for you, like some languages would. Rust makes you be explicit about the conversion. This type fussiness feels restrictive at first, but it prevents countless subtle bugs. You get introduced to Rust's data types in Book 1, [Chapter 3](#), and then learn more in Book 2, [Chapters 2](#) through [5](#).
- » **Ownership:** In Rust, every piece of data has one and only one owner at any time. When that owner disappears, the owner's data is automatically dropped from memory. This simple rule eliminates entire categories of bugs related to memory leaks and dangling pointers. Ownership is the topic of Book 2, [Chapter 1](#).
- » **Borrowing:** The owner of some data can lend out temporary access to that data without giving up ownership. Borrowing prevents you from wasting precious memory by creating a copy of the data every time you want to use it somewhere else. Borrowing also gives you a lot more flexibility because you can allow other parts of your program to work with the same data without any of them having to consume the data and therefore make it unavailable to others. Borrowing is also covered in Book 2, [Chapter 1](#).
- » **Explicit error handling:** Operations that might fail return a `Result`, and things that might not exist return an `Option`. Rust forces you to acknowledge that uncertainty exists instead of hoping nothing goes wrong. No more null pointer exceptions at runtime. I introduce you

to error handling in Book 2, [Chapter 3](#), and then get into it in more detail in Book 3, [Chapter 4](#).

- » **Scope as an organizing principle:** When variables come into existence and when they cease to exist is predictable and visible in your code. Scope isn't just about memory management; it's about understanding the lifetime of all your data. You get your first look at scope in Book 1, [Chapter 6](#), and then learn more in Book 1, [Chapter 7](#). The concept of lifetimes gets its time in the spotlight in Book 3, [Chapter 3](#).
- » **The compiler as teacher:** Rust's error messages are notably helpful compared to many other languages. Rather than just saying "this is wrong," the compiler often explains why and suggests fixes. That said, some error messages can still be cryptic, especially when you're learning. Don't be discouraged if you encounter one that leaves you scratching your head. You get your first look at Rust's wondrous compiler messages in Book 1, [Chapter 3](#).

Chapter 3

Speaking Rust: The Basics

IN THIS CHAPTER

- » Messing around with variables
 - » Peppering your code with comments
 - » Getting to know Rust’s data types
 - » Getting your standard library card
-

It should be noted that the authentic Rust learning experience involves writing code, having the compiler scream at you, and trying to figure out what the heck that means.

—ARIA DESIRES

Rust is a programming *language*. That is, it’s not a programming *instruction set* or a programming *system* or a programming *scheme*. It’s a *language* and that metaphor can be helpful because in a very real sense you’re learning to speak Rust. You’re learning the words, phrases, grammar, and idioms that enable you, via Rust, to make a computer do your bidding. But instead of asking “Where’s the bathroom?,” you’re building cool things such as utilities, apps, and even (way down the road) servers.

In this chapter, you get your first lesson in speaking Rust by learning about variables, comments, data types, and a few other basics. Don’t worry even a teensy bit if some of this feels overwhelming at first. Learning a programming language is like learning any language: You start with the basics, practice a lot, and gradually become more fluent. Let this chapter be your first step toward Rust fluency.

main(): A Home for Your Code

As I describe in Book 1, [Chapter 2](#), when you run `cargo new name`, where *name* is the moniker you want Cargo to assign to your new project, one of the essential tasks that Cargo performs is to create a file named `main.rs`, which contains the following Rust code:

```
fn main() {  
    println!("Hello, world!");  
}
```

Here, the `fn` part tells Rust that `main()` is a function. You learn all about functions in Book 1, [Chapter 7](#); for now it's enough to know that a function is just a way to run whatever Rust code is shoehorned between the braces (`{` and `}`). In particular, when you run a compiled Rust executable, the first things Rust does are look for `main()` and then run that function's code.

The upshot is that, at least for now, `main()` is where the action begins. Each time you use Cargo to create a new Rust project, the next thing you should do is open that project's `main.rs` file in your text editor, delete the `println!("Hello, world!");` statement, and replace it with your own code.

Variables: Storing Your Digital Stuff

Most good cooks and bakers, consciously or not, follow the principle of *mise en place* (in French, putting in place), which means to gather and organize ingredients before getting started.

Good programmers, too, follow something like *mise en place*, but instead of gathering flour, water, salt, and yeast, a programmer gathers data. And where does a programmer place each bit of data? In a variable, which is a memory location for storing a particular piece of data.

Variables are like labeled containers where you can store information. For example, as part of the prep for making bread, a baker might put the required amount of salt in a container labeled “Salt” and the required amount of yeast in another container labeled “Yeast.”

In Rust, variables work similarly: they’re named containers that each hold a different type of data. Here’s how you create — or *declare*, in programming lingo — your first variable:

```
let favorite_number = 42;
```

It’s a short statement, but there’s a lot going on here:

- » `let` is the keyword that tells Rust “I want to create a new variable.”
- » `favorite_number` is the name assigned to the variable (the label on the container).
- » `=` assigns a value to the variable (that is, it tells Rust that the value that you want to put in the container is coming next). In this context, `=` is called the *assignment operator*.
- » `42` is the value that gets stored in (is *bound* to, in the vernacular) the variable.
- » The semicolon (`;`) is required at the end of most statements to tell Rust, “This statement is complete.”

Here’s an example that shows the variable declaration in its native habitat (in this book’s example files, refer to `bk01ch03/example01/`):

```
fn main() {  
    let favorite_number = 42;  
    println!("My favorite number is {favorite_number}!");  
}
```

Running this code produces the following output in the terminal:

```
My favorite number is 42!
```

(For more about what `println!` is doing here, check out the “[Outputting Text](#)” section.)

Variable names in Rust must follow a few mercifully simple rules. They

- » Can contain only letters, numbers, and underscores.
- » Can’t start with a number.

- » Should use *snake case* for multiword names, where each word is separated by an underscore (`_`), as in `snake_case`. (Using snake case is optional, but if you don't use it, Rust will display warning:
variable 'x' should have a snake case name each time you compile your program. Who needs the hassle?)
- » Can't be any of Rust's *reserved words*, which are keywords used by the language itself, such as `let` and `fn`. For the full list, head over to <https://doc.rust-lang.org/stable/std/index.html#keywords>.



REMEMBER Good variable names are as short as possible while still describing what the variable contains. The variables `user_age`, `total_cost`, and `is_complete` are examples of good names. Some not-so-good variable names are `item` (too general), `total_cost_of_all_purchased_items` (too long-winded), `a` or `x` (too cryptic), and `fy26` (too short-winded).



REMEMBER

MEMORY 101: HOW VARIABLES LIVE IN MEMORY

When you write something like `let favorite_number = 42;` in Rust, you're essentially asking Rust to do three *mise en place* tasks for you:

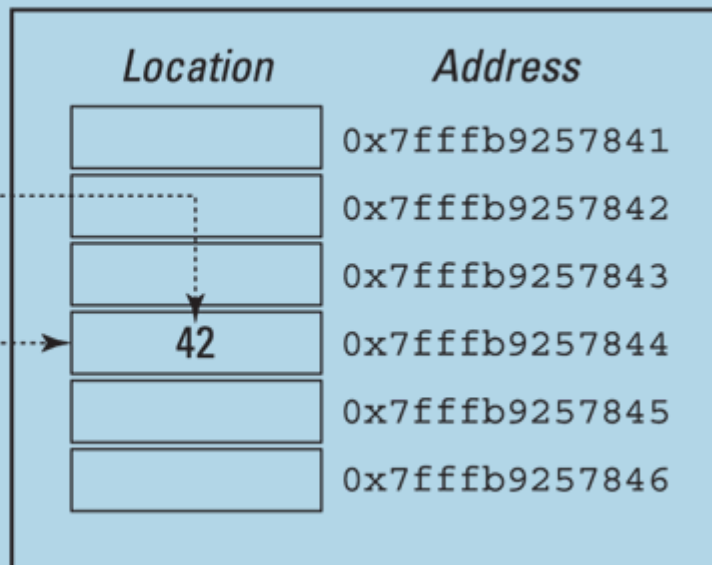
1. Find an empty spot in memory.
2. Write the number 42 into that spot.
3. Create a label named `favorite_number` that points to that memory location.

Imagine memory as a set of lockers, each of which is assigned a number that represents the locker's address. In the preceding procedure, Rust locates an empty locker and takes note of its address, which is, say, `0x7fffb9257844`. Rust stores the number 42 in that locker (technically, Rust would store the hexadecimal [base 16] equivalent of 42, which is `2a`), and then records that "favorite_number" refers to locker `0x7fffb9257844`. Now, the next time you need to use the contents of that locker, you can

just use `favorite_number` instead of the meaningless (to us humans, anyway) collection of numbers and letters that make up the memory address. In other words, the variable name is a human-friendly way to refer to a memory address.

Memory

```
fn main() {  
    let x = 42;  
    ...  
}
```



Different types (refer to the section “[Talking About Types](#),” coming up shortly) take up different amounts of space. An integer might take up 4 bytes, like the size of a small locker. A string might require dozens or hundreds of bytes, so it would be like renting a storage unit.

Knowing that variables are labels pointing to memory locations is the first step in creating a mental model of how Rust deals with memory, which is crucial for truly understanding the Rust way of doing things.

If you want to examine the memory address used to store a variable, you can use code similar to the following: (bk01ch03/example02/):

```
fn main() {  
    let favorite_number = 42;  
    println!("My favorite number is {favorite_number}!");  
    println!("Memory address of my favorite number is {:p}",  
        &favorite_number);  
}
```

This code will display something like the following (the actual address will vary each time):

```
My favorite number is 42!  
Memory address of my favorite number is 0x7fffb9257844
```

Make Mine a Mutable

One of the strange things about variables in Rust is that, by default, they don't vary! That is, if you declare a variable and give it a value, and then later try to change that value, the Rust compiler will stop you in your tracks. Try this and check out what happens (bk01ch03/example03/):

```
fn main() {
    let favorite_number = 42;
    println!("My favorite number is {favorite_number}!");
    favorite_number = 16;
    println!("I changed my mind. My actual favorite is {favorite_number}!");
}
```

This code first sets `favorite_number` to 42, and then later tries to change `favorite_number` to 16. Here's what the Rust compiler has to say about this:

```
% cargo run
   Compiling example02 v0.1.0 (/Users/paul/Documents/git/rust-aio-
fd/bk01ch03/example02)
error[E0384]: cannot assign twice to immutable variable 'favorite_number'
--> src/main.rs:4:5
   |
2 |     let favorite_number = 42;
   |     ----- first assignment to 'favorite_number'
3 |     println!("My favorite number is {favorite_number}!");
4 |     favorite_number = 16;
   |     ^^^^^^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable variable
   |
help: consider making this binding mutable
   |
2 |     let mut favorite_number = 42;
   |         +++

For more information about this error, try 'rustc --explain E0384'.
error: could not compile 'example02' (bin "example02") due to 1 previous
error
```

Okay, wow, that's a lot! You'll be getting this kind of compiler feedback a *ton* in your Rust career. I want to take a bit of time here to dig deep into the message because Rust's compiler messages are one of its greatest teaching tools. Here we go:

```
error[E0384]: cannot assign twice to immutable variable 'favorite_number'
```

This line tells you the specific error that occurred. As I mention, by default Rust variables can't be changed — that is, they are *immutable* — so here the compiler is telling you that you tried to “assign (a value) twice” to the “immutable variable” named `favorite_number`.

```
--> src/main.rs:4:5
```

This line tells you where the error occurred: in the `src/main.rs` file, on line 4, beginning in column 5.

```
2 |     let favorite_number = 42;
  |     ----- first assignment to 'favorite_number'
```

These lines show you the first time you assigned a value to the `favorite_number` variable. The dashes act as a kind of underline to point out the relevant code.

```
4 |     favorite_number = 16;
  |     ^^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable variable
```

And these lines show you the second time you tried to assign a value to the `favorite_number` variable. Again, the carets (^) underline the relevant code, so you can determine exactly where you went wrong. Helpfully, the compiler also repeats the error message.

```
help: consider making this binding mutable
```

Now the compiler switches into teaching mode. First it offers the suggestion `consider making this binding mutable`. Here, *mutable* means that the value of the variable is allowed to change. (Note, too, that a *binding* is an association between a variable name and a value.) That's helpful, but then the compiler does something quite wonderful:

```
2 |     let mut favorite_number = 42;
  |         +++
```

That's right: the compiler shows you *exactly* what you need to do to, in this case, make `favorite_number` mutable! That is, you insert the keyword `mut` (for *mutable*) between `let` and the variable name. The compiler also underlines `mut` with plus signs (+) to make everything crystal clear.

Clear, specific, and helpful messages such as this one are why, year after year, developers vote Rust as the most admired and desired language on Stack Overflow (refer to Book 1, [Chapter 1](#)).

Okay, so now you follow the compiler's advice and make `favorite_number` mutable by inserting the `mut` keyword (bk01ch03/example04/):

```
fn main() {  
    let mut favorite_number = 42;  
    println!("My favorite number is {favorite_number}!");  
    favorite_number = 16;  
    println!("I changed my mind. My actual favorite is {favorite_number}!");  
}
```

This time the program compiles and the following output appears:

```
My favorite number is 42!  
I changed my mind. My actual favorite is 16!
```

Annotating Your Code with Comments

To help you decipher your code, particularly when you're looking at the code a few weeks or months after you first wrote it, it's good programming practice to make liberal use of comments throughout the script. A *comment* is text that describes or explains a statement or group of statements. Comments are ignored by the compiler, so you can add as many as you deem necessary.

For short, single-line comments, use the double-slash (`//`). Put the `//` at the beginning of the line and then type your comment after it (with a space in between for readability). Here's an example:

```
// Set favorite number  
let favorite_number = 42;
```

You can also add a comment after a statement:

```
let mut favorite_number = 42; // Favorite number (I think)
```

It's fine to use `//` for two or three comment lines in a row. However, Rust also supports multiple-line comments that begin with the `/*` symbol and end with the `*/` symbol. Here's an example:

```
/*
This code is released under the Creative Commons Zero
v1.0 Universal (CC0 1.0) license. To the fullest extent
permitted by law, I (the author) have dedicated this
work to the public domain.

For more details, refer to:
https://creativecommons.org/publicdomain/zero/1.0/
*/
```

Outputting Text

It's common for Rust programs to output the program's result as text for the user to read. Where does this output appear? In the *standard output* (often abbreviated as *stdout*), which on most systems means the terminal window in which the program is running.

In Rust, you print (the verb preferred over *output*, oddly) to standard output using `println!`. That exclamation point isn't a mark of excitement at the prospect of displaying text to the user. Instead, it signifies that you're dealing with a Rust macro, which is a script that runs multiple Rust statements (refer to Book 3, [Chapter 5](#) for more on macros).

The simplest way to use `println!` is to output a text message (bk01ch03/example05/):

```
fn main() {
    println!("Hello, world!");
    println!("This is a simple message.");
}
```

Dutifully, Rust displays the text in the terminal window when you run the compiled program:

```
% cargo run ./target/debug/example05
Hello, world!
This is a simple message.
```


Note that `println!` outputs the messages on separate lines. That's what the `ln` part of `println!` means: that Rust outputs the message, followed by a *newline* character, which ensures that the next output starts on a fresh line.

But `println!` can do much more than just print simple text. You can use *named placeholders* — a variable name surrounded by braces `{ }` — to insert variables into your output (bk01ch03/example06/):

```
fn main() {
    let name = "Alice";
    let age = 30;

    println!("Hello, {name}!");
    println!("{name} is {age} years old.");
}
```

When the compiler stumbles upon a variable name surrounded by braces (such as `{name}`), it replaces the variable name with its value when outputting the message:

```
% cargo run ./target/debug/example06
Hello, Alice!
Alice is 30 years old.
```

To include the result of an expression (refer to Book 1, [Chapter 4](#)) or function (Book 1, [Chapter 7](#)), you need to use a *blank placeholder* — just the braces `{ }` with nothing inside — in the message, type a comma, and then specify the expression or whatever (bk01ch03/example07/):

```
fn main() {
    let name = "Alice";
    let age = 30;

    println!("In 10 years, {name} will be {}. ", age + 10);
}
```

Here's the output:

```
% cargo run ./target/debug/example07
In 10 years, Alice will be 40 years old.
```

You can also use numbered placeholders if you want to use the same value multiple times or use them in a different order

(bk01ch03/example08/):

```
fn main() {  
    let first_name = "Alice";  
    let last_name = "Johnson";  
  
    println!("{0} {1} is also known as {1}, {0}.", first_name, last_name);  
}
```

Output:

```
Alice Johnson is also known as Johnson, Alice
```

For more control over formatting, you can specify how numbers should be displayed (bk01ch03/example09/):

```
fn main() {  
    let pi = 3.14159265359;  
  
    println!("Pi to 2 decimal places: {:.2}", pi); // Output: 3.14  
    println!("Hexadecimal: {:x}", 255);           // Output: ff  
    println!("Binary: {:b}", 255);                 // Output: 11111111  
}
```

There's also `print!` (without the `ln`), which doesn't print the newline character after the output (bk01ch03/example010/):

```
fn main() {  
    print!("Loading");  
    print!(".");  
    print!(".");  
    println!(".");  
    println!("Done!");  
}
```

Output:

```
Loading...  
Done!
```

Talking about Types

Every piece of data that you use in Rust has a type that tells you the general category into which the data fits. In real life, you might slot things into categories such as food, clothing, or electronics. In

programming, there are categories that cover the different kinds of values you can bind to a variable, including whole numbers, text, and true/false values. A data type is a specific instance in such a category.

Scalar values

When you step on a bathroom scale, one value appears: your weight. When you examine a temperature scale, one value appears: the temperature. So, broadly speaking, a device with a scale is something that provides a single value and you can therefore describe that device as *scalar*.

In Rust, a *scalar value* (or just a *scalar*) is a data type that stores just one value. You might be thinking that’s kind of obvious because how could anything store more than one value? Rust actually offers a fistful of ways to store multiple values in a single variable, and that’s such an important topic that I devote several chapters to it a bit later in the book (refer to Book 2, [Chapters 2](#) through [4](#)).

For now, though, I’m going to keep things simple and talk just about scalar values. Rust has four fundamental scalar types: integer, floating-point, character, and Boolean.

The integer types

An *integer* is a number without a fractional component, such as 42, 625, and -27 (but not 3.14159 or -2.4). What could be simpler, right? If only! Rust defines a *dozen* different integer data types. [Table 3-1](#) lists them all.

TABLE 3-1 Rust’s Integer Data Types

Type	Description	Range	Most Appropriate Use
u8	8-bit unsigned integer	0 to 255	Raw bytes, ASCII chars, color channels, compact storage in large arrays
i8	8-bit signed integer	-128 to 127	Small signed values, data compression formats, niche cases in embedded systems

Type	Description	Range	Most Appropriate Use
u16	16-bit unsigned integer	0 to 65,535	Small counters, pixel dimensions, network protocol fields
i16	16-bit signed integer	−32,768 to 32,767	Audio samples, signed pixel data, compact numerical storage
u32	32-bit unsigned integer	0 to 4,294,967,295	Large counts, IDs, bit flags, memory sizes on 32-bit systems
i32	32-bit signed integer	−2,147,483,648 to 2,147,483,647	Default integer type for general-purpose math and logic
u64	64-bit unsigned integer	0 to 18,446,744,073,709,551,615	Large file sizes, 64-bit timestamps, hashing
i64	64-bit signed integer	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	High-precision counters, financial calculations
u128	128-bit unsigned integer	0 to 3.4×10^{38}	Cryptography, very large IDs, special scientific computations
i128	128-bit signed integer	-1.7×10^{38} to 1.7×10^{38}	Cryptography, specialized scientific and financial workloads
usize	Pointer-sized unsigned integer	Platform-dependent (32-bit: 0–4,294,967,295; 64-bit: 0–18 quintillion)	Indexing collections, memory sizes, pointer arithmetic
isize	Pointer-sized signed integer	Platform-dependent (32-bit: −2,147,483,648 to 2,147,483,647; 64-bit: −9 quintillion to 9 quintillion)	Offsets in collections, pointer differences

What a list! Don't worry, though, I have notes. First and foremost, know these two things:

- » You don't have to memorize this list or even worry about it all that much! I put it here in all its glory as a reference when you need it and when you come across integer types in other folks' code.

- » Rust has a default integer data type: `i32`. This type is fine for the vast majority of integer work you'll do in Rust.

Okay, with those important reminders in your pocket, here are some notes to help you decipher [Table 3-1](#).

- » The number of bits tells you how much space in memory each value consumes. (Refer to the sidebar “Memory 101: 1s and 0s” in Book 1, [Chapter 1](#).)
- » *Signed* means the integer is allowed to have both positive and negative values; *unsigned* means the integer is allowed to have only positive values (or 0).
- » When you do want to choose a particular integer type, first check the size range to know which type is appropriate. For example, if you know your values will always be positive and no greater than 255, `u8` is the data type you want. Remember, though, that you can't go wrong here because the compiler will catch data type errors for you.
- » For the `usize` and `isize` types, *platform-dependent* means that the actual size depends on the size used by the memory model of whatever system the code is running on. For example, a 64-bit computer (which most modern computers are) will use 64-bit values.
- » Some of the use cases in the Most Appropriate Use column won't mean much to you now but are there for later when you might need to know which type to choose for which application.



TECHNICAL
STUFF

Are you wondering where the numbers in [Table 3-1](#)'s Range column come from? Okay, cool, here you go: A bit can be either 1 or 0, so when you have n bits, there are 2^n possible values. However, 2^n produces a binary number with $n + 1$ digits, which is one too many, so the actual number of possibilities is $2^n - 1$. So, for every unsigned integer data type, the range is from 0 to $2^n - 1$.

(With `u8`, for example, $2^8 - 1 = 255$, so the `u8` range is from 0 to 255.) Signed integers use up one digit for the sign (0 for positive, 1 for negative), so the range must be from $-(2^{n-1})$ to $(2^{n-1}) - 1$. (With `i8`, for example, $2^{8-1} = 2^7 = 128$, so the `i8` range is from -128 to 127.)

To specify a data type for a variable, modify the `let` statement to include a colon (`:`) right after the variable name, followed by the data type, surrounded by spaces. Here's an example (bk01ch03/example11/) that sets the data type of the `favorite_number` variable to `u8`:

```
fn main() {  
    let favorite_number: u8 = 42;  
    println!("My favorite number is {favorite_number}!");  
}
```



TIP Rather than specifying the type using `let name: type = value;` (for example, `let x: u16 = 1024;`), you can specify the type as a suffix using the following alternate form:

```
let name = valuetype;
```

where *name* is the variable name and *valuetype* is the value followed by the type, with no space between them. For example, the following two statements do the same thing:

```
let x: u16 = 1024;  
let x = 1024u16;
```



TIP When you need to write really large integers, Rust lets you use the underscore (`_`) as a separator for easier reading. That is, instead of writing the hard-to-decipher value `1872365940`, you can instead write `1_872_365_940`. Note that the positions of the underscores aren't significant, meaning you can put them anywhere that makes the number readable for you, such as `187_236_59_40`. Many Rust

programmers use an underscore to make data type suffixes easier to read:

```
let x = 1024_u16;
```



REMEMBER If you need to write a number in a base other than 10, Rust is happy to go along:

» **Hexadecimal (base 16):** Precede the hexadecimal value with `0x`.

Example: `0x2a`.

» **Octal (base 8):** Precede the octal value with `0o`. Example: `0o52`.

» **Binary (base 2):** Precede the binary value with `0b`. Example:

`0b101010`.

» **Byte:** Precede the byte value with `b` and surround the value with single quotation marks. Example: `b'H'`.

As I mention in the notes for [Table 3-1](#), the compiler will let you know if you use values that are out of range for a specified integer type.

Suppose, for example, you wrote the following (bk01ch03/example12/):

```
let favorite_number: u8 = 422;
```

When you try to run the program, the Rust compiler springs into action:

```
error: literal out of range for 'i8'
--> src/main.rs:2:31
   |
2 |     let favorite_number: i8 = 422;
   |                               ^^^
   |
   = note: the literal '422' does not fit into the type 'i8' whose range is
'-128..=127'
   = help: consider using the type 'i16' instead
```

Once again, you get a helpful error value, the location of the error, and the reason for the error. (Interpretation note: When the compiler offers a range of values such as `-128..=127`, it means everything from `-128` up to and *including* `127`. If the upper value wasn't to be included in the

range, the compiler would omit the = and you'd get something like -128..127, instead.) But the following is my favorite part of this message:

```
help: consider using the type 'i16' instead
```

The compiler doesn't assume you made a silly typo. Instead, it assumes you knew what you were doing when you typed `422` and is helping you use that value in your program!

The floating-point types

A *floating-point number* is a number with a fractional component, such as 1.618 or -1.5. Thankfully, Rust only defines two floating-point types:

- » `f32`: A 32-bit floating point number.
- » `f64`: A 64-bit floating point number. This is the default floating-point type because it offers more precision during floating-point calculations.

Here's an example (bk01ch03/example13/):

```
fn main() {  
    let golden_ratio:f32 = 1.61803399;  
    println!("The famous Golden Ratio is approximately {golden_ratio}.");  
}
```

As with the integer types, you can also use a type suffix for the floating-point types (remember that the underscore is there for readability; it's not a required part of the type):

```
let golden_ratio = 1.61803399_f32;
```

The character type

The `char` data type represents a single Unicode scalar character. Wait, what? Unicode? Unicode is a technology that keeps track of (give or take) every possible character from writing systems around the world — Latin, Cyrillic, Chinese, Arabic, and many more — over 150,000 of them as I write this. Each of those characters is given a unique numeric value. For example, the asterisk (*) has the value 42, whereas the digit 5 has the value 53. This sounds like ASCII, but Unicode extends way

beyond that to include math symbols (such as ∞), letters with accents (such as ç), oddball punctuation marks (such as ¶), and even emoji (such as 🥳)!

You specify a `char` value by hugging it with single quotation marks, as in the following examples (bk01ch03/example14/):

```
fn main() {  
    let infinity_symbol: = '∞';  
    let cedilla = 'ç';  
    let pilchrow: char = '¶';  
    let party_face: char = '🥳';  
}
```

It's okay that I didn't explicitly declare the `infinity_symbol` and `cedilla` variables with the `char` data type because the Rust compiler can figure that out on its own. To understand why, check out the section [“Type inference: The compiler knows all \(most of the time\).”](#)

The Boolean type

The `bool` data type is the simplest of all the literal data types because it can assume only one of two values: `true` or `false`. That simplicity may make it seem as though Booleans aren't particularly useful, but the capability to test whether a particular variable or condition is `true` or `false` is invaluable in Rust programming.

You can assign Boolean literals directly to a variable, like this (bk01ch03/example15/):

```
let task_completed: bool = true;
```

Alternatively, you can work with Boolean values implicitly using expressions:

```
current_month == "August"
```

The comparison expression `current_month == "August"` asks the following: Does the value of the `current_month` variable equal the string "August"? If it does, the expression evaluates to the Boolean value `true`; if it doesn't, the expression evaluates to `false`. I discuss comparison expressions in Book 1, [Chapter 4](#).

String literals

A *string* is a collection of one or more characters surrounded by double quotation marks (" and "). Compare this to the definition of a `char` type: a single character surrounded by *single* quotation marks (' and '). If you're skeptical about whether the type of quotation mark really makes a difference, try compiling this program (bk01ch03/example16/):

```
fn main() {  
    let greeting = 'Hello, world!';  
    println!("{}", greeting);  
}
```

The compiler is having none of it:

```
error[E0762]: unterminated character literal  
--> src/main.rs:2:34  
  |  
2 |     let greeting = 'Hello, world!';  
  |                                   ^^  
  |  
help: if you meant to write a string literal, use double quotes  
  |  
2 -     let greeting = 'Hello, world!';  
2 +     let greeting = "Hello, world!";
```

There it is in stark black and white: if you meant to write a string literal, use double quotes.



TIP Before moving on, I want to point out a helpful but easily missed teaching aid in the compiler's message. Examine the message once more, this time focusing on the final two lines. Note that near the start of the second-to-last line a minus sign (-) appears, and near the start of the last line a plus sign (+) appears. The minus sign acts as a kind of thumb down symbol, because what follows is your incorrect code. The plus sign, then, is like a thumb's up symbol, because what follows is the corrected code.

In all this talk of the string literal, what's missing? Right: I haven't told you the data type of a string literal. That's because it's complicated. You

can think of a string literal as a little note you've permanently scribbled in the margins of your program. It's built right into the program when you compile your code, and the string literal never changes because the computer keeps it tucked away in a special safe place in memory.

Later in the book (Book 2, [Chapter 5](#)), you learn that the official Rust name for this type is `&'static str`. But for now, just remember:

- » String literals are fixed text, baked into your program.
- » You can make a variable point to a string literal, but you can't go in and change the literal's letters.
- » If you want changeable text, you have to use a different type (called `String`), which you meet down the road (also in Book 2, [Chapter 5](#)).

Type inference: The compiler knows all (most of the time)

The Rust compiler is usually smart enough to figure out what type a variable takes based on how your code uses the variable. This is called *type inference*, and it takes some of the pressure off having to come up with the right type for every variable. For example:

```
fn main() {  
    let age = 25;           // Rust infers this is an integer  
    let name = "Bob";       // Rust infers this is a string literal  
    let is_awesome = true;  // Rust infers this is a Boolean  
}
```

However, sometimes it's just not possible for the compiler to know before the program is run what type a particular variable should have. In such cases, the compiler will display a `type must be known at this point` error, which is your cue to add a type annotation to that variable.

Constants: Values carved in stone

Variables are immutable by default, and if you're just storing an unchanging number in a single place in your program, `let` works perfectly fine. But Rust also has `const` (short for *constant*) which is like

a big, bold label that tells the reader of your code that this value is not meant to change:

```
const MAX_PLAYERS: u32 = 100;
```

Note the (`u32`) type annotation. When you use `const`, you must also supply a type. Also, Rust programmers always write `const` variable names in `ALL_CAPS` to really hammer home the “this value will never change” message.

Using `const` doesn’t make the program behave any differently than if you’d used `let`, but it does make the meaning clearer for humans reading the code. (There are also a few technical reasons why it’s better to use `const` over `let` for unchanging values, but there’s no point muddying the waters with all that here. You come across some examples as you work through the book.)

Talking about type safety

Rust is what programming nerds call a *type-safe* language, which is just a fancy way of saying that the Rust compiler is very strict about mixing different types, especially when doing so will cause problems. Imagine a kitchen where the salt and sugar are in unlabeled containers and the chef just grabs a spoonful of whatever’s closest to hand, no matter what the recipe. Disaster awaits!

Rust treats data types like they’re salt and sugar: It keeps them separate and allows you to combine them only under very strict conditions. If you try to add a string such as “hello” to the number 42, Rust stops you immediately instead of letting your program do something bizarre or unpredictable.

Why does this matter? In languages with looser rules, type mismatches often lead to subtle bugs, program crashes, or even security problems. In Rust, the compiler forces you to be clear about what type of data you’re working with, which means:

» Fewer crashes at runtime

- » Easier-to-read code, since you always know what you're dealing with
- » Peace of mind that your program isn't silently doing the wrong thing

So, when the Rust compiler is asking you yet again to fix a problem related to types, don't think of it as being picky. Think of it as having a helpful sous chef who won't let you reach for the salt when the recipe calls for sugar.

Working with the Standard Library

Every Rust installation comes with a collection of ready-to-use tools that you can use in your projects. This collection is called the *standard library* (or often just *std*, for short) and includes most of the core functionality of Rust. A special subset of the standard library is called the *prelude* and includes a carefully curated set of Rust tools that are imported into every program. The `println!` macro is an example of a prelude item. (You can peruse the full contents of the prelude here: <https://doc.rust-lang.org/std/prelude/index.html>.)

The big deal about the prelude is that the Rust compiler knows where to find items such as `println!`, so no extra work is required from you. For everything else in the standard library, you have to tell the compiler where to find it.

The standard library is organized into separate sections called *modules* to keep things neat and tidy, and some modules contain their own modules. This modules-within-modules structure means that when you want to use something from the standard library, you need to specify its full module path.

This path-specifying process is like specifying a path on your computer's file system. Instead of saying something like "Navigate to the C: drive, open the Windows folder, open the System32 subfolder, and there you'll find Notepad.exe," you can just use the following:

```
C:\Windows\System32\Notepad.exe
```

The backslash (`\`) is called the *path separator*. The `::` operator in Rust works like the path separator in a file system because it, too, shows the path that the Rust compiler has to follow to locate a particular Rust resource in the standard library.

For example, suppose you're writing some code that requires the use of `pi` for a mathematical calculation, such as the area of a circle. You could use `let pi = 3.14159;`, but there's no need for that because Rust comes with a built-in `PI` constant.

Here's one way you can use `PI`:

```
fn main() {
    let radius: f32 = 5.0;
    let area: f32 = std::f32::consts::PI * radius.powf(2.0);
    println!("The area of a circle with radius {radius} is {area}.")
}
```

I tell the compiler where to find the `PI` constant by specifying `std::f32::consts::PI`, which you can read as follows: “Open the standard library (`std`), look in the 32-bit floating-point module (`f32`), look in the constants submodule (`consts`), and there you'll find the `pi` constant (`PI`).”

Specifying a standard library path once is no big deal, but if you find yourself using the same resource multiple times, typing the full path gets old in a hurry. You can save yourself time and effort by importing the resource into your project once, at the top of `main.rs`, using the `use` keyword. Here's an example ([bk01ch03/example17/](#)):

```
use std::f32::consts::PI;

fn main() {
    let radius: f32 = 5.0;
    let area: f32 = PI * radius.powf(2.0);
    let circumference: f32 = 2.0 * PI * radius;
    println!(
        "The area of a circle with radius {radius}
        is {area}."
    );
    println!(
        "The circumference of a circle with radius {radius}"
    );
}
```

```
        is {circumference}."
    )
}
```

I imported the `PI` constant at the top with `use std::f32::consts::PI;`. Now I can refer to `PI` on its own, without the path, in my `area` and `circumference` calculations.

Chapter 4

Crafting Expressions

IN THIS CHAPTER

- » Understanding what expressions are
 - » Figuring out numeric expressions
 - » Getting the hang of comparison expressions
 - » Learning about logical expressions
 - » Building expression blocks
-

Rust: The language that makes you take the safety off before shooting yourself in the foot.

—MATT BRUBECK

The Rust variables described in the preceding chapter can't do all that much by themselves. They don't become useful members of your code community until you give them something productive to do. For example, you can assign values to them, use them to assign values to other variables, use them in calculations, and so on.

This productive side of variables in particular, and Rust-based code in general, is brought to you by a Rust feature known as the expression. When coding in Rust, you use expressions constantly, so it's vital to understand what they are and to get comfortable with the types of expressions available to you. Every Rust coder is different, but I can say without fear of contradiction that every *good* Rust coder is fluent in expressions.

This chapter takes you through everything you need to know about expressions. You discover some expression basics and then you explore a number of techniques for building powerful expressions using numbers, Boolean values, and blocks.

Understanding Expression Structure

An *expression* is a collection of symbols, words, and numbers that performs a calculation and produces a result. In expression lingo, the inputs are called *operands*, and they're combined by using special symbols called *operators*:

- » **operand:** An input value for an expression. An operand is the raw data that the expression manipulates to produce its result. The operand could be a number, a string, a variable, or a function result.
- » **operator:** A symbol that represents a particular action performed on one or more operands. For example, the `*` operator represents multiplication, and the `+` operator represents addition.

Assuming that your code has already declared a variable named `item_total` that holds a total for several items as well as a non-zero variable named `item_count` that holds the number of items, here's an expression that calculates the average of the items and assigns the result to a variable named `item_average`:

```
let item_average = item_total / item_count;
```

The expression is everything to the right of the equals sign (`=`). Here, `item_total` and `item_count` are the operands, and the division sign (`/`) is the operator.

Building Numeric Expressions

A mathematical calculation is often called a *numeric expression*, and it combines numeric operands and arithmetic operators to produce a numeric result. This section discusses all the Rust arithmetic operators and shows you how best to use them to build useful and handy numeric expressions.

A quick look at the arithmetic operators

Rust's basic arithmetic operators are more or less the same as those found in your smartphone's calculator app or on the numeric keypad of your computer's keyboard. [Table 4-1](#) lists the basic arithmetic operators you can use in your code expressions.

TABLE 4-1 Basic Arithmetic Operators

<i>Operator</i>	<i>Name</i>	<i>Example (where $n = 12$)</i>	<i>Result</i>
+	Addition	$n + 4$	16
-	Subtraction	$n - 4$	8
-	Negation	$-n$	-12
*	Multiplication	$n * 4$	48
/	Division	$n / 4$	3
%	Remainder	$n \% 4$	0

Here are some examples (bk01ch04/example01/ in this book's example files):

```
fn main() {
    let x = 20;
    let y = 20.0;

    // Integer examples
    println!("{}", x + 10);      // 30
    println!("{}", x / 8);       // 2
    println!("{}", x - 5 * 2);   // 10
    println!("{}", (x - 5) * 2); // 30

    // Floating-point examples
    println!("{}", y * 5.0);      // 100
    println!("{}", y / 8.0);      // 2.5
    println!("{}", x as f32 * y); // 400
}
```

Some notes about these results:

- » When you divide two integers, Rust discards any fractional part of the result. In the second example $x / 8$ is $20 / 8$, and the result is 2

because Rust truncates (removes) the decimal part of the answer (2.5), leaving just the integer part.

- » You can use parentheses to control the order of operations. Normally, multiplication and division are carried out before addition and subtraction (as in the third example), but by using parentheses, you can tell Rust to perform an addition or subtraction first (as in the fourth example). To learn more, head down to the “[Understanding Operator Precedence](#)” section.
- » When you build floating-point expressions, literal values must use a decimal point, even when they’re whole numbers (for example, use 5.0 instead of just 5, as in the fifth example).
- » The result of floating-point division includes the fractional part (as in the sixth example).
- » To use an integer type with a floating-point type in an expression, you can tell Rust to treat the integer type temporarily as a floating-point type by following the integer with `as type`, where `type` is `f32` or `f64` (as in the seventh and final example).



WARNING The `as` keyword is Rust’s *conversion operator*. You mostly use it to temporarily convert one numeric type to another, usually by *promoting* (sometimes called *widening*) a value to a bigger type, such as `i16` to `i32`. You can also *demote* (or *narrow*) a value to a smaller type, such as going from `u16` to `u8`. But if the original value is larger than the narrower type, Rust will modify the temporary value by taking the remainder of the original when divided by the number of distinct values in the narrower type. For example:

```
let x: u16 = 260;
let y: u8 = x as u8; // y = 4 because 260 % 256 = 4
```

Similarly, if you convert a floating-point value to an integer type, Rust will lop off the decimals from the original value:

```
let x: f32 = 3.14159;  
let y: u8 = x as u8; // y = 3
```



TIP You can use `as` to convert a Boolean value to an integer, where `true` becomes 1 and `false` becomes 0. However, Rust doesn't allow you to convert an integer to a Boolean.

What's going on with the remainder operator (%)?

The remainder operator (%) divides one number by another and then returns the remainder as the result:

```
dividend % divisor
```

For example, the following code outputs the value 1 because 5 (the `my_divisor` value) divides into 16 (the `my_dividend` value) three times and leaves a remainder of 1:

```
let my_dividend = 16;  
let my_divisor = 5;  
let my_remainder = my_dividend % my_divisor;  
println!("{}", my_remainder); // Output: 1
```

On a more practical level, suppose that you're trying to come up with a color scheme and want to use two colors that are complements of each other. Complementary means that the two hues are on the opposite side of the color wheel, so one way to calculate the second color is by adding 180 to the first color's hue value. That approach works when the hue of the first color is between 0 and 179, which gives second color hue values between 180 and 359. However, an initial hue of 180 or higher produces values of 360 or higher. But hue values go from only 0 to 359, so you need to wrap around back to the beginning of the color wheel. You can handle this wraparound using a remainder expression like this:

```
let complementary_color = (original_color + 180) % 360;
```

For example, if your original color has a hue of 200, adding 180 gives 380. Using `% 360` returns the remainder when 380 is divided by 360,

which is 20. So the complementary hue wraps around to 20.

Using the arithmetic assignment operators

Your code will often update the value of a variable by adding to it the value of some other operand. Here's an example:

```
running_total = running_total + 1;
```

Coders are an efficiency-loving bunch, so the fact that the `running_total` variable appears twice in that statement is like chewing tin foil to your average programmer. The Rust brain trust hates that kind of thing, too, so they came up with *arithmetic assignment operators*, a combination of some of the arithmetic operators and the assignment operator, which is the humble equals sign (=) that assigns a value to a variable. [Table 4-2](#) lists these operators.

For example, the following two statements are equivalent:

```
running_total = running_total + 1;
running_total += 1;
```

Yep, the second statement does the same thing as the first one but with 15 fewer characters. Sweet!

TABLE 4-2 Arithmetic Assignment Operators

<i>Operator</i>	<i>Example</i>	<i>Equivalent</i>
<code>+=</code>	<code>x += y</code>	<code>x = x + y</code>
<code>-=</code>	<code>x -= y</code>	<code>x = x - y</code>
<code>*=</code>	<code>x *= y</code>	<code>x = x * y</code>
<code>/=</code>	<code>x /= y</code>	<code>x = x / y</code>
<code>%=</code>	<code>x %= y</code>	<code>x = x % y</code>

Floating-point constants

The Rust floating-point types also come with quite a few handy predefined constants that are commonly used in mathematical operations. [Table 4-3](#) lists the most useful of these constants

(<https://doc.rust-lang.org/stable/std/f64/consts/index.html#constants> has the full list).

TABLE 4-3 Some Floating-Point Constants

<i>Constant</i>	<i>What It Represents</i>	<i>Approximate Value</i>
E	Euler's constant	2.718281828459045
FRAC_1_SQRT1_2	Square root of 1/2	0.7071067811865476
LN_10	Natural logarithm of 10	2.302585092994046
LN_2	Natural logarithm of 2	0.6931471805599453
LOG10_E	Base 10 logarithm of E	0.4342944819032518
LOG2_E	Base 2 logarithm of E	1.4426950408889633
PHI	Golden ratio	1.6180339887498948
PI	Constant pi	3.141592653589793
SQRT2	Square root of 2	1.4142135623730951

To use any of these constants in your code, precede the constant with a reference to the `consts` module in Rust's standard library, which is where the constants are stored. (Refer to Book 1, [Chapter 3](#) to learn more about the standard library.) The module you specify must match the floating-point type you're using (`f32` or `f64`):

```
std::f32::consts::PI
std::f64::consts::SQRT2
```

Floating-point methods

The Rust floating-point types also come with a large set of *methods* (predefined code) that enable you to perform mathematical operations such as square roots, powers, rounding, and trigonometry. A few of the most useful methods are summarized in [Table 4-4](#).

TABLE 4-4 Some Floating-Point Methods

Method Syntax	What It Returns
<code>number.abs()</code>	The absolute value of <i>number</i> (that is, the number without any sign)
<code>number.cbrt()</code>	The cube root of <i>number</i>
<code>number.ceil()</code>	The smallest integer greater than or equal to <i>number</i> (<i>ceil</i> is short for <i>ceiling</i>)
<code>number.cos()</code>	The cosine of <i>number</i> ; returned values range from -1 to 1 radians
<code>number.exp()</code>	E raised to the power of <i>number</i>
<code>number.floor()</code>	The largest integer that is less than or equal to <i>number</i>
<code>number.ln()</code>	The natural logarithm (base E) of <i>number</i>
<code>number1.max(number2)</code>	The larger of <i>number1</i> and <i>number2</i>
<code>number1.min(number2)</code>	The smaller of <i>number1</i> and <i>number2</i>
<code>number1.powf(number2)</code>	<i>number1</i> raised to the power of <i>number2</i> , where both numbers are floating-point values
<code>number1.pow(number2)</code>	<i>number1</i> raised to the power of <i>number2</i> , where both numbers are integers
<code>number.round()</code>	The integer closest to <i>number</i> ; if <i>number</i> is halfway between two integers, rounds away from 0.0
<code>number.sin()</code>	The sine of <i>number</i> ; returned values range from -1 to 1 radians
<code>number.sqrt()</code>	The square root of <i>number</i> (which must be greater than or equal to 0)
<code>number.tan()</code>	The tangent of <i>number</i> , in radians
<code>number.trunc()</code>	The integer portion of <i>number</i>

For example, to calculate the area of a circle, you use the formula πr^2 , where π (pi) is the ratio of the circumference of a circle to its diameter and *r* is the radius of the circle. Here's a program that takes a radius value and returns the area of the circle (bk01ch04/example02/):

```
fn main() {
    let radius: f32 = 5.0;
    let area: f32 = std::f32::consts::PI * radius.powf(2.0);
}
```

```
println!("The area of a circle with radius {radius} is {area}.")
}
```

The code uses `PI` to represent `pi` and `radius.pow(2)` to raise the `radius` value to the power of 2.

Building Comparison Expressions

You use *comparison expressions* to compare the value of two or more numbers, strings, variables, function results, or method results. If the expression is true, the expression result is set to the Boolean value `true`; if the expression is false, the expression result is set to the Boolean value `false`. You'll use comparisons with alarming frequency in your code, so it's important to understand what they are.

[Table 4-5](#) summarizes Rust's comparison operators.

TABLE 4-5 Comparison Operators

Operator	Name	Example	Result
<code>==</code>	Equality	<code>10 == 4</code>	<code>false</code>
<code>!=</code>	Nonequality	<code>10 != 4</code>	<code>true</code>
<code>></code>	Greater than	<code>10 > 4</code>	<code>true</code>
<code><</code>	Less than	<code>10 < 4</code>	<code>false</code>
<code>>=</code>	Greater than or equal	<code>10 >= 4</code>	<code>true</code>
<code><=</code>	Less than or equal	<code>10 <= 4</code>	<code>false</code>

The following example puts a few comparison expressions through their paces (`bk01ch04/example03/`):

```
fn main() {
    let x = 10;
    let y = 15;
    let a = "alpha";
    let b = "beta";

    println!("{}", x == y); // false
    println!("{}", x != y); // true
}
```



```
println!("{}", x <= y); // true
println!("{}", a == b); // false
println!("{}", a < b); // true
println!("{}", a >= b); // false
}
```

Building Logical Expressions

You use *logical expressions* to combine or manipulate Boolean values, particularly comparison expressions. For example, if your code needs to test whether two different comparison expressions are both `true` before proceeding, you can do that with a logical expression.

[Table 4-6](#) lists Rust’s logical operators.

TABLE 4-6 Logical Operators

Operator	Name	General Syntax	Returned Value
<code>&&</code>	AND	<code>expr1 && expr2</code>	true if both <code>expr1</code> and <code>expr2</code> are true; false otherwise
<code> </code>	OR	<code>expr1 expr2</code>	true if one or both of <code>expr1</code> and <code>expr2</code> are true; false otherwise
<code>!</code>	NOT	<code>!expr</code>	true if <code>expr</code> is false; false if <code>expr</code> is true

Here’s some Rust code that gives a few logical expressions a whirl (bk01ch04/example04/):

```
fn main() {
    let x = 10;
    let y = 15;
    let z = 20;

    println!("{}", x < y && x < z); // true
    println!("{}", y > x || y > z); // true
    println!("{}", !(x < z)); // false
}
```

Understanding Operator Precedence

Your Rust code will often use expressions that are blissfully simple: just one or two operands and a single operator. Alas, *often* here doesn't mean *mostly*, because many expressions you use will have a number of values and operators. In these more complex expressions, the order in which the calculations are performed becomes crucial. For example, consider the expression $3+5*2$. If you calculate from left to right, the answer you get is 16 ($3+5$ equals 8, and $8*2$ equals 16). However, if you perform the multiplication first and then the addition, the result is 13 ($5*2$ equals 10, and $3+10$ equals 13). In other words, a single expression can produce multiple answers depending on the order in which you perform the calculations.

To control this ordering problem, Rust evaluates an expression according to a predefined order of precedence. This order of precedence lets Rust calculate an expression unambiguously by determining which part of the expression it calculates first, which part second, and so on.

The order of precedence

The order of precedence that Rust uses is determined by the various expression operators that I've covered so far in this chapter. [Table 4-7](#) summarizes the complete order of precedence used by Rust. (Since this table is complete, it means that it includes some Rust operators that you haven't encountered yet.)

TABLE 4-7 Rust Order of Precedence for Operators

<i>Precedence (High to Low)</i>	<i>Operator(s)</i>	<i>Order of Evaluation</i>	<i>Notes</i>
1	::	N/A	Path
2	.	N/A	Method call
3	.	Left to right	Field access
4	() ,	N/A	Function call, array index
5	?	N/A	Error propagation operator
6	-x, *x, !x, &x, &mut x	Right to left	Unary minus, dereference operator, logical NOT, borrow, mutable borrow

<i>Precedence (High to Low)</i>	<i>Operator(s)</i>	<i>Order of Evaluation</i>	<i>Notes</i>
7	as	Left to right	Type casting
8	*, /, %	Left to right	Multiplication, division, remainder
9	+, -	Left to right	Addition, subtraction
10	<<, >>	Left to right	Bit shifts
11	&	Left to right	Bitwise AND
12	^	Left to right	Bitwise XOR
13		Left to right	Bitwise OR
14	==, !=, <, <=, >, >=	Left to right	Comparison operators
15	&&	Left to right	Logical AND
16		Left to right	Logical OR
17	.., ..=	N/A	Range operators
18	=, +=, -=, *=, /=, %=, &=, ^=, `	Right to left	Assignment operators
19	return, break, closures	N/A	None

For example, [Table 4-7](#) tells you that Rust performs multiplication before addition. Therefore, the correct answer for the expression `3+5*2` (just discussed) is 13.

Note, as well, that some operators in [Table 4-7](#) have the same order of precedence (for example, multiplication and division). Having the same precedence means that the order in which Rust evaluates these operators doesn't matter. For example, consider the expression `5*10/2`. If you perform the multiplication first, the answer you get is 25 (`5*10` equals 50, and `50/2` equals 25). If you perform the division first, you also get an answer of 25 (`10/2` equals 5, and `5*5` equals 25).

However, Rust does have a predefined order for these kinds of expressions, which is what the Order of Evaluation column tells you. A

value of left to right means that operations with the same order of precedence are evaluated from left to right; an order of right to left means the operations are evaluated from right to left.

Controlling the order of precedence

Sometimes Rust's default order of precedence can lead to an incorrect expression result. For example, you calculate the total cost of a retail item by multiplying the retail price by the tax rate, and then adding that result to the retail price:

```
Total Price = Retail Price + Retail Price * Tax Rate
```

However, what if you want to reverse this calculation? That is, suppose you know the final price of an item and, given the tax rate, you want to know the original (that is, pre-tax) price. Applying a bit of algebra to the preceding equation, it turns out that you can calculate the original price by dividing the total price by 1 plus the tax rate. So, if the total price is \$11.00 and the tax rate is 10 percent, you divide 11 by 1.1 and get an answer of \$10.00.

Okay, now I'll convert this calculation to Rust code. A first pass at the new equation might look something like this:

```
let retail_price = total_price / 1 + tax_rate;
```

The following code implements this formula (bk01ch04/example05/):

```
let total_price = 11.00;
let tax_rate = 0.1;
let retail_price = total_price / 1.0 + tax_rate;
println!("The pre-tax price is ${retail_price:.2}.");
```

Output:

```
The pre-tax price is $11.10.
```

Whoops, the result is incorrect. What happened? Well, according to the rules of precedence, Rust performs division before addition, so the `total_price` value first is divided by 1 and then is added to the `tax_rate` value, which isn't the correct order.

To get the correct answer, you have to override the order of precedence so that the addition `1 + tax_rate` is performed first. You override precedence by surrounding that part of the expression with parentheses, as shown in the following code (bk01ch04/example06/):

```
let total_price = 11.00;
let tax_rate = 0.1;
let retail_price = total_price / (1.0 + tax_rate);
println!("The pre-tax price is ${retail_price:.2}.");
```

Output:

```
The pre-tax price is $10.00.
```

Using this revised code, you get the correct answer. That's because terms inside parentheses are always calculated first, and terms outside parentheses are calculated sequentially (according to the order of precedence).

To gain even more control over your expressions, you can place parentheses inside one another; this is called *nesting* parentheses, and Rust always evaluates the innermost set of parentheses first.



WARNING One of the most common mistakes when using parentheses in expressions is to forget to close a parenthetical term with a right parenthesis or to close it in the wrong place, which creates a different grouping than you intended. Most modern code editors will automatically add a right parenthesis as soon as you type a left one. If your editor doesn't do this, you need to make sure you've closed each parenthetical term. One method you can use is to count all the left parentheses and then count all the right parentheses. If these totals don't match, you know you've left out a parenthesis somewhere along the line.

Using parentheses to determine the order of calculations allows you to gain full control over Rust expressions. This way, you can make sure that the answer given by an expression is the one you want.

Corralling Code into Expression Blocks

In the expression examples in the past few sections, each expression appears to the right of the equals sign (=) and consists of just a few operands and operators. But you won't get very far into your Rust journey before you come across scenarios where you need a calculation that's a bit more involved.

Rust has a few features that can help you in these situations, including functions (the topic of Book 1, [Chapter 7](#)) and closures (one of the topics of Book 2, [Chapter 6](#)). But most of the time, the Rust tool of choice for next-level expressions is the expression block.

First, know that in Rust a *block* is one or more statements and expressions surrounded by braces ({ and }):

```
{  
    Statements and expressions go here  
}
```

You've already encountered Rust's default block, the one associated with the `main()` function:

```
fn main() {  
    println!("Hello, world!");  
}
```



REMEMBER Notice that the `println!` statement is indented by four spaces.

This is a standard programming convention that helps to make blocks more readable and easier to troubleshoot. You can use whatever number of spaces makes you feel good inside, but four spaces is the Rust community standard (and it what `rustfmt`, the official Rust formatting tool, uses by default).

An *expression block* is a block with one or more statements that ends by using an expression to return a value. The key here is that while all the

expression block's statements end with semicolons, the final expression does not. That semicolon-less line is your signal to the Rust compiler that you're returning the final value of the expression block. Here's an example (bk01ch04/example07):

```
fn main() {  
    let area = {  
        let width = 5;  
        let height = 4;  
        width * height  
    };  
    println!("The area is: {area}");  
}
```

This example's variable assignment begins innocently enough with `let area =`, but then instead of a simple expression, an expression block shows up. The three lines within the block consist of two statements that assign values to the `width` and `height` variables, followed by an expression (no semicolon!) that multiplies these variables (`width * height`). The result of that expression is what gets stored in the `area` variable.

The expression block is one of Rust's more powerful and most expressive tools, and you'll be using it a ton as you master Rust.

Chapter 5

Controlling the Flow

IN THIS CHAPTER

- » Making your code make decisions
 - » Repeating code on purpose
 - » Getting your first taste of iteration
-

Rust is one of those friends that take some time to get along with but that you'll finally want to engage with for a long-term relationship.

—SYLVAIN WALLEZ

A simple Rust program, such as any of the example programs in Book 1, [Chapters 3](#) and [4](#), runs as though it was following the advice that the King offers gravely to the White Rabbit in *Alice's Adventures in Wonderland*: “Begin at the beginning, and go on till you come to the end: Then stop.” That is, the `main()` function gets called and then the statements in that function are executed, one after the other, until there's nothing left to do but stop.

Programming nerds call this top-to-bottom, one-statement-after-the-other process *sequential flow*. But you can make your Rust programs more powerful, more flexible, and far more useful when you practice what those same nerds call *flow control*.

In this chapter, you explore Rust's main tools for controlling program flow, including getting your code to make decisions and learning how to loop over repetitive code.

Making Decisions with if

A smart program performs tests on its environment and then decides what to do next based on the results of each test. For example, suppose you've declared a variable that you later use as a divisor in an expression. You should test the variable before using it in the expression to make sure that the variable's value isn't 0.

The most basic test is the simple true/false decision (which could also be thought of as a yes/no or an on/off decision). In this case, your program looks at a certain condition, determines whether it's currently true or false, and acts accordingly. Comparison and logical expressions (covered in Book 1, [Chapter 4](#)) play a big part here because they always return a `true` or `false` result.

In Rust, simple true/false decisions are handled by the `if` statement. You can use either the *single-line* syntax:

```
if expression { statement; }
```

or the *block* syntax:

```
if expression {  
    statement1;  
    statement2;  
    ...  
}
```

In both cases, *expression* is a comparison or logical expression that returns `true` or `false`, and *statement(s)* represent the Rust statement or statements to run if *expression* returns `true`. If *expression* returns `false`, Rust skips over the statements.



TECHNICAL
STUFF

If you're coming to Rust from another programming language, you might be familiar with the concepts of *truthy* values, which are non-Boolean values that the language treats as being equivalent to `true` (such as 1), and *falsy* values, which are non-Boolean values that the language treats as being equivalent to `false` (such as 0).

Well, there'll be none of that here! Rust has no concept of truthy or falsy: A condition is either `true` or `false`, end of story.

Whether you use the single-line or block syntax depends on the statements you want to run if the *expression* returns a `true` result. If you have only one statement, you can use either syntax. If you have multiple statements, use the block syntax.

Consider the following example (check out `bk01ch05/example01/` in this book's example files):

```
fn main() {  
    let age = 19;  
    if age >= 18 {  
        println!("You can vote!");  
    }  
}
```

This program uses `if` to check whether the value of the `age` variable is greater than or equal to 18. If the conditional expression `age >= 18` returns `true`, the program flow diverts into the `if` block and runs the `println!` macro; if instead the `age >= 18` expression returns `false`, the program skips over the `if` block.



REMEMBER It might seem silly to specify a value for the `age` variable and then run a test on that value, as though you somehow forgot the value you just typed! However, sometimes for the sake of simplicity an example has to be silly in just this way. In a real program, the value of `age` would be determined at runtime by asking the user to input their age or by reading the age value from a file or database.

Branching with `if...else`

Using the `if` statement to make decisions adds a powerful new weapon to your Rust arsenal. However, the simple version of `if` suffers from an important limitation: A `false` result only bypasses one or more

statements; it doesn't execute any of its own. This is fine in many cases, but there will be times when you need to run one group of statements if the condition returns `true` and a different group if the result is `false`. To handle these scenarios, you need to use an `if...else` statement:

```
if expression {  
    statements-if-true  
} else {  
    statements-if-false  
}
```

The *expression* is a comparison or logical expression that returns `true` or `false`. *statements-if-true* represents the block of statements you want Rust to run if *expression* returns `true`, and *statements-if-false* represents the block of statements you want executed if *expression* returns `false`.

As an example, consider the following code (bk01ch05/example02/):

```
fn main() {  
    let age = 16;  
    if age >= 18 {  
        println!("You can vote now!");  
    } else {  
        println!("You can vote in {} year(s).", 18 - age);  
    }  
}
```

If the `age` value is greater than or equal to 18, the message `You can vote now!` appears; otherwise, the flow switches to the `else` branch and the message `You can vote in x year(s)` appears, where `x` is determined by the expression `18 - age`.



TIP `if...else` statements are much easier to read when you indent the statements in each block, as I did in my examples. This indentation lets you easily identify which block will run if there's a `true` result and which block will run if the result is `false`. An indent of four spaces does the job, but some programmers prefer two spaces.

Making multiple decisions

The `if...else` control structure makes only a single decision. The `if` part calculates a single logical result and performs one of two actions.

However, plenty of situations require multiple decisions before you can decide which action to take.

Stringing together multiple if statements

A third syntax for the `if...else` statement lets you string together as many logical tests as you need using a *multiblock statement* (so-called because it contains multiple `if/else` blocks):

```
if expression1 {  
    statements-if-expression1-true  
} else if expression2 {  
    statements-if-expression2-true  
}  
etc.  
else {  
    statements-if-false  
}
```

Rust first tests `expression1`. If `expression1` returns `true`, Rust runs the block represented by `statements-if-expression1-true` and skips over everything else. If `expression1` returns `false`, Rust tests `expression2`. If `expression2` returns `true`, Rust runs the block represented by `statements-if-expression2-true` and skips over everything else. Otherwise, if all the `if` tests return `false`, Rust runs the block represented by `statements-if-false`.



REMEMBER Rust checks these conditions from top to bottom and executes the first one that's `true` and skips the rest. Even if multiple conditions are `true`, only the first one gets executed.

The following code shows a program that strings together several `if` statements (bk01ch05/example03/):

```
fn main() {
    let greeting;
    let current_hour = 10;

    if current_hour < 12 {
        greeting = "Good morning!";
    } else if current_hour < 18 {
        greeting = "Good afternoon!";
    } else {
        greeting = "Good evening!";
    }

    println!("{}", greeting);
}
```

The first `if` checks whether `current_hour` is less than 12. If so, the string literal "Good morning!" is stored in the `greeting` variable; if not, the next `if` checks whether `current_hour` is less than 18 (that is, less than 6:00 p.m.). If so, `greeting` is assigned the string literal "Good afternoon!"; if not, `greeting` is assigned "Good evening!" instead.

Using the AND (&&) and OR (||) operators

Another solution to a multiple-decision problem is to combine multiple comparison expressions in a single `if` statement. As I discuss in Book 1, [Chapter 4](#), you can combine comparison expressions by using Rust's AND (&&) and OR (||) operators.

The following code shows an example `if` statement that combines two comparison expressions using the `&&` operator ([bk01ch05/example04/](#)):

```
fn main() {
    let tax_rate = 5.0;
    let total_price = 52.45;

    if tax_rate >= 0.0 && tax_rate < 1.0 {
        let retail_price = total_price / (1.0 + tax_rate);
        println!("The retail price is ${retail_price:.2}.");
    } else {
        println!("Hmm. Please enter a tax rate between 0 and 1.");
    }
}
```

The key here is the `if` statement:

```
if tax_rate >= 0.0 && tax_rate < 1.0 {
```

This tells Rust that only if the `tax_rate` value is both greater than or equal to 0 *and* less than 1 should the statements in the true block be executed. If either one is `false` (or if both are `false`), the program displays the message in the false block instead.

Using if as an expression

In Book 1, [Chapter 4](#), I talk a bit about expression blocks, which consist of multiple statements in a block and a final expression (a line that doesn't end in a semicolon) that returns the value of the expression block.

Interestingly, an `if` structure is a variation on the expression block theme. Specifically, each block in an `if` structure — the initial `if` block, each `else if` block, and the final `else` block — can be an expression block that returns a value. And, since the program will flow into only one of those blocks, that block's return value becomes the return value of the entire `if` structure.

That's right: `if` is an expression!

Here's an example (bk01ch05/example05):

```
fn main() {  
    let score = 87;  
    let grade = if score >= 90 {  
        'A'  
    } else if score >= 80 {  
        'B'  
    } else if score >= 70 {  
        'C'  
    } else if score >= 50 {  
        'D'  
    } else {  
        'F'  
    };  
  
    println!("Your grade: {grade}");  
}
```

This code declares `let grade =`, but instead of a literal value or a simple expression, the value is set to the result of the `if` structure. Note that none of the lines in each block end with a semicolon to ensure that Rust knows that these are expressions that return a value.



WARNING Each branch of an `if` expression must return the same type. You can't return, say, a `char` from one branch and an integer from another. If you try, you'll get an error.

Handling Repetitive Code with Loops

Every reasonably complex Rust program, and even many a relatively simple Rust program, contains repetitive code. Here's a snippet to show you what I mean (bk01ch05/example06/):

```
fn main() {  
    let mut num = 2;  
    let mut num_squared = num * num;  
    println!("{num} squared = {num_squared}");  
  
    num += 1;  
    num_squared = num * num;  
    println!("{num} squared = {num_squared}");  
  
    num += 1;  
    num_squared = num * num;  
    println!("{num} squared = {num_squared}");  
}
```

This code initializes the `num` variable with the value 2 and the `num_squared` variable with the expression `num * num`, and prints both values. The code then increments `num` and repeats the squaring and the printing. Then the whole thing repeats once again.

That is some *boring* code! It's also remarkably inefficient because so much is repeated. Imagine if you needed the squares of `num` up to `num`

equal to 100! Forget about it!

Fortunately, Rust offers a solution to repetitive code: the loop. A *loop* is a code block that repeats a certain number of times, until some condition is met or until the code has cycled through every item in some collection of items. The next few sections provide the looping details.

Looping with a basic loop

The simplest loop is the `loop` structure, which repeats a block of code until you tell it to stop by using the `break` statement. Here's an example (bk01ch05/example07/):

```
fn main() {  
    let mut num = 2;  
    loop {  
        let num_squared = num * num;  
        println!("{num} squared = {num_squared}");  
  
        num += 1;  
  
        if num > 10 {  
            break;  
        }  
    }  
}
```

After initializing the `num` variable, this code heads right into the `loop` structure, which does the following:

1. Calculate the square of `num` and store the result in `num_squared`.
2. Print the value of `num` and `num_squared`.
3. Increment `num`.
4. Use `if` to test whether the incremented value of `num` is now greater than 10.
5. Repeat Steps 1 through 4 until `num` is greater than 10, at which point the flow enters the `if` block and runs `break` to exit the loop.

Here's the output of this program:


```
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
6 squared = 36
7 squared = 49
8 squared = 64
9 squared = 81
10 squared = 100
```

Using loop as an expression

A loop is a code block (that is, a bunch of statements between `{` and `}`), so it might not surprise you to learn that you can turn it into an expression that returns a value. To turn a `loop` structure into an expression, follow the `break` keyword with the value you want to return, and be sure to leave out the semicolon to tell Rust that you're returning an expression result. Here's an example ([bk01ch05/example08/](#)):

```
fn main() {
    let mut num = 2;
    let result = loop {
        let num_squared = num * num;
        println!("{num} squared = {num_squared}");

        if num_squared > 50 {
            break num
        }
        num += 1;
    };
    println!("The first number with a square larger than 50 is {result}.");
}
```

This code sets the `result` variable equal to the `loop` expression. In the loop, when the `if` condition `num_squared > 50` returns `true`, the code flow goes into the `if` block, where `break num` is an expression that returns the current value of `num`, which is what gets stored in the `result` variable. Note, too, the semicolon added to the closing brace of the `loop` block.

Doing conditional repetition with while loops

An alternative to `loop` is `while`, which loops using the following syntax:

```
while expression {  
    statements  
}
```

Here, *expression* is a comparison or logical expression (that is, an expression that returns `true` or `false`) that, as long as it returns `true`, tells Rust to keep executing the *statements* in the block.

Essentially, Rust interprets a `while` loop as follows: “Okay, as long as *expression* remains `true`, I’ll keep running through the loop statements, but as soon as *expression* becomes `false`, I’m out of there.”

Here’s a closer look at how a `while` loop works:

1. Evaluate the *expression* in the `while` statement.
2. If *expression* is `true`, continue with Step 3; if *expression* is `false`, skip to Step 5.
3. Execute each of the statements in the block.
4. Return to Step 1.
5. Exit the loop (that is, execute the next statement that occurs after the `while` block).

Here’s an example (bk01ch05/example09/):

```
fn main() {  
    let mut num = 2;  
    while num <= 10 {  
        let num_squared = num * num;  
        println!("{num} squared = {num_squared}");  
  
        num += 1;  
    }  
}
```

Okay, so when should you use `while` versus `loop`? Here are some criteria that might help:

- » Use `while` when you want the condition checked before going into the loop for the first time. Note this means that if the condition

returns `false` the first time it's checked, your loop code will never run (which might be exactly what you want).

- » Use `loop` when you want the code to flow into the `loop` block at least once. The loop structure is also useful when you need more complex control over the condition that decides whether the loop should continue.



REMEMBER In case you're wondering, no you can't use a `while` loop (or a `for` loop, covered next) as an expression. Of all the Rust loop types, only a `loop` structure can be an expression.

Iterating with for loops

If `while` structures are the fussy aunts of basic loops (they work only if the conditions are right) and `loop` structures are the drunk uncles (you never know when they'll stop!), `for` loops are the no-nonsense cousins. A `for` loop is designed to repeat — programming nerds prefer the phrase *iterate over* — a specific collection of data. There are no conditions and no mid-loop `if` tests, which usually makes the `for` loop the best choice when you know exactly what you want to loop over.

The `for` loop uses the following syntax:

```
for element in collection {  
    statements  
}
```

Here, *collection* is the group of elements you want to iterate over, and *element* represents one of those elements.

Rust interprets a `for` loop as follows: “Okay, run through every element in *collection* one at a time, and each time through the loop store the current element in the *element* variable. When you come to the end of *collection*, bail out of the loop.”

Here's a closer look at how a `for` loop works:

1. Get the next element in the *collection*. (When first entering the loop, the “next” element is the first element in *collection*.) If there are no more elements in *collection*, skip to Step 5.
2. Assign that element to the *element* variable.
3. Execute each of the statements in the loop block, often by performing operations on *element*.
4. Return to Step 1.
5. Exit the loop (that is, execute the next statement that occurs after the `for` block).

Rust offers different collections that you can iterate over with `for`, including many of the data structures that I discuss in Book 2, [Chapter 4](#), such as arrays and vectors. However, one of the most common use cases of `for` loops is iterating over a range of numbers (bk01ch05/example10/):

```
fn main() {  
    // Loop from 1 to 5 (inclusive)  
    for number in 1..=5 {  
        println!("Number: {number}");  
    }  
    // Loop from 0 to 4 (exclusive of 5)  
    for number in 0..5 {  
        println!("Index: {number}");  
    }  
    // Count backwards  
    for number in (1..=5).rev() {  
        println!("Countdown: {number}");  
    }  
}
```

The `x..y` syntax creates a range of values that run from `x` up to but not including `y`. So, `0..5` includes 0, 1, 2, 3, and 4. The `x..=y` syntax creates a range of values that run from `x` up to and including `y`. So, `1..=5` includes 1, 2, 3, 4, and 5.



TIP

When your loop code doesn't need to use the *element* part of the syntax, you can use `_` instead to create an *anonymous loop*:

```
for _ in 0..5 {
    println!("Hello, loop world!");
}
```

Here's the `for` loop version of the code for displaying the squares of the numbers 2 through 10 (bk01ch05/example11/):

```
fn main() {
    for num in 2..=10 {
        let num_squared = num * num;
        println!("{num} squared = {num_squared}");
    }
}
```



REMEMBER

Although you'll generally allow a `for` loop to run through every element in its collection, you don't have to let the loop run its full course. If some condition comes up in the loop that requires the loop to end prematurely, you can insert a `break` statement to shut down the loop.

Bypassing loop statements using `continue`

The `continue` statement is similar to `break`, but instead of exiting a loop entirely, `continue` tells Rust to bypass the rest of the statements in the loop block and begin a new iteration of the loop. Here's an example (bk01ch05/example12/):

```
fn main() {
    for num in 2..=10 {
        // Skip odd numbers
        if num % 2 != 0 {
            continue;
        }
        let num_squared = num * num;
        println!("{num} squared = {num_squared}");
    }
}
```

```
}  
}
```

This code uses the remainder operator (%) to check for odd numbers using `num % 2 != 0`. (`num % 2` returns 0 if `num` is even.) If that expression returns `true`, the flow goes into the `if` block and runs the `continue` statement, which tells Rust to bypass the rest of the loop code and return to the top of the loop for the next iteration.

Nesting loops

The code inside a loop can be any Rust code, even another loop! That might sound like something you'd rarely need, but you'd be surprised. Certain Rust data structures have multiple dimensions and it's common to have to iterate one dimension nested inside another.

Here's an example of nested `for` loops ([bk01ch05/example13/](#)):

```
fn main() {  
    for i in 1..=10 {  
        for j in 1..=10 {  
            print!("{:3} ", i * j);  
        }  
        println(); // New line after each row  
    }  
}
```

This code creates a simple multiplication table by nesting one `1..=10` range inside another. Here's the output of this program:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100



WARNING You'll want to exercise a bit of caution when it comes to nested loops. A 10×10 nested loop (that is, 10 iterations on the outer loop and 10 iterations on the inner loop) has 100 total iterations, but a $1,000 \times 1,000$ nested loop has a million! This tells you that nested loops can get computationally expensive very quickly. If you have a loop inside a loop inside a loop, that's usually a sign that you should rethink your approach.

Labeling loops

What happens when you have one loop nested inside the other and a condition occurs in the inner loop that means you need to shut down *both* loops? If you include a `break` statement in the inner loop, all that does is bail out of the inner loop; the outer loop keeps iterating.

Whenever you're juggling multiple loops, a label enables you to tell Rust which loop to break (or continue). A *label* is a name, preceded by a single quote and followed by a colon, that you insert just before the loop. Here's an example (bk01ch05/example14):

```
fn main() {
    'outer: for i in 1..=3 {
        for j in 1..=3 {
            if i * j == 4 {
                println!("Found 4 at i={i}, j={j}.");
                // Break the outer loop, not just the inner one
                break 'outer;
            }
        }
    }
}
```

The outer (*i*) for loop is given the label `'outer:`. In the inner (*j*) for loop, the code bails out of both loops by running `break 'outer`.

Chapter 6

Pattern Matching: The Swiss Army Knife of Rust

IN THIS CHAPTER

- » Matching patterns
 - » Matching ranges
 - » Matching with conditions
 - » Understanding block scope
 - » Coding a text-based adventure game
-

Rust is like doing parkour while suspended on strings and wearing protective gear. Yes, it will sometimes look a little ridiculous, but you'll be able to do all sorts of cool moves without hurting yourself.

—ANDRE BOGUS

Controlling the flow of your Rust programs using conditionals (such as `if` and `if/else`) and loops (such as `loop` and `while`) is a powerful way to take control of how your program works. But when it comes to altering a program's flow, pattern matching is where Rust really starts to show off. It's like having a super-smart and super-fast switchboard operator that can route your code based on the shape and content of your data.

In this chapter, you learn the basics of pattern matching in Rust, which will stand you in good stead when you learn about Rust features such as enums — especially `Option` and `Result` (refer to Book 2, [Chapter 3](#)) — that rely on pattern matching.

Becoming a match Master

The `match` expression is Rust's pattern-matching powerhouse. Here's the general syntax:

```
match value {  
    pattern1 => {  
        pattern1_statements  
    },  
    pattern2 => {  
        pattern2_statements  
    },  
    etc.  
    _ => {  
        catch_all_statements  
    }  
}
```

Here, *value* is a number, character, string, Boolean, or one of the many types covered in [Book 2](#) (such as tuple, struct, reference, or enum, particularly `Option` or `Result`). *pattern1*, *pattern2*, and so on are *patterns*, which are rules for matching a value. Rust examines each pattern to determine whether one of them matches *value*. If *pattern1* matches *value*, the code represented by *pattern1_statements* is executed and Rust stops processing the rest of the `match` expression. Otherwise, if *pattern2* matches *value*, the code represented by *pattern2_statements* is executed and Rust stops processing the rest of the `match` expression.

It's important to understand that Rust insists that the patterns you specify must be exhaustive. You're not allowed to leave anything out. That might sound impossible when the number of possible patterns is technically infinite, but Rust makes it easy by offering the wildcard character (`_`), which matches anything. So, if your patterns aren't naturally exhaustive, the final pattern must be the wildcard (`_`) pattern, which is used to handle situations where none of the other patterns matches *value*. When this "everything else" pattern matches, Rust executes the code represented by *catch_all_statements*.

The combination of a pattern (such as *pattern1*) and an associated code block (such as *pattern1_statements*) is called an *arm* of the `match` expression. If an arm runs just a single statement, you can omit the braces (but you still need the comma). (Technically, when an arm's code block uses braces, the trailing comma is optional.)

Here's an example (bk01ch06/example01/):

```
fn main() {
    let number = 3;

    match number {
        1 => println!("One"),
        2 => println!("Two"),
        3 => println!("Three"),
        4 => println!("Four"),
        5 => println!("Five"),
        _ => println!("Something else"),
    }
}
```

Output:

```
Three
```

Using match as an expression

Handily, you can use `match` as an expression to return a value. Here's an example (bk01ch06/example02/):

```
fn main() {
    let month = "January";
    let year = 2026;
    let days_in_month = match month {
        "January" => 31,
        "February" => {
            if year % 4 == 0 {
                29
            }
            else {
                28
            }
        },
        "March" => 31,
        "April" => 30,
        "May" => 31,
    }
}
```

```

        "June" => 30,
        "July" => 31,
        "August" => 31,
        "September" => 30,
        "October" => 31,
        "November" => 30,
        "December" => 31,
        _ => 0
    };

    if days_in_month > 0 {
        println!("{month} has {days_in_month} days.")
    }
}

```

Output:

```
January has 31 days.
```

This code uses `match` to look for a month name and then return the number of days in that month, which is stored in the `days_in_month` variable.



WARNING Each arm of a `match` expression must return the same type. You can't return, say, a floating-point value from one arm and an integer from another. If you try, the Rust compiler will raise a `'match' arms have incompatible types` error.

Matching a range of values

Instead of matching individual values, you can take advantage of the range syntax (which I introduce in Book 1, [Chapter 5](#)) to match a range of values. Here's an example (bk01ch06/example03/):

```

fn main() {
    let age = 25;
    match age {
        0..=12 => println!("Child"),
        13..=19 => println!("Teenager"),
        20..=64 => println!("Adult"),
        65..=120 => println!("Senior"),
        _ => println!("That's not a realistic age!"),
    }
}

```

```
}  
}
```

Output:

```
Adult
```

Matching one value or another

When you need to match at least one out of two or more values, use the *pattern alternative* syntax, where each possible value is separated by the pipe symbol (`|`). Check out this example (bk01ch06/example04/):

```
fn main() {  
    let today = "Saturday";  
  
    match today {  
        "Saturday" | "Sunday" => {  
            println!("It's the weekend! Time to relax.");  
        }  
        "Monday" => {  
            println!("Back to work!");  
        }  
        "Tuesday" | "Wednesday" | "Thursday" => {  
            println!("You're doing great!");  
        }  
        "Friday" => {  
            println!("TGIF!");  
        }  
        _ => {  
            println!("You need a vacation!");  
        }  
    }  
}
```

Output:

```
It's the weekend! Time to relax.
```

Matching with a condition

A common pattern is “match anything that meets the following condition,” where you use an `if` statement to set up the condition. Here’s an example (bk01ch06/example05/):

```
fn main() {  
    let n = 15;
```

```
match n {
  _ if n % 15 == 0 => println!("FizzBuzz"),
  _ if n % 3 == 0  => println!("Fizz"),
  _ if n % 5 == 0  => println!("Buzz"),
  _               => println!("{n}"),
}
```

Output:

```
FizzBuzz
```

Each arm uses the “match anything” wildcard character (`_`), but the first three use `if` conditions as guards that define the matching pattern. For example, the arm expression `_ if n % 15 == 0` means “match every value of `n` where the remainder when `n` is divided by 15 is 0.”

An alternative conditional pattern is to introduce a variable binding in place of the wildcards:

```
match n {
  x if x % 15 == 0 => println!("FizzBuzz"),
  x if x % 3 == 0  => println!("Fizz"),
  x if x % 5 == 0  => println!("Buzz"),
  _               => println!("{n}"),
}
```

You read the first three arm conditionals as “For any value `x`, if `x` meets the following condition, the pattern matches.”

Introducing Block Scope

One of Rust’s foundational ideas is a concept called scope. This concept is foundational because without it you can’t hope to make sense of how Rust manages memory, how ownership and borrowing work (refer to Book 2, [Chapter 1](#)), why some variables just seem to disappear from your code, and what on Earth the compiler is talking about when it tells you that it “cannot find value ‘x’ in this scope.”

The *scope* of a variable defines where in the program the variable can be used and where it can’t be used. To put it another way, a variable’s scope

determines when the variable is “alive” (accessible in memory) and “dead” (no longer accessible in memory).

When a variable has *block scope* (it’s also called *lexical scope*), in particular, it means that the variable is accessible only to other statements in that block. When does a variable have block scope? When the variable is declared using `let` (or `const`) within the `{` and `}` braces that define a block.

For example, consider the following code (bk01ch06/example06/):

```
fn main() {
    let area = {           // The block starts here
        let width = 5;    // width has block scope
        let height = 4;   // height has block scope
        width * height    // This works
    };                    // The block ends here
    println!("{width} * {height} = {area}"); // Error!
}
```

The `area` variable is declared using an expression block. Within that block, the `width` and `height` variables are declared, so they have block scope. It’s fine to use these variables in the return expression `width * height`, but then the block ends, which means Rust releases the `width` and `height` variables. They no longer exist, so when the program tries to use them in the `println!` statement, the Rust compiler has its say:

```
error[E0425]: cannot find value 'width' in this scope
--> src/main.rs:8:16
|
8 |     println!("{width} * {height} = {area}"); // Error!
|               ^^^^^
|
help: binding 'width' is available in a different scope in the same function
--> src/main.rs:3:13
|
3 |         let width = 5;    // width has block scope
|         ^^^^^

error[E0425]: cannot find value 'height' in this scope
--> src/main.rs:8:26
|
8 |     println!("{width} * {height} = {area}"); // Error!
|                      ^^^^^^^
```

```
|
help: binding 'height' is available in a different scope in the same function
--> src/main.rs:4:13
|
4 |         let height = 4; // height has block scope
  |         ^^^^^^

For more information about this error, try 'rustc --explain E0425'.
error: could not compile 'example20' (bin "example20") due to 2 previous
errors
```

But block scope doesn't just apply to expression blocks. You create scope also when you set up a block with `if`, `else if`, and `else`; with `loop`, `while`, and `for`; and within each arm of a `match` structure.



MEMORY 101: WHEN A VARIABLE GOES OUT OF SCOPE

All this talk of memory can seem hopelessly abstract and otherworldly, but scope brings memory management down to the ground and into the real world. That's because, in a concrete and predictable way, scope is where memory management happens.

Think of it this way: Creating a variable is like checking into a hotel room: You're given a specific room (Rust allocates memory), you move your stuff in (Rust stores the variable's value), and you can use that room for the duration of your stay (the variable is in scope). When you check out (come to the end of the block), the room is immediately available for the next guest (Rust frees the memory). When you check out (come to the end of the block), the room is immediately available for the next guest (Rust frees the memory) because you don't have to clean up (Rust handles everything automatically).

For example, suppose you declare `let width = 5;` inside an expression block. The moment that block's closing brace is reached, Rust automatically calls a special cleanup function for `width`. The memory that was holding the value `5` gets immediately freed and returned to the system. No waiting around, no guessing when it might happen, no separate garbage collector process running in the background slowing down everything as it tries to figure out what's safe to clean up.

This automatic cleanup is what makes Rust's memory management so elegant. You don't have to remember to manually free memory (as you do in C or C++), and you don't have to wait for a garbage collector to eventually get around to cleaning things up (as happens in Java or Python). The scope rules tell you exactly when cleanup happens. It's as predictable as turning off the lights when you leave a room. Your programs use the memory they need and when they need it, not a moment longer.

Project: Escape the Dungeon (a Tiny Text Adventure)

To bring all this chapter's concepts together, I close with a project named *Escape the Dungeon*, which is a text-based adventure game. The user is given a description of a scene, and then provided with several choices for how they want to proceed, with the overall goal being to escape the dungeon they find themselves in at the start of the game.

To get started with the project, open a terminal window, navigate to whatever folder you use for your Rust projects, and enter the following commands:

```
% cargo new escape-the-dungeon
% cd escape-the-dungeon
```

Getting input from the user

This game requires the user to type responses at the keyboard; the game then reads and processes those responses. In Book 1, [Chapter 3](#), I mention that the job of the `println!` macro is to display text to standard output (`stdout`), which is the terminal window. The opposite process is to read text from standard input (`stdin`), which is usually the keyboard when running the program in a terminal window.

Reading from standard input is the job of the `stdin()` function, which is part of the standard library's `io` (input/output) module. So, you begin the project by importing the `io` module:

```
use std::io;
```

With that done, you can then use code similar to the following to prompt the user and then read the data they enter via the keyboard:

```
print!("What do you do? ");
let mut choice = String::new();
let _ = io::stdin().read_line(&mut choice);
```


The `print!` statement displays the prompt to the user. Then the code declares a mutable variable named `choice` to hold the input. That input comes in the form of a `String` data type, which you learn all about in Book 2, [Chapter 5](#). For now, just know that a `String` is a more flexible and more powerful version of the string literals I talk about in Book 1, [Chapter 3](#). Finally, the last line uses the `read_line()` function to get the data from standard input. That data is stored in the `choice` variable, so the declaration uses `let _`, where the use of Rust’s wildcard character (`_`) tells Rust to go ahead and process everything to the right of the assignment operator (`=`), but don’t create a variable for it.

Also, I should mention the weird ampersand (`&`) that appears before `mut` in the final line. That’s called a reference in Rust and it’s a major Rust concept that I describe in glorious detail in Book 2, [Chapter 1](#).

Note that this section gives you just the barest of barebones introductions to getting input from the user. You get the full treatment when you start building full-fledged command-line utilities in Book 4, [Chapter 1](#).

Setting up the game

To get started with the rest of the game, remove the default “Hello, world!” statement, replace it with the user’s game introduction, and set up your game state:

```
fn main() {
    println!("=== Escape the Dungeon ===");
    println!("You wake up in a torchlit corridor with three paths.");
    println!("Survive long enough to find the exit!");
    println!();

    // Simple game state
    let mut health: i32 = 3;
    let mut treasure: i32 = 0;
    let mut escaped: bool = false;
```

The game continues as long as the `health` value is greater than 0 (the value gets decremented if the user makes some incorrect choices along

the way) and as long as the `escaped` Boolean is `false`. With those conditions in mind, you can set up your main game loop:

```
while health > 0 && !escaped {  
    // Game logic goes here  
}
```

Until I say otherwise, all the code from here on goes inside the main game loop.

Now display the user's current status, the input choices, and the prompt:

```
println!("Health: {health}    Treasure: {treasure}");  
println!("You see three choices: left (l), right (r), or wait (w).");  
print!("What do you do? ");
```

Now you're ready to read the user's input:

```
let mut choice = String::new();  
let _ = io::stdin().read_line(&mut choice);
```

Introducing variable shadowing

With the user's input stored in the `choice` variable, the code needs to clean and normalize that input in two ways:

- » Trim the input to remove any leading or trailing whitespace, which includes spaces, tabs and, especially, newline characters, because by default the `read_line()` function includes the newline character that becomes part of the input when the user presses Enter or Return. This cleaning part of the process is handled by the `str` type's `trim()` method.
- » Convert the input to all-lowercase letters, which makes our later `match` pattern easier to implement because we no longer have to include uppercase letters as a possible input. This normalization part of the process is handled by the `str` type's `lowercase()` method.

Here's one way you could do this:

```
let choice_trimmed = choice.trim();  
let choice_lower = choice_trimmed.to_lowercase();
```

You would then use `choice_lower` throughout the rest of the code. That'll work, for sure, but it's a very un-Rust pattern because you're creating extra variables that don't do much.

The true Rust way of approaching this type of situation (which, by the way, is common) is to keep using the `choice` variable throughout and use `let` to declare a new variable with the same name each time:

```
let choice = choice.trim();
let choice = choice.to_lowercase();
```

I know: that just *looks* weird. But this is a legit and common Rust pattern that's called *variable shadowing*. Each time you use `let` to declare a new variable with the same name, Rust makes the original unavailable and, in a sense, overshadows it with the new variable of the same name.

Pattern matching the user input

Now the game logic must process the cleaned and normalized user input, and that means setting up a `match` structure:

```
match choice.as_str() {
    // Match arms go here
}
```

(To learn what the `as_str()` method is doing here, refer to Book 2, [Chapter 5](#).) The three `match` arms are `"left" | "l"`, `"right" | "r"`, and `"wait" | "w"`. I won't go through each arm's code, though. Here's the code for the `"left" | "l"` arm:

```
"left" | "l" => {
    println!("You creep left into a narrow passage...");
    // Small scene loop that resolves quickly
    loop {
        println!("A riddle is carved on the wall:
                \"Two doors: one lies, one tells truth.\");
        println!("Pick door 1 (1) or door 2 (2).");

        // Get the user's input
        let mut door = String::new();
        let _ = io::stdin().read_line(&mut door);
        let door = door.trim(); // More variable shadowing!

        if door == "1" {
```

```

println!("Clank! A trap nicks you as you slip through. Ouch!");
health -= 1;
println!("But hey, a coin purse! (+1 treasure)");
treasure += 1;
break;
} else if door == "2" {
    println!("The door opens smoothly. A friendly rat hands you a
coin. (+1 treasure)");
    treasure += 1;
    break;
} else {
    println!("That's not a valid door. Try again.");
    continue;
}
}
}

```

This arm sets up an inner game loop that presents the user with another scene and another prompt, which is read, cleaned, and then handled by an `if` structure. The other arms create similar inner game loops:

```

"right" | "r" => {
    println!("You stride right into a damp chamber. Something glitters under
murky water.");

    // A quick gamble: Take treasure with a health risk or play it safe.
    println!("Do you reach in for the glitter (g) or play it safe (s)?");

    let mut action = String::new();
    let _ = io::stdin().read_line(&mut action);
    let action = action.trim().to_lowercase();

    if action == "g" || action == "glitter" {
        println!("You reach in ... Something bites! (-1 health) But you grab a
gem! (+2 treasure)");
        health -= 1;
        treasure += 2;
    } else if action == "s" || action == "safe" {
        println!("You keep your hands clean. Wise ... but you find a single
coin near the edge. (+1 treasure)");
        treasure += 1;
    } else {
        println!("You hesitate too long; the water settles. Nothing
gained.");
    }
}
}

```

```

"wait" | "w" => {
    println!("You wait. Footsteps echo, then fade. A breeze hints at fresh
air ahead.");
    println!("You follow the breeze and find a heavy door with a number wheel
(0-9).");
    println!("The inscription reads: \"The code is the first odd number
greater than 6.\");");
    println!("Enter a single digit (0-9):");

    let mut digit = String::new();
    let _ = io::stdin().read_line(&mut digit);
    let digit = digit.trim();

    match digit {
        "7" => {
            println!("Click! The door swings open. Sunlight pours in—you're
free!");
            escaped = true;
        }
        "8" | "9" | "0" | "1" | "2" | "3" | "4" | "5" | "6" => {
            println!("Wrong code. The door shocks you lightly. (-1 health)");
            health -= 1;
        }
        _ => {
            println!("That's not a single digit. The door hums ominously. (-1
health)");
            health -= 1;
        }
    }
}

_ => {
    println!("You mumble indecisively and wander in circles. (-1 health)");
    health -= 1;
}

```

Finishing up

Once the main game loop is complete, the code closes by displaying the results:

```

if escaped {
    println!("You escaped with {treasure} treasure. Nice work!");
} else {
    println!("Your adventure ends here. Final treasure: {treasure}");
    println!("Tip: Try \"wait\" next time—fresh air is a clue.");
}

```

Chapter 7

Functions: Teaching Rust New Tricks

IN THIS CHAPTER

- » Getting to know Rust functions
 - » Creating and using custom functions
 - » Passing and returning function values
 - » Getting the scoop on function scope
 - » Building a temperature converter
-

It's enjoyable to write Rust, which is maybe kind of weird to say, but it's just the language is fantastic. It's fun. You feel like a magician, and that never happens in other languages.

—PARKER TIMMERMAN

Rust's standard library is bristling with built-in features that perform specific tasks. In Book 1, [Chapter 6](#), for example, I talk about using `read_line()` to get user input from the keyboard and `trim()` to remove any spaces and newlines attached to that input. Hundreds of these ready-to-roll features perform tasks that range from the indispensable to the obscure. But Rust can't possibly do everything that you'd like or need it to do. What happens if your project requires a particular task or calculation that isn't part of the Rust language or a third-party crate (external library)? Are you stuck? Not even close! The solution is to roll up your sleeves and then roll your own code that accomplishes the task or runs the calculation.

This chapter shows you how to create such do-it-yourself code. In the pages that follow, you explore the powerful and infinitely useful realm

of functions, where you craft reusable code that performs tasks that out-of-the-box Rust can't do.

What Is a Function?

A *function* (to get right down to it) is a group of Rust statements that are separate from the rest of the program and perform a designated task. Technically, a function can perform any number of chores, but as a rule it's best to have each function focus on a specific task. When your program needs to perform that task, you tell it to run — or *execute*, in the vernacular — the function.

You already have, in fact, extensive experience with Rust functions in the form of the `main()` function that populates every new Rust project. However, in this chapter you deal with functions that reside outside of `main()`.

Why Functions Matter

Before diving into the function particulars, it's probably a good idea to get a sense of why functions are a big deal. An example will help. Suppose you're writing a program that needs to calculate the area of five different circles. One way to approach this would be to copy and paste the same formula each time:

```
let radius1 = 5.0;
let area1 = 3.14159 * radius1 * radius1;

let radius2 = 10.1;
let area2 = 3.14159 * radius2 * radius2;

let radius3 = 7.5;
let area3 = 3.14159 * radius3 * radius3;

let radius4 = 3.6;
let area4 = 3.14159 * radius4 * radius4;

let radius5 = 12.0;
let area5 = 3.14159 * radius5 * radius5;
```

Man, it *hurt* a little to write that code because this approach has a long list of problems:

- » **It's not scalable.** Repeating the formula five times won't kill you but writing it a *hundred* and five times just might.
- » **Repetition is the enemy of maintenance.** If you later realize you need a more precise value of pi or that you want to use the standard library's `PI` constant, you have to hunt down and change that `3.14159` everywhere it's used. Miss one, and your calculations will be inconsistent.
- » **Bugs can multiply rabbit-like.** Each copy of the code is another opportunity for a typo or logic error. And if the original formula contains an error, you just propagated the error into every copy!
- » **Your code becomes a novel instead of the short story it's meant to be.** Rather than being concise and readable, your code becomes a wall of repetitive text that's hard to read, harder to understand, and even harder to troubleshoot.

Functions make short work of all these problems by giving you a way to write the logic once and reuse it as many times as you need. A function is like creating a recipe that you can follow whenever you need to make the same dish, rather than writing out all the steps from scratch each time.

If all functions were good for is avoiding repetition, they'd still be worth the price of admission. But functions are much more powerful than that. Functions also help you

- » Break down complex problems into smaller, more manageable pieces.
- » Test your code more easily by isolating specific functionality.
- » Make your code more readable by giving descriptive names to operations.
- » Share functionality between different parts of your program.

Even tiny functions that do seemingly simple things can make your code dramatically better. A function that calculates the area of any circle might not seem like a big deal, but if circle area calculations are important to your project, giving that task its own function makes your code easier to read and understand.

Writing Your First Function

The basic structure of a function looks like this:

```
fn function_name()><i>arguments</i><![CDATA) {  
    Rust statements  
}
```

where:

- » `fn` identifies the block of code that follows it as a function.
- » `function_name` is a unique name for the function. The naming rules and guidelines that I outline for variables in Book 1, [Chapter 3](#) also apply to function names:
 - Can contain only letters, numbers, and underscores.
 - Can't start with a number.
 - Should use `snake_case` for multiword names, where each word is separated by an underscore (`_`).
 - Can't be any of Rust's *reserved words*, which are keywords used by the language itself, such as `let` and `fn`. For the full list, head over to <https://doc.rust-lang.org/stable/std/index.html#keywords>.
- » `arguments` are one or more optional values that are passed to the function and act as variables within the function. Arguments (or *parameters*, as they're sometimes called) are typically one or more values that the function uses as the raw materials for its tasks or calculations. You always enter arguments between parentheses after the function name, and you separate multiple arguments with

commas. If you don't use arguments, you must still include the parentheses after the function name.

» *Rust statements* are the code that performs the function's tasks or calculations.

Note, too, the use of braces (`{` and `}`). These are used to enclose the function's statements within a block, which tells you (and the compiler) where the function's code begins and ends.



TIP Note how the *Rust statements* line in the example is indented slightly from the left margin. This is a standard and highly recommended programming practice because it makes your code easier to read. This example is indented four spaces, which is enough to do the job but isn't excessive. Some programmers use two spaces.

Here's an example of a function with no arguments:

```
fn say_hello() {  
    println!("Hello, world!");  
}
```

To run the function, you call it from the `main()` function by typing the function name, followed by parentheses (refer to `bk01ch07/example01/` in this book's example files):

```
fn main() {  
    say_hello(); // Call the say_hello() function  
}
```

Output:

```
Hello, world!
```



REMEMBER Rust doesn't care if your custom functions reside before or after the `main()` function. Put them wherever you like in `main.rs`

because the compiler will find them wherever they are.

Arguments: How to Pass 'Em, How to Use 'Em

Functions become truly useful members of Rust society when they take in one or more arguments and then use those values in the function code.

Passing a single argument

Here's an example of a function that takes a single argument (bk01ch07/example02/):

```
use std::f32::consts::PI;

fn area_of_circle(radius: f32) {
    let area = PI * radius * radius;
    println!("The area of a circle with radius {radius:.1} is {area:.1}.");
}
```

Note that within the function's parentheses, you type the argument name (radius, in the example), which must be accompanied by the argument's type (f32, in this case). Then, inside the function, you're free to use the argument name just like any other variable:

```
let area = PI * radius * radius;
```

With your function all set up, you can call it as many times as you like:

```
fn main() {
    area_of_circle(5.0);
    area_of_circle(10.1);
    area_of_circle(7.5);
    area_of_circle(3.6);
    area_of_circle(12.0);
}
```

Here's the output:

```
The area of a circle with radius 5.0 is 78.5.
The area of a circle with radius 10.1 is 320.5.
The area of a circle with radius 7.5 is 176.7.
The area of a circle with radius 3.6 is 40.7.
The area of a circle with radius 12.0 is 452.4.
```

Passing multiple arguments

When your function requires multiple inputs, specify each argument's name and type, separating the argument definitions with commas.



REMEMBER Rust uses *positional parameters*, meaning the order of the arguments matters. When you call a function, you must pass values in the same order as they're defined in the function signature. For example, if you swap `person1` and `person2` in the following call, you'll get different output!

Here's an example (bk01ch07/example03/):

```
fn introduce_people(person1: &str, person2: &str) {
    println!("{person1}, meet {person2}. {person2}, meet {person1}.");
}

fn main() {
    introduce_people("Alice", "Bob");
}
```

Output:

```
Alice, meet Bob. Bob, meet Alice.
```

What's up with the `&str` type for both arguments? It's what's known in the Rust world as a string slice, which is a read-only view of a string. For now, think of it as a string I can read but not change. You learn about the deeper meaning of `&` later when I cover ownership and references (Book 2, [Chapter 1](#)), and you get the details of string slices when I cover strings (Book 2, [Chapter 5](#)). Until then, feel free to use `&str` for read-only text arguments.



WARNING If you define a function to have multiple arguments, you must always pass values for each of those arguments to the function. If you don't, the compiler will tell you about the missing argument and refuse to compile your code until you fix the problem.



TIP You can make your code more readable if, when you call a function, you precede each passed value with the corresponding argument name, followed by a colon (:). Here are a couple of examples:

```
area_of_circle(radius: 5.0);  
introduce_people(person1: "Alice", person2: "Bob");
```

Making an argument mutable

Just like regular variables, function arguments are immutable. Suppose you try something like this (bk01ch07/example04/):

```
fn circumference_of_circle(radius: f32) {  
    radius = radius * 2.0;    // Good luck with this!  
    let circumference = PI * radius;  
    println!("The circumference of a circle with diameter {radius:.1} is  
[CA]{circumference:.1}.");  
}
```

The code attempts to multiply the `radius` argument by 2. Here's what you get for your trouble:

```
error[E0384]: cannot assign to immutable argument 'radius'  
--> src/main.rs:4:5  
  |  
4 |     radius = radius * 2.0;  
  |     ^^^^^^^^^^^^^^^^^^^^^ cannot assign to immutable argument  
  |  
help: consider making this binding mutable  
  |  
3 | fn circumference_of_circle(mut radius: f32) {  
  |                               +++
```

Following the compiler's sage advice (consider making this binding mutable), you can make the argument mutable by adding the `mut` keyword (bk01ch07/example05/):

```
fn circumference_of_circle(mut radius: f32) {  
    radius = radius * 2.0;    // This is fine now  
    let circumference = PI * radius;  
    println!("The circumference of a circle with diameter {radius:.1} is  
[CA]{circumference:.1}.");  
}
```



REMEMBER When you call a function using a variable as an argument, note that using `mut` creates a new, mutable local variable. So, when you make changes to that variable inside the function, you're changing just the local copy, not the caller's variable:

```
fn main() {  
    let radius = 5.0;    // radius value is 5.0  
    circumference_of_circle(radius); // Pass radius to the function as mut  
    println!("{radius}");    // radius is still 5.0 here  
}
```

Return Values: Bringing Data Back Alive

One of the major benefits that functions bring to the Rust table is that you can use them to perform calculations and then return the result.

In Rust, just as you need to specify a type for each argument, you also need to specify a type for the return value. You tell the compiler a function's return type with a so-called *thin arrow* (`->`) — that's a hyphen (`-`) followed by a greater-than sign (`>`) — after the parameter list but before the opening brace:

```
fn function_name(<i>arguments</i><![CDATA) -> type {  
    Rust statements  
}
```

where *type* is the type of the return value. The combination of the function name, the function arguments, and the function return type is called the *function signature*.

Here's an example:

```
fn add_numbers(x: i16, y: i16) -> i16 {
    x + y
}

fn main() {
    let result = add_numbers(17, 25);
    println!("The sum is {result}."); // Output: The sum is 42.
}
```

In the `add_numbers()` function, note that the line `x + y` has no ending semicolon. In Rust, the value of the last expression in a function is automatically returned if it doesn't have a semicolon. This is called an *expression-based* return. In this example, the return value gets stored in the `result` variable.

You can also use the `return` keyword explicitly:

```
fn add_numbers(x: i16, y: i16) -> i16 {
    return x + y;
}
```

That's fine, but the expression-based return is the preferred style in Rust. The `return` statement does have its place, however. You use it whenever your code needs to return early from a function ([bk01ch07/example07/](#)):

```
fn divide_if_positive(x: i32, y: i32) -> i32 {
    if y == 0 {
        return 0; // Early return to avoid division by zero
    }
    x / y
}

fn main() {
    let result = divide_if_positive(100, 0);
    println!("The result is {result}.") // Output: The result is 0.
}
```

The unit type return

What if your function doesn't need to return a value? In Rust, such functions have a return type of `()`, which is known as the *unit type*. (Technically, it's an empty tuple. *A what now?* A tuple, which I introduce you to in Book 2, [Chapter 4](#).) However, you usually don't need to write the unit type explicitly:

```
// No return value here
fn print_greeting(name: &str) {
    println!("Hello, {name}!");
}

// This is equivalent to:
fn print_greeting_explicit(name: &str) -> () {
    println!("Hello, {name}!");
}
```

A practical return example

Here's a practical use case for return values (bk01ch07/example08/):

```
fn calculate_tip(bill: f64, percentage: f64) -> f64 {
    bill * (percentage / 100.0)
}

fn format_currency(amount: f64) -> String {
    format!("{:.2}", amount)
}

fn main() {
    let bill_amount = 50.0;
    let tip = calculate_tip(bill_amount, 18.0);
    let total = bill_amount + tip;
    println!("Bill: {}", format_currency(bill_amount));
    println!("Tip: {}", format_currency(tip));
    println!("Total: {}", format_currency(total));
}
```

This example uses two functions: `calculate_tip()` returns the tip amount given a bill amount and a tip percentage; and `format_currency()` returns a `String` (refer to Book 2, [Chapter 5](#)) that formats a floating-point value as a currency string with a dollar sign (\$) and two decimal places. (To learn more about the `format!` macro, head over to Book 3, [Chapter 5](#).)

Digging Deeper into Variable Scope

In Book 1, [Chapter 6](#), I discuss variable scope, particularly block scope, where a variable declared inside a block (between { and }) can be accessed only by statements in that block and is dropped from memory as soon as the block ends.

Rust has lots of different scope types besides block scope, so this is as good a time as any to let you in on two more:

- » Function scope
- » Global scope

The next two sections describe each type in detail.

Function scope

When a variable has *function scope* (often also known as *local scope*), the variable was either declared inside a function or passed as a function argument, and the only statements that can access the variable are the ones in that same function. Statements outside the function and statements in other functions can't access the variable. The local variable goes out of scope (and is dropped from memory) as soon as the function ends.

The following code demonstrates function scope (bk01ch07/example09/):

```
fn demonstrate_scope() {  
    let local_variable = "I only exist inside this function";  
    println!("{}", local_variable);  
}  
  
fn main() {  
    demonstrate_scope();  
    println!("{}", local_variable); // Not going to happen!  
}
```

Here's what the compiler thinks of this:

```
% cargo run
error[E0425]: cannot find value 'local_variable' in this scope
--> src/main.rs:8:20
|
8 |     println!("{}", local_variable); // Not going to happen!
|                               ^^^^^^^^^^^^^^^^^ not found in this scope

For more information about this error, try 'rustc --explain E0425'.
error: could not compile 'example09' (bin "example09") due to 1 previous
error
```

The variable named `local_variable` is declared inside the function, so it has function scope. As soon as the function ends, that variable disappears off the face of the Earth, so when `main()` tries to use `local_variable`, it can't find it and the compiler tosses a `not found in this scope` error your way.

A big advantage to function scope is that you're free to use the same variable name in multiple functions (bk01ch07/example10/):

```
fn function_one() {
    let name = "Alice";
    println!("In function_one: {name}");
}

fn function_two() {
    let name = "Bob";
    println!("In function_two: {name}");
}

fn main() {
    let name = "Charlie";
    println!("In main: {name}");
    function_one();
    function_two();
    println!("Still in main: {name}");
}
```

Output:

```
In main: Charlie
In function_one: Alice
In function_two: Bob
Still in main: Charlie
```

Each instance of the `name` variable is independent of the others, which saves you the bother of coming up with unique variable names for the same thing. Nice!

Global scope

What if you want to use the same variable in multiple functions in the same program? In that case, you need to use *global scope*, which makes a variable accessible to any function in your program.

To set up a variable with global scope, the Rust-approved way is to use `const` and declare the variable outside of any function. The following code gives this a whirl (bk01ch07/example11/):

```
const APP_NAME: &str = "The Global Game";
const MAX_PLAYERS: u16 = 100;

fn initialize() {
    // MAX_PLAYERS has global scope, so it can be used here
    println!("The maximum number of players is {MAX_PLAYERS}.");
}

fn main() {
    // APP_NAME has global scope, so it can be used here    println!("Welcome
to {APP_NAME}!");
    initialize();
}
```

Output:

```
Welcome to The Global Game!
The maximum number of players is 100.
```

Both the `APP_NAME` and `MAX_PLAYERS` variables are declared with `const` outside of any function, so they have global scope. To prove it, the `MAX_PLAYERS` variable is used successfully in the `initialize()` function and the `APP_NAME` variable is used without mishap in the `main()` function.



TECHNICAL
STUFF

Using a constant as a global variable is the way to go when you know the value of the variable in advance. However, what if the variable value isn't known until the program is running, such as program configuration data loaded at runtime from a file? In that case, you can replace `const` with `static`. This technique is a bit advanced for us here, but you'll see plenty of examples that take advantage of `static` later in the book.



REMEMBER Note, however, that the universal availability of a variable doesn't mean you should go crazy and litter the top of your code with a thousand `const` declarations. Rust generally discourages global variables because they make troubleshooting a bit harder. Rust prefers that you pass values around as function arguments.

Functions That Call Other Functions

In all the function-calling examples I've presented so far in this chapter, the statement that calls the function has resided in `main()`. But that doesn't mean that you can call a function only from `main()`. You can call any of your custom functions from any other custom function, which can lead to code that's more understandable, more flexible, easier to test, and easier to debug. Somewhat mind-bendingly, you can even have a function call itself!

Separating responsibilities

I mention earlier that, in an ideal Rust world, each function would be responsible for accomplishing a single task. Keeping every function focused in this way brings so many benefits:

- » Testing (covered in Book 3, [Chapter 7](#)) becomes so much easier because now you have to test only that each function is doing the task it was designed to do.
- » Your code becomes self-documenting because, as long as you use descriptive function names, it's easy to get a high-level read on what your program does. In a sense, the functions act like a kind of table of contents for your program.
- » Your code becomes easier to maintain and troubleshoot because it mostly consists of a bunch of small, tightly focused functions.

So, yes, small functions are great, but small functions that work together are awesome. Check out an example (bk01ch07/example12/):

```
fn calculate_tax(bill: f64, tax_rate: f64) -> f64 {
    bill * (tax_rate / 100.0)
}

fn calculate_tip(bill: f64, percentage: f64) -> f64 {
    bill * (percentage / 100.0)
}

fn calculate_total(bill: f64, tax_rate: f64, percentage: f64) -> f64 {
    let tax = calculate_tax(bill, tax_rate);
    let tip = calculate_tip(bill, percentage);
    bill + tax + tip
}

fn format_currency(amount: f64) -> String {
    format!("{:.2}", amount)
}

fn main() {
    let bill_amount = 50.0;
    let tax_rate = 8.5;
    let tip_percentage = 18.0;
    let total = calculate_total(bill_amount, tax_rate, tip_percentage);
    println!("Bill amount: {}", format_currency(bill_amount));
    println!("Total with tax and tip: {}", format_currency(total));
}
```

Here's the output:

```
Bill amount: $50.00
Total with tax and tip: $63.25
```

Note that each function has a single responsibility: `calculate_tax()` deals with only tax and `calculate_tip()` deals with only tips. Also, the functions are reusable. For example, `main()` uses `format_currency()` twice. It would be easy to imagine using the `calculate_tax()` function in other parts of your program.

Using helper functions

Sometimes even a big program needs a little help. A *helper function* is a small, reusable function that accomplishes a task designed to support or supplement the program's main logic. In the preceding section's code example, the `format_currency()` function is an example of a helper function. Here are a few more examples (bk01ch07/example12/):

```
fn is_leap_year(year: i32) -> bool {
    (year % 4 == 0 && year % 100 != 0) || year % 400 == 0
}

fn days_in_month(month: i32, year: i32) -> i32 {
    match month {
        1 | 3 | 5 | 7 | 8 | 10 | 12 => 31,
        4 | 6 | 9 | 11 => 30,
        2 => if is_leap_year(year) { 29 } else { 28 },
        _ => 0 // Invalid month
    }
}

fn is_valid_date(day: i32, month: i32, year: i32) -> bool {
    if day < 1 {
        return false;
    }

    if month < 1 || month > 12 {
        return false;
    }

    day <= days_in_month(month, year)
}

fn main() {
    println!("Is 2/29/2028 a valid date? {}", is_valid_date(29, 2, 2028));
    println!("Is 0/1/2027 a valid date? {}", is_valid_date(1, 0, 2027));
}
```

```
println!("Is 4/31/2026 a valid date? {}", is_valid_date(31, 4, 2026));
println!("Is 8/23/2025 a valid date? {}", is_valid_date(23, 8, 2025));
}
```

Here's the output:

```
Is 2/29/2028 a valid date? true
Is 0/1/2027 a valid date? false
Is 4/31/2026 a valid date? false
Is 8/23/2025 a valid date? true
```

In this code, `is_leap_year()` is a helper function that makes the `days_in_month()` helper function easier to understand, and both helper functions make the `is_valid_date()` function much cleaner than if you tried to do all the logic in one place.

Getting a function to call itself

One of the stranger things you can do with a function is have it execute itself. That is, you place a statement in the function that calls the function. This is called *recursion*, and such a function is called a *recursive function*.

Here's a simple program that demonstrates the basic procedure (bk01ch07/example14/):

```
const MAX_COUNT: u16 = 10;

fn main() {
    add_one(1);
}

fn add_one(counter: u16) {
    if counter <= MAX_COUNT {
        println!("counter is now {counter}.");
        add_one(counter + 1);
    } else {
        println!("All done!");
    }
}
```

The `add_one()` function does just what it says on the box: Each time it's called, it adds 1 to the `counter` variable. When `counter` goes past the `MAX_COUNT` global, the recursion ends.



WARNING Whenever your code calls a function recursively, you need to ensure that you've implemented some sort of brake on your recursive function so that it doesn't call itself forever. Having a function call itself without stopping is called *infinite recursion*, and the maximum number of recursive calls depends on the operating system and how much memory your device has installed.

In any case, it's important to build in some kind of test to ensure that the function will stop calling itself after a certain number of calls. In the `add_one()`'s function, for example, the test `if counter <= MAX_COUNT` ensures that the recursion ends once `counter` tries to go beyond `MAX_COUNT`.

Here's another example (bk01ch07/example15/):

```
fn factorial(n: u128) -> u128 {
    if n <= 1 {
        1
    } else {
        n * factorial(n - 1)
    }
}

fn main() {
    let number = 5;
    println!("factorial({number}) = {}", factorial(number));
}
```

The *factorial* of a number n is the product of every number from 1 to n . The `factorial()` function computes this value for the passed argument using recursion. The brake is that the variable `n` is decremented each time, and when it becomes less than or equal to 1, the recursion stops.

Project: Temperature Converter

To close this chapter, I take you through a practical project that puts all this function know-how together. The project is a temperature converter that converts Fahrenheit to Celsius and Celsius to Fahrenheit.

To begin, use a terminal window to navigate to the folder where you store your Rust projects, and then run the following command:

```
cargo new temperature-converter
```

Open this project's `main.rs` file, remove the default `println!` statement, and get ready to code!

Checking out the project's functions

Besides `main()`, this project contains no less than eight functions, several of which are simple helper functions that consist of a single expression.

The first function is `show_menu()`:

```
fn show_menu() {
    println!();
    println!("=====");
    println!("Temperature Converter");
    println!("=====");
    println!("1. Convert °F to °C");
    println!("2. Convert °C to °F");
    println!("Choose an option (1 or 2; press a letter to quit):");
}
```

The purpose of the `show_menu()` function is to display the program's interface, which consists of the title, a simple two-item menu, and a prompt to press 1 or 2 to choose a menu option or to press a letter to quit the program.

The next function is `get_menu_choice()`:

```
fn get_menu_choice() -> i32 {
    let mut choice = String::new();
    let _ = io::stdin().read_line(&mut choice);
    let choice = choice.trim().parse();
    choice.unwrap_or(0)
}
```

The job of the `get_menu_choice()` function is to use the standard library's `std::io` module to read whatever the user enters in response to the `show_menu()` prompt. That entry comes in as a string, so after using the `trim()` method to remove the newline character added when the user

presses Enter or Return, the `parse()` method converts the string to (in this case) an `i32` integer. (The Rust compiler infers the `i32` type from the return type in the function signature.) The final expression returns the value.



TECHNICAL
STUFF

YOUR FIRST LOOK AT RUST ERROR HANDLING

There's a lot more going on with `parse()` than just a simple type conversion, but feel free to skip this part if you're not interested in the details. The key idea here is that because the function signature infers an `i32` return type, the Rust compiler is going to insist that only `i32` is acceptable from the user, which is an example of the type safety I mention in Book 1, [Chapter 3](#).

There are three main possibilities for what the user can enter:

- **The user enters 1 or 2.** These are valid menu choices.
- **The user enters some other integer.** This isn't a valid menu choice, but it *is* a legal (`i32`) value, so the code must take this into account.
- **The user enters a non-integer.** This could be a letter or a floating-point number. Either one is both an invalid menu choice *and* an illegal (non-`i32`) value. What happens now? The program crashes!

Or, I should say, the program *would* crash, but Rust's built-in error handling comes to the rescue. Technically, the `parse()` method returns a data type called `Result`, which, if the user entered an `i32` integer, contains a valid value that the code extracts using the `unwrap` part of `unwrap_or(0)`. If, instead, the user entered anything else, such as a letter, the `Result` type will contain an error that would normally cause the entire program to crash. However, the `or` part of `unwrap_or(0)` tells the compiler to return the `i32` value 0, instead. This is a teensy taste of a huge Rust topic: error handling. Book 2, [Chapter 3](#) introduces you to error handling and the `Result` type, and then Book 3, [Chapter 4](#) fills in the details.

So, the upshot of the `get_menu_choice()` function is that it returns either 1 or 2 if the user made a menu choice, or 0 if the user pressed a letter to quit the program. (The function could also return any other integer the user enters, so the code has to take that into account as well, as you learn a bit later.)

Next up is the `get_temp()` function:

```
fn get_temp() -> f64 {
    let mut choice = String::new();
    let _ = io::stdin().read_line(&mut choice);
    let choice = choice.trim().parse();
    // If the user enters a non-integer,
    // return a value (-500.0) that's
    // invalid in both Fahrenheit and Celsius
    choice.unwrap_or(-500.0)
}
```

This function runs if the user presses 1 or 2 in the main menu, and it reads the temperature that the user wants to convert. This code is nearly identical to the `get_menu_choice()` function. The only differences here are that the function return value is `f64` and the `unwrap_or()` method specifies a floating-point value (`-500.0`). Why that value? It's an invalid temperature, as you learn next.

Next you need to include a couple of helper functions that check whether the temperature value entered by the user is valid:

```
fn is_valid_fahrenheit(fahrenheit: f64) -> bool {
    fahrenheit >= -459.67
}

fn is_valid_celsius(celsius: f64) -> bool {
    celsius >= -273.15
}
```

These one-liners both return `false` if the entered temperature is below absolute zero (that is, below `-459.67` Fahrenheit or below `-273.15` Celsius).

Now comes the heart of the program, the two functions that perform the temperature conversions:

```
fn fahrenheit_to_celsius(fahrenheit: f64) -> f64 {
    (fahrenheit - 32.0) * 5.0 / 9.0
}

fn celsius_to_fahrenheit(celsius: f64) -> f64 {
    (celsius * 9.0 / 5.0) + 32.0
}
```

These functions use the standard formulas for converting between Fahrenheit and Celsius.

You always want to offer the user a decent output and to that end the `print_converted_temp()` function takes the original and converted temperatures and units and displays them nicely:

```
fn print_converted_temp(original_temp: f64,
                        original_unit: &str,
                        converted_temp: f64,
                        converted_unit: &str) {
    println!();
    println!("{:.2}°{} = {:.2}°{}",
             original_temp, original_unit, converted_temp, converted_unit);
}
```

Putting everything together

With the project's functions defined, it's time to lash it all together in the `main()` function:

```
use std::io;

fn main() {
    // Loop until the user quits
    loop {
        // Show the main menu and get the user's choice
        show_menu();
        let choice = get_menu_choice();
        println!();

        // Match the user's choice
        match choice {
            // 0 means the user pressed a letter, so break out of the loop to
quit
            0 => {
                println!("Thanks for using Temperature Converter!");
                break;
            }
            // Convert °F to °C
            1 => {
                // Get the Fahrenheit temperature to convert
                println!("Type the °F temperature you want to convert:");
                let temp = get_temp();

                // Is it valid?
```

```

        if is_valid_fahrenheit(temp) {
            // If so, convert it and display the conversion
            let converted_temp = fahrenheit_to_celsius(temp);
            print_converted_temp(temp, "F", converted_temp, "C");
        } else {
            // If not, prompt the user to try again
            println!();
            println!("Not a valid temperature! Please try again.");
        }
    }
}
// Convert °C to °F
2 => {
    // Get the Celsius temperature to convert
    println!("Type the °C temperature you want to convert:");
    let temp = get_temp();

    // Is it valid?
    if is_valid_celsius(temp) {
        // If so, convert it and display the conversion
        let converted_temp = celsius_to_fahrenheit(temp);
        print_converted_temp(temp, "C", converted_temp, "F");
    } else {
        // If not, prompt the user to try again
        println!();
        println!("Not a valid temperature! Please try again.");
    }
}
// Handle integers not equal to 1 or 2
_ => println!("Invalid choice! Please try again."),
}
}
}
}

```

First the project imports `std::io` so that the user's input can be read in the `get_menu_choice()` and `get_temperature()` functions. The `main()` function starts off with a loop that runs until the user presses a letter to quit. The loop begins by calling `show_menu()` to display the program's main menu:

```

=====
Temperature Converter
=====
1. Convert °F to °C
2. Convert °C to °F
Choose an option (1 or 2; press a letter to quit):

```

Then `get_menu_choice()` reads the menu selection and stores the result in the `choice` variable.

Next comes a `match` structure that has four arms:

- » **0:** The user pressed a letter to quit the program, so the arm code runs `break` to quit the loop.
- » **1:** The user wants to convert a Fahrenheit temperature to Celsius, so `get_temp()` returns the Fahrenheit value and runs `is_valid_fahrenheit()`. If the temperature is valid, the arm runs `print_converted_temp()` and the loop returns to the top for another iteration.
- » **2:** The user wants to convert a Celsius temperature to Fahrenheit, so `get_temp()` returns the Celsius value and runs `is_valid_celsius()`. If the temperature is valid, the arm runs `print_converted_temp()` and the loop returns to the top for another iteration.
- » **_:** The user entered an integer that wasn't 1 or 2, so the arm displays an `Invalid choice` message and prompts the user to try again.

And there you have it: 100+ lines of pure Rust usefulness. If you're interested in building even more useful command-line utilities, be sure to check out [Book 4](#), which focuses exclusively on building for the command line.

Book 2

Learning the Rust Way of Doing Things

Contents at a Glance

Chapter 1: Ownership: Rust's Secret Sauce

[Understanding Ownership without Losing Your Mind](#)

[Moving: How Rust Transfers Ownership](#)

[Borrowing: Sharing Your Toys](#)

[References: Pointing to Things Safely](#)

[Move? Borrow? Clone? Which One Should You Use?](#)

[Project: Creating a Text Analyzer](#)

Chapter 2: Structs: Rolling Your Own Types

[You Need More Struct in Your Life](#)

[Meet the Struct](#)

[A Couple of Useful Struct Shortcuts](#)

[Methods: Teaching Structs New Behaviors](#)

[Implementing Associated Functions](#)

[Project: Building a Simple Bank Account Manager](#)

Chapter 3: Enums: Embracing Possibilities

[Meet the Enum](#)

[Option: It's Always Something \(or Is It?\)](#)

[Result: Dealing with Errors Gracefully](#)

[Project: Modeling an Authentication Flow](#)

Chapter 4: Storing Data in Rust

[Tuples: Groups of Things](#)

[Arrays: Fixed-Sized Storage](#)

[Slices: Versions of Your Stuff](#)

[Vectors: Dynamic Arrays That Grow](#)

[Hash Maps: Key-Value Storage Made Easy](#)

[Project: The Prime Explorer](#)

Chapter 5: The String Section: Text in Rust

[String Instruments: String and &str](#)

[Making String Music: Creating Strings](#)

[String Works: The Greatest Hits](#)

[String Pieces: Slicing Strings](#)

[String Movements: Common String Patterns](#)

[Project: Slugifying a String](#)

Chapter 6: Functional Programming with Closures and Iterators

[What on Earth Is Functional Programming?](#)

[Closures: Anonymous Functions with Attitude](#)

[Meet the Iterator Pattern](#)

[Creating an Iterator from a Collection](#)

[Consuming Iterators](#)

[Adapting Iterators](#)

[Project: Word Wrangler](#)

Chapter 7: Organizing Your Code

[How Rust Treats Code Organization](#)

[Starting Small: Organizing Code in One File](#)

[Growing Up: Moving to Multiple Files](#)

[Controlling Access: Public versus Private](#)

[Importing and Using Code](#)

[Working with External Dependencies](#)

[Real-World Organization Patterns](#)

Chapter 1

Ownership: Rust's Secret Sauce

IN THIS CHAPTER

- » Getting your head around ownership
 - » Coming to terms with borrowing and the borrow checker
 - » Understanding references
 - » Crafting a basic text-analysis program
-

Learning @rustlang: Borrow checker -> borrow checker -> borrow checker -> borrow checker -> understanding -> productivity -> happiness.

—LORI HOLDEN

Most of the programming constructs that I talk about in [Book 1](#) — including variables, comments, conditionals, loops, and functions — are implemented in every programming language worthy of the moniker. Variables and loops and whatnot are the meat and potatoes of programming.

Now, here in [Book 2](#), it's time to level up the cuisine and learn what unique delicacies Rust has to offer. You begin here by taste-testing what I like to call Rust's secret sauce: ownership. Over the next few pages, you're about to learn what is arguably Rust's most important concept. No pressure!

Seriously, though, so much of what makes Rust both fast and safe begins with the ideas you learn in this chapter. I won't lie to you: Most folks new to Rust find these ideas hard to grasp at first. But those same folks

report almost universally that once they get it, everything just clicks into place. Let's get you there!

Understanding Ownership without Losing Your Mind

Here are two fundamental truths about program data and memory:

- » Every piece of data, from a 1-byte Boolean to a multimegabyte string, must live somewhere in memory.
- » Someone or something must be responsible for cleaning up that data when it's no longer needed.

Cleaning up the data means removing it from memory, which frees that memory to be used elsewhere. In most programming languages, that cleanup procedure is handled in one of the following two ways:

- » **Manual memory management:** In languages such as C and C++, the programmer is responsible for both allocating memory and then freeing it when the data is no longer needed. Humans, being the eminently fallible creatures that we are, fail at these tasks distressingly often, leading to program instability, crashing, and security vulnerabilities.
- » **Garbage collection:** In languages such as JavaScript and Python, memory is allocated as needed, and then an automated garbage collection routine periodically looks for unused memory and releases it. Sounds good, but you can't control when the garbage is collected; and when it does run, it can have a substantial (negative) effect on your program's performance.



REMEMBER

MEMORY 101: MEMORY MANAGEMENT FAILS

It's one thing to understand the *how* of Rust's ownership model, but you won't really get it until you also understand the *why* of it. Why jump through all the ownership hoops laid out in this chapter? Because when memory management is left up to humans, a long list of things can — and, following Murphy's law, always do — go wrong:

- **Memory leak:** A program allocates memory for some data but never releases that memory when the data is no longer needed. If this happens enough times, the program consumes more and more of the computer's memory, eventually causing the program to crash or become unresponsive.
- **Dangling pointer:** A *pointer* — a data type that references (points to) a particular address in memory — points to an address that's no longer valid. The memory might have been freed elsewhere, reused for something else, or just doesn't belong to your program any longer. When you try to read that location, any number of bad things can happen. Your program might crash, you might get garbage data, or you might inadvertently access memory that's now being used for something else entirely in your program.
- **Double-free:** A program frees the same piece of memory twice. The first time the program frees the memory, everything works fine. But the second time? Trouble. It might be freeing memory already being used for something else, which could crash the program or could corrupt that memory, leading to mysterious and hard-to-trace bugs elsewhere in your program.
- **Buffer overflows:** A program writes more data into a memory location than that location was designed to hold. The extra data spills over into adjacent memory locations, potentially corrupting other data or critical control information that determines what your program does next. This overflow could cause your program to crash or behave erratically. In the worst case, it makes your program vulnerable to third-party attackers who try to exploit the buffer overflow to take control of the program.

Rust's ownership rules and general focus on memory safety prevent these and many more memory-related failures. The tradeoff is that the Rust compiler will wag an admonishing finger at you more often. Just remember that it's all for a good cause.

Rust thumbs its nose at both manual memory management and garbage collection. Instead, Rust implements an ownership model that ensures

the same performance as manually managed memory languages and the same thoroughness as garbage-collection languages.

Rather than having either the programmer or the garbage collector be responsible for a piece of data, in Rust's ownership model, every piece of data is responsible for its *own* cleanup. Specifically, when a variable is initialized with some data, that variable becomes the sole owner of that data. No one else can own it at the same time. And when that variable goes out of scope, the data that it owns is automatically dropped from memory.

Here's the simplest possible example of ownership in action:

```
fn main() {           // The function scope starts here
    let x = 42;        // x owns the value 42
}                     // x goes out of scope here, so 42 is dropped from memory
```

Now you can appreciate why scope is such a vital concept in Rust. When a variable's scope ends, the data that it owns is automatically freed. No manual memory management needed. No garbage collector required.

The three rules of ownership club

If you came into this chapter with fear and loathing based on what you'd heard about the complexities of ownership, let me help you set that fear aside (at least a little) by telling you that ownership in Rust operates based on just three simple rules. Yep, that's right: *three*. You can count them on one hand and have a couple of fingers left over to wipe the now unnecessary sweat from your brow.

Here are the rules:

- » Every value has an owner.
- » Each piece of data can have only one owner.
- » When the owner of a piece of data goes out of scope, the data is dropped from memory.

Here's an example of how the Rust compiler enforces these rules (refer to `bk02ch01/example01/` in this book's example files):

```
fn main() {
    { // A block scope starts here
        let y = 5; // y comes into scope and owns 5
        println!("The in-scope value of y is {y}."); // This is fine
    } // The block scope ends here, so y's value is dropped
    println!("The out-of-scope value of y is {y}."); // Not going to work!
}
```

Here's what the Rust compiler thinks of these shenanigans:

```
% cargo run
error[E0425]: cannot find value 'y' in this scope
--> src/main.rs:5:34
|
|
6 |     println!("The out-of-scope value of y is {y}.");
|                                     ^
|
help: the binding 'y' is available in a different scope in the same function
--> src/main.rs:3:13
|
3 |         let y = 5; // y comes into scope and owns 5
|             ^

For more information about this error, try 'rustc --explain E0425'.
error: could not compile 'example01' (bin "example01") due to 1 previous
error
```

The error is `cannot find value 'y' in this scope` (which I've bolded to make it stand out from the rest of the message), which is what you'd expect given Rust's ownership rules. That is, `y` went out of scope as soon as the block ended, so `y` doesn't exist for the second `println!` macro to use.

Copying owned values

Check out the following code (bk02ch01/example02/):

```
fn main() {
    let x = 16;
    let y = x;
    println!("The value of x is {x}.");
    println!("The value of y is {y}.");
}
```

Here's the output:

```
The value of x is 16.  
The value of y is 16.
```

Now answer the following question: Are `x` and `y` owners of the same value of `16` in memory? The answer is no, because according to the rules of ownership, a piece of data can only have a single owner. So, what is `y` the owner of? It owns a *different* value of `16` in memory. That is, when the program runs `let y = x`, the original value of `16` that's owned by `x` is copied and stored in another location, and `y` is assigned ownership of that copy. Two pieces of data and two owners: the compiler is happy.

So, doing something like `let variable2 = variable1` always copies the value of `variable1` into `variable2`, right? Ah, if only the Rust world was that simple!

Copying happened automatically in the example because integers in Rust implement the `Copy` trait, which is a kind of label on the data that says, “This type is small enough and simple enough that Rust can quickly and safely duplicate it just by copying its bits.” (Traits are a fascinating Rust feature that I devote an entire chapter to later; check out Book 3, [Chapter 1](#).) The `Copy` trait is automatically implemented on Rust's fixed-size, primitive types, including integers, floating-point numbers, `char`, and `bool`.

The fly in the ownership ointment here is that Rust has a long list of types that do *not* implement the `Copy` trait, which means that running `let variable2 = variable1` on any of those types won't work. But to really understand this, you need to work with a type that doesn't implement the `Copy` trait, so let's take a brief detour to examine such a type.

Introducing the String type

Back in Book 1, [Chapter 3](#), I discuss the string literal, which is an immutable and fixed-size value:

```
let greeting = "Hello, world!"; // Can't change this
```

String literals have their place, especially since they're embedded directly in the program's compiled binary file in a read-only memory

area, which ensures fast access to the string.

However, it's a rare program of even a minimal level of complexity that doesn't need to manipulate string values in some way. When your program needs to mess around with strings, you need to turn to the `String` type, which is a flexible text object that gets stored on the heap. (Check out the sidebar "[Memory 101: The heap](#)" to learn what the heap is all about.)

Rust offers several different ways to create a new `String`:

```
let s1 = "codswallop".to_string();
let s2 = String::from("hullaballoo");
let mut s3 = String::new();
```

The first two examples each create a `String` from a string literal. In the last example, an empty `String` is created as a mutable, which means you can add text to it later. When you declare a mutable `String`, you can append a `char` value using the `push()` method or append a string slice with the `push_str()` method:

```
let mut greeting = String::new();
greeting.push_str("Hello");
greeting.push(',');
greeting.push_str(" world!");
println!("{}", greeting); // Output: Hello, world!
```

You learn more about the `String` type in Book 2, [Chapter 5](#). For our purposes here, you should know that whatever variable is bound to a `String` type owns that data on the heap and obeys Rust's ownership rules, meaning that no other variable can own the same `String` value and the `String` data on the heap is released when the owning variable goes out of scope.



REMEMBER

MEMORY 101: THE HEAP

As I mention in an earlier "[Memory 101](#)" sidebar (in Book 1, [Chapter 7](#)), the stack is a chunk of memory assigned to each function to store fixed-size variable data, such as

integers, floating-point numbers, and Booleans. The stack is like a pigeonhole rack behind a hotel's reception desk, with fixed-size compartments that the staff can access quickly to insert or retrieve guest messages.

But what happens when a guest asks to store several pieces of luggage temporarily? For such scenarios, the hotel usually has a storeroom in the back of the building or off the main lobby. The guest's luggage is placed in the storeroom and the guest is given a claim ticket.

Similarly, not all program data comes in a nice, neat, stack-friendly package such as a `u32` (always 32 bytes) or a `bool` (always 1 byte). String data, for example, can be quite large, might not be assigned until runtime, and can grow or shrink as the program runs. For such variables, Rust stores the value in a separate memory area called the *heap* (like the hotel storeroom). Unlike the strict, last-in, first-out regimentation of the stack, the heap (as its name implies) is far less organized. When it's time to put a value on the heap, the program searches for any empty memory location that's big enough to store the data. The variable stores a data structure that includes the address of that memory location in the form of a *pointer* (analogous to our hotel guest's claim check), the length of the string in bytes, and the total capacity of the heap allocation. Interestingly, the pointer, the length, and the capacity are all fixed-size pieces of data themselves, so the program stores them on the stack! When the program needs the heap data, it grabs the pointer from the stack, looks up its memory address, and retrieves the data.

So, just as it takes longer for the hotel staff to retrieve the guest's stored luggage than it does to retrieve a message from the pigeonhole rack, so too does it take a Rust program longer to retrieve data from the heap than from the stack. But the tradeoff is that the heap is much more flexible in terms of the types of data it can store.

The key takeaway about the heap is that Rust's ownership system is mostly about managing heap memory safely.

Moving: How Rust Transfers Ownership

Okay, now try copying a `String` to find out what happens (bk02ch01/example03/):

```
fn main() {  
    let s1 = String::from("Rumpelstiltskin");  
    let s2 = s1;  
    println!("The value of s1 is {s1}.");  
    println!("The value of s2 is {s2}.");  
}
```


This code stores a new `String` in the `s1` variable, and then runs `let s2 = s1` as we did before with the integer values. What happens? I'll let the compiler speak for itself:

```
% cargo run
error[E0382]: borrow of moved value: 's1'
  --> src/main.rs:4:34
   |
2 |   let s1 = String::from("Rumpelstiltskin");
   |       -- move occurs because 's1' has type 'String',
   |         which does not implement the 'Copy' trait
3 |   let s2 = s1;
   |           -- value moved here
4 |   println!("The value of s1 is {s1}.");
   |                                           ^^^^ value borrowed here after move
help: consider cloning the value if the performance cost is acceptable
   |
3 |   let s2 = s1.clone();
   |               ++++++++
```

Whoa. There's a lot to unpack here. First, the curious word *borrow* appears a couple of times. Borrowing is a vital Rust concept, but let's set it to one side and come back to it later (in the “[Borrowing: Sharing Your Toys](#)” section). For now, I want you to zero in on the following (which I've bolded in the error message):

```
let s1 = String::from("Rumpelstiltskin");
-- move occurs because 's1' has type 'String',
   which does not implement the 'Copy' trait
```

This message tells you that the `String` type does *not* implement the `Copy` trait I mention earlier. The `Copy` trait is designed for types that are, in a sense, cheap to copy (such as integers and Booleans) because the copy has a predefined size and a known location (the top of the stack). But for types stored on the heap, copying would be too expensive because such types can be quite large and because the heap is slower to access.

So, instead of copying heap data, Rust *moves ownership* of that data to the new variable. The data itself stays exactly where it is in memory; only the ownership changes hands. Let's go to the videotape replay and determine what really happened in the code:

```
fn main() {
    let s1 = String::from("Rumpelstiltskin"); // s1 owns the String
    let s2 = s1; // s2 takes over ownership; s1 is now invalid
    println!("The value of s1 is {s1}."); // Error!
    println!("The value of s2 is {s2}."); // We don't get this far
}
```

Note, too, that a move happens even when you use a heap-allocated variable as part of an expression to the right of the assignment operator (=):

```
let s1 = "Hello".to_string(); // s1 owns the String
let message = s1 + ", world!"; // Now message owns it
```

By transferring ownership, Rust is enhancing memory safety by ensuring that the `String` data only ever has one owner.

Cloning: Deep-copying heap data

At this point you might be frustrated because it can seem wrong that once a variable transfers ownership, that variable becomes invalid and you can no longer use it. If you need such a variable to stay valid within the current scope, Rust offers the `clone()` method, which creates a full copy (called a *deep copy*, because it copies everything). The compiler even suggests this in the error message:

```
help: consider cloning the value if the performance cost is acceptable
|
3 |     let s2 = s1.clone();
|                   ++++++++
```

Here's an example (bk02ch01/example04/):

```
fn main() {
    let s1 = String::from("Rumpelstiltskin"); // s1 owns the String
    let s2 = s1.clone(); // s2 gets a deep copy of s1
    println!("The value of s1 is {s1}."); // This works!
    println!("The value of s2 is {s2}."); // So does this
}
```



WARNING Note that the Rust compiler's suggestion to consider cloning comes with a proviso: if the performance cost is acceptable.

Cloning is considered a computationally expensive operation, and the larger the data, the greater the cost, so you shouldn't be sprinkling `clone()` methods around like confetti. Instead, it's best to borrow instead of clone, as I describe later in the “[Borrowing: Sharing Your Toys](#)” section. Although it's not a universal solution, the compiler *will* let you know when the borrowed value “doesn't live long enough” or a variable needs to own its data, in which case you can try cloning instead of borrowing.

Passing values into functions moves them

One surprising result of Rust's ownership rules is that, because function arguments are passed by value, when you pass a non-`Copy` type (such as a `String`) to a function, ownership of that variable's value is transferred to the function's local variable. Check this out ([bk02ch01/example05/](#)):

```
fn main() {
    let msg = "Important message".to_string();
    let str_length = get_length(msg);
    println!("'{msg}' is {str_length} characters long.");
}

fn get_length(s: String) -> usize {
    s.len()
}
```

This looks straightforward enough: A `String` is stored in the `msg` variable, and then `msg` is passed to the `get_length()` function, which returns the number of characters in the string using the `len()` method. Then the code attempts to print both `msg` and the `length` result. Here's what happens:

```
error[E0382]: borrow of moved value: 'msg'
--> src/main.rs:4:16
|
2 |     let msg = "Important message".to_string();
|         --- move occurs because 'msg' has type 'String', which does not
implement the 'Copy' trait
3 |     let str_length = get_length(msg);
|                                   --- value moved here
4 |     println!("'{msg}' is {str_length} characters long.");
|         ^^^ value borrowed here after move
```

```
|
note: consider changing this parameter type in function 'get_length'
to borrow instead if owning the value isn't necessary
--> src/main.rs:7:18
|
7 | fn get_length(s: String) -> usize {
|   ----- ^^^^^^ this parameter takes ownership of the value
|   |
|   in this function
```

The compiler is letting you know that passing the value as a `String` *moves* the value into the `get_length()` function's `s` parameter, which takes ownership of the value. When the function ends, the local `s` variable goes out of scope and the `String` data vanishes in a puff of digital smoke. The `println!` macro tries to use the original `msg` variable, but that data is long gone, and the compiler shuts everything down.

Moving a non-`Copy` value into the function in this way might seem like perverse behavior on Rust's part, but it's an important part of Rust's memory safety because it ensures that when the function finishes and `s` goes out of scope, the memory that it owned is properly cleaned up. There are no ambiguities about who's responsible for that cleanup, and there's now zero risk of the same memory being released twice. Win, win!

That's all good, I hear you muttering to yourself, but there will be tons of situations where you want to let a function use a value *and* still have that value available afterward. True! And the compiler, ever optimistic, offers a way to make that happen:

```
note: consider changing this parameter type in function 'get_length'
to borrow instead if owning the value isn't necessary
```

Okay, borrowing has come up a bunch of times now, so it's time you got to know one of Rust's most important concepts.

Borrowing: Sharing Your Toys

Imagine you have a locked warehouse full of products. You want to know how much of each product is in there, so you hire someone to take

an inventory. It would be impractical (and, frankly, just plain weird) to sell all the products to the inventory-taker just so they could have access to the warehouse! Instead, you sensibly lend the person a key to the warehouse. Then, after the job is complete, that person returns the key.

Similarly, in programming, there will be situations where it makes sense for a variable to retain ownership of a value stored in memory while still allowing some other variable to access the data. In the same way that our inventory-taker gained access to the warehouse by borrowing a key, a non-owning variable can gain access to a value in memory by borrowing a reference to that value.

In Rust, a *reference* is a pointer to a value stored in memory, not the value itself. It's like having the GPS coordinates to some location. The coordinates tell you where something is located without being the thing itself.

You turn a variable into a reference to that variable by appending an ampersand (&) to the variable name (bk02ch01/example06/):

```
fn main() {  
    let s1 = String::from("Inflatable Dartboard"); // s1 owns the String  
    let s2 = &s1; // s2 borrows s1's value  
    println!("The value of s1 is {s1}."); // s1 is still valid, so this  
    works!  
    println!("The value of s2 is {s2}."); // And this runs, too, no problem  
}
```

Output:

```
The value of s1 is Inflatable Dartboard.  
The value of s2 is Inflatable Dartboard.
```

Here, instead of using `let s2 = s1` and moving ownership of the `String` from `s1` to `s2`, this code uses `let s2 = &s1` and lets `s2` borrow the `String` from `s1`, which means two things:

- » `s2` now contains only a pointer to `s1`'s value.
- » `s1` remains valid and, in particular, it retains ownership of the `String`.

So, when the program tries to use `s1` in the first `println!` macro, the compiler waves the code through because `s1` is still valid. The second `println!` macro also runs without a hitch, and the preceding program output shows that `s2` has borrowed the `s1` value.

In Rust lingo, this process is known as an *immutable borrow* because `s2` can only view `s1`'s value, it can't make changes to that value. Immutable borrowing is also often referred to as *sharing*. To learn how to set things up so that a borrow can also make changes to a value, head down to the [“Mutable borrowing: Looking and touching”](#) section.



REMEMBER Although a value can have only one owner at a time, that same value is allowed to have multiple immutable borrows at once. Having a bunch of borrowers isn't a safety issue because each borrow is just a read-only view into the data, so two or more borrowers reading the same data simultaneously won't cause a problem. (Just as it's not a problem for you to read a book while two or more people read along over your shoulder. Annoying? Yes. Does it prevent you from reading? No.)

Borrowing and function calls

Borrowing also works when passing a variable to a function (bk02ch01/example07/):

```
fn main() {
    let msg = "Important message".to_string(); // msg owns the String
    let str_length = get_length(&msg); // A reference to msg is passed
    println!("{}", msg) is {} characters long. // This works now!
}

fn get_length(s: &String) -> usize { // s borrows the passed value
    s.len()
}
```

Output:

```
Important message is 17 characters long.
```

By using `get_length(&msg)`, this revised version of the code passes to the function a reference to the `msg` value. The function's local variable `s` borrows that value, returns the length, and then goes out of scope. When the code returns to the `main()`, `msg` is still valid and still owns the `String` value, so the compiler gives a thumb's up to the `println!` statement.



REMEMBER When you examine Rust code, you won't often come across a function parameter annotated with `&String`. Instead, string parameters are almost always annotated with `&str`, which is a reference to a string slice. I explain why experienced Rust programmers almost always use `&str` when I talk more about both the `String` type and slices in Book 2, [Chapter 5](#). But for now, I can tell you that using `&str` makes your code more flexible because it means that the parameter can accept not only `String` data but also string literals.

Mutable borrowing: Looking and touching

Prepending `&` to a variable name gives you an immutable borrow of the variable's value, but sometimes the borrower needs to make changes to that value. That means the original variable has to transfer ownership to the new variable, right? Not so fast! Rust offers a much better alternative: the *mutable borrow*, which enables the borrower to make changes to the referenced value while still leaving the ownership of the value in the hands of the original variable.

In the same way that you turn an immutable variable into a mutable one by including the `mut` keyword in the declaration, you transform an immutable borrow into a mutable one by preceding the variable name with `&mut` (followed by a space):

```
let mut s1 = String::from("Change is gonna come!"); // Note the mut!
let s2 = &mut s1; // s2 borrows s1's value mutably
```

One important subtlety here is that you also have to declare the owner as mutable because you can't create a mutable borrow from an immutable

binding.

Here's a full example (bk02ch01/example08/):

```
fn main() {  
    let mut warn = String::from("Careful"); // warn owns the String mutably  
    let shout = &mut warn; // shout is a mutable borrow of warn  
    shout.push_str("!!!"); // shout can modify the original String  
    println!("{}", warn); // Output: Careful!!!  
}
```

The golden rule of borrowing

Borrowing can be a real head-scratcher, but it can help if you remember the inviolable rule of borrowing in Rust:

At any time, you can have only one of the following:

- » Multiple immutable borrows
- » One mutable borrow

The compiler simply won't allow you to have both at the same time. More accurately, the part of the compiler known as the *borrow checker* won't allow you to have both simultaneously. The borrow checker's job is to ensure that the golden rule is followed to the letter.

In practical terms the golden rule means the following:

- » While an immutable borrow is in effect, the borrow checker doesn't allow you to create a mutable borrow on the same value (bk02ch01/example09/):

```
fn main() {  
    let mut original = String::from("Original string"); // Mutable  
    owner  
    let immutable_share = &original; // Immutable share of the String  
    let mutable_share = &mut original; // Not allowed!  
    println!("{}", immutable_share); // immutable_share still alive here  
}
```

Run this code and the borrow checker displays the `cannot borrow 'original' as mutable because it is also borrowed as immutable` error. The reason is that the `immutable_share` borrow

remains in effect to the end of the function because that variable is used in the `println!` macro.

- » While a mutable borrow is in effect, the borrow checker doesn't allow you to create an immutable borrow on the same value (bk02ch01/example10/):

```
fn main() {  
    let mut original = String::from("Original string"); // Mutable  
    owner  
    let mutable_share = &mut original; // Mutable share of the String  
    let immutable_share = &original; // Not allowed!  
    println!("{mutable_share}"); // mutable_share is still alive here  
}
```

Run this code and the borrow checker raises the `cannot borrow 'original' as immutable because it is also borrowed as mutable` error. The reason is that the `mutable_share` borrow remains in effect to the end of the function.

- » While a mutable borrow is in effect, the borrow checker doesn't allow you to create another mutable borrow on the same value (bk02ch01/example11/):

```
fn main() {  
    let mut original = String::from("Original string"); // Mutable  
    owner of String  
    let mutable_share_1 = &mut original; // Mutable share of the  
    String  
    let mutable_share_2 = &mut original; // Not allowed!  
    println!("{mutable_share_1}"); // mutable_share_1 is still alive  
    here  
}
```

Run this code and the borrow checker produces the `cannot borrow 'original' as mutable more than once at a time` error. Again, the reason is that the `mutable_share_1` borrow remains in effect to the end of the function.



TECHNICAL
STUFF

SHARING XOR MUTATING

In Boolean logic, the `AND` operator means that the expression `a AND b` is true if and only if both `a` and `b` are true. The `OR` operator means that the expression `a OR b` is true if `a` is true, or if `b` is true, or if *both* `a` and `b` are true. The `XOR` (exclusive OR) operator means that the expression `a XOR b` is true if and only if either `a` is true or `b` is true. That is, `a XOR b` is false if *both* `a` and `b` are true (and, of course, if they're both false).

I'm telling you all this because a famous expression in Rust circles reduces the borrow checker's golden rule down to its essence:

Sharing XOR mutating

That is, you can either immutably borrow (share) a value or you can mutably borrow it. You can't do both. Here's the variation that I keep handy in my brain:

Share often XOR mutate once

It's not as pithy, but it covers every possibility of the golden rule.

The golden rule promotes memory safety

To understand why the golden rule of borrowing is necessary, here's an analogy: password sharing. Suppose you have a favorite streaming service, and you share guest access with family members by telling everyone the Guest account password. This is immutable sharing because guests aren't allowed to change the account password. You can share the password with as many people as you like.

Now suppose you decide to change the password. First you have to tell everyone to stop using the old password so that no one tries to use it while you're making the change. When you've finished changing the password, you then share the new password with everyone.

If you were irresponsible about your password ownership, chaos would reign because you might change the password without telling people, or you might allow multiple people to have an account with password-changing privileges.

The golden rule is the borrow checker's way of ensuring responsible ownership and thus solving some common memory problems:

- » No dangling pointers because the golden rule means you can't reference data that has been dropped
- » No data races where multiple mutable borrows make changes to the same data
- » No double-free errors because there's always just one owner of any memory value
- » No invalid memory access where you have both immutable borrows and at least one mutable borrow on a value at the same time and the mutable borrow changes the value

Working around the golden rule

Sometimes the golden rule of borrowing will get in the way of something you want to do. The borrow checker has no sympathy for your plight, believe me. However, you can take advantage of a Rust feature called the *non-lexical lifetime*, which means that the compiler keeps track of not only when a variable goes out of scope but also when it's last used.

For example, although the borrow checker won't allow you to create a mutable borrow when an immutable borrow is in effect, it *is* possible if the immutable borrow happens to complete its work *before* the mutable borrow needs to happen (bk02ch01/example12/):

```
fn main() {  
    let mut original = String::from("Original string"); // Mutable owner  
    let immutable_share = &original; // Immutable share of the String  
    println!("{}", immutable_share); // immutable_share is done after this  
    let mutable_share = &mut original; // No problem now!  
}
```

In this version, the last usage of the `immutable_share` borrow comes before we declare `mutable_share`, so, since there are no longer any immutable borrows in effect, the borrow checker allows the mutable borrow to proceed.

References: Pointing to Things Safely

The key to both immutable and mutable borrowing in Rust is the reference, which is essentially a pointer to a memory address. For example (bk02ch01/example13/):

```
fn main() {  
    let n = 42;  
    let r = &n;  
    println!("Reference r points to: {:p}", r);  
}
```

In this code, the `println!` macro uses the pointer formatter (`{:p}`) to print a reference. The output here will look something like this:

```
Reference r points to: 0x7ff7b55f3664
```

It's a little scary to have a raw memory address like that apparently within your grasp. Not to worry, though, because Rust doesn't just hand you a raw memory address and wish you luck. A reference is a safe, typed pointer managed by Rust's borrowing rules. Even if you tried, you couldn't use that address to wander off and poke around random memory location. The Rust compiler enforces that the thing the reference points to actually exists (that is, it's not a dangling reference).

So, while a reference contains a memory address under the hood, in everyday Rust you should think of it more as a borrowed view into a value rather than a raw memory address.

Dereferencing: Getting at the referenced value

Let's make a brief return to an earlier example (bk02ch01/example06/):

```
fn main() {  
    let s1 = String::from("Inflatable Dartboard");  
    let s2 = &s1;  
    println!("The value of s1 is {s1}.");  
    println!("The value of s2 is {s2}.");  
}
```

Now look again at the following statement:

```
println!("The value of s2 is {s2}.");
```

And here's its output:

```
The value of s2 is Inflatable Dartboard.
```

You might be wondering how, if `s2` contains only a reference to the "Inflatable Dartboard!" `String` value owned by `s1`, the `{s2}` part resolves to that original `String` value. Weird, no? It's not once you learn that when Rust comes upon `s2` in the code, behind the scenes it automatically *dereferences* the `&s1` reference associated with `s2`, which means `s2` is given access to the value that it points to.



MEMORY 101: FAT POINTERS AND CONTROL BLOCKS

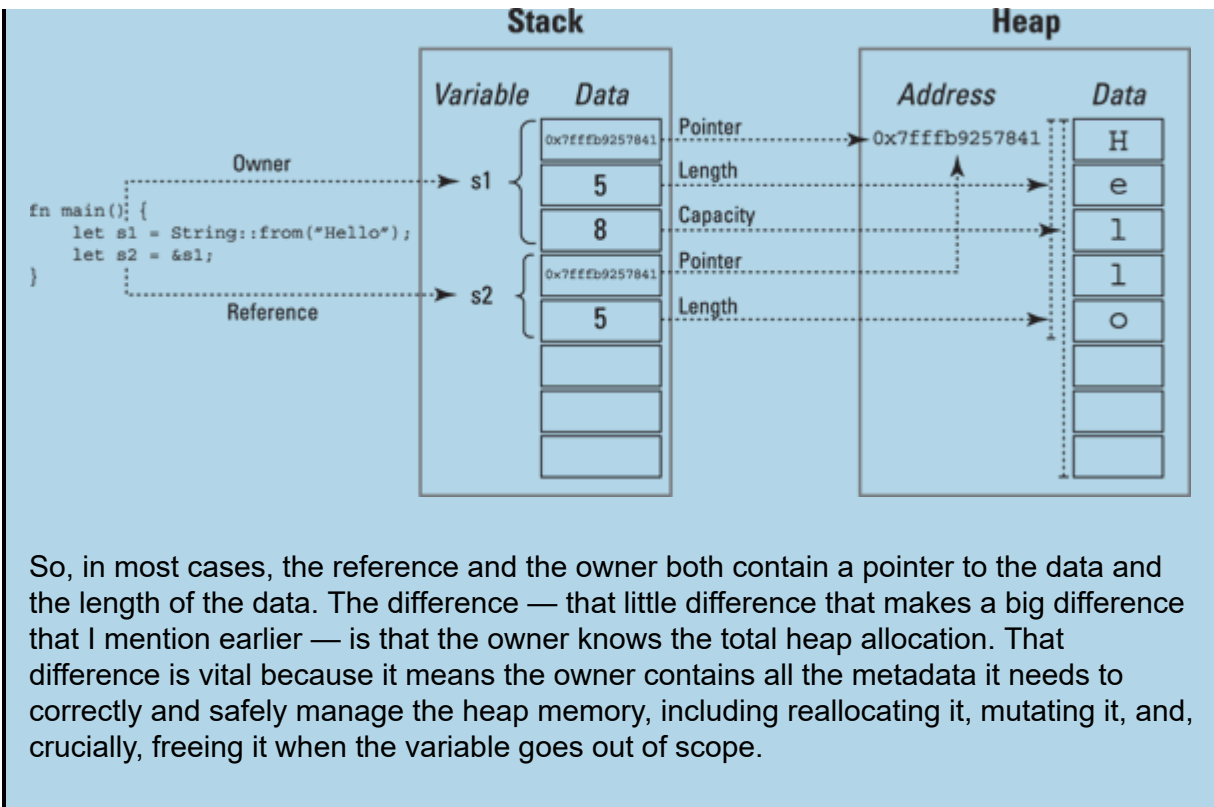
From the perspective of memory (and you could say that the underlying point of these *Memory 101* sidebars is that, with Rust, memory is the primary perspective), what is the difference, really, between a variable that owns heap data and a variable that's a reference to that heap data?

A reference (immutable or mutable, it doesn't matter) lives on the stack and is basically what's known in the memory game as a *fat pointer*:

- For most types, including `&i32` and `&String`, the reference is just a pointer to the data.
- For dynamically sized types, such as `&str`, the reference is a pointer plus the length of the data (which is why it's called a "fat" pointer).

A variable that's the owner of a heap-allocated type (such as a `String`) is a small data structure (technically, it's a `struct`, which you learn about in Book 2, [Chapter 2](#)) that also lives on the stack and is known as a *control block*, which includes:

- A pointer to the heap allocation
- The length of the data
- The total capacity of the allocation (that is, how much heap space was allocated when the variable was initialized)



References and the `println!` macro

I'd like to clear up what might at this point be a minor mystery from much earlier in this chapter. Way back in the “[Moving: How Rust Transfers Ownership](#)” section, I presented the following example (bk02ch01/example03/):

```
fn main() {
    let s1 = String::from("Rumpelstiltskin");
    let s2 = s1;
    println!("The value of s1 is {s1}.");
    println!("The value of s2 is {s2}.");
}
```

Here's part of the borrow checker's complaint about this code:

```
error[E0382]: borrow of moved value: 's1'
--> src/main.rs:4:34
...
4 |     println!("The value of s1 is {s1}.");
  |                                     ^^^^ value borrowed here after move
```

Why does the borrow checker think the use of `{s1}` in this `println!` statement is a borrow? After all, it's `{s1}` not `{&s1}`. The secret is that

when you expand the `println!` macro (which I talk more about in Book 3, [Chapter 5](#)), the underlying code can work only with references. So, when the Rust compiler comes across this:

```
println!("The value of s1 is {s1}.");
```

It automatically changes it to this:

```
println!("The value of s1 is {}. ", &s1);
```

There's your borrow! Mystery solved. You're welcome.

Reference lifetimes

Here's another crucial borrow checker rule to write on your forearm so you always remember it:

A reference can't outlive the value it points to.

Hunh? Let's make this concrete with an example (bk02ch01/example14/):

```
fn main() {
    let r;
    {
        // Start of a new scope.
        let x = 5; // x is local to this scope.
        r = &x;    // Initialize r to be a reference to x.
    }            // The scope ends here, so x goes away. Uh oh.
    println!("{}", r); // r is now pointing to freed memory. Not good!
}
```

Try to compile this code and the borrow checker raises both an eyebrow and the following error:

```
'x' does not live long enough
```

The borrow checker is protecting you from the classic *dangling reference* memory blunder, where a reference points to memory that has previously been freed. When you create a reference to a value, you need to ensure that the value sticks around longer than the reference.

Move? Borrow? Clone? Which One Should You Use?

Given an existing variable that owns a chunk of data on the heap, when your code needs to declare a new variable that uses the same heap value, you have four main choices: move the original variable into the new one; immutably borrow the value; mutably borrow the value; or clone the value. How do you decide which one to use in your code?

That question often does not have a straightforward answer, so here are a few things to think about when deciding which route to take:

» Use a move when

- Your code no longer needs the original variable.
- The new variable will outlive the original variable.
- The new variable is meant to be the true owner of the data going forward in your code. A common example is moving, say, a `String` value into a collection, such as a vector (refer to Book 2, [Chapter 4](#)) because the vector needs to own the data.

» Use an immutable borrow when

- The original variable is the true owner of the data.
- You just want the new variable to read the data without changing it.
- You want to share the data with multiple variables (since the borrow checker allows multiple immutable borrows).

» Use a mutable borrow when

- The original variable is the true owner of the data.
- You want the new variable to be able to change the data.
- The data is large or complex, so you want to avoid either an entirely new allocation of the data or a clone of the data, both of which would be computationally expensive.

» Clone the data when

- Both the original owner and the new variable need their own copies of the data.
- Your code requires two independent copies of the data so that it can make changes to one without affecting the other.
- The data is neither large nor complex, so the cloning won't be expensive.

Project: Creating a Text Analyzer

To put many of these concepts of ownership and borrowing through their paces, this chapter's project is a simple text analysis program that examines a lengthy-ish (just over 1,000 words) book excerpt and returns the number of bytes, characters, and words and also the longest word.

In a terminal, navigate to your main Rust projects folder, and then run the following command:

```
cargo new text-analyzer
```

Load this project's `main.rs` file in your text editor, delete the `println!` statement that's there, and then add this project's code.

Here's the code:

```
fn main() {  
    let excerpt = String::from("It was the best of times, it was the worst of  
times, it was the age of wisdom, it was the age of foolishness, it was the  
epoch of belief, it was the epoch of incredulity, it was the season of Light,  
it was the season of Darkness, it was the spring of hope, it was the winter  
of despair, we had everything before us, we had nothing before us, we were  
all going direct to Heaven, we were all going direct the other way-in short,  
the period was so far like the present period, that some of its noisiest  
authorities insisted on its being received, for good or for evil, in the  
superlative degree of comparison only...");  
  
    println!("Length: {} bytes", get_length(&excerpt));  
    println!("Characters: {}", get_character_count(&excerpt));  
    println!("Words: {}", get_word_count(&excerpt));  
    println!("Longest word: {}", get_longest_word(&excerpt));  
}
```

```

fn get_length(text: &str) -> usize {
    text.len()
}

fn get_character_count(text: &str) -> usize {
    text.chars().count()
}

fn get_word_count(text: &str) -> usize {
    text.split_whitespace().count()
}

fn get_longest_word(text: &str) -> String {
    let text_without_dashes = text.replace("-", " ");
    let words = text_without_dashes.split_whitespace();
    let mut longest_word = "";
    for word in words {
        if word.len() > longest_word.len() {
            longest_word = word;
        }
    }
    longest_word.to_string()
}

```

Here's the output:

```

Length: 5774 bytes
Characters: 5762
Words: 1001
Longest word: supernaturally

```

The `excerpt` variable holds the text, only part of which is shown here. (The text, in case you don't recognize it, consists of Chapter 1 of a public domain version of the Charles Dickens classic novel, *A Tale of Two Cities*.)

The code then calls four functions, passing a reference to the text (`&excerpt`) in each case. Each function defines a `text` argument with the string slice (`&str`) type:

» **`get_length()`**: Uses `text.len()` to return the number of bytes (a `usize` value) in the `excerpt`.

- » **get_character_count():** Uses `text.chars()` to return a collection of all the characters in the excerpt, and then uses `.count()` to return the number of characters (a `usize` value) in the collection.
- » **get_word_count():** Uses `text.split_whitespace()` to return a collection of string slices that are sub-slices of the excerpt, separated by any amount of whitespace (such as spaces and newlines). The function then uses `.count()` to return the number of items (a `usize` value) in the collection.
- » **get_longest_word():** Returns the longest word in the excerpt. This is the most involved function, so I explain it in detail next.

You might be wondering why the `get_longest_word()` function returns a `String` rather than a string slice (`&str`). After all, the longest word, whatever it might be, will definitely be a word from the text, so by definition it's a slice of that text, right? It's true that the longest word is found in the original text, but there's an important ownership subtlety that happens in `get_longest_word()`. It's also a wonderfully instructive subtlety because it's a great example of the borrow checker being super-safe to prevent dangling references.

The subtlety that I'm talking about happens in the first statement:

```
let text_without_dashes = text.replace("-", " ");
```

The `slice.replace(from, to)` method searches `slice` (a string slice) for every instance of the text `from` and replaces it with the text `to`. So, `text.replace("-", " ")` replaces every hyphen (-) in the text with a space. This is necessary to avoid compound words such as “five-and-twentieth” from being considered as a single word.

However, `replace()` doesn't modify the original slice. Instead, it creates a `newString`, which in the function is stored in the `text_without_dashes` local variable, which owns this newly created `String`. Keep that ownership in mind as you progress through the rest of the function.

Next the code applies the `split_whitespace()` method to `text_without_dashes`:

```
let words = text_without_dashes.split_whitespace();
```

Whitespace refers to characters such as spaces, tabs, and newlines. What `split_whitespace()` does is use these and other whitespace characters to recognize the individual words in the text, each of which it saves to a collection named `words`. These words are `&str` slices that borrow from `text_without_dashes`.

The `words` collection is called an *iterator* (refer to Book 2, [Chapter 6](#)), which has the fundamental characteristic that you can run through (iterate) its items. The code takes advantage of that to find the longest word in the collection:

```
let mut longest_word = "";
for word in words {
    if word.len() > longest_word.len() {
        longest_word = word;
    }
}
```

After initializing the `longest_word` variable as a mutable, the `for word in words` loop runs through every word. In the loop, an `if` conditional checks each `word` slice to determine if it's longer than the longest word it has encountered so far. When it finds a longer one, it assigns that slice to `longest_word` and the loop continues until the last word has been checked.

Once the loop ends, `longest_word` is a string slice that holds the longest word. But recall that the function's return type is `String`, so the final chore is to convert `longest_word` to a `String` and make it the function's return value by leaving off the semicolon:

```
longest_word.to_string()
```

So, can you figure out why this function had to return a `String` rather than a string slice (`&str`)?

Suppose, instead, I had set the function's return type to `&str` and then just returned `longest_word` as a slice. That would have been perfectly fine if `longest_word` was a slice from `text`, which was the original, borrowed slice passed as the function argument. Instead, `longest_word` is a slice from `text_without_dashes`, a `String` that was created inside the function and, crucially, *owns* the revised version of the text. As soon as the function ends, `text_without_dashes` gets dropped, so the function would be returning a slice (`longest_word`) that's pointing to memory that no longer exists — a dangling pointer. Rust's borrow checker prevents this by insisting that references can never outlive the data they refer to.

By setting the return value to `String` (owned) instead of `&str` (borrowed), the code transfers ownership of the `longest_word` value back to the function caller.



TIP Okay, I can hear you thinking out loud: “It’s fine for you to tell me that `replace()` creates a new `String`, but how can I find that out for myself for other Rust methods and functions that you don’t happen to mention? Give a person a fish, etc., etc.” Great question! The answer is to turn to the Rust documentation for the standard library, available at <https://doc.rust-lang.org/std/index.html>. In the search box, type the standard library feature you want to research, and then click the appropriate item in the search results that show up. For example, type **replace**, and then click `str::replace` in the results. The resulting page begins with the function signature for `replace()`:

```
pub fn replace<P>(&self, from: P, to: &str) -> String
```

Okay, there’s lots of weird stuff here that you can fortunately ignore because what you want to take away from this information is the return value at the end, which as shown in the preceding code is `String`. Now you know!

Chapter 2

Structs: Rolling Your Own Types

IN THIS CHAPTER

- » Getting to know structs
 - » Creating struct instances
 - » Making your struct do something useful
 - » Building a simple bank account manager
-

The C and C++ compiler[s], when I make mistakes, make me feel like a terrible person. The Rust compiler is more like it's guiding you to write super-safe code.

—NELL SHAMRELL-HARRINGTON

Rust is a language that practically invites you to look under the hood. For example, when you have a heap-allocated `String` value, Rust offers methods that tell you how many bytes `String` has stored on the heap (`len`) and the total capacity of the allocation (`capacity`).

When you pop the hood on Rust and start poking around, you'll notice two things over and over: structs and enums. These are two of Rust's foundational data types because so much of the rest of Rust is built on them. You could say that structs and enums, together, define the overall shape of data in Rust. They make up the fundamental structure that determines what a value contains and how it can be used. In this chapter, you discover the power and usefulness of structs, and you explore enums.

You Need More Struct in Your Life

When you're gathering all the things you need for a camping trip, one strategy would be to lay out a large tarp on the floor, create a pile of all your gear in the middle, and then tie it all up. But, of course, that's madness. It's too inefficient, too disorganized, too helter-skelter. Instead, you put the food into a cooler, your personal stuff into a backpack, the fishing gear into a tackle box, and so on. Ah, that's better: efficient, organized, and neither helter nor skelter.

The big pile of camping stuff in a tarp sounds ridiculous, but that's essentially how your program variables are organized. If `main.rs` is the tarp, all your variables are just sitting in there with no organization, no structure.

For example, suppose you're building a program to track the books in a library. You might start with a few variables to represent some basic book data:

```
let title = "The Mystery of the Missing Sock";  
let author = "Millicent Peeved";  
let pages = 320;  
let available = true;
```

Okay, fine. But what if you have to send all this data to a function? Now you have to remember to send all four variables. And this is just a simplified example. An actual library program might need to track a dozen or more variables for each book. And this is just *one* book. Even a small library has *thousands*!

Having all these separate-but-related variables floating around is like making a big, disorganized pile of your camping stuff. That disorganized state isn't a problem for small Rust programs, but once your projects grow beyond a certain size, you need some structure. You need the coding equivalent of a backpack, a cooler, and a tackle box.

Meet the Struct

That equivalent is a Rust feature called the struct (short for, appropriately enough, *structure*), which is designed to enable you to bring multiple bits of related data together into a unit. Here's the general syntax:

```
struct Name {  
    field1: type1,  
    field2: type2,  
    ...  
    fieldn: typen,  
}
```

where *Name* is the name of the struct; *field1*, *field2*, and so on are the names of the struct data fields; and *type1*, *type2*, and so on specify the type for each field. For example, here's a struct that could be used to hold book data:

```
struct Book {  
    title: String,  
    author: String,  
    pages: u32,  
    available: bool,  
}
```



REMEMBER By convention, struct names use *PascalCase* (also known as *UpperCamelCase*), where the first letter of each word in the name is capitalized. Field names use the standard Rust `snake_case`.

What happened here is that you created a shiny, brand-spanking-new, custom data type in Rust named `Book`. Look at you! And note, too, that you're free to define your custom types with any of Rust's standard types, including integer, floating-point number, `String`, `char`, `bool`, `enum` (refer Book 2, [Chapter 3](#)), and even `struct`!

With that just-out-of-the-oven type defined, now you can use it to keep the data for any book in a single place, a single *structure*. How? That's the topic of the next section.

Creating an instance of a struct

On its own, a struct doesn't do much. Going back to the camping trip metaphor, a struct isn't a backpack; it's a set of instructions for *making* a backpack. So, a struct is a kind of template. For example, the `Book` struct definition from the preceding section isn't itself a record for a library book; it's a template for making — creating an *instance* of — such a record. (The know-it-alls prefer to use the word *instantiating*.)

Here's the general syntax for creating an instance of a `struct` type:

```
let variable = Name {  
    field1: value1,  
    field2: value2,  
    ...  
    fieldn: value,  
}
```

where *variable* is the name of the variable that will hold the instance; *Name* is the name of the `struct`; *field1*, *field2*, and so on are the names of the `struct` data fields; and *value1*, *value2*, and so on are the values for each field.

For example, the following creates an instance of the `Book` struct:

```
let missing_sock_book = Book {  
    title: String::from("The Mystery of the Missing Sock"),  
    author: String::from("Millicent Peeved"),  
    pages: 320,  
    available: true,  
};
```

But just as you can use, say, the `String` type to create as many strings as you need, with your `Book` type defined, you're free to create a new instance for every book your program needs to work with:

```
let tiktok_book = Book {  
    title: String::from("The Old Person's Guide to TikTok"),  
    author: String::from("I. M. Young"),  
    pages: 1584,  
    available: false,  
};
```



REMEMBER In practice, when you're dealing with a large number of data elements (such as books in a library), you wouldn't create individual variables for each element. Instead, you'd store everything in one of Rust's collections, such as a vector. I introduce you to vectors and other collection types in Book 2, [Chapter 4](#).

Getting at the data

Okay, it's one thing to get data into a struct, but how do you get it out? Use *dot notation*, like so:

```
variable.field
```

where *variable* is the name of the variable that contains the instance of the struct and *field* is the name of the struct field that has the data you want to work with. Here's an example (bk02ch02/example01/ in this book's example files):

```
struct Book {
    title: String,
    author: String,
    pages: u32,
    available: bool,
}

fn main() {
    let missing_sock_book = Book {
        title: String::from("The Mystery of the Missing Sock"),
        author: String::from("Millicent Peeved"),
        pages: 320,
        available: true,
    };

    println!("Title: {}", missing_sock_book.title);
    println!("Author: {}", missing_sock_book.author);
    println!("Pages: {}", missing_sock_book.pages);

    if missing_sock_book.available {
        println!("This book is available for checkout.");
    } else {
        println!("This book is currently checked out.");
    }
}
```

```
}  
}
```

Here's the output:

```
Title: The Mystery of the Missing Sock  
Author: Millicent Peeved  
Pages: 320  
This book is available for checkout.
```

Making it mutable

If you want your program to make changes to any of the field data inside a struct instance, you need to create that instance using the `mut` keyword (bk02ch02/example02/):

```
let mut missing_sock_book = Book {  
    title: String::from("The Mystery of the Missing Sock"),  
    author: String::from("Millicent Peeved"),  
    pages: 320,  
    available: true,  
};  
  
// Someone checks out the book  
missing_sock_book.available = false;  
println!("{}", missing_sock_book.title);
```

Yep: You have to make the entire `struct` mutable. Rust doesn't offer any way to make only selected fields in the `struct` mutable.

A Couple of Useful Struct Shortcuts

You'll likely use `struct` to roll your own types frequently, so it'll help to let you in on a couple of handy shortcut features that Rust has implemented: the field init shorthand and the struct update syntax.

The field init shorthand

Since creating an instance of a particular struct is such a common task, many programmers delegate that task to a function:

```
fn create_book(title: String, author: String, pages: u32) -> Book {  
    Book {  
        title: title,  
        author: author,
```

```

        pages: pages,
        available: true,
    }
}

fn main() {
    let missing_sock_book = create_book(
        String::from("The Mystery of the Missing Sock"),
        String::from("Millicent Peeved"),
        320,
    );
}

```

That function works like a charm, but it feels redundant and wasteful to write `title: title` and `author: author`. Rust takes pity on us fingertip-sore coders by offering the *field init shorthand*, which, when the function parameter name matches the field name, enables you to specify just the field name when you instance the struct (bk02ch02/example03/):

```

fn create_book(title: String, author: String, pages: u32) -> Book {
    Book {
        title,
        author,
        pages,
        available: true,
    }
}

```

The struct update syntax

You often need to create a *new* instance of a struct that's mostly the same as an existing instance, with just a few fields changed. For example, if your library gets a new edition of a book, the new edition might have a different page count but keep the same title and author. Rather than typing every field again, you can use the *struct update syntax* to copy most fields from an existing instance while specifying new values for the fields you want to change (bk02ch02/example04/):

```

// First, here's the original instance
let missing_sock_book = Book {
    title: String::from("The Mystery of the Missing Sock"),
    author: String::from("Millicent Peeved"),
    pages: 320,
}

```

```

        available: false,
    };

    // Now create a new instance, changing only 'available'
    let missing_sock_book2 = Book {
        available: true,
        ..missing_sock_book // Copy all other fields from
                              'missing_sock_book'
    };
    println!("Available: {}", missing_sock_book.available); // This works
    println!("Pages: {}", missing_sock_book.pages); // Yep, this works, too
    println!("Title: {}", missing_sock_book.title); // Error!

```

If you have one or more fields you want to initialize to a specific value for the new instance, include those fields first. Read the `..instance` syntax to mean “everything else in the existing *instance*.” This line must come last.

The ownership and move/copy details of the struct update syntax are a bit tricky, so here are the possibilities you need to bear in mind:

- » When you specify a field and a value, that field remains valid in the original instance. For example, I specified a value for the `available` field in the `missing_sock_book2` instance, so the following is still a valid reference:

```
missing_sock_book.available
```

- » For every variable value included in the `..instance` syntax, if that variable’s type implements the `Copy` trait (refer to Book 2, [Chapter 1](#)), Rust *copies* the data to the new instance and leaves the data in the old instance as is. For example, the `pages` field is included in `..instance` and `pages` uses the `u32` type, which implements the `Copy` trait. Therefore, the following is still a valid reference:

```
missing_sock_book.pages
```

- » For every variable value included in the `..instance` syntax, if that variable’s type does *not* implement the `Copy` trait, Rust *moves* the data to the new instance, an ownership change that invalidates the data in the old instance. For example, the `title` field is included in `..instance` and `title` uses the `String` type, which does not

implement the `Copy` trait. Therefore, the following is now an invalid reference:

```
missing_sock_book.title
```

Methods: Teaching Structs New Behaviors

Being able to create custom data types and use them to organize related data in a single data structure makes the struct worth learning. But you can take the struct to an entirely new level of usefulness by customizing it to perform tasks on its data. You specify a struct task by implementing a *method*, which is a function that belongs to a specific struct data type. Here's the general syntax for defining a method:

```
impl Name {  
    fn method1(&self) -> return_type {  
        // Method statements go here  
    }  
    fn method2(&mut self) -> return_type {  
        // Method statements go here  
    }  
    etc.  
}
```

where *Name* is the name of your struct; *method1*, *method2*, and so on are the names of the methods; and the optional *return_type* is the type that each method returns (if any).

The `impl` keyword starts an *implementation block* in which you define one or more methods (`fn` blocks) for the task or tasks you want your struct to do. Note that each method's first parameter is some form of the `self` keyword, which refers to the instance of the struct on which the method is called:

- » Use `&self` to borrow the struct immutably, which means the method code can read the instance data but can't change it.

- » Use `&mut self` to borrow the struct mutably, which means the method code can both read and change the instance data.

Here's an example (bk02ch02/example05/):

```
impl Book {
    fn checkout(&mut self) {
        self.available = false;
        println!("The book {} has been checked out.", self.title);
    }
    fn return_book(&mut self) {
        self.available = true;
        println!("The book {} has been returned.", self.title);
    }
    fn get_info(&self) -> String {
        format!("{}", self.title, "{} pages.", self.author, self.pages)
    }
    fn is_long(&self) -> bool {
        self.pages > 400
    }
}
```

With your methods implemented, you can call them on your instances:

```
fn main() {
    let mut tiktok_book = Book {
        title: String::from("The Old Person's Guide to TikTok"),
        author: String::from("I. M. Young"),
        pages: 1584,
        available: false,
    };

    println!("{}", tiktok_book.get_info());

    if tiktok_book.is_long() {
        println!("");
        println!(
            "At {} pages, {} is a long book!",
            tiktok_book.pages, tiktok_book.title
        );
    }

    tiktok_book.checkout();
    tiktok_book.return_book();
}
```

Output:

```
The Old Person's Guide to TikTok by I. M. Young, 1584 pages.  
At 1584 pages, The Old Person's Guide to TikTok is a long book!  
The book The Old Person's Guide to TikTok has been checked out.  
The book The Old Person's Guide to TikTok has been returned.
```

Implementing Associated Functions

Sometimes it makes sense to implement functions for a particular struct that aren't called on a particular instance, so they don't use the `self` keyword. These are called *associated functions* and are like utility functions that belong to the struct itself.

The most common type of associated function is a *constructor*, which is a special function that creates a new instance of its struct, usually with a special set of default field values. Take a look (bk02ch02/example06/):

```
impl Book {  
    fn new(title: String, author: String, pages: u32) -> Book {  
        Book {  
            title,  
            author,  
            pages,  
            available: true,  
        }  
    }  
  
    fn new_unavailable(title: String, author: String, pages: u32) -> Book {  
        Book {  
            title,  
            author,  
            pages,  
            available: false,  
        }  
    }  
}  
  
fn main() {  
    let book1 = Book::new(  
        String::from("Welcoming Our New Insect Overlords"),  
        String::from("K. Brockman"),  
        352,  
    );  
  
    let book2 = Book::new_unavailable(  
        String::from("Welcoming Our New Insect Overlords"),  
        String::from("K. Brockman"),  
        352,  
    );  
}
```



```

        String::from("ChatGPT and Me: A Love Story"),
        String::from("Anonymous"),
        704,
    );
}

```

Note that this code creates an `impl` block for the `Book` struct that's separate from the one shown earlier. Rust allows you to set up as many `impl` blocks for a struct as you need.

MEMORY 101: RUST'S DYNAMIC TYPES ARE STRUCTS

The struct is a useful and powerful feature, so useful and powerful, in fact, that Rust itself uses the struct frequently. For example, a slice reference such as `&str` is essentially a struct that's a fat pointer (a pointer plus a length):

```

struct StrRef {
    ptr: *const u8, // Pointer to UTF-8 bytes
    len: usize,     // The number of bytes
}

```

A heap-allocated type, such as a `String`, has a pointer, length, and capacity:

```

struct String {
    ptr: *mut u8, // Pointer to heap-allocated bytes
    len: usize,   // The number of bytes currently used
    cap: usize,   // Total bytes allocated
}

```

Similarly, a vector (which you learn about in Book 2, [Chapter 4](#)) is another heap-allocated type, so it, too, has a pointer, length, and capacity:

```

struct Vec<T> {
    ptr: *mut T, // Pointer to heap-allocated elements
    len: usize,  // The number of elements currently stored
    cap: usize,  // The number of elements that can fit
}

```

Project: Building a Simple Bank Account Manager

Okay, let's put structs to work. In this project, you use a struct and quite a few methods and associated functions to build a program that manages a bank account. Is it realistic? Not even close! But it will give you a taste of how structs work in the real world.

To get started, crank open a terminal, `cd` into the directory where you store your Rust projects, and then run the following command:

```
cargo new bank-account-manager
```

Open that project in your text editor and then follow along as I take you through the project code.

Setting up the struct

This entire project is built around a struct that models a simplified bank account. At the top of your `main.rs` file, add the following:

```
use std::io;

struct BankAccount {
    account_holder: String,
    balance: f64,
    account_number: String,
}
```

Yep, it's about as simple as it gets: your name (as a `String`), the account balance (`f64`), and the account number (another `String`).

Later you add some input code to read menu choices and deposit and withdrawal amounts, so that's why the import of `std::io` is on the first line.

Implementing the struct methods

Just below the `struct` definition, add an implementation block:

```
impl BankAccount {

}
```

Throughout the rest of this section, I take you through the methods and the single associated function for this project, all of which you enter in

this impl block.

First up is the constructor function for the struct, which is the only associated function in this project:

```
fn new(name: String, init_balance: f64, num: String) -> BankAccount {  
    BankAccount {  
        account_holder: name,  
        balance: init_balance,  
        account_number: num,  
    }  
}
```

Next is the `deposit()` method, which processes a deposit to the account:

```
fn deposit(&mut self, amount: f64) {  
    if amount > 0.0 {  
        self.balance += amount;  
        println!("Successfully deposited ${:.2}", amount);  
        println!("New balance: ${:.2}", self.balance);  
    } else {  
        println!("Deposit amount must be positive!");  
    }  
}
```

If the passed `amount` value is positive, that value is added to the account and the new balance is displayed.

Next is the `withdraw()` method, which processes a withdrawal from the account:

```
fn withdraw(&mut self, amount: f64) {  
    if amount < 0.0 {  
        println!("Withdrawal amount must be positive!");  
    } else if amount > self.balance {  
        println!("Insufficient funds! Your balance is ${:.2}", self.balance);  
    } else {  
        self.balance -= amount;  
        println!("Successfully withdrew ${:.2}", amount);  
        println!("Remaining balance: ${:.2}", self.balance);  
    }  
}
```

This code first checks that a positive withdrawal amount was passed to the function. Then it checks that sufficient funds are in the account to

make the withdrawal. If everything is okay, the funds are withdrawn and the remaining balance is displayed.

The `check_balance()` method is a one-liner that displays the account's current balance:

```
fn check_balance(&self) {  
    println!("Current balance: ${:.2}", self.balance);  
}
```

The final method is `display_account_info()`, which just does what it says:

```
fn display_account_info(&self) {  
    println!("\n--- Account Information ---");  
    println!("Account Holder: {}", self.account_holder);  
    println!("Account Number: {}", self.account_number);  
    println!("Current Balance: ${:.2}", self.balance);  
    println!("-----");  
}
```

Adding the user input functions

This project asks the user to enter both strings and numbers. To handle string input, the program uses the `get_user_input()` function:

```
fn get_user_input() -> String {  
    let mut input = String::new();  
    let _ = io::stdin().read_line(&mut input);  
    input.trim().to_string();  
}
```

To handle numeric input, the program uses the `get_amount()` function:

```
fn get_amount() -> f64 {  
    println!("Enter $ amount:");  
    let input = get_user_input();  
    input.parse::<f64>().unwrap_or_default()  
}
```

This code calls `get_user_input()`, and then parses the input string as an `f64` value. The `unwrap_or_default()` method handles non-numeric inputs by returning `0.0` (the default value for the `f64` type). I explain this method when I talk about the `Result` enum in Book 2, [Chapter 3](#).

Creating an instance of the struct

Okay, now we're in the `main()` loop, and the first task is to create an instance of the `BankAccount` struct. Here's the code that does that:

```
println!("Welcome to Simple Bank!");
println!("=====");

// Create a new account
println!("Let's set up your account.");

println!("Enter your name:");
let name = get_user_input();

println!("Enter your desired account number:");
let account_num = get_user_input();

println!("Enter your initial deposit amount:");
let initial_deposit = get_amount();

let mut account = BankAccount::new(name, initial_deposit, account_num);

println!("\nAccount created successfully!");
account.display_account_info();
```

After welcoming the user to the app, the code prompts the user for their name, account number, and initial deposit. The program uses that data to create an instance of the `BankAccount` struct:

```
let mut account = BankAccount::new(name, initial_deposit, account_num);
```

The code then calls the `display_account_info()` method. Here's a sample session so far:

```
Welcome to Simple Bank!
=====
Let's set up your account.
Enter your name:
Bilbo
Enter your desired account number:
12-3456
Enter your initial deposit amount:
Enter $ amount:
5000

Account created successfully!
```

```
--- Account Information ---  
Account Holder: Bilbo  
Account Number: 12-3456  
Current Balance: $5000.00
```

Implementing the main menu loop

The final part of the program is the main loop:

```
loop {  
  println!("\nWhat would you like to do?");  
  println!("1. Check Balance");  
  println!("2. Make Deposit");  
  println!("3. Make Withdrawal");  
  println!("4. View Account Info");  
  println!("5. Exit");  
  println!("Enter your choice (1-5):");  
  
  let choice = get_user_input();  
  
  match choice.as_str() {  
    "1" => {  
      account.check_balance();  
    }  
    "2" => {  
      println!("How much would you like to deposit?");  
      let amount = get_amount();  
      account.deposit(amount);  
    }  
    "3" => {  
      println!("How much would you like to withdraw?");  
      let amount = get_amount();  
      account.withdraw(amount);  
    }  
    "4" => {  
      account.display_account_info();  
    }  
    "5" => {  
      println!("Thank you for using Simple Bank!");  
      println!("Have a great day, {}!", account.account_holder);  
      break;  
    }  
    _ => {  
      println!("Invalid choice! Please enter 1, 2, 3, 4, or 5.");  
    }  
  }  
}
```

This code displays a menu of the tasks the user can perform, calls `get_user_input()` to process the user's menu choice, and then uses `match` to run through the possibilities.

Chapter 3

Enums: Embracing Possibilities

IN THIS CHAPTER

- » Handling possibilities with enums
 - » Working with Option, the all-or-nothing enum
 - » Handling errors with Result
 - » Building a simple state machine
-

Rust: safe, fast, productive — pick three.

—AARON TURON

It's often the case that a Rust program finds itself in a particular *state*, where one out of a finite set of possibilities is currently true. Think of a traffic light, for example. A basic light is always in one of three possible states: red, yellow, or green. Similarly, a connection to a remote resource is always in one of four possible states: connecting, connected, disconnected, or failed. Whether your program is simulating a traffic light or attempting a connection to a resource, the code must take the current state into account. But it's not enough just to check whether the light is green or the connection was successful. To avoid surprises that might cause erratic behavior or even crashes, your code must handle *all* the possible states for a given situation.

Embracing all the possibilities is where the enum comes in. Short for *enumerations*, the enum is a custom data type that enables you to list all the possible states that something can be in. Crucially, once that list is defined, the Rust compiler won't allow you to proceed unless your code explicitly handles every one of those states.

Meet the Enum

Here's the general syntax for defining an enum:

```
enum Name {  
    state1,  
    state2,  
    ...  
    staten,  
}
```

where *Name* is the name of the enum and *state1*, *state2*, and so on specify all the possible states — called the *variants* — you want the enum to model. For example, here's an enum that could be used as part of a traffic light simulation:

```
enum TrafficLight {  
    Red,  
    Yellow,  
    Green,  
}
```



REMEMBER By convention, enum names and states use *PascalCase* (also known as *UpperCamelCase*), where the first letter of each word in the name is capitalized.

This enum is a custom type and when you create an instance of that type, you're specifying that the instance can only ever be in one of the defined states. An instance of the `TrafficLight` enum, for example, can only ever be in one of three states: `Red`, `Yellow`, or `Green`.

Creating an instance of an enum

An enum is a custom type so, like all types, you use it to create an instance of that type. Here's the general syntax for creating an instance of an enum type:

```
let variable = Name::state;
```

where *variable* is the name of the variable that will hold the instance; *Name* is the name of the enum; and *state* is the initial state of the enum, which must be one of its defined states.

For example, the following creates an instance of the `TrafficLight` enum:

```
let light1 = TrafficLight::Red;
```

Here, `light1` is a `TrafficLight` type that starts off in the `Red` state.

Pattern matching enum states

I introduce you to pattern matching with the `match` keyword in Book 1, [Chapter 6](#), but it's when you apply pattern matching to an enum that this technique becomes a superpower. That's because you can set up your `match` structure so that each arm covers a state specified in the enum. No convoluted logic is required. No Hydra-headed `if/else if` monstrosities need clutter your code. Just the blessed straightforwardness of `match` (bk02ch03/example01):

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

fn main() {
    let light1 = TrafficLight::Red;

    match light1 {
        TrafficLight::Red => println!("Stop!"),
        TrafficLight::Yellow => println!("Proceed with caution!"),
        TrafficLight::Green => println!("Go!"),
    }
}
```

Output:

```
Stop!
```

Best of all, the Rust compiler keeps your code on the straight and narrow by enforcing exhaustive handling of all enum states. For example, if you were to leave off the `TrafficLight::Green` arm in your `match` structure, the compiler will call you out on it:

```
non-exhaustive patterns: 'TrafficLight::Green' not covered
```

This is the real superpower: The compiler won't let you forget to handle a case. Similarly, if you were to initialize a variable with a non-existent state such as `TrafficLight::Purple`, the compiler will have none of it:

```
no variant or associated item named 'Purple' found for
enum 'TrafficLight' in the current scope
```

Adding data to an enum

An enum is a collection of possible states, but each of those states can also have data associated with it. For example, suppose you're creating a game and you want to use an enum to track the player's current state:

- » If the player is currently active, you might store the player's name (a `String` value).
- » If the player is currently idle, you might store how many seconds (a `u32` value) they've been inactive.
- » If the player is dead, you might store `true` or `false` (a `bool` value) to indicate whether the player can respawn.

To specify that an enum state can take a value, you put the type in parentheses after the state (`bk02ch03/example02/`):

```
enum PlayerState {
    Active(String), // Stores the player's name
    Idle(u32),      // Stores the idle time, in seconds
    Dead(bool),     // true = can respawn; false = permanently dead
}
```

These states would naturally change as the game progresses, but you can create the instances yourself:

```
fn main() {
    let state1 = PlayerState::Active(String::from("Alice"));
    let state2 = PlayerState::Idle(120);
    let state3 = PlayerState::Dead(true);

    match state1 {
        PlayerState::Active(name) => println!("{name} is playing!"),
        PlayerState::Idle(secs)   => println!("Player idle for {secs}
seconds."),
        PlayerState::Dead(can_respawn) => {
```

```
        if can_respawn {
            println!("Player is dead but can respawn.");
        } else {
            println!("Player is permanently dead.");
        }
    }
}
```

Output:

```
Alice is playing!
```

Option: It's Always Something (or Is It?)

There's an old (some say it's the oldest) philosophical question:

Why is there something rather than nothing?

In other words, why does the universe exist when it seems that it could just as easily have not existed? I'll leave the solution to that conundrum as an exercise for the reader. We're here to talk Rust, and it turns out that Rust has its own version of that philosophical question:

Is there something or is there nothing?

The assumption with the philosophical version is that the universe can only ever be in one of two states: there is something or there is nothing. In Rust, there are many operations that return either something or nothing, and those are the only two states.

If all this talk of states has you thinking about the `enum` type from the preceding section, you're on the right track. Rust defines a special kind of enum called `Option`:

```
enum Option<T> {
    None,
    Some(T),
}
```

where `<T>` is a generic type parameter, which means the `T` part represents any type. (Head over to Book 3, [Chapter 2](#) to get the details on generics in Rust.) `Option` is a type that represents an optional value. It's an enum, so from the preceding definition you know that every `Option` must be one of the following variants:

» **Some:** The `Option` contains a value of type `T`.

» **None:** The `Option` doesn't contain any value.

A concrete example will help at this point. The `String` type has a method called `find()` that searches a string for a specified substring:

```
string.find(substring) -> Option<usize>
```

where `string` is the `String` to search and `substring` is what you're searching for. Crucially, note that the return value for this method is `Option<usize>`, which means the method returns one of the following:

» The byte position (a value of `usize` type) of the first character of the first instance of `substring` that's found in `string`. This value is contained in the `Some` state of the `Option` result. Note that this is the *byte* position, not the character position, which is an important distinction when working with non-ASCII text, where a single character might span multiple bytes. (If that last sentence has your brow fully furrowed, see Book 2, [Chapter 5](#), where I explain everything in more detail.)

» `None` if `substring` was not found in `string`.

Let's put this to work (bk02ch03/example03/):

```
fn main() {
    let question = String::from("Why is there something rather than
nothing?");

    let result1 = question.find("something");
    match result1 {
        Some(num) => println!("Substring found at position {num}!"),
        None => println!("Substring not found!"),
    }
}
```

```

    }

    let result2 = question.find("nothin");
    match result2 {
        Some(num) => println!("Substring found at position {num}!"),
        None => println!("Substring not found!"),
    }
}

```

This code binds a `String` to the `question` variable, and then searches the string using `question.find("something")`, with the returned `Option` bound to `result1`. Then the code uses `match` to process all the possible states of the `Option` enum. In particular, here's the first part of the `match` statement:

```

match result1 {
    Some(num)

```

Here's some pseudocode that tells you how to read this:

```

if result1 contains something
    let num = the extracted value of result1

```

That is, two things are happening:

1. Rust checks `result1` to determine whether it contains a value.
2. If `result1` does contain a value, that value is extracted and bound to the `num` variable.

In the example, the `something` substring *is* found in the original string, so the `Some` arm runs its code and produces the following output:

```

Substring found at position 13!

```

To give the `None` result a whirl, the code next uses `question.find("nothin")` (a typo) with the returned `Option` bound to `result2`. Once again, the code uses `match` to process all the possible states of the `Option` enum. In this case, though, the `nothin` substring is *not* found in the original string and `find()` returns `None`, so the `None` arm of the `match` runs and displays the following:

```

Substring not found!

```



TIP

If you only care about the *existence* of an `Option` value, not the value itself, use the `result.is_some()` method, which returns `true` if the `Option result` contains a value or `false` if it doesn't.

Similarly, if you care about only the *nonexistence* of an `Option` value, use the `result.is_none()` method, which returns `true` if the `Option result` does not contain a value or `false` if it does.

Handling only Some with if let

Wanting to know only whether an `Option` enum contains a value — that is, wanting to process just the `Some` arm of the `match` statement — is a common Rust scenario that often results in code like this:

```
match result {
    Some(value) => // Code to run if result contains a value
    None => {}      // Do nothing
}
```

It's a pain to include a `None` arm that does nothing, so the Rust team took the preceding code and created a pattern-matching shortcut for it:

```
if let Some(num) = result {
    // Code to run if result contains a value
}
```



REMEMBER

The equals sign (`=`) in the `if let` statement works differently than you might expect. It's not the equality operator (`==`), nor is it the assignment operator (`=`). Instead, it's Rust's pattern-matching syntax. Read it as “Try to match the value on the right — `result` — against the pattern on the left — `Some(num)`. If it matches, bind the inner value to `num` and run the block.” So, in other words, `if let Some(num) = result` means that if `result` is the `Some` variant, Rust will extract that value into `num` and execute the code block.

Here's an example (bk02ch03/example04/):

```

fn main() {
    // The goal: sum the numbers in this string
    let code = String::from("398w4x881b");

    // Initialize the sum
    let mut num_sum = 0;

    // Loop through all the characters in the string
    for c in code.chars() {
        // Try to convert the char c to a digit
        let result: Option<u32> = c.to_digit(10);

        // We only care if the Option contains a value
        if let Some(digit) = result {
            num_sum += digit;
        }
    }
    println!("The sum of the digits is {num_sum}");
}

```

This program defines a `String` that contains a mix of numbers and letters; the goal of the program is to sum just the numbers. After initializing `num_sum` to 0, the program sets up a `for` loop to iterate over all the characters in the `code` `String`, which are returned by the `chars()` method. For each character `c` (a `char` type), the loop runs `c.to_digit(10)` to attempt to convert the character to a base-10 digit. The `to_digit()` method returns `Option<u32>`. If the conversion is successful, the converted digit is contained in `Some`; if the conversion was not successful (say, because `c` is a letter), `None` is returned. Since the program cares about only the `Some` result, an `if let` statement looks for that result and, when found, adds the digit to `num_sum`.

Assigning a default value to None

By definition, `None` is an enum variant that has no value. But you'll find yourself in coding situations where it would be convenient for `None` to default to a value, usually because your code requires a value no matter which `Option` variant gets returned.

For example, suppose you want to search for a substring in a larger string and then do one of two things:

- » If the substring was found, return the slice of the original string from the substring to the end.
- » If the substring was not found, return the entire original string.

In Rust, you take a slice of a string as follows:

```
&string[start..end]
```

where *string* is a `String` value, *start* is the byte position in *string* that marks the first character you want to include in the slice, and *end* is the byte position that marks the last character you want to include in the slice. (If you omit *end*, the slice extends to the end of the original string.) Note that string bytes are 0-based, meaning the first byte is 0, the second is 1, and so on.

So, if we once again use the `find()` method to search for a substring, the resulting `Option` will include either the byte position of the substring wrapped in `Some` if the substring was found or `None`.

In this case, defaulting to 0 when the substring isn't found would make sense: If we can't find the substring, we want to return the entire original string, which means starting the slice at position 0 (the beginning). How do we tell Rust to use the value 0 if the `None` variant is returned? By using the `unwrap_or()` `Option` method:

```
option.unwrap_or(default)
```

where *option* is the `Option` result and *default* is a value with a type that matches the `Option` type. The `unwrap_or()` method does one of two things:

- » If *option* matches the `Some` variant, `unwrap_or(default)` returns (*unwraps*) the value contained within it.
- » If *option* matches the `None` variant, `unwrap_or(default)` returns *default*.

Here's an example (bk02ch03/example05/):

```

fn main() {
    // The original string
    let question = String::from("Why is there something rather than
nothing?");

    // Search for the substring
    let result = question.find("something");

    // Use the result to set the start of the slice
    // (0 if the substring wasn't found)
    let start = result.unwrap_or(0);

    // Display the slice
    println!("Slice: {}", &question[start..]);
}

```

In this code, if `find()` locates the substring in the `question` `String`, the resulting byte position is unwrapped from the `Some` variant and used as the `start` value. If `find()` doesn't locate the substring in the `question` `String`, the `None` variant is returned, so the `unwrap_or()` method's default value of `0` is assigned to the `start` variable.

Returning an Option in your own functions

The `Option` enum isn't available only by using standard library methods and functions. You can also incorporate it into your own custom functions. For example, suppose you create a pizza slice calculator that determines the cost per slice given a total cost and the number of slices. That function can return an `Option` that contains the cost per slice if the calculation was successful or returns `None` if the divisor (the number of slices) is `0` (bk02ch03/example06):

```

fn cost_per_slice(total_cost: f64, slices: u32) -> Option<f64> {
    if slices == 0 {
        None
    } else {
        Some(total_cost / slices as f64)
    }
}

fn main() {
    let total_cost = 19.99;
    let slices = 0;
    let result = cost_per_slice(total_cost, slices);
}

```

```
match result {
    Some(cost) => println!("Each slice costs ${:.2}", cost),
    None => println!("No pizza for you!"),
}
}
```

The `cost_per_slice()` function accepts the `total_cost` and `slices` arguments, and returns an `Option<f64>` enum. If the `slices` value is 0, the function returns `None`; otherwise, the function returns `Some(total_cost / slices as f64)`.

Result: Dealing with Errors Gracefully

As your Rust programs become more sophisticated, they'll start to reach out to external sources for data, such as input from the user, a local file, a database, or a remote API. These and similar sources widen the possibilities of what your program can do, but each of them also brings something less welcome — uncertainty:

- » Did the user enter an invalid response?
- » Was the file not found?
- » Did the database query fail?
- » Did the API fail to connect?

These and myriad other negative results can cause your program to produce invalid results or even to crash spectacularly. To prevent your users from having to deal with these consequences, your code needs to anticipate potential errors and handle them.

In Rust, you handle operations that might fail using a special enum called `Result`:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

where `<T>` is a generic type parameter and `<E>` is a generic error value. (Book 3, [Chapter 2](#) has the info on generics in Rust.) `Result` is a type that represents either success or failure. It's an enum, so from the preceding definition you know that every `Result` must be one of the following variants:

- » **ok**: The operation was successful and `Ok(T)` contains a success value of type `T`.
- » **Err**: The operation was not successful and `Err(E)` contains an error value.



REMEMBER This section offers just an introduction to error handling in Rust. Dealing with errors is such an important topic that I devote an entire chapter to it later in the book. To learn more, head over to Book 3, [Chapter 4](#).

Okay, let's bring this down to earth with an example (bk02ch03/example07/):

```
use std::io;

fn main() {
    loop {
        println!("Enter a red component value (0 - 255):");
        let mut choice = String::new();
        let _ = io::stdin().read_line(&mut choice);
        let choice: Result<u8, _> = choice.trim().parse();
        match choice {
            Ok(color) => {
                println!("Your red component value is {color}.");
                break;
            }
            Err(e) => println!("Error: {e}."),
        }
    }
}
```

The purpose of this code is to prompt the user for a numeric value from 0 to 255 to be used as the red component of a color (presumably the green and blue components are known or are gathered elsewhere; who knows?). A `loop` is set up, a prompt displayed, and then the user's input is read into the `choice` variable by using `io::stdin().read_line(&mut choice)`.

Now the code needs to parse the string input into an integer using the `parse()` method, which returns a `Result` enum. Every numeric type has a `parse()` method, so the Rust compiler needs to know in advance which type to use for the conversion. This code needs a `u8` integer, so that type is specified as part of the `Result` type:

```
let choice: Result<u8, _> = choice.trim().parse();
```

The error part of the result is handled with the `_` wildcard character (meaning any error is accepted).

A `match` statement then checks the `Result` variants. Note, in particular, that the `Ok` arm includes a `break` statement to exit the loop. Here's some sample output from this program:

```
Enter a red component value (0 - 255):  
q  
Error: invalid digit found in string  
Enter a red component value (0 - 255):  
256  
Error: number too large to fit in target type  
Enter a red component value (0 - 255):  
200  
Your red component value is 200!
```



TIP If you only care whether an operation was successful and don't require the success value, use the `result.is_ok()` method, which returns `true` if the `result` is the `Ok` variant. Similarly, if you only care whether an operation had an error and don't require the error value, use the `result.is_err()` method, which returns `true` if the `result` is the `Err` variant.

Assigning a default value to Err

Rust offers many ways to handle errors. One of the simplest is to assign a value to the `Err` variant that matches the type of the `Ok` variant. You can assign a default value to the `Err` variant by handling the `Result` enum using the `unwrap_or()` method:

```
result.unwrap_or(default)
```

where `result` is the `Result` and `default` is a value with a type that matches the `Result` type. The `unwrap_or()` method does one of two things:

- » If `result` matches the `Ok` variant, `unwrap_or(default)` returns (that is, it *unwraps*) the success value contained within it.
- » If `result` matches the `Err` variant, `unwrap_or(default)` returns `default`.

Here's an example (bk02ch03/example08/):

```
use std::io;

fn main() {
    println!("Enter a red component value (0 - 255):");
    let mut choice = String::new();
    let _ = io::stdin().read_line(&mut choice);
    let choice: Result<u8, _> = choice.trim().parse();
    let color = choice.unwrap_or(255);
    println!("Your red component value is {color}.");
}
```

In this version of the color component choice code, if the user enters anything other than an integer between 0 and 255, the code assigns the value 255 to the `color` variable.



TIP

An alternative to the `unwrap_or()` method is the `unwrap_or_default()` method, which, when the `Result` is the `Err` variant, returns the default value for the `Result` type: 0 for an

integer; `0.0` for a floating-point type; `false` for a Boolean; and the empty string `""` for a `String`.

Returning a Result in your own functions

Any time you create a function that includes code that might fail in some way, you can return a `Result` enum, where the `Ok` variant contains the function result and the `Err` variant contains an error message.

For example, a greatly simplified banking program might include a struct to model a banking account (bk02ch03/example09/):

```
struct BankAccount {  
    balance: f64,  
}
```

Next you might implement a `withdraw()` method on the bank account:

```
impl BankAccount {  
    fn withdraw(&mut self, amount: f64) -> Result<f64, String> {  
        if amount <= 0.0 {  
            Err(String::from("Withdrawal amount must be positive."))  
        } else if amount > self.balance {  
            Err(String::from("Insufficient funds."))  
        } else {  
            self.balance -= amount;  
            Ok(amount)  
        }  
    }  
  
    fn get_balance(&self) -> f64 {  
        self.balance  
    }  
}
```

The `withdraw()` method returns the type `Result<f64, String>`, which means it returns either a success value of type `f64` (wrapped in `Ok`) or an error message of type `String` (wrapped in `Err`). Inside the function, an `if` conditional checks for two errors: a non-positive withdrawal amount and a withdrawal amount greater than the account balance. In both cases, `Err` is populated with a string describing the error. Otherwise, `Ok` is returned with the withdrawal amount.

The `main()` function puts this struct through its paces:

```
fn main() {
    let mut account = BankAccount { balance: 100.0 };
    println!("Initial balance: ${:.2}", account.get_balance());

    match account.withdraw(200.0) {
        Ok(amount) => println!("Withdrew: ${:.2}", amount),
        Err(error) => println!("Withdrawal failed: {}", error),
    }

    match account.withdraw(50.0) {
        Ok(amount) => println!("Withdrew: ${:.2}", amount),
        Err(error) => println!("Withdrawal failed: {}", error),
    }
}
```

Here's the output:

```
Initial balance: $100.00
Withdrawal failed: Insufficient funds
Withdrew: $50.00
```

Handling only Ok with if let

You'll often find yourself handling `Result` along these lines:

```
match result {
    Ok(value) => // Code to run if result contains a value
    Err(_) => {} // Do nothing
}
```

That is, your code needs to know only whether a `Result` enum contains a value, so it processes only the `Ok` arm of the `match` statement.

Rather than adding that do-nothing `Err` arm every time, you can use the `if let` pattern-matching shortcut instead:

```
if let Ok(value) = result {
    // Code to run if result contains a value
}
```

Here's an example (bk02ch03/example10/):

```
use std::io;

fn main() {
    loop {
        println!("Enter a red component value (0 - 255):");
```



```

        let mut choice = String::new();
        let _ = io::stdin().read_line(&mut choice);
        let choice: Result<u8, _> = choice.trim().parse();
        if let Ok(color) = choice {
            println!("Your red component value is {color}.");
            break;
        }
    }
}

```

Project: Modeling an Authentication Flow

To close this chapter, I present a simple state machine designed to model the simplest possible authentication flow for a user. At a terminal, navigate to the project where you store your Rust projects, and then run the following command:

```
cargo new authentication-flow
```

The program uses stdin to get input from the user, so start with the following:

```
use std::io::{self, Write};
```

The program tracks a user's state using the following enum:

```

enum State {
    LoggedOut,
    EnteringName { name: String },
    LoggedIn { user: String },
}

```

Here's the function that gets the user's input:

```

fn prompt(label: &str) -> Result<String, io::Error> {
    print!("{label}");
    io::stdout().flush()?;
    let mut s = String::new();
    io::stdin().read_line(&mut s)?;
    Ok(s.trim().to_string())
}

```

Note that the error propagation operator (?) is used to catch input/output errors, which use the type `io:Error`, so the function's return type is `Result<String, io:Error>` (where the `String` returned on `Ok` is the command the user entered).

Here's the `main()` function:

```
fn main() {
    let mut state = State::LoggedOut;

    loop {
        match &mut state {
            State::LoggedOut => {
                println!("Commands: login, quit");
                let cmd = prompt("> ").unwrap_or_default();
                match cmd.as_str() {
                    "login" => state = State::EnteringName { name:
String::new() },
                    "quit" => break,
                    _ => println!("Try 'login' or 'quit'."),
                }
            }
            State::EnteringName { name } => {
                if name.is_empty() {
                    *name = prompt("Enter your name: ").unwrap_or_default();
                } else {
                    println!("Welcome, {name}!");
                    state = State::LoggedIn { user: name.clone() };
                }
            }
            State::LoggedIn { user } => {
                println!("You are logged in as {user}.");
                println!("Commands: logout, whoami, quit");
                let cmd = prompt("> ").unwrap_or_default();
                match cmd.as_str() {
                    "logout" => state = State::LoggedOut,
                    "whoami" => println!("You are {user}."),
                    "quit" => break,
                    _ => println!("Try 'logout', 'whoami', or 'quit'."),
                }
            }
        }
    }

    println!("Goodbye!");
}
```

This is a `loop` that uses `match` to process the user's changing state. For each state, the user is presented with a prompt, such as a list of available commands, and then an `unwrap_or_default()` method handles the `Result` returned from the `prompt()` function.

Here's an example session:

```
% cargo run
Commands: login, quit
> login
Enter your name: Furio
Welcome, Furio!
You are logged in as Furio.
Commands: logout, whoami, quit
> whoami
You are Furio.
You are logged in as Furio.
Commands: logout, whoami, quit
> logout
Commands: login, quit
> quit
Goodbye!
```

Chapter 4

Storing Data in Rust

IN THIS CHAPTER

- » Grouping stuff with tuples and arrays
 - » Viewing data with slices
 - » Gathering data into vectors
 - » Storing key-value pairs with hash maps
 - » Exploring prime numbers with vectors
-

Something I think Rust gets right, the community, the book, even the compiler, is that it does believe in you, but it also doesn't expect you to just "get it" overnight.

—JARCANE

There is no such thing as a program that uses no data. Oh, sure, it's technically possible to create a Rust program that runs without data. For example, `fn main() { }` would be such a program. You could even run `fn main() { loop { } }` if you wanted to kill some CPU cycles. So, let me rephrase:

There is no such thing as a program that *does something* that does that thing without data. Even if it's just a `"Hello, world!"` string literal printed to `stdout`, that string is data. Numbers are data. Booleans are data.

Rust programs need data, and it's perhaps because of that need that Rust defines a seemingly limitless number of ways to store data. Earlier in the book I introduce you to several of these ways, including scalar types (integers, floats, Booleans, and characters), string literals, references

(immutable and mutable), structs, and enums. Now, in this chapter, you explore the various ways that Rust offers to group pieces of data together, including the compound types (tuples and arrays) and the basic collection types (vectors and hash maps). It's a veritable feast of data, so let's dig in.

Tuples: Groups of Things

A *tuple* is a data structure that enables you to store multiple values of varying types in a single variable:

```
let variable = (value1, value2, ...);
```

where *variable* is the name of the variable that gets bound to the tuple and *value1*, *value2*, and so on are the values stored in the tuple. You can mix and match any Rust data type, including strings, integers, floats, Booleans, chars, references, structs, enums, other tuples, and the types you learn about in the rest of this chapter: arrays, vectors, and hash maps. The only catches are that once you define a tuple, you can't add or remove items, nor can you change an item's type. Also, tuples are *ordered*, which means that the position of each value matters and is fixed when you create the tuple. (In case you're wondering, yes; if you declare a tuple as mutable, you can make changes to the values inside the tuple, as long as your new values match the existing value types.)

For example, if you're making a game in which you have to track player positions on a 2D plane using (x, y) coordinates (where x is the horizontal position and y is the vertical position), the following conditions are true:

- » A given position has only two values: x and y, which are numeric values.
- » You don't want the type of either x or y to be changed accidentally.
- » You don't want a third value added to the position, nor do you want either x or y to be removed.

- » You do want the capability to change one or both values (for example, when a player's position changes).

You can meet all these conditions by storing a player's current position in a tuple:

```
let mut player_position = (125, 280);
```

Tuple values are indexed, with the first value at index 0, the second at index 1, and so on. To extract a value from a tuple, you use the syntax `tuple.i`, where `tuple` is the name of the tuple and `i` is the index. Here's an example (bk02ch04/example01/ in this book's example files):

```
let mut player_position = (125, 280);
let x = player_position.0;
let y = player_position.1;
println!("The player is at position ({x}, {y}).");
```

Output:

```
The player is at position (125, 280).
```



REMEMBER A special tuple called the *unit tuple* — or sometimes the *empty tuple* — contains no values: `()`. That sounds quite useless, but Rust uses the unit tuple under the hood quite a bit. For example, if a function or an expression has no explicit return value, technically it's returning `()` behind the scenes. Cool fact: `()` is the default return value of the `main()` function!

Why tuples are useful

Tuples seem awfully limited, but they're useful when you need to group a small number of related values. A tuple is the go-to type for coordinates, RGB color values, or whenever you need to return multiple values from a function. Here's an example (bk02ch04/example02/):

```
fn angle_to_point(angle: i16) -> (f64, f64) {
    // Convert angle to radians
    let radians = (angle as f64).to_radians();
```

```

    // Return the point
    (radians.cos(), radians.sin())
}

fn main() {
    let angle = 90;

    // Convert the angle to a point (x, y)
    let point = angle_to_point(angle);

    println!(
        "The angle {angle}° corresponds to the point ({:.1}, {:.1}).",
        point.0, point.1
    );
}

```

Output:

```
The angle 90° corresponds to the point (0.0, 1.0).
```

Just to be clear, when you want to return a tuple from a function, the return type is `(<T>, <T>, ...)`, where `<T>` is the type of each value in the returned tuple (and the number of `<T>`s is the number of elements in the tuple).

Tuple destructuring

Rust supports tuple *destructuring*, where you assign variables to each item in the tuple in one fell swoop. That is, rather than this:

```

let x = player_position.0;
let y = player_position.1;

```

You can do this:

```
let (x, y) = player_position;
```

Here's a slightly more involved example (bk02ch04/example03/):

```

fn main() {
    // Here comes the tuple
    let person = ("Alice", 30, true);

    // Destructure it into separate variables
    let (name, age, is_student) = person;
    println!("Name: {name}, Age: {age}, Is student: {is_student}");

    // While destructuring, you can also ignore values you don't need

```

```
let (name, _, _) = person;
println!("Hello {name}!");
}
```

Output:

```
Name: Alice, Age: 30, Is student: true
Hello Alice!
```

Note, in particular, that if you want to skip a value while destructuring, specify `_` instead of a variable name.

Tuple limitations

Tuples will be an oft-used part of your Rust toolkit, but they do have some important limitations that you need to consider when deciding whether to use them in your code:

- » **No named access:** Indexes (0, 1, 2, and so on) aren't intuitive, so without named fields, you have to remember that, say, age is in position 1, not position 2.
- » **Fixed size:** Once you create a tuple, you can't add or remove elements.
- » **Confusion is just around the corner:** A small tuple with just two values (such as a point) or three values (such as an RGB color) shouldn't tax your brain, but once the number of elements grows beyond these small sizes, tuples can become total head-scratchers:

```
fn main() {
    // This is getting unwieldy
    let student_record = ("Alice", 30, true, 3.8, "Computer Science",
2026);

    // Skip a few hundred lines of code
    // Six months later, you're staring at this code thinking:
    // "What the heck is student_record.4 supposed to be?"
    println!("Report: {}", student_record.4);
}
```

Arrays: Fixed-Sized Storage

A variable usually stores a single item of data, such as a grade, a day of the week, or an ice cream flavor. But your code will commonly work with two or more (usually many more) items of related data, such as the grades for an entire class, every day of the week, or the names of all the ice cream flavors that a store sells.

Sure, you *could* create a separate variable for each item, but Rust offers a faster, simpler, *and* more powerful way: the array. An *array* is a collection of items that resides in a single container. I know: That sounds just like a tuple! But arrays come with a few extra bells and even a few added whistles that make arrays more convenient and more powerful in certain situations.

Making an array

You construct an array by starting with a pair of square brackets [and], and then filling those brackets with the values you want to store, separated by commas. An array can hold any data type that has a fixed length known at compile time, including integers, floats, Booleans, chars, string literals, strings, references, tuples, structs, enums, and even other arrays.

The key, though, is that an array can store data of only a single type. No mixing and matching of types as with a tuple:

```
// This is a valid array because every element is the same type (&str)
let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

// This is an invalid array because it mixes different types
let person = ["Alice", 30, true];
```

The number of items in the array is part of its type. For example, in the preceding code, the `days` array consists of 7 items of type `&str`, so the data type of that array is `[&str; 7]`. (If the type `&str` has you flummoxed, Book 2, [Chapter 5](#) explains all.) Because the compiler knows the type and number of elements in advance, arrays are stable and fast. Rust can lay them out in memory with precision, typically on the stack.

Another way to make an array is to use the *repeat expression* syntax:

```
let array = [value; number];
```

where *array* is the name of the array, *value* is a fixed-length value, and *number* is how many elements of *value* you want in the array. For example, the following two declarations are equivalent:

```
let zeros = [0; 6];  
let zeros = [0, 0, 0, 0, 0, 0];
```

Getting an item from an array

As with tuples, array items are numbered according to their position in the array, and that number is called the *index*. The first item in the array has index 0, the second item has index 1, and so on.

You can grab something from an array by specifying the item's index number in square brackets after the array name. Here's an example (bk02ch04/example04/):

```
fn main() {  
    let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];  
    let first_day = days[0];  
    println!("{}", first_day);  
}
```

Output:

```
Mon
```

Array bounds checking: Rust has your back

One of Rust's safety features is that it checks for an invalid array index. If your code tries to access an array element that doesn't exist, Rust simply won't permit it because doing so would mean allowing your code to access random memory, which would open up huge security holes:

```
let numbers = [1, 2, 3];  
  
// This is fine  
println!("{}", numbers[1]); // Prints 2  
  
// This is a problem  
println!("{}", numbers[10]);
```

The compiler offers the following admonition when it trips over that last line:

```
println!("{}", numbers[10]); // Index out of bounds!  
      ^^^^^^^^^^^ index out of bounds: the  
                  length is 3 but the index is 10
```

Thanks, Rust!

Iterating through an array

One of the nice features you get with an array that's not available with a tuple is that you can loop — or *iterate*, to those in the know — through all the elements in an array. Here's the general syntax:

```
for element in array {  
    // Do something with each element  
}
```

where *array* is the name of the array you're working with and *element* is a local variable that takes on the next array value during each loop. That is, during the first loop, *element* is set to *array*[0], during the second loop *element* is *array*[1], and so on. The loop ends when it has gone through all the elements in *array*.

Here's an example (bk02ch04/example05/):

```
fn main() {  
    let berries = [  
        "strawberry",  
        "blueberry",  
        "blackberry",  
        "raspberry",  
        "gooseberry",  
    ];  
    for berry in berries {  
        println!("Mmm, the {berry} is delicious!");  
    }  
}
```

Output:

```
Mmm, the strawberry is delicious!  
Mmm, the blueberry is delicious!  
Mmm, the blackberry is delicious!
```

```
Mmm, the raspberry is delicious!
Mmm, the gooseberry is delicious!
```



TIP If your code needs to use the number of elements in an array, use `array.len()`, where `array` is the name of the array. For example, in the preceding code, `berries.len()` returns 5.

Array types and functions

One of the little gotchas to watch for with arrays is that they're not a single data type. Consider the following two arrays:

```
let array1: [i32; 4] = [1, 2, 3, 4];
let array2: [i32; 5] = [1, 2, 3, 4, 5];
```

As shown in the type annotations, `array1` is `[i32; 4]` and `array2` is `[i32; 5]`. They're both arrays of `i32` integers, but to Rust these types are as different as `f64` and `bool`! This difference means you can't hand, say, an `[i32; 5]` to a function that expects an `[i32; 4]` ([bk02ch04/example06/](#)):

```
fn sum_four(nums: [i32; 4]) -> i32 {
    nums[0] + nums[1] + nums[2] + nums[3]
}

fn main() {
    let array1: [i32; 4] = [1, 2, 3, 4];
    let array2: [i32; 5] = [1, 2, 3, 4, 5];

    // This works fine:
    println!("{}", sum_four(array1));

    // Error: expected an array with a size of 4, found one with a size of 5
    println!("{}", sum_four(array2));
}
```

The workaround is to use an array slice as the function argument. I show you how to do that when I introduce you to slices in the upcoming [“Slices: Versions of Your Stuff”](#) section.

You also use an array's data type when you have a function that returns an array. For example, if the function returns a three-element array of

`f32` floating-point numbers, the function's return type is `[f32; 3]`. For example, a common operation when working with colors is to take some RGB values in the range 0 to 255 and *normalize* them to values in the range 0 to 1. Here's an example program that performs this normalization and returns the normalized values as an array (bk02ch04/example07/):

```
fn normalize_rgb(r: u8, g: u8, b: u8) -> [f32; 3] {
    [
        r as f32 / 255.0,
        g as f32 / 255.0,
        b as f32 / 255.0,
    ]
}

fn main() {
    let rgb = normalize_rgb(221, 160, 221); // Plum
    println!("Normalized [0-1] RGB values: {:.2?}", rgb);
}
```

Output:

```
Normalized [0-1] RGB values: [0.87, 0.63, 0.87]
```



REMEMBER In the `println!` macro's display format — `{:.2?}` — you might have figured out that the `.2` part tells Rust to display the values to two decimal places, but what's up with the `?` that I slipped in there? If you try running this code without `?`, the compiler will complain that `'[f32; 3]'` cannot be formatted with the default formatter. Some formats are like that. (Technically, they don't implement the `Display` trait.) You get around this problem by adding `?`, which specifies a debug output that works for almost all types. (Technically, they implement the `Debug` trait. I fill in the many missing details in this explanation when I talk about traits in Book 3, [Chapter 1](#).)

When to use an array (and when not to)

You'll want to pull out an array when you come across data that meets the following criteria:

- » The data consists of a fixed number of elements and that number won't change.
- » The size of every element is known at compile time.
- » The number of elements is relatively small.

If your data meets these requirements, an array will give you the fastest possible access to the elements.

But an array is the wrong choice when your data requires any of the following:

- » You need to add or remove elements.
- » The size of the elements depends on user input, external data, or some other dynamic runtime consideration.
- » The number of elements is very large.

When your data checks any of these boxes, you'll want to consider a vector, which is the subject of the "[Vectors: Dynamic Arrays That Grow](#)" section, coming along real soon.

Slices: Versions of Your Stuff

When you're working with a collection in Rust, particularly the arrays of the preceding section and the vectors of the next one, you'll often find it convenient to work with just a subset of that collection. In an array that contains the days of the week, for example, you might want to work with only the weekdays. In Rust, a subset of a collection is called a *slice*, and it's one of the most useful and common Rust patterns.

The slice type is generally written using square brackets: `[T]` (immutable) or `&mut [T]` (mutable), where `T` is the type of the data from which the slice comes. A slice is a *view* into a collection, by which I mean it's a borrowed portion of the original, not owned. So, a slice

enables you to use part of an array or vector (say, by passing it to a function) without copying it or taking ownership of it.

Creating a slice

Here's the general syntax for a slice:

```
let slice = &original[start..end];
```

where *slice* is the name of the slice, *&original* is a reference to the collection from which the slice is taken, *start* is the index of the first element of *original* to include in the slice, and *end* is the index of the element that is one more than the last element to include in the slice.

The *end* part is tricky at first because, by default, the element at position *end* is *not* included in the slice (bk02ch04/example08/):

```
fn main() {  
    let array = [0, 1, 2, 3, 4, 5];  
    let slice = &array[0..3];  
    println!("{:?}", slice);  
}
```

Output:

```
[0, 1, 2]
```

Besides the standard `[start..end]` slice syntax, there are four variations that you'll use often in your code:

- » `[start..]`: The slice includes everything from *start* through to the last element of the original collection (inclusive, because there's no explicit end to exclude).

```
let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];  
let weekend = &days[5..] // ["Sat", "Sun"]
```

- » `[..end]`: The slice includes everything from the first element of the original collection up to but not including *end*.

```
let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];  
let weekdays = &days[..5]; // ["Mon", "Tue", "Wed", "Thu", "Fri"]
```

- » `[start..=end]`: The slice includes everything from *start* up to and *including* *end*.

```
let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
let long_weekend = &days[4..=6]; // ["Fri", "Sat", "Sun"]
```

» `[..]`: The slice includes everything from the original collection.

```
let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
let days2 = &days[..]; // ["Mon", "Tue", "Wed", "Thu", "Fri",
"Sat", "Sun"];
```

Modifying data with mutable slices

If the original data is mutable, you can take a mutable slice of that data and modify the slice's elements. Because a slice is a *reference* to the original data (not a copy), any changes you make through the slice are changing the original data. This is useful when you just want to make changes to some subset of the original data. Rather than figuring out some tricky way to apply your change to the original data subset directly, a mutable slice makes it easy (bk02ch04/example09/):

```
fn zero_out(nums: &mut [i32]) {
    for x in nums {
        // We need to dereference here!
        *x = 0;
    }
}

fn main() {
    // Declare a mutable array
    let mut arr = [12, 22, 31, 69, 82, 91, 5, 44, 50, 79];

    // Send a mutable slice to the function
    zero_out(&mut arr[..5]);
    // Check out the values in arr now
    println!("{:?}", arr);
}
```

Output:

```
[0, 0, 0, 0, 0, 91, 5, 44, 50, 79]
```

Wondering about that cryptic comment regarding dereferencing in the `zero_out()` function? The next section explains all.

Dereferencing with a mutable slice

In Book 2, [Chapter 1](#), I talk briefly about dereferencing, which extracts the value that a reference points to in memory. For an immutable reference, dereferencing happens automatically:

```
let s1 = String::from("Inflatable Dartboard");
let s2 = &s1;
println!("The value of s2 is {s2}.");
// Output:
The value of s2 is Inflatable Dartboard.
```

Here, `s2` is an immutable borrow of `s1`. When the code uses `s2` in the `println!` macro, Rust automatically dereferences `s2` to get the `String` value from memory.

Rust automatically dereferences immutable borrows because no harm can come from that read-only view into the data. But when you have a mutable reference — such as a mutable slice — you have a much more dangerous weapon in your hands, so Rust does not automatically dereference the borrow. Instead, it forces you to declare your intentions by explicitly dereferencing the borrow using the dereference operator (`*`). When Rust comes upon `*` just before a reference, it follows the reference to access the value that it points to:

```
// Mutable value
let mut hue = 30;

// Mutable reference
let complement = &mut hue;

// Gotta dereference explicitly to make the change
*complement += 180;
println!("{hue}"); // Output: 210
```

Here, `*complement` dereferences the reference `&mut i32` into a `mut i32` so that it's now legal to add the value. The same process takes place in the `zero_out()` function from the preceding section:

```
fn zero_out(nums: &mut [i32]) {
    for x in nums {
        // We need to dereference here!
        *x = 0;
    }
}
```

`*x` dereferences the `&mut i32` reference into a `mut i32` value to make the change.

Using slices to make functions more flexible

When a data type, such as an array, has a fixed length, using that data type as a function argument type leads to code inflexibility:

```
fn sum(nums: [i32; 4]) -> i32 {
    let mut total = 0;
    for n in nums {
        total += n;
    }
    total
}
```

Thanks to the `[i32; 4]` argument type, this function works only if you pass it a four-element array of `i32` values. Try it with an array that has two or five values, and the compiler will gently admonish you.

To make this function more flexible, change the array type (`[i32; 4]`) to an array slice type (`&[i32]`), which enables the function to accept an array that contains any number of `i32` values (bk02ch04/example10/):

```
fn sum(nums: &[i32]) -> i32 {
    let mut total = 0;
    for n in nums {
        total += n;
    }
    total
}

fn main() {
    let a = [1, 2, 3];
    let b = [1, 2, 3, 4];
    println!("{}", sum(&a)); // Output: 6
    println!("{}", sum(&b)); // Output: 10
}
```



REMEMBER The `sum()` function is now so flexible that it will accept not only any size array of `i32` values, but also any size slice of `i32` values

and any size vector of `i32` values. (I cover vectors in the next section.)

Vectors: Dynamic Arrays That Grow

If you asked Rust developers to vote for an MVT (most valuable type), I'd be willing to bet that the vector would top that poll. A *vector* is a collection of items that resides in a single container. I know: That sounds just like a tuple and an array! But vectors are much more powerful than either tuples or arrays because they are flexible, are dynamic, and can handle the common case where you need to store data but you don't know in advance how much.

Creating a vector

One of the ways that Rust signals the importance of a type is by offering a long list of ways to create that type in your code. This section takes you through the main ways you can create a vector in Rust.

Creating a new vector

The official type designation for a vector is `Vec<T>`, where `T` represents the type of each element in the vector. Yep, just like arrays, a vector can only hold elements of a single type.

To create a new, empty vector, you use the following syntax:

```
let mut vector: Vec<T> = Vec::new();
```

where `vector` is the name of the variable and `T` is the type of the elements you'll eventually add to this vector (refer to the “[Adding and removing elements](#)” section for details). The fact that you'll add elements to this vector later is why it's declared as mutable.

For example, the following declares a new, empty vector that can store integers of type `i32`:

```
let mut my_new_vector: Vec<i32> = Vec::new();
```

Creating a vector from an array

If you have an existing array and need a dynamic version, you can create a vector from that array by using two methods. The method you choose depends on the type of the array data and whether you still need to use the array after creating the vector.

The first route you can take is to use the `Vec` type's `from()` method:

```
let vector = Vec::from(array);
```

where `vector` is the name of the variable and `array` is the existing array that you want to move into the vector. Here's how `from()` works:

» **If the array consists of a type that implements the `Copy` trait:** For values such as integers, floats, or string literals, Rust copies the array data into the vector. The array is still valid after the vector has been created:

```
let rgb_array = [128, 0, 128]; // Purple
let rgb_vector = Vec::from(rgb_array);
println!("{:?}", rgb_array); // This works!
```

» **If the array consists of a type that does *not* implement the `Copy` trait:** For a type such as `String`, Rust moves the array data into the vector. The array is invalid after the creation of the vector:

```
let string_array = [String::from("this"), String::from("that")];
let string_vector = Vec::from(string_array);
println!("{:?}", string_array); // This doesn't work!
```

(I first mention the `Copy` trait in Book 2, [Chapter 1](#). For the full treatment, check out Book 3, [Chapter 1](#).)

If you know for sure that you'll need the array after creating the vector, consider using the array type's `to_vec()` method, which always clones the array data into the vector, so the array remains valid:

```
let vector = array.to_vec();
```

where `vector` is the name of the variable and `array` is the existing array that you want to copy into the vector:

```
let string_array = [String::from("this"), String::from("that")];
let string_vector = string_array.to_vec();
```

```
println!("{:?}", string_array); // This works!
```

Since `to_vec()` always clones the array, the original is still available and can be used in the rest of your code.

Creating a vector using the `vec!` macro

Rust offers a macro named `vec!` that greatly simplifies the creation of a macro when you already know which values you want to populate it with. (To learn about Rust macros, refer to Book 3, [Chapter 5](#).) Here's the syntax:

```
let vector = vec![elements];
```

where *vector* is the name of the variable and *elements* is one of the following:

- » A comma-separated list of the values you want to use to populate the vector:

```
let primes = vec![2, 3, 5, 7, 11, 13, 17, 19];
```

- » A repeat expression of the form `[expr; N]`, where *expr* evaluates to a value and *N* is the number of elements in the vector, each of which is assigned the *expr* value:

```
let init = vec![true; 5]; // [true, true, true, true, true]
```

Accessing vector elements (a story about risk)

Rust gives you two ways to access vector elements.

The first way is to use the *indexing operator* `vector[index]`, where *vector* is the name of the vector, *index* is the position of the element, and 0 is the first element, 1 is the second element, and so on in *index*.

For example (bk02ch04/example11/):

```
let test_scores = vec![85, 77, 62, 91, 48, 80, 71, 66, 75, 53];
println!("Last score: {}", test_scores[9]);
// Output: Last score: 53
```

The second way is to use the `vector.get()` method, where *vector* is the name of the vector. The `get()` method returns an `Option` (refer to Book

2, [Chapter 3](#)):

```
let test_scores = vec![85, 77, 62, 91, 48, 80, 71, 66, 75, 53];
match test_scores.get(9) {
    Some(score) => println!("Last score: {}", score),
    None => println!("Score not found!"),
}
```

If the index value is within bounds, `get()` returns `Some` with the vector value wrapped inside; otherwise, `get()` returns `None` so that your program can handle the bad index instead of crashing (or *panicking*, in the Rust vernacular). Remember, too, that if you don't need to handle the `None` result, you can make your code a bit cleaner by using `if let`:

```
if let Some(score) = test_scores.get(9) {
    println!("Last score: {}", score);
}
```

Obviously the indexing operator method is much simpler and easier to read, so why would you ever use the `get()` method with all its extra `Option` baggage? It's all about risk. That is, using the indexing operator is generally considered risky because if the index is out of bounds, your program will crash.

So, does that mean you should always use `get()`, just to be safe? Some programmers do that at the beginning of a project, and then refactor to use `vector[index]` in places where testing has shown that it's safe to do so. For example, in certain tightly controlled loops or well-defined algorithms, it's just not possible for the index to go out of bounds, so it's overkill to use `get()`. But when the index or vector comes from user data, a file, or some other external source, you can't guarantee that a vector access will never go out of bounds, so you should always use `get()` in those situations.

Adding and removing elements

One of the big-time advantages you get with a vector over an array is that vectors are growable types, meaning you can add and remove elements.

To add an element to a mutable vector, you have two choices (bk02ch04/example12/):

» **Add an element to the end of the vector.** Use the `v.push(value)` method to add `value` to the end of the vector `v`:

```
let mut css_colors = vec!["bisque", "lemonchiffon"];
css_colors.push("papayawhip");
println!("{:?}", css_colors);
// Output:
// ["bisque", "lemonchiffon", "papayawhip"]
```

» **Add an element at a specified position within the vector.** Use the `v.insert(value, index)` method to add `value` at position `index` in vector `v`:

```
css_colors.insert(1, "honeydew");
println!("{:?}", css_colors);
// Output:
// ["bisque", "honeydew", "lemonchiffon", "papayawhip"]
```

To remove an element from a mutable vector, you also have two choices (bk02ch04/example12/, continued):

» **Remove the last element of the vector.** Use the `v.pop()` method on the vector `v`. Since the array might be empty, `pop()` might return nothing, so `pop()` returns an `Option`, where the `Some` result contains the value of the last element if it exists and the `None` result applies if the vector was empty:

```
if let Some(color) = css_colors.pop() {
    println!("Removed {color}!");
    println!("{:?}", css_colors);
}
// Output:
// Removed papayawhip!
// ["bisque", "honeydew", "lemonchiffon"]
```



TIP

You can check in advance whether a vector contains no elements by running `v.is_empty()`, where `v` is the name of the

vector. This method returns `true` if the vector has no elements, or `false` if it does.

» **Remove an element from a specified position within the vector.**

Use the `v.remove(index)` method to remove the element at position `index` in vector `v`:

```
css_colors.remove(1);
println!("{:?}", css_colors);
// Output:
// ["bisque", "lemonchiffon"]
```



TIP

If you want to keep the first n elements of the vector and remove all the rest, use `v.truncate(n)`, where `v` is the name of the vector.



TIP

If you want to remove every element from a vector in one shot, use `v.clear()`, where `v` is the name of the vector.

Iterating through a vector

When you need to perform some operation over the elements of a vector, you can iterate over it using the following general syntax:

```
for element in &vector {
    // Do something with each element
}
```

where `vector` is the name of the vector and `element` is a local variable that takes on the next vector value during each loop. Note the `&` in front of the vector name. In most cases, you want the vector to retain ownership of its values so that you can still use the vector after the loop; therefore, the loop just borrows the vector. If you want to make changes to the vector value, then you'd borrow the vector mutably for the loop:

```
for element in &mut vector {
    // Modify each element
}
```


Here's an example (bk02ch04/example13/):

```
fn main() {
    let mut test_scores = vec![85, 77, 62, 91, 48, 80, 71, 66, 75, 53];

    // Calculate the average test score
    let mut sum_of_scores = 0;
    for score in &test_scores {
        sum_of_scores += score;
    }
    let test_average = sum_of_scores as f64 / test_scores.len() as f64;
    println!("Average test score: {test_average}.");

    // Good behavior bonus: 5 extra points!
    for score in &mut test_scores {
        // Note the dereference!
        *score += 5;
    }
    println!("{:?}", test_scores)
}
```

Output:

```
Average test score: 70.8.
[90, 82, 67, 96, 53, 85, 76, 71, 80, 58]
```



TIP

If your code needs to use the number of elements in a vector, use `v.len()`, where `v` is the name of the vector. For example, in the preceding code, `test_scores.len()` returns 10.

Hash Maps: Key-Value Storage Made Easy

A *hash map* is a data structure that enables you to store multiple items in a single variable. Yes, I know, that sounds just like a tuple, an array, and a vector. However, in a hash map, each item is a pair: the value you want to store and a label that describes the value. That label is called a *key*, so these pairs are known as *key-value pairs*.

That label is important because it can make your code more readable. For example, it's not at all clear what the following refers to:

```
person[0]
```

But even without looking at the rest of the code, it's pretty obvious what the following means:

```
person["age"]
```

Creating a hash map

When you're planning on using a hash map in your code, note that Rust doesn't automatically include the `HashMap` data type in the standard library's prelude. (Refer to Book 1, [Chapter 3](#) to learn about the prelude.) To bring that data type into the scope of your project, add the following at the top of your file:

```
use::std::collections::HashMap;
```

The official type designation for a hash map is `HashMap<K, V>`:

- » **k**: The type of every key in the hash map. You can use most of the standard types, such as `String`, `&str`, integers, and so on), but you can't mix them. Every key must use the same type.
- » **v**: The type of every value in the hash map. You can use any type you want but every value must use the same type.

With that done and said, you can now create a hash map using either of the techniques I present in the next two sections.

Creating a shiny, new hash map

To create a new, empty hash map, you use the following syntax:

```
let mut hash_map: HashMap<K, V> = HashMap::new();
```

where `hash_map` is the name of the variable, `K` is the type used by the keys, and `V` is the type used by the values. The fact that you'll later add elements to this hash map is why it's declared as mutable.

For example, the following declares a new, empty hash map that can store key-value pairs with keys of type `String` and values of type `i32`:

```
let mut my_new_hash_map: HashMap<String, i32> = HashMap::new();
```

Creating a hash map from an array of tuples

If you have an existing array of tuples of the form (K, V) , where the K elements would work as keys and the V elements would work as values, you can create a hash map from that array using the `from()` method:

```
let hash_map: HashMap<K, V> = HashMap::from(array);
```

where *hash_map* is the name of the variable, K is the type used by the keys, V is the type used by the values, and *array* is the existing array that you want to use to populate the hash map.

For example, the following declares an array of test score tuples, and then converts that array to a hash map (bk02ch04/example14/):

```
let score_array = [("Alphonse", 77), ("Biff", 61), ("Carmine", 92)];
let score_hashmap: HashMap<&str, i32> = HashMap::from(score_array);
println!("{}", score_hashmap);
```

Output:

```
{
  "Alphonse": 77,
  "Biff": 61,
  "Carmine": 92,
}
```



REMEMBER Note that the `println!` macro's display format — `{:#?}` — includes a hashtag (`#`). That's the pretty print symbol, and it produces the nice output just shown instead of the standard output for a hash map, which is to put all the key-value pairs on a single line.



WARNING In the array of tuples, if the key type or the value type or both don't implement the `Copy` trait, the `from()` method moves the array data into the hash map. That move means the original array is dropped and you can't use it anymore. If both the key type and the value type implement the `Copy` trait, the array data is copied to the hash map and the array remains valid.

Accessing hash map values

Rust gives you two ways to access hash map values. The first way is to access a key directly:

```
hash_map[key]
```

where `hash_map` is the name of the hash map and `key` is the key associated with the value you want to access. For example (bk02ch04/example15/):

```
fn main() {
    let score_array = [("Alphonse", 77), ("Biff", 61), ("Carmin", 92)];
    let score_hashmap: HashMap<&str, i32> = HashMap::from(score_array);

    let biff_score = score_hashmap["Biff"];
    println!("Biff's score is {biff_score}.");
}
// Output: Biff's score is 61.
```

This method is fine for quick-and-dirty code and when you're certain the key exists. However, it's often the case that either the key or the hash map itself comes from an external source, such as a file or database. In those cases, you simply can't be sure if the key exists, and if your code tries to access a non-existent key, Rust panics and shuts down the program in a fiery explosion.

When there's no certainty that the key you want exists, use the second possibility for accessing a hash map value: the `hash_map.get()` method, where `hash_map` is the hash map. The `get()` method returns an `Option` (refer to Book 2, [Chapter 3](#)), as shown in this example (bk02ch04/example16/):

```
fn main() {
    let score_array = [("Alphonse", 77), ("Biff", 61), ("Carmin", 92)];
    let score_hashmap: HashMap<&str, i32> = HashMap::from(score_array);

    let result = score_hashmap.get("Biff");
    match result {
        Some(biff_score) => println!("Biff's score is {biff_score}.",),
        None => println!("The score wasn't found!"),
    }
}
// Output: Biff's score is 61.
```

If the key exists, `get()` returns `Some` with the key's value wrapped inside; otherwise, `get()` returns `None` so that your program can handle the missing key instead of panicking.

Adding and removing key-value pairs

If you declare your hash map as mutable, you can add and remove key-value pairs.

To add a key-value pair, use the `hash_map.insert(key, value)` method, where `hash_map` is the hash map, `key` is the key, and `value` is the value (bk02ch04/example17/):

```
fn main() {
    let mut words: HashMap<&str, &str> = HashMap::new();
    words.insert("lipthinking", "Thinking out loud.");
    words.insert("koumpounophobia", "The fear of buttons.");
    words.insert("grammo", "A grammatical error.");
    words.insert("wasband", "A woman's ex.");
    println!("{:?}", words);

    words.insert("wasband", "A woman's ex-husband.");
    println!("{:?}", words);
}
```

Output:

```
{
  "lipthinking": "Thinking out loud.",
  "wasband": "A woman's ex.",
  "grammo": "A grammatical error.",
  "koumpounophobia": "The fear of buttons.",
}
{
```

```
"lipthinking": "Thinking out loud.",  
"wasband": "A woman's ex-husband.",  
"grammo": "A grammatical error.",  
"koumpounophobia": "The fear of buttons.",  
}
```

Note that the code performs a second `insert()` on the "wasband" key, but with a different value. In this case, Rust doesn't insert the key all over again. Instead, it just updates the key to the new value.



TIP Interestingly, if you run this code multiple times, the order of the key-value pairs will change with each run! This apparent bug is actually a feature: Hash maps are stored unordered, which enables them to access data a bit faster. If you want your data stored in key order, use a B-Tree map, instead. In this section, you can replace all instances of the `HashMap` type with the `BTreeMap` type to get an ordered map.

To remove an element from a mutable hash map, use the `hash_map.remove(key)` method, where `hash_map` is the hash map and `key` is the key you want to delete. Since the key might not exist, `remove()` returns an `Option`, where the `Some` result contains the value of the removed key if it exists and the `None` result applies if the key doesn't exist (bk02ch04/example18/):

```
let result = words.remove("wasband");  
match result {  
    Some(def) => println!("Key removed!"),  
    None => println!("Key not found!"),  
}
```

Iterating through a hash map

When you need to perform some operation over the elements of a hash map, you can iterate over the key-value pairs, just the keys, or just the values using the following general syntaxes:

```
for (key, value) in hash_map {  
    // Do something with each key-value pair
```

```

}
for key in hash_map.keys() {
    // Do something with each key
}
for value in hash_map.values() {
    // Do something with each value
}

```

where *hash map* is the name of the hash map and *key* and *value* are local variables that take on the next hash map key or value during each loop. Here's an example (bk02ch04/example19/):

```

fn main() {
    let mut words: HashMap<&str, &str> = HashMap::new();
    words.insert("lipthinking", "Thinking out loud.");
    words.insert("koumpounophobia", "The fear of buttons.");
    words.insert("grammo", "A grammatical error.");
    words.insert("wasband", "A woman's ex.");

    for (key, value) in words {
        println!("{key} means '{value}'");
    }
}

```

Output:

```

lipthinking means 'Thinking out loud.'
koumpounophobia means 'The fear of buttons.'
wasband means 'A woman's ex.'
grammo means 'A grammatical error.'

```

Project: The Prime Explorer

To close this chapter, this section takes you through part of a project to explore prime numbers. Why only part? This chapter is quite long, so I leave most of the code for you to study on your own as an exercise. Here I'll just focus on the code that uses vectors to implement a famous mathematical algorithm called the Sieve of Eratosthenes. This algorithm determines all the prime numbers up to a specified limit by systematically eliminating (sieving out) the multiples of each prime number, leaving only the gold of the primes behind.

To get started, use a terminal to navigate to your Rust projects folder, and then run the following command:

```
cargo new prime-explorer
```

Load the new project's `main.rs` into your text editor and add the program code (bk02ch04/prime-explorer/). Here's the function that, given a limit (such as 10,000), returns a vector of all the prime numbers up to that limit:

```
fn sieve_primes(limit: usize) -> Vec<usize> {
    // Start by assuming every number is a prime
    // is_prime is a vector with limit elements, all of which start as true
    let mut is_prime = vec![true; limit + 1];

    // 0 and 1 are not prime
    is_prime[0] = false;
    if limit >= 1 { is_prime[1] = false; }

    // Mark off prime multiples starting at p*p and stepping by p
    // For each multiple m, is_prime[m] is set to false (not prime)
    let mut p = 2;
    while p * p <= limit {
        if is_prime[p] {
            let mut m = p * p;
            while m <= limit {
                is_prime[m] = false;
                m += p;
            }
        }
        p += 1;
    }

    // At this point, is_prime[i] is true for every i that's prime
    // So, now we collect those prime indices into a separate vector
    let mut primes = Vec::new();
    let mut i = 2;
    while i <= limit {
        if is_prime[i] {
            primes.push(i);
        }
        i += 1;
    }
    // Done
    primes
}
```


The key here is to first assume that every number up to `limit` is prime by setting `is_prime` to `vec![true; limit + 1]`. Then a `while` loop runs through all the multiples of each prime number, marking those multiples as `false` in the `is_prime` vector. When the loop completes, `is_prime[i]` will be `true` for every `i` that's prime, so the code finishes by pushing each of the prime `i` indices into the `primes` array.

Chapter 5

The String Section: Text in Rust

IN THIS CHAPTER

- » Making sense of UTF-8 text
 - » Understanding string literals versus owned strings
 - » Creating and manipulating strings
 - » Slugifying a string
-

Writing Rust for me is a gradual process of the compiler patiently guiding me towards the program I should have written in the first place, and at the end I take all the credit.

—FELIX WATTS

When you're just getting started with programming, bits of text — or *strings* as the coders prefer to say — seem refreshingly straightforward. Numbers can be hard, Booleans can take a minute to get your head around, and collections such as vectors and hash maps have a lot going on. But text? A few characters between quotation marks, right?

Yes, in programming languages such as JavaScript and even Python. Alas, there's no such straightforwardness when it comes to text in Rust. But Rust's treatment of text, which can seem arbitrary, even cruel, at first sight, is all in the service of type safety, global compatibility, and ensuring that even mere bits of text pass muster with the borrow checker.

In this chapter, you not only explore Rust's impressive set of text tools but also, I hope, come to understand and appreciate why Rust treats text the way it does.

String Instruments: String and &str

In most programming languages, text is just text. You write “hello world” and the language figures out what to do with it. Rust, being the safety-conscious language it is, wants to be very clear about who owns that text and where it lives in memory.

Text for the whole world: Rust and UTF-8

Rust’s treatment of text begins with today’s de facto standard for working with digital text: UTF-8 (Unicode Transformation Format-8-bit), which encodes characters as binary representations. UTF-8 uses *variable-length encoding*, which means some characters use just one byte (8 bits; these are the ASCII characters I mention in Book 1, [Chapter 1](#)), while some use two, three, or even four bytes:

```
let english = "Hello";           // 5 bytes (1 byte each)
let accented = "Olá";            // 4 bytes (á is 2 bytes)
let japanese = "こんにちは";    // 15 bytes (each character is 3 bytes)
let emoji = "Hello 🐼";         // 10 bytes (🐼 takes 4 bytes)
```

Encoding text this way ensures that UTF-8 encompasses not just basic English letters and numbers but also accented letters, non-Latin scripts such as Cyrillic and Japanese, ancient writing systems, mathematical symbols, and even emoji. The world’s characters and symbols all fall under the UTF-8 umbrella.

The potential problem with variable-length encoding is that when you ask for a chunk of memory that contains text, how can you be sure you’re getting a full character? If you ask for one byte, you might get lucky and get an ASCII character; but if that byte is part of a multibyte character, all you’re getting is part of that character, which is just gibberish.

Returning gibberish from memory just won’t do with Rust, so it enforces an iron-clad rule: All text types must contain valid UTF-8.

Text types in Rust: Why two?

Text in Rust appears confusing at first because the language treats text in two fundamentally different ways:

- » **Owned string:** This is the `String` type. It's a full-fledged type like `Vec<u8>` (a vector) or `[i32; 5]` (an array) but with an extra guarantee: Its contents are always valid UTF-8. The `String` type is heap-allocated, owns its memory, is growable, and can be mutated (to, say, add or remove text).
- » **String slice:** This is the `&str` type. This type represents a borrowed view into string data, which is why some Rust coders informally call it a “borrowed string.” It can point into a `String` and also into other strings, such as string literals and string slices. However, like a `String`, the `&str` type is guaranteed to be a valid slice of UTF-8 text. Think of `&str` as a window into valid UTF-8.

Okay, I hear you asking, but why the need for two separate text types? After all, when you want to borrow, say, an `i32`, you use `&i32`. When you want to borrow a `Vec<f64>`, you use `&Vec<f64>`. Why, when you want to borrow a `String`, can't you just use `&String`?

That would be fine if all you ever wanted to borrow was `String` data. If you only had `&String` to work with, all you could ever do is borrow from specific heap-allocated text buffers. But that wouldn't cover all the other sources of UTF-8 slices, including the following:

- » String literals
- » String slices (substrings)
- » Borrowed views into text stored in other types, such as structs

Rust solved this problem by creating `&str` as a dedicated type that works with *any* slice of valid UTF-8, no matter what the source. With `&str`, Rust gives you a universal borrowed text type, separate from the owned, growable buffer (`String`).

With `&str` on the job, functions can accept not just an `&String` reference but also a string literal, a substring, or any string slice, all with a single function signature:

```
fn do_something(s: &str) { ... }
```

The text twosome of `String` and `&str` balances flexibility, safety, and efficiency. `String` handles text ownership and growth, while `&str` provides a universal, lightweight view into UTF-8 text. Together they ensure that text manipulation in Rust is memory-safe and Unicode-friendly, while making explicit the difference between owning text and merely borrowing it.

When to use which

Okay, so when should you use `&str` and when should you use `String`? There are just two rules to keep in your back pocket:

- » Use `&str` when you're just reading or passing text around.
- » Use `String` when you need to create, modify, or own text data.

For function arguments and return types, keep the following in mind:

- » When a function requires text input, you'll almost always make the argument `&str` to take advantage of that type's flexibility.
- » When a function needs to return text that it owns, make the return type `String` to pass ownership of the text back to the caller.
- » When a function needs to return just a portion of some text, make the return type `&str`.

Making String Music: Creating Strings

Since strings in Rust aren't as simple as you might expect, you won't be even a little bit surprised to learn that creating strings isn't as simple as just wrapping some text in quotation marks. Fortunately, sometimes it *is* just that simple, as you learn in the next section.

Creating string literals

By far the most straightforward way to create a string in Rust is to enfold some text in double quotation marks (`"`). Yes, they must be double quotation marks; single quotation marks (`'`) are reserved for the `char` type. For example:

```
let currently_playing = "Adagio for Strings";
```

If your string literal is very long, you can break it up into multiple lines by adding a backslash (`\`) to the end of each line but the last one:

```
let music = "Serenade for Strings \
    in C major, Op. 48 \
    by Pyotr Ilyich Tchaikovsky");
```

Ah, but what if the text itself includes one or more double quotation marks? Doing the following will just confuse the compiler:

```
let next_piece = "Symphony No. 6, the \"Pastoral\" symphony";
```

Instead, you need to tell the compiler that those internal quotation marks are to be treated as characters rather than as string delimiters, a process known as *escaping* the character. You escape any character by preceding it with our old friend the backslash (`\`):

```
let next_piece = "Symphony No. 6, the \"Pastoral\" symphony";
```

The combo `\"` is called an *escape sequence*. Another useful escape sequence is `\n`, which starts a new line within the string:

```
let track = "Serenade for Strings\nby Dag Wirén";
println!("{}", track);
```

Output:

```
Serenade for Strings
by Dag Wirén
```

Another escape sequence you'll use frequently is `\\` to escape a backslash:

```
let playlist = "C:\\Users\\Ludwig\\Documents\\faves.txt";
```

Oof, that's a *lot* of backslashes. If you find yourself endlessly typing escape sequences such as `\\` and `\"` (and, more often than you care to admit, missing one or two and having the compiler point out your gaffe),

take pity on yourself and use *raw strings*, which tell the compiler to read everything in the string literally. You build a raw string with the letter `r` followed by the string literal, blissfully free of any escape sequences:

```
let next_piece = r"Symphony No. 6, the \"Pastoral\" symphony";  
let playlist = r"C:\Users\Ludwig\Documents\faves.txt";
```

String literals use the `&str` type. However, because string literals are valid for the entire duration of the program (I explain why in the sidebar, “[Memory 101: How string literals are stored](#)”), they have a special lifetime (refer to Book 3, [Chapter 3](#)) called `static`. So, the official type of a string literal is `&'static str` (the `'` character is called a *lifetime specifier* or just a *lifetime*).



REMEMBER

MEMORY 101: HOW STRING LITERALS ARE STORED

Okay, quiz time. When your code includes something like `let fave_composer = "Johann Sebastian Bach";`, are the bytes in that string literal stored on the stack or on the heap? Trick question! The answer is neither. Unlike other data that gets created at runtime and then stored on the stack or on the heap, the compiler doesn't create a string literal when the program runs. Instead, it creates the string literal *during compilation*, which means the text `Johann Sebastian Bach` gets embedded directly into your program's executable file.

When the program runs, it stores all its string literals in a special section of memory known as the *read-only data segment* (also called the *rodata segment* or just *.rodata* by all the cool kids). This memory area is marked as immutable by the operating system, so your program can read from it, but any attempt to change anything in there will crash the program. Since the rodata segment exists unchanged for the duration of the program, the program's string literals are guaranteed to live for the entire time the program is running, hence the `static` lifetime of any string variable.

The actual UTF-8 bytes (`J`, `o`, `h`, `a`, `n`, `n`, and so on) are stored at a particular address in the rodata segment and the `fave_composer` variable is a pointer to that address, plus the length of the string, making `&str` a fat pointer (address plus location).

Fun fact: Suppose you have two variables set to the same string literal:

```
let fave_composer = "Johann Sebastian Bach";  
let best_composer = "Johann Sebastian Bach";
```

The program does *not* store two separate versions of the string literal in the rodata segment. Instead, both variables point to the *same* string literal (a process Rust calls, endearingly, *deduplication*).

Creating Strings

When your code needs an owned `String`, Rust provides you with a just-this-side-of-infinite number of ways to go about it:

- » **Forging an empty string:** To create an empty string that you'll populate with text later (refer to the “[Adding to a string](#)” section, a bit later), use the `String` type's `new()` method:

```
let cello = String::new();
```

- » **Making a new string with a specified capacity:** If you know approximately how big your `String` will eventually grow, use the `String` type's `with_capacity(bytes)` method to allocate enough memory (where `bytes` is the amount of memory to allocate) to hold the text (which makes your program a teensy bit faster because it won't have to allocate memory as the string grows):

```
let double_bass = String::with_capacity(100);
```

- » **Creating a new string from an `&str` (part I):** Use the `String` type's `from(str)` method (where `str` is the `&str` value):

```
let violin = String::from("My other violin is a Stradivarius");
```

- » **Creating a new string from an `&str` (part II):** Use the `str.to_string()` method (where `str` is the `&str` value):

```
let viola = "Definitely not Viola Davis".to_string();
```



REMEMBER What's the difference between `String::from()` and `to_string()`? They do the same thing, so go with whichever one makes your code more readable. For good measure, you can also

use a third method: `str.to_owned()` method (where `str` is the `&str` value).

Converting String to &str

If you have an existing `String` value and want to create a new `&str` value from it, this conversion usually happens automatically, depending on how you create the `&str`. However, Rust does give you a couple of ways to be explicit about it:

```
let owned = String::from("String quartets");
let borrowed: &str = &owned; // Explicit conversion
let borrowed2 = owned.as_str(); // Another way
```

String Works: The Greatest Hits

The `String` type is rife with methods that return information about the text and enable you to perform tons of string-related tasks, such as adding, replacing, trimming, and searching text. In this section, you learn the most basic of these `String` operations. Later, in the “[String Movements: Common String Patterns](#)” section, you explore a few time-tested recipes for manipulating strings in your Rust programs.

Getting info about a string

When you need to get to know a particular string a little better, Rust offers a few methods that return information about that string:

- » `len()`: The number of bytes in the string. Remember that a UTF-8 character can take up between one and four bytes in memory, so you can’t just count the characters, especially when dealing with international text (bk02ch05/example01/ in this book’s example files):

```
fn main() {
    let english = String::from("Symphony"); // 8 English characters
    let russian = String::from("Симфония"); // 8 Cyrillic characters
    let korean = String::from("교향곡"); // 3 Korean characters
    let emoji = String::from("🎵🎶🎷🎹"); // 4 emoji characters

    println!("English string uses {} bytes.", english.len()); // 8
```

```

bytes
    println!("Russian string uses {} bytes.", russian.len()); // 16
bytes
    println!("Korean string uses {} bytes.", korean.len());    // 9
bytes
    println!("Emoji string uses {} bytes.", emoji.len());      // 16
bytes
}

```

» **chars()**: The characters in the string. Technically, this method returns an *iterator* over the string's characters. You learn what an iterator is in Book 2, [Chapter 6](#). For now, it's enough to know that you can use `chars().count()` to return the number of characters (not necessarily the same as the number of bytes!) in a string (bk02ch05/example02/):

```

fn main() {
    let english = String::from("Symphony"); // 8 English characters
    let russian = String::from("Симфония"); // 8 Cyrillic characters
    let korean = String::from("교향곡"); // 3 Korean characters
    let emoji = String::from("🎵🎶🎷🎹"); // 4 emoji characters

    println!("English string has {} characters.",
english.chars().count()); // 8
    println!("Russian string has {} characters.",
russian.chars().count()); // 8
    println!("Korean string has {} characters.",
korean.chars().count()); // 3
    println!("Emoji string has {} characters.",
emoji.chars().count()); // 4
}

```



TIP

If you want to know whether a particular character is in the ASCII range, use `char.is_ascii()`, which returns `true` if `char` is an ASCII character. If you want to know whether `char` is an ASCII letter or number, use `char.is_ascii_alphanumeric()`, instead.

» **is_empty()**: Returns `true` if the string has a length of 0 bytes; returns `false`, otherwise. Here's an example (bk02ch05/example03/):

```
fn main() {
    let h_string = String::new();
    let g_string = String::from("Air");

    println!("{}", h_string.is_empty()); // true
    println!("{}", g_string.is_empty()); // false
}
```

Adding to a string

When you declare a `String` as mutable, you can have all kinds of fun adding things to it, including individual characters, string slices, and other `String` values. Here are the possibilities:

- » **Adding a character to the end of the string:** Use the `string.push(char)` method, where `string` is the string to which you want to add and `char` is the `char` value that's appended. (If it's a character literal, it must be surrounded by single quotation marks `' '` since you're specifying a `char` type here.) Here's an example (bk02ch05/example04/):

```
fn main() {
    let mut sarcasm = String::from("World's tiniest");

    // Add single characters
    sarcasm.push(' ');
    sarcasm.push('🎻'); // Violin emoji
    sarcasm.push('!');
    println!("{}", sarcasm); // Output: World's tiniest 🎻!
```

- » **Adding an `&str` slice to the end of the string:** Use the `string.push_str(slice)` method, where `string` is the string to which you want to add and `slice` is the `&str` value that's added. (If it's a string literal, it must be surrounded by double quotation marks `" "`.) Here's an example (bk02ch05/example05/):

```
fn main() {
    let mut eww = String::from("String");
    eww.push_str(" cheese");
    println!("{}", eww); // Output: String cheese
}
```

Concatenating a String and a string slice

In Rust (as in many other programming languages), the `+` operator does double duty as the *concatenation operator* used to combine existing chunks of text into a single chunk. In Rust's case, the `+` operator concatenates a `String` value and a string slice. There are three ways you can go here:

- » **Concatenating a `String` and an `&str`:** In this case, you use `string` + `slice`, where `string` is the `String` value and `slice` is the `&str`. Here is an example (bk02ch05/example06/):

```
fn main() {
    let string1 = String::from("String");
    let string2 = " together.";
    let apt = string1 + string2;
    println!("{apt}"); // Output: String together.
}
```

- » **Concatenating two `String` values:** In this case, you use `string1` + `&string2`, (that is, you borrow the second `String` value) where `string1` and `string2` are `String` values. An example follows (bk02ch05/example07/):

```
fn main() {
    let string1 = String::from("Learning Rust adds ");
    let string2 = String::from("another string to your bow.");
    let backup_plan = string1 + &string2;
    println!("{backup_plan}");
    // Output: Learning Rust adds another string to your bow.
}
```

- » **Concatenating two `&str` slices:** In this case, you use `slice1.to_string()` + `slice2` (that is, you convert the first `&str` to a `String`), where `slice1` and `slice2` are the `&str` slices. For example (bk02ch05/example08/):

```
fn main() {
    let string1 = "Shoe";
    let string2 = "string";
    let budget_type = string1.to_string() + string2;
    println!("{budget_type}"); // Output: Shoestring
}
```



WARNING Whenever you concatenate using `string1 + string2`, where `string1` is a `String` value, the concatenation operation *moves* `string1` into the concatenated string. This means that after concatenating, `string1` is no longer valid, so trying to use it on your code will raise a `borrow of moved value: 'string1'` error.

If you still need to use `string1` after the concatenation, consider cloning it (bk02ch05/example09/):

```
fn main() {
    let string1 = String::from("Learning Rust adds ");
    let string2 = String::from("another string to your bow.");
    let backup_plan = string1.clone() + &string2;
    println!("{}", backup_plan);
    // Output: Learning Rust adds another string to your bow.
    println!("{}", string1); // This works!
}
```

Even better, you can ignore ownership and borrowing and cloning altogether by using the `format!` macro to glue some strings together (bk02ch05/example10/):

```
fn main() {
    let string1 = String::from("puppet");
    let string2 = String::from("lonely");
    let string3 = String::from("string");
    let lyric = format!("Just a {string1} on a {string2} {string3}.");
    println!("{}", lyric); // Output: Just a puppet on a lonely string.
}
```

Replacing text in a string

When your code needs to replace some or all instances of some substring in a larger string, Rust offers two methods:

```
string.replace(from, to)
string.replacen(from, to, count)
```

where in both methods `string` is the original string, `from` is the substring you want to replace, and `to` is the replacement text (which must be an `&str`). In the `replacen()` method, `count` specifies the number of

instances of *from* that you want replaced, starting for the beginning of *string*. Both methods return a newly allocated `String`; the original string is unchanged. Here's an example (bk02ch05/example11/):

```
fn main() {
    let string1 = String::from("First string");
    let demotion = string1.replace("First", "Second");
    println!("{}", demotion); // Output: Second string
}
```

Clearing a string

If your code ever needs to wipe a string's slate clean for a fresh start, use the `string.clear()` method, where *string* is the mutable string you want to reduce to the empty string (`""`). Here's an example (bk02ch05/example12/):

```
fn main() {
    let mut no_strings_attached = String::from("You see where this is
going.");
    no_strings_attached.clear();
    println!("No strings attached? {}", no_strings_attached.is_empty());
    // Output:
    // No strings attached? true
}
```



TIP If you just want to remove the last character of a mutable string, use `string.pop()`, where *string* is the string you want to modify. The `string.pop()` method returns the character it removed.

String Pieces: Slicing Strings

String literals and borrowed `String` values are valid `&str` slices because they're views into some text residing in memory. However, your Rust code will often need to take the word *slice* at face value and carve out a subset of some existing text, such as a portion of a string literal or a `String` value.

Slicing a string with the range syntax

In Book 2, [Chapter 4](#), I introduce you to slices in the context of arrays and vectors, where a slice is defined as some subset of the elements. The same range syntax — `[start..end]` — that I talk about in that chapter can also be applied to strings, where the first byte of the string is at index 0, the second byte is at index 1, and so on.

Here's an example (bk2ch05/example13/):

```
fn main() {
    let advice = "Time to cut the apron strings and move out!";
    println!("{}", &advice[..5]);    // Time
    println!("{}", &advice[16..29]); // apron strings
    println!("{}", &advice[34..]);   // move out!
}
```

Notice that each slice is taken from a borrow of the original string.

The preceding code works because the original string uses only English characters. Rust stores string literals as UTF-8 bytes but, as I mention earlier, it isn't 1-byte-per-character storage, as you might expect. Yes, most English characters get stored in a single byte each, but other UTF-8 characters might take 2, 3, or 4 bytes.



TIP You can find out the number of bytes that a particular `char` value requires when encoded in UTF-8 by using the `char.len_utf8()` method, where `char` is the character you're curious about. For example, `'é'.len_utf8()` returns 2.

Why might that be a problem? Recall that, from the perspective of the range syntax, the string indices 0, 1, 2, and so on are *byte* positions, not character positions. Here's an example where not taking that distinction into account can come back to bite you (pun not even close to intended; bk02ch05/example14/):

```
fn main() {
    let name = "Dag Wirén";
    let slice = &name[..8];
}
```

```
println!("{}", slice);  
}
```

The compiler doesn't like this:

```
byte index 8 not a char boundary; it is inside 'é' (bytes 7..9) of 'Dag  
Wirén'
```

This error means that your slice cleaved the character `é` — a two-byte UTF-8 character — right down the middle. Panic time!



WARNING Therefore, unless you're absolutely certain your string contains only one-byte UTF-8 characters (or in the unlikely event you're comfortable working with UTF-8 character byte boundaries!), and you can be absolutely certain that your start and end range values will always be within the bounds of the string, slicing strings with the range syntax is a dangerous game. Let's get you a helmet!

Safer slicing with `get()`

Rather than risking program crashes by slicing strings with the range syntax, Rust offers a safer way: the `string.get(start..end)` method, where `string` is any string value and `start` and `end` define the range of bytes you want in the slice. The `get()` method returns an `Option` (refer to Book 2, [Chapter 3](#)), where the `Some` result wraps the slice if it was valid and the `None` result applies if the slice was invalid. Here's an example (bk02ch05/example15/):

```
fn main() {  
    let name = "Dag Wirén";  
    let result = name.get(..8);  
    match result {  
        Some(slice) => println!("{}", slice),  
        None => println!("Whoops! Illegal slice operation!"),  
    }  
}
```

Output:

```
Whoops! Illegal slice operation!
```


If you don't need to handle the `None` result, you have the option (these puns just write themselves!) to make your code a bit leaner by using `if let`:

```
if let Some(slice) = name.get(..3) {  
    println!("{}", slice);  
}
```

Understanding deref coercion

Check out the following code (bk02ch05/example16/):

```
fn main() {  
    let msg = "Important message".to_string();  
    let str_length = get_length(&msg);  
    println!("{}", msg, " is {} characters long.", str_length);  
    // Output: 'Important message' is 17 characters long.  
}  
  
fn get_length(s: &str) -> usize {  
    s.len()  
}
```

Since the function argument's type annotation is a string slice (`&str`) but the function is called with a reference to the full `String` value, what is the local variable `s` getting here, really?

What's going on here is an example of one of Rust's key design choices: the automatic coercion — known to nerds as a deref coercion — between `String` and `&str`. *Deref coercion* means that `String` implements the `Deref` trait with `Target = str`. That's just a geeky way of saying that `String` can automatically coerce into `&str`. This isn't a copy or a new allocation. Instead, it gives a slice view into the string's contents. That is, `&msg`, which is a reference to the entire `String`, gets coerced into `&msg[..]`, which is a string slice covering all of `msg`.

So, in the example code, the parameter `s` is a string slice (`&str`), which means it's a reference consisting of the following:

- » A pointer to the start of the data owned by `msg`
- » The length of that data

You can think of it like this:

```
msg (String)
├ pointer => heap buffer "Important message"
├ length = 17
└ capacity = 17

&msg (reference to String)
└ automatically coerced into &str
    ├── pointer => "Important message"
    └ length = 17
```

On a related note, string slices also explain why, when you annotate a function parameter with `&str`, you can pass not only a `String` value but also a string literal. For example, suppose you write:

```
let greeting = "hello";
```

What is the `"hello"` part of the assignment? It's not a `String` but a string literal defined to use the type `&'static str`. Let's break down that type:

- » The `'static` part is the lifetime of the data, which in this case means the data is guaranteed to be available for as long as the program runs (since string literals are baked into the program's executable code). Book 3, [Chapter 3](#) tells you all you need to know about lifetimes in Rust.
- » What's left is `&str`, which tells you that a string literal is, by definition, a borrowed string slice!

So, when your function signature uses a parameter type of `&str`, you can pass either a `String`, which gets coerced to `&str`, or a string literal, which is already an `&str`.

String Movements: Common String Patterns

As you get more experienced with Rust, you'll find yourself performing a few common string-related tasks over and over. This section takes you through a few of these useful patterns.

Trimming whitespace from a string

Whitespace refers to invisible characters typically used for spacing effects. The most common whitespace characters are the space, the tab, and the newline but can also include the en space and the em space.

When your Rust code receives a string from an external source, one of the first things you should do is trim any leading or trailing (or both) whitespace characters. This is particularly true of user input, which always includes a newline character at the end from when the user pressed Enter or Return.

Rust offers quite a few methods for trimming a string, but the following three are the ones you'll turn to most often:

- » **`string.trim()`**: Returns an `&str` with the leading and trailing whitespace removed from *string*
- » **`string.trim_end()`**: Returns an `&str` with the trailing whitespace removed from *string*
- » **`string.trim_start()`**: Returns an `&str` with the leading whitespace removed from *string*

Here's an example (bk02ch05/example17/):

```
fn main() {
    let messy_input = " \t Pull some strings \t ";

    // Remove whitespace from both ends
    let cleaned_input = messy_input.trim();
    println!("'{messy_input}' -> '{cleaned_input}'");

    // Remove whitespace from the start
    let cleaned_start = messy_input.trim_start();
    println!("'{messy_input}' -> '{cleaned_start}'");

    // Remove whitespace from the end
    let cleaned_end = messy_input.trim_end();
```

```
println!("{messy_input}" -> "{cleaned_end}");
}
```

Output:

```
'      Pull some strings      ' -> 'Pull some strings'
'      Pull some strings      ' -> 'Pull some strings      '
'      Pull some strings      ' -> '      Pull some strings'
```



REMEMBER In case you're wondering, the `\t` escape sequence in the preceding code represents a tab character.

Searching and testing a string

Your Rust code will often need to interrogate a string to find out if it contains, starts with, or ends with a specified substring. And if a string does contain a particular substring, it can often be helpful to know the byte position where that substring begins. The following string methods enable you to perform all these tasks:

- » **`string.contains(substring)`**: Returns `true` if *substring* (an `&str` or `char` value) is found in *string*; returns `false`, otherwise. Note that `contains()` is case-sensitive, so `contains("c")` is a different search from `contains("C")`.
- » **`string.ends_with(substring)`**: Returns `true` if *substring* (an `&str` or `char` value) matches the end of *string*; returns `false`, otherwise.
- » **`string.starts_with(substring)`**: Returns `true` if *substring* (an `&str` or `char` value) matches the beginning of *string*; returns `false`, otherwise.
- » **`string.find(substring)`**: Searches *string* for *substring* (an `&str` or `char` value) and returns an `Option`. If a match was found, the `Some` result is returned and it wraps the byte position of the first character of *substring*; if no match was found, `None` is returned. Note that

`find()` is case-sensitive, so `find("cello")` is different from `find("Cello")`.

Here's an example (bk02ch05/example18/):

```
fn main() {
    let email = "cellos@orchestra.org";
    let url = "https://orchestra.org";

    // Check if a string contains a substring
    if email.contains("@") {
        println!("{email} looks like a legit address. Nice.");
    }

    // Check the start of a string
    if url.starts_with("https://") {
        println!("{url} is a secure site. Awesome.")
    }

    // Check the end of a string
    if url.ends_with(".org") {
        println!("{url} is a non-profit. Cool.")
    }

    // Find the position of a substring
    if let Some(at_position) = email.find("@") {
        let username = &email[..at_position];
        let domain = &email[at_position + 1..];
        println!("Username: {username}. Domain: {domain}. Boom!");
    }
}
```

Output:

```
cellos@orchestra.org looks like a legit address. Nice.
https://orchestra.org is a secure site. Awesome.
https://orchestra.org is a non-profit. Cool.
Username: cellos. Domain: orchestra.org. Boom!
```

Converting case

In the preceding section, I mention that both the `contains()` and `find()` methods are case-sensitive. That case-sensitivity is good because it forces you to be exact with your string searches, but it can be a pain when the text might have multiple case versions. For example, in the

name *Acme Strings*, the first word might also be spelled *ACME* or even *AcMe*.

Rather than setting up convoluted tests for every possible case, go easy on yourself by converting the string to either all-lowercase letters or all-uppercase letters. That way, you need only test a single case.

Rust offers two case-conversion methods:

» **`string.to_lowercase()`**: Returns a new `String` value with the letters in *string* converted to lowercase

» **`string.to_uppercase()`**: Returns a new `String` value with the letters in *string* converted to uppercase

Here's an example (bk02ch05/example19/):

```
fn main() {
    let address = "Herbert-von-Karajan-Straße 1, 10785 Berlin, Germany";
    println!("{}", address);
    println!("Characters in address: {}\n", address.chars().count());

    let address_upper = address.to_uppercase();
    println!("{}", address_upper);
    println!("Characters in address: {}", address_upper.chars().count());
}
```

Output:

```
Herbert-von-Karajan-Straße 1, 10785 Berlin, Germany
Characters in address: 51

HERBERT-VON-KARAJAN-STRASSE 1, 10785 BERLIN, GERMANY
Characters in address: 52
```

Did you notice what happened there? That's right: When converting the string to uppercase, the number of characters *increased* from 51 to 52! That happened because the German lowercase letter ß traditionally becomes SS when uppercased. (A capital ß does exist in Unicode, but Rust's `to_uppercase()` follows the traditional convention of expanding ß to SS.) The possibility that a case conversion can result in greater or fewer characters in the converted string is why both `to_lowercase()` and `to_uppercase()` return a new `String` value.



TIP

If you prefer to convert a string in-place rather than creating a string, use the `string.make_ascii_lowercase()` method or the `string.make_ascii_uppercase()` method. These methods are safe to use because they convert only ASCII letters. Non-ASCII letters are left as is.

Splitting a string

When a Rust program gets input from an external source, such as a file or database, one of the most common string patterns is to split the string into parts. For example, if your program reads data from a comma-separated values (CSV) file, each line will contain a single record, with each field separated by a comma. The entire line as a whole isn't much use, but the values in the individual fields are often extremely useful.

Splitting on whitespace

Happily, Rust makes it easy to extract parts from strings by offering a fistful of methods that split strings in different ways. I'll keep things simple for now by taking you through the `string.split_whitespace()` method, which splits `string` on any whitespace character, such as a space, tab, or newline. Here's the `split_whitespace()` method at work (bk02ch05/example20/):

```
fn main() {  
    let strings_you_bow = "Bass\tCello\tErhu\tViola\tViolin";  
    for instrument in strings_you_bow.split_whitespace() {  
        println!("{}", instrument);  
    }  
}
```

Output:

```
Bass  
Cello  
Erhu  
Viola  
Violin
```

Here, the `strings_you_bow` variable is bound to a string literal that contains several words separated by tabs, represented by the `\t` escape sequence. (Programmers call this a *tab-delimited* string.) The `split_whitespace()` method splits the string on those tabs.

The `split_whitespace()` method returns what Rust calls an *iterator*, a type that represents a sequence of values that you can process one at a time (that is, iterate through them). You learn all about iterators in Book 2, [Chapter 6](#). For now, it's enough to know that one way to handle what gets returned by `split_whitespace()` is to loop through the split value using a `for` loop.

Another route you can take is to add the `collect()` method, which handily bundles the split values into a vector ([bk02ch05/example21/](#)):

```
fn main() {
    let strings_you_bow = "Bass\tCello\tErhu\tViola\tViolin";
    let strings: Vec<&str> = strings_you_bow.split_whitespace().collect();
    println!("{:#?}", strings);
}
```

Output:

```
[
    "Bass",
    "Cello",
    "Erhu",
    "Viola",
    "Violin",
]
```

Splitting on a substring

The other split method that you'll use frequently is `string.split(substring)`, which splits `string` using `substring` as the separator. Here's an example ([bk02ch05/example22/](#)):

```
fn main() {
    let strings_you_pluck = "Banjo,Guitar,Harp,Mandolin,Ukulele";
    for instrument in strings_you_pluck.split(',') {
        println!("{}", instrument);
    }
}
```


Output:

```
Banjo  
Guitar  
Harp  
Mandolin  
Ukulele
```

In this example, the string is comma-separated, so the code uses `split(',')` to extract the words between the commas.

Project: Slugifying a String

Back in the days of print (remember print? No? Sigh.), editors kept their lives organized by representing each article using a label consisting of a word or the stub of the article’s headline. That label was called a *slug*.

Here in the digital era, the concept of the slug has been recycled for identifiers such as file names and URLs. When you *slugify* some text, you turn it into a URL- and file-system-safe, lowercase, hyphen-separated identifier. For example:

<i>Original Text</i>	<i>Slugified Text</i>
Serenade for Strings	serenade-for-strings
\t Pull some strings \t	pull-some-strings
Strings & snake_case	strings-and-snake-case
Dag Wirén	dag-wiren

Slugifying a string usually includes the following steps:

1. Lowercase everything.
2. Trim leading and trailing whitespace.
3. Replace spaces and common “separating” characters such as underscores with hyphens (-).
4. Remove punctuation and characters that aren’t URL-friendly.

5. Transliterate non-ASCII Unicode characters (for example, è, é, and ê become e).

Why bother? Because slugs make clean, human-readable identifiers for file names, blog post links, and database keys. Compare:

» <https://example.com/post?id=918273>

» <https://example.com/post/serenade-for-strings>

The second is nicer for readers and search engines alike.

For this project, I take you through a function named `slugify()` that takes an `&str` input and return a slugified `String` value.

Open a terminal and navigate to the directory where you store your Rust projects. Enter the following command:

```
% cargo new slugify-string
```

Open the project's `main.rs` file in your text editor and begin with the skeleton of the `slugify()` function:

```
fn slugify(input: &str) -> String {  
}
```

Until I tell you otherwise, you add the code that follows within the `slugify()` function.

First up is a helper function for appending a hyphen to the slug (which is stored in the `slug` variable):

```
fn push_hyphen(slug: &mut String, prev_hyphen: &mut bool) {  
    if !*prev_hyphen && ! slug.is_empty() {  
        slug.push('-');  
    }  
    *prev_hyphen = true;  
}
```

The main idea behind the `push_hyphen()` function is to avoid adding consecutive hyphens to the slug. The `prev_hyphen` Boolean is set to `true` each time the code appends a hyphen to the slug. If this function is

called and `prev_hyphen` is `false` (and the slug isn't empty), `slug.push('-')` appends the hyphen to the slug.

Next up is another helper function called `transliterate()`, which makes a few common substitutions of things such as accented characters into ASCII equivalents:

```
fn transliterate(ch: char) -> Option<&'static str> {
    match ch {
        // Single-letter replacements
        'à' | 'á' | 'â' | 'ã' | 'ä' | 'å' | 'ä' | 'ä' | 'ä' => Some("a"),
        'ç' | 'ç' | 'ç' | 'ç' | 'ç' => Some("c"),
        'd' | 'd' => Some("d"),
        'è' | 'é' | 'ê' | 'ë' | 'ë' | 'ë' | 'ë' | 'ë' | 'ë' => Some("e"),
        'f' => Some("f"),
        'g' | 'g' | 'g' | 'g' => Some("g"),
        'h' | 'h' => Some("h"),
        'i' | 'í' | 'î' | 'ï' | 'ï' | 'ï' | 'ï' | 'ï' => Some("i"),
        'j' => Some("j"),
        'k' => Some("k"),
        'l' | 'l' | 'l' | 'l' => Some("l"),
        'ñ' | 'ñ' | 'ñ' | 'ñ' | 'ñ' => Some("n"),
        'ò' | 'ó' | 'ô' | 'õ' | 'õ' | 'õ' | 'õ' | 'õ' | 'õ' => Some("o"),
        'r' | 'r' | 'r' => Some("r"),
        's' | 's' | 's' | 's' | 's' => Some("s"),
        't' | 't' | 't' | 't' => Some("t"),
        'ù' | 'ú' | 'û' | 'ü' | 'ü' | 'ü' | 'ü' | 'ü' | 'ü' => Some("u"),
        'ß' | 'ß' => Some("ss"),
        'ý' | 'ý' | 'ý' => Some("y"),
        'z' | 'z' | 'z' => Some("z"),
        // Common ligatures
        'æ' => Some("ae"),
        'œ' => Some("oe"),
        _ => None,
    }
}
```

This function returns a `Result` where `Some` wraps the transliteration and `None` is returned for every character not covered by the function (such as emojis).

Okay, that's it for the helper functions. Now the `slugify()` function's main code begins:

```
let mut slug = String::with_capacity(input.len());
let mut prev_hyphen = false;

let trimmed = input.trim();
```

This code initializes the `slug` variable as a new `String` with a capacity equal to the length of the function's `input` argument. The `prev_hyphen` Boolean is initialized as `false`. Finally, the `input` is trimmed.

Next up, the code loops through every character in the input (now stored in the `trimmed` variable) and the first task will be to lowercase each character. However, some care needs to be taken here because some characters get lowercased into multiple characters. To handle this, you need two loops:

```
for trimmed_char in trimmed.chars() {
    for ch in trimmed_char.to_lowercase() {
        // Code to handle each character goes here
    }
}
```

The outer `for` loop runs through characters returned by the `trimmed.chars()` method, and then the inner `for` loop handles the possibility of each `trimmed_char` containing multiple characters. For the next little bit, the code we're adding goes inside this inner `for` loop.

The inner `for` loop first checks whether `ch` is an ASCII alphanumeric character:

```
if ch.is_ascii_alphanumeric() {
    slug.push(ch);
    prev_hyphen = false;
    continue;
}
```

If so, the code pushes `ch` to `slug` as is, sets `prev_hyphen` to `false`, and skips over the rest of the loop.

Next, the code checks whether `ch` is a whitespace character:

```
if ch.is_whitespace() {
    push_hyphen(&mut slug, &mut prev_hyphen);
    continue;
}
```

If so, the code pushes `ch` to `slug` as a hyphen and skips over the rest of the loop.

Next up, the code looks out for a few common separators:

```
match ch {
    '-' | '_' | '-' | '-' | '-' | '/' | '\\\' | '\.' | '\.' | '|' => {
        push_hyphen(&mut slug, &mut prev_hyphen);
        continue;
    }
    _ => {}
}
```

If `ch` matches any of these separators, the code pushes `ch` to `slug` as a hyphen and skips over the rest of the loop.

Next the code tackles the case of characters that represent words:

```
match ch {
    '&' => {
        push_hyphen(&mut slug, &mut prev_hyphen);
        slug.push_str("and");
        prev_hyphen = false;
        continue;
    }
    '+' => {
        push_hyphen(&mut slug, &mut prev_hyphen);
        slug.push_str("plus");
        prev_hyphen = false;
        continue;
    }
    '@' => {
        push_hyphen(&mut slug, &mut prev_hyphen);
        slug.push_str("at");
        prev_hyphen = false;
        continue;
    }
    _ => {}
}
```

If `ch` is `&` (and), `+` (plus), or `@` (at), the corresponding word is pushed to the slug and the rest of the loop is skipped.

Now the code checks for a few common punctuation characters:

```
match ch {
    '.' | ',' | ':' | ';' | '!' | '?' |
```

```

    "'" | '"' | "'" | '„' | '«' | '»' |
    '\'' | ']' | ']' | '(' | ')' | '[' |
    ']' | '{' | '}' | '#' | '$' | '%' |
    '^' | '=' | '<' | '>' => {
        continue;
    }
    _ => {}
}

```

If `ch` is any one of these characters, it's bypassed (not added to the slug) and the rest of the loop is skipped.

Finally, the inner loop passes `ch` the `transliterate()` function:

```

if let Some(transliterated_ch) = transliterate(ch) {
    slug.push_str(transliterated_ch);
    prev_hyphen = false;
}

```

If `transliterate()` returns `Some`, the wrapped transliterated string is pushed to `slug` and `prev_hyphen` is set to `false`.

This ends the inner `for` loop. If any character wasn't covered by any of the `match` expressions or by the `transliterate()` function, it's simply discarded at this point.

Okay, now we're out of both `for` loops, so it's time to complete the `slugify()` function:

```

// Remove any trailing hyphens
while slug.ends_with('-') {
    slug.pop();
}

// Is our slug empty?
if slug.is_empty() {
    // Return a default slug
    String::from("slug")
} else {
    // Return the slug
    slug
}

```

This code first removes any trailing hyphens from `slug`. Then it checks whether `slug` is empty. If it is, the function returns a default string

("slug"); otherwise, the function returns the `slug` string.

You use the `main()` function to send your string to the `slugify()` function:

```
fn main() {
    let samples = [
        " Rust For Dummies! ",
        "Strings-Without-Tears", // em dashes
        "Café-noir & crème brûlée", // diacritics + ampersand
        "Rust: Tips, Tricks, & Gotchas", // punctuation
        "AC/DC + C++ @ 3am", // slash, plus, at
        "\"Smart\" quotes - 'everywhere'", // curly quotes + dash
        "Straße & Æsir vs. coöperate", // ß, ligatures, diaeresis
        "路径/пример/δοκιμή", // non-Latin drops
        " -- already-sluggo -- ", // leading/trailing separators
    ];

    for s in samples {
        println!("{:35} -> {}", s, slugify(s));
    }
}
```

Output:

Rust For Dummies!	-> rust-for-dummies
Strings-Without-Tears	-> strings-without-tears
Café-noir & crème brûlée	-> cafe-noir-and-creme-brulee
Rust: Tips, Tricks, & Gotchas	-> rust-tips-tricks-and-gotchas
AC/DC + C++ @ 3am	-> ac-dc-plus-c-plus-plus-at-3am
"Smart" quotes - 'everywhere'	-> smart-quotes-everywhere
Straße & Æsir vs. coöperate	-> strasse-and-aesir-vs-cooperate
路径/пример/δοκιμή	-> slug
-- already-sluggo --	-> already-sluggo

Chapter 6

Functional Programming with Closures and Iterators

IN THIS CHAPTER

- » Learning about functional programming
 - » Getting acquainted with closures
 - » Creating and working with iterators
 - » Building a text analysis program with iterators
-

Rust iterators are the best thing since [Bread].

—YANIEL

At times in this book, I've described Rust code as *elegant*, and I've tried to give you a sense that Rust programming can be a genuine pleasure. And there have been some nice Rust features so far in the book. The `match` structure is a flexible way to control program flow; the `if let` statement is a sweet way to handle `Option` and `Result` returns; vectors are powerful collections.

But when I think about Rust code being truly elegant and pleasurable both to write and read, I'm really talking about the two topics in this chapter: closures and iterators. Maybe it's the coding nerd in me talking, but these are beautiful Rust features that make your code more powerful, more efficient, and more expressive. I can't wait to tell you about them!

What on Earth Is Functional Programming?

Way, way back in Book 1, [Chapter 1](#), I defined a computer as “a machine that does nothing until someone or something tells it what to do.” What I’ve been doing since that point is showing you how to write Rust code that enables you to tell your computer what to do, step by finicky step. Along the way, you’ve learned how to use `if` statements to make decisions, loops to repeat actions, and functions to organize your code. This “do this, then do that” approach is called *imperative programming* by the computer science professors.

But another approach called *functional programming* changes how you think about the problem. Instead of writing step-by-step instructions that modify variables (“add 1 to the counter, then check if it’s even”), you write expressions that transform data (“take this list and give me back only the even numbers”). The key idea is that you describe *what* you want, and the system applies your transformations to get the result. It’s not magic because you still write the logic, but that logic is expressed as functions that transform inputs into outputs rather than as commands that change state.

Functional programming is the subject of this chapter. Over the next few pages, you discover two powerful Rust features that make functional programming not just possible but something you’ll wonder why I waited until this far in the book to tell you about: closures and iterators. These features are the royal road to Rust code that’s more readable, more concise, and almost always more efficient than the equivalent imperative code.

Closures: Anonymous Functions with Attitude

When I introduced you to functions in Book 1, [Chapter 7](#), I tried to emphasize that good coding practice is to make each function perform just a single task or calculation. Okay, so here’s an example of a function that’s about as simple as it gets:

```
fn doubler(x: i32) -> i32 {  
    x * 2  
}
```

True to coding best practices, this function does just one thing: It takes an input and returns its value doubled.

But even with an example this simple, the function itself still *looks* complex. So, here’s a question for you: Out of everything in the preceding function definition, what part or parts define the true *essence* of the function? Given what this function is designed to accomplish, what is essential to that task? With a bit of thought, I think you’ll see that just two things could be described as essential:

- » The input value: `x`
- » The expression: `x * 2`

Everything else — the `fn` keyword, the `doubler` name, the `i32` argument type, the `i32` return type, and the braces `{` and `}` — are the scaffolding that supports the function’s essence.

Meet the closure

A *closure* is a regular Rust function reduced to that true essence. The term *closure* comes from the fact that these functions can “close over” their environment, meaning they can capture and use variables from the surrounding scope where they’re defined. This ability to capture context is what makes closures more than just anonymous functions. Here’s the preceding function written as a closure:

```
|x| x * 2
```

Isn’t that beautiful? This closure says, “Give me an `x` and I’ll give you back `x * 2`.” Here’s how you might use such a closure (bk02ch06/example01/ in this book’s example files):

```
fn main() {  
    let doubler = |x| x * 2;  
    println!("{}", doubler(10)); // Output: 20  
}
```

Here's the general syntax for a closure:

```
|arguments(s)| expression_or_block
```

where *argument(s)* is one or more comma-separated arguments and *expression_or_block* is either an expression or an expression block. There are no types because the compiler can almost always infer the types used by the arguments. Here's another example (bk02ch06/example02/):

```
fn main() {  
    let greeting = |name| format!("Hello there, {name}!");  
    println!("{}", greeting("Ferris"));  
}
```

Output:

```
Hello there, Ferris!
```

Here, the compiler infers that `name` is an `&str` because the closure gets passed a string literal in the `println!` statement.

But if the compiler stumbles on an inference (it happens from time to time), you can help it along by annotating the closure arguments with the required types:

```
|x: i32, y: i32| x * y
```

Capturing local variables

But a closure isn't just a barebones version of a regular Rust function. Closures also have a superpower that functions lack: A closure can immutably or mutably borrow variables that exist in the surrounding scope! This process is known as *capturing* local variables.

Here's an example of an immutable borrow of a local variable (bk02ch06/example03/):

```
fn main() {  
    let threshold = 1000;  
    let is_big = |n| n >= threshold; // Immutable borrow of threshold  
    println!("{}", is_big(1001)); // Output: true  
}
```

The closure expression `n >= threshold` borrows the local variable `threshold` immutably.

What makes this special? Regular functions can't do this. For example, if you tried to reference `threshold` inside a regular function defined elsewhere, the compiler would complain that `threshold` doesn't exist in that scope. But a closure, because it's defined right there in the same block, can reach out and grab variables from its surroundings. The closure essentially carries those captured variables along with it, like a backpack of context.

Here's an example of a mutable borrow (bk02ch06/example04/):

```
fn main() {
    let mut total = 0;
    let mut add = |x| total += x; // Mutable borrow of total
    add(5);
    add(7);
    println!("{}", total); // Output: 12
}
```

Sorting with a closure

One of the most common and most useful collection operations is sorting the elements, and sorting is where closures first start to earn their keep. Rust gives you two closure-tinted sorting methods: `sort_by_key()` and `sort_by()`.

`sort_by_key()`: Sorting based on a key

When you want to sort the elements in a collection based on a property of each element, such as word length, the `sort_by_key()` method is the door to go through:

```
collection.sort_by_key(closure)
```

where *collection* is the collection you want to sort (such as a vector or an array), and *closure* is a *key extraction function*, which is a closure that derives a property — called the *key* — from the collection data and sorts the collection based on those property values.

A common example is sorting a collection of strings based on their length. Here's an example that illustrates this (bk02ch06/example05/):

```
fn main() {  
    let mut dogs = [  
        "Flat-Coated Retriever",  
        "Pug",  
        "Bulldog",  
        "Poodle",  
        "Basenji",  
        "Chihuahua",  
        "Siberian Husky",  
    ];  
    dogs.sort_by_key(|dog| dog.len());  
    println!("{:?}", dogs);  
}
```

Output:

```
[  
    "Pug",  
    "Poodle",  
    "Bulldog",  
    "Basenji",  
    "Chihuahua",  
    "Siberian Husky",  
    "Flat-Coated Retriever",  
]
```

The money statement in this code is the following:

```
dogs.sort_by_key(|dog| dog.len());
```

This code tells Rust to derive the length of the string for each string in the array, and then use those lengths to sort the array.

sort_by(): Sorting based on an ordering

Rust defines an enum called `Ordering` that has three variants:

- » **Less:** An ordering where a compared value is less than the value with which it's being compared
- » **Equal:** An ordering where a compared value is equal to the value with which it's being compared

» **Greater:** An ordering where a compared value is greater than the value with which it's being compared

Rust's `sort_by()` method uses an ordering to sort the values in a collection:

```
collection.sort_by(closure)
```

where *collection* is the collection you want to sort (such as a vector or an array), and *closure* is a closure that takes two values, say *a* and *b*, and returns an `Ordering` enum containing the `Less` variant if $a < b$, the `Equal` variant if $a = b$, or the `Greater` variant if $a > b$.

By far the easiest way to generate an `Ordering` enum is to use the `a.cmp(b)` method, which compares *a* and *b* and returns whatever `Ordering` variant applies. Here's an example (bk02ch06/example06/):

```
fn main() {
    let mut dogs = [
        "Flat-Coated Retriever",
        "Pug",
        "Bulldog",
        "Poodle",
        "Basenji",
        "Chihuahua",
        "Siberian Husky",
    ];
    dogs.sort_by(|a, b| a.cmp(b));
    println!("{:?}", dogs);
}
```

Output:

```
[
    "Basenji",
    "Bulldog",
    "Chihuahua",
    "Flat-Coated Retriever",
    "Poodle",
    "Pug",
    "Siberian Husky",
]
```

In this example, `a` and `b` represent elements from the array, so `a.cmp(b)` just compares the elements straight up. That's usually what you want for a sort, but within the closure expression you can use `a` and `b` in whatever way makes sense for your sort. For example, you could use `a.len().cmp(b.len())` to compare lengths.

Different ways to write closures

Rust gives you a bunch of ways to write closures, from the succinct to the verbose (bk02ch06/example07/):

```
fn main() {
    // Concise version - Let the compiler figure it out
    let square = |x| x * x;
    println!("{}", square(10)); // Output: 100

    // Version 2: Add a type to the closure argument
    let square2 = |x: i32| x * x;
    println!("{}", square2(10)); // Output: 100

    // Version 3: Add types to the closure argument and result
    let square3 = |x: i32| -> i32 { x * x };
    println!("{}", square3(10)); // Output: 100

    // Expression block example
    let hypotenuse_length = |a, b| {
        let sum_of_squares: f64 = square(a) as f64 + square(b) as f64;
        sum_of_squares.sqrt()
    };
    println!("{}", hypotenuse_length(3, 4)); // Output: 5
}
```

The style you use comes down to how much detail you want to provide or is necessary to either get the closure to do what you want or to make the compiler happy. Most of the time, it's best to let Rust figure out the types involved and keep your closures short and sweet, as nature intended.

Closures are a lovely addition to Rust all by themselves, but they really shine when you combine them with an iterator, which is what the rest of this chapter is all about.

Meet the Iterator Pattern

I mention iteration in several places in previous chapters, most notably in relation to the `for` loop. But now it's time to pop the hood and examine the engine that powers not only the `for` loop but all iteration in Rust: the iterator. In Rust, an *iterator* is an object (technically, it's a trait; refer to Book 3, [Chapter 1](#)) that can go through a collection of items — such as a vector or an array — one at a time.

Why would you want to iterate a collection? Let me count the ways:

- » **Perform some operation on each element of the collection.** For example, you might want to print each string in a collection of strings.
- » **Calculate a value based on the elements of the collection.** For example, given a collection of integers, you might want to find the sum or the average.
- » **Find a value based on some criterion.** For example, in a collection of strings, you might want to find the first value that begins with Q.
- » **Filter the elements based on some criterion.** For example, in a collection of integers, you might want to work with just the even numbers.
- » **Transform each element of the collection.** For example, you might want to double each number in a collection of numeric values.

As you can tell from this list, iterators are one of Rust's most powerful and useful features. Examine just about any Rust code that does something non-trivial, and you'll find one or more iterators doing the heavy lifting.

Creating an Iterator from a Collection

In Book 2, [Chapter 1](#), I talk about ownership and borrowing. One of the main ideas in that chapter is that when you declare a variable based on some existing data, that new variable can be only one of the following:

- » An immutable borrower of the original data
- » A mutable borrower of the original data
- » The new owner of the original data

Since an iterator is a new object based on some existing data — in this case a collection, such as a vector — you won't be surprised to learn that the iterator, too, can be only one of the following:

- » An immutable borrower of the collection
- » A mutable borrower of the collection
- » The new owner of the collection

Borrowing immutably with `iter()`

When your iterator just needs to read the elements in a collection, use the `iter()` method to create an iterator that immutably borrows the collection:

```
collection.iter()
```

where `collection` is the collection you want to iterate. Here's an example (bk02ch06/example08/):

```
fn main() {  
    // Here's the collection  
    let primes = vec![2, 3, 5, 7, 11, 13];  
  
    // Iterate with an immutable borrow via iter()  
    for prime in primes.iter() {  
        print!("{}", prime); // Each prime is &i32  
    }  
  
    // The primes vector is still valid here  
    println!("\n{:?}", primes);  
}
```

Output:

```
2, 3, 5, 7, 11, 13,  
[2, 3, 5, 7, 11, 13]
```

This code creates a vector named `primes`, uses `primes.iter()` to create an immutably borrowed iterator, and then uses a `for` loop to run through that iterator, storing each vector element in `prime`, a borrowed (&i32) value. Because the `primes` vector was borrowed only to use with the iterator, `primes` is still valid after the `for` loop completes its chores.

Borrowing mutably with `iter_mut()`

When your iterator needs to make changes to the elements in a collection, use the `iter_mut()` method to create an iterator that mutably borrows the collection:

```
collection.iter_mut()
```

where `collection` is the collection you want to iterate. Here's an example (bk02ch06/example09/):

```
fn main() {  
    // A collection of boastful words  
    let mut words = vec![  
        String::from("BLUSTER"),  
        String::from("BRAGGADOCIO"),  
        String::from("GASCONADE"),  
        String::from("RODOMONTADE"),  
    ];  
  
    // Iterate with a mutable borrow via iter_mut()  
    for word in words.iter_mut() {  
        // Lowercase the word  
        word.make_ascii_lowercase(); // Each word is &mut String  
    }  
  
    // The words vector is still valid here  
    // but the words have been changed  
    println!("{:?}", words);  
}
```

Output :

```
["bluster", "braggadocio", "gasconade", "rodomontade"]
```

This code creates a vector named `words`, uses `words.iter_mut()` to create a mutably borrowed iterator, and then uses a `for` loop to run through that iterator, storing each vector element in `word`, a mutably borrowed (`&mut String`) value. The loop uses the `make_ascii_lowercase()` method to lowercase each word in place.

Owning with `into_iter()`

If, after creating an iterator, you no longer need the original collection, you can use the `into_iter()` method to create an iterator that takes ownership of the collection:

```
collection.into_iter()
```

where `collection` is the collection you want to iterate. Here's an example (bk02ch06/example10/):

```
fn main() {
    // A temporary vector to hold the old wholesale prices
    let old_prices = vec![15.75, 22.2, 28.15, 30.6, 42.85];

    // Set up a vector for the new prices
    let mut new_prices = Vec::new();

    // Iterate by taking ownership with into_iter()
    for price in old_prices.into_iter() {
        new_prices.push(price * 1.05);
    }

    // Print the new prices
    println!("{:.2?}", new_prices);

    // This won't work because old_prices is now invalid
    //println!("{:.2?}", old_prices);
}
```

Output:

```
[16.54, 23.31, 29.56, 32.13, 44.99]
```

This code creates a vector named `old_prices` and an empty vector named `new_prices`, and then uses `old_prices.into_iter()` to create an iterator that takes ownership of `old_prices`. The code then uses a `for` loop to run through that iterator, storing each vector element in `price`, an

owned (f64) value. The loop multiplies `price` by 1.05 and pushes that value to the `new_prices` vector.

Watch what happens when you implement a garden-variety `for` loop and then try to use the collection after the loop (bk02ch06/example09/):

```
fn main() {  
    let numbers = vec![1, 2, 3, 4, 5];  
    for number in numbers {  
        println!("{}", number);  
    }  
    println!("{:?}", numbers);  
}
```

The borrow checker puts a Stop sign in front of the second `println!` statement:

```
error[E0382]: borrow of moved value: 'numbers'
```

What does this tell you? Right: A regular `for` loop is using an `into_iter()` iterator behind the scenes. Interesting!

Consuming Iterators

When you create an iterator using, say, the `iter()` method, you're creating a lightweight object that knows how to step through the collection one element at a time. Think of it as a cursor or bookmark that starts at the beginning and knows how to move to the next item. But here's the key: Creating the iterator doesn't do anything with the data yet. The iterator just sits there, ready to go, until you ask it to start moving. As you learn in the "[Adapting Iterators](#)" section, Rust offers a bunch of methods that enable you to transform an iterator in some way, such as filtering the collection to include only the values that meet some criteria. But even then, the transformation is only a potential one.

In Rust vernacular, iterators are said to be *lazy* because they don't do anything useful until you ask them to. The way you ask an iterator to do something useful is to apply a *consuming* method on the iterator.

Consuming methods that perform calculations

Rust offers a nice collection of consuming iterator methods that reduce a collection down to a single result, such as a sum or the maximum value:

- » `iterator.count()` : Returns the number of iterations it would take to run through all the elements in the original collection
- » `iterator.last()` : Returns an `Option` that contains `Some` wrapping the last element in the original collection or `None` if the collection is empty
- » `iterator.max()` : Returns an `Option` that contains `Some` wrapping the largest element in the original collection or `None` if the collection is empty
- » `iterator.min()` : Returns an `Option` that contains `Some` wrapping the smallest element in the original collection or `None` if the collection is empty
- » `iterator.product()` : Returns the product of the elements in the original collection
- » `iterator.sum()` : Returns the sum of the elements in the original collection

These methods are called *consuming* because they use up the iterator. When you call, say, `sum()` or `count()`, the iterator moves through all its elements and can't be used again. You've "consumed" it to produce a result.

Here's an example that tries out each of these methods (bk02ch06/example11/):

```
fn main() {  
    let numbers: Vec<i32> = vec![4, 6, 3, 2, 1, 5];  
  
    let how_many_nums = numbers  
.iter().count();  
    let last_result = numbers.iter().last();  
    let largest_result = numbers.iter().max();  
    let smallest_result = numbers.iter().min();  
    let product_of_nums: i32 = numbers.iter().product();  
    let total_of_nums: i32 = numbers.iter().sum();  
}
```

```
println!("Elements: {}", how_many_nums);
if let Some(last_num) = last_result {
    println!("Last element: {}", last_num);
}
if let Some(largest_num) = largest_result {
    println!("Largest element: {}", largest_num);
}
if let Some(smallest_num) = smallest_result {
    println!("Smallest element: {}", smallest_num);
}
println!("Product of elements: {}", product_of_nums);
println!("Total of elements: {}", total_of_nums);
}
```

Output:

```
Elements: 6
Last element: 5
Largest element: 6
Smallest element: 1
Product of elements: 720
Total of elements: 21
```



TIP

Method chaining is when multiple methods are applied one right after the other, as in `numbers.iter().count()`. You can make your method chains a bit easier to read by placing each method on its own line, like so:

```
let how_many_nums = numbers
    .iter()
    .count();
```

find(): Finding the first item that matches

If you need to find the first item in a collection that matches some condition, use the `find()` method:

```
iterator.find(closure)
```

where *iterator* is the iterator and *closure* is a closure that returns `true` or `false` when applied to each element of *iterator*. The `find()` method returns an `Option`: the first element for which the closure evaluates to

`true` is returned as `Some(element)`; if no element evaluates to `true`, the `Option` returns `None`.

Here's an example (bk02ch06/example12):

```
fn main() {
    let rates = vec![0.45, 0.73, 0.33, 1.15, 0.9, 2.76, 3.25, 1.55];

    let result = rates
        .iter()
        .find(|&n| *n > 1.0);
    match result {
        Some(first_over_one) => println!(
            "The first rate over 1.0 is {first_over_one}"),
        None => println!("No rate over 1.0 found!"),
    }
}
```

This code uses a closure to look for the first item in `rates` that has a value greater than 1.0.

Let's take a closer look at the `find()` closure:

```
.iter().find(|&n| *n > 1.0)
```

Here's what's going on here:

1. `iter()` produces (in this case) `&f64` items (that is, references to the floating-point numbers in the vector).
2. `find()` always hands its closure a reference to the item, so that means the closure here receives an `&&f64`. Weird, right?
3. In the closure, `|&n|` pattern-matches one layer of reference away, meaning that `n` is passed into the closure expression as `&f64`.
4. You can't compare a reference to a floating-point value, so you need to dereference `n` with `*n` to get the actual `f64` value for comparison.

If you no longer need the original collection, you can simplify this process by using `into_iter()` instead of `iter()`:

```
.into_iter().find(|&n| n > 1.0)
```

Let's go through what happens now:

1. `into_iter()` produces (in this case) `f64` items (that is, values, not references, because `into_iter()` takes ownership).
2. `find()` hands the closure a reference to the item, so this time the closure receives an `&f64`.
3. In the closure, `|&n|` destructures one layer of reference away, meaning that `n` is passed into the closure expression as `f64`.
4. You can compare two floating-point values, so there's no need to dereference `n`.



TIP If you just want to know whether a particular value exists in an iterator, use the `iterator.any(closure)` method, where `closure` is a closure that returns `true` or `false` when applied to each element of `iterator`:

```
let result = rates
    .iter()
    .any(|n| *n > 1.0);
```

If the closure returns `true` for at least one element, `any()` returns `true`; `any()` returns `false`, otherwise.

A similar method is `iterator.all(closure)`. If the closure returns `true` for every item in `iterator`, `all()` returns `true`; `any()` returns `false`, otherwise.

collect(): The ultimate consumer

The `iterator.collect()` method transforms `iterator` into a collection. The usual workflow here is as follows:

1. Create an iterable from your original collection using `iter()`, `iter_mut()`, or `into_iter()`.
2. Perform one or more transformations using the iterator adapter methods covered in the next section.

3. Run `collect()` at the end of the chain to gather the transformed data to a *new* collection. The original collection may or may not still be available, depending on which iterator method you used in Step 1.

You already know how to perform Step 1, but before you can get to Step 3 you need to know about the iterator adapter methods that are the heart of Step 2. That's the subject of the next section.

Adapting Iterators

An *iterator adapter* is a method that transforms one iterator into another. For example, after starting with an iterator from a collection of integers, you might filter those values to include only even numbers, which creates a new iterator just for the even integers. You could then run `collect()` to save those values to a new collection.

filter(): Keeping only what you want

You use the `filter()` method to keep only those items in the iterator that match a condition:

```
iterator.filter(closure)
```

where *iterator* is the iterator and *closure* is a closure that returns `true` or `false` when applied to each element of *iterator*. The `filter()` method returns an iterator that includes the elements for which the closure evaluates to `true`.

Here's an example (bk02ch06/example13):

```
fn main() {
    let numbers: Vec<i32> = vec![4, 6, 3, 2, 1, 5];

    let evens: Vec<i32> = numbers
        .iter()
        .filter(|&n| *n % 2 == 0)
        .collect();
    println!("{:?}", evens);
}
```

Output:

```
[4, 6, 2]
```

In this code, `filter(|&n| *n % 2 == 0)` returns an iterator containing just the even numbers from the original collection. That iterator is saved into a `Vec<i32>` vector by the `collect()` method, and the new vector is printed.



TIP

You can limit the number of items in an iterator by using the `iterator.take(n)` method, where `n` means that only the first `n` items are taken from `iterator`.

map(): Transforming each item

You use the `map()` method to use a closure to transform each item in some way:

```
iterator.map(closure)
```

where `iterator` is the iterator and `closure` is a closure that takes a reference to each `iterator` item and modifies the item. The `map()` method returns an iterator that includes the transformed elements.

Here's an example (bk02ch06/example14):

```
fn main() {
    let old_prices = vec![15.75, 22.2, 28.15, 30.6, 42.85];

    let new_prices: Vec<f64> = old_prices
        .into_iter()
        .map(|price| price * 1.05)
        .collect();

    // Print the new prices
    println!("{:.2?}", new_prices);
}
```

Output:

```
[16.54, 23.31, 29.56, 32.13, 44.99]
```

This code performs the same price updating task that earlier I used a `for` loop to accomplish (refer to the “[Owning with into_iter\(\)](#)” section). We

get the same result, but with a neater and easier-to-understand method chain.

enumerate(): Adding index numbers

Like arrays and vectors, iterators use index numbers behind the scenes, where the first iterator item is 0, the second item is 1, and so on. If your code requires both the item and its index, the `iterator.enumerate()` method makes the index number of `iterator` available. Here's an example (bk02ch06/example15/):

```
fn main() {  
    let fruits = vec![  
        "apple",  
        "banana",  
        "cherry",  
        "peach",  
        "grape",  
        "blueberry",  
        "mandarin",  
        "blackberry",  
        "strawberry",  
        "pear",  
    ];  
  
    fruits  
        .iter()  
        .enumerate()  
        .for_each(|(i, fruit)| println!("{}", i + 1, fruit));  
}
```

Output:

```
1: apple  
2: banana  
3: cherry  
4: peach  
5: grape  
6: blueberry  
7: mandarin  
8: blackberry  
9: strawberry  
10: pear
```

Note that this code uses the `iterator.for_each(closure)` method, which acts as a kind of one-line loop that uses the `closure` to perform an

operation (in this example, running `println!`) on each element of *iterator*.

Project: Word Wrangler

I finish this chapter with a small project called Word Wrangler that makes good use of iterators and closures to process and analyze a large chunk of text.

To get the project off the ground, head to your main Rust project directory and enter the following command:

```
cargo new word-wrangler
```

Load that project's `main.rs` into your text editor. You need a hash map (Book 2, [Chapter 4](#)) in this project, so add the following to the top of the file:

```
use std::collections::HashMap;
```

Tokenizing the text

The data in this project is a large run of text (Chapter 1 from a public domain version of George Eliot's novel, *Middlemarch*), plopped unceremoniously as a raw string into the `main()` function. To analyze this text, you need to *tokenize* it, which means to break apart the text into smaller, meaningful chunks called tokens. In general, a *token* is the smallest unit of text that still carries meaning, where the definition of “meaning” depends on the context.

In the context of this project, that unit is the word, so the first thing you want to do with the text is tokenize it:

```
const STOPWORDS: &[&str] = &[
    "the", "a", "an", "and", "or", "but", "of",
    "to", "in", "on", "for", "is", "it", "this",
    "that", "as", "at", "by", "be", "with", "are",
    "was", "were", "from", "has", "have", "had",
    "not", "no", "so", "if", "then",
];

fn tokenize(text: &str) -> Vec<String> {
```

```

    text.split(|c: char| !c.is_alphanumeric() && c != '\')
        .filter(|w| !w.is_empty())
        .map(|w| w.to_lowercase())
        .filter(|w| w.len() >= 3)
        .filter(|w| !STOPWORDS.contains(&w.as_str()))
        .collect()
}

```

This code begins by declaring a constant named `STOPWORDS`, which is an array of trivial words — known in the tokenizing trade as *stopwords* — that should be excluded when the tokenizing happens. But I’m sure right now your brow is significantly furrowed by the definition of this constant:

```
const STOPWORDS = &[&str] = &[...];
```

The `[&str]` part is straightforward: It’s an array (a slice) of string literals, each of which is an `&str`. But why is the type annotation `&[&str]` instead of just `[&str]`? `STOPWORDS` isn’t an array with a fixed length baked into its type annotation, such as `[&str; 32]` for 32 stopwords. You *could* do that, but then you’d have two hurdles to overcome:

- » You’d have to remember to update the type any time you added words to or removed words from the array.
- » The `contains()` method used in the `tokenize()` function expects a slice, so you’d need to take the extra step of borrowing `STOPWORDS` to use it with that method.

The code steps nimbly over these obstacles by declaring `STOPWORDS` to be a reference to a slice of string slices; that’s what `&[&str]` means. Now you can add and remove words at will, and the code can use the `STOPWORDS` constant directly with methods such as `contains()` without the compiler needing to know the number of elements in advance.

However, since the type is a reference to a slice, the data has to match, so that’s why an `&` is in front of the array.

Okay, next is the `tokenize()` function, which takes the text to be tokenized as input (in the `text` argument) and returns a vector of `String` values: the tokens — words, in this case — extracted from the text.

The tokenization requires no less than six method calls on the text:

- » `split(|c: char| !c.is_alphanumeric() && c != '\')`; This method uses a closure to split the text. The closure checks every character `c` and splits if the character isn't alphanumeric. The exception is the apostrophe, which we want to preserve in words such as *don't* and *she'll*. This method returns an iterator over the split substrings.
- » `filter(|w| !w.is_empty())`: This method iterates over the split substrings and discards any that are empty.
- » `map(|w| w.to_lowercase())`: This method converts every word to lowercase so that we don't have to worry about case when analyzing the tokens.
- » `filter(|w| !STOPWORDS.contains(&w.as_str()))`: This method filters out all the stopwords.
- » `collect()`: This method gathers all the tokens into a `Vec<String>` (the function's return type) and returns it.

Okay, the project has its tokens, so now it's time to analyze them. Well, *analyze* is probably too strong a word here because we're really just going to perform a few straightforward interrogations of the data. But the point isn't so much to truly analyze the text but to give you some practice using closures and iterators. That said, onward we go.

Counting word frequencies

An often-useful text analysis is to count the frequency of each word in the text. Now you see why ignoring stopwords is so important because filler words such as *the* and *or* would dominate any list of word frequencies.

In this project, the word frequencies are generated by the `word_counts()` function:

```
fn word_counts(words: &[String]) -> HashMap<String, usize> {
    let mut counts = HashMap::new();
    for w in words {
        if let Some(count) = counts.get_mut(w) {
            *count += 1;
        } else {
            counts.insert(w.clone(), 1);
        }
    }
    counts
}
```

The goal here is to save each unique word and the number of times it appears in the text, which is a perfect task for a hash map. So, the `word_counts()` function takes the word tokens as input (in the `words` argument) and returns a hash map of type `HashMap<String, usize>`, where each word is the key and each word's frequency is the value.

The function declares a new hash map (`counts`), and then iterates through the words. The point of this iteration is, first, to check whether the current word is already a key in the hash map. To that end, the `counts.get_mut(w)` method checks whether the word exists in `counts`. The `get_mut()` method is the mutable version of `get()` because, if the word does exist, the code updates the count, so it needs to be a mutable reference.

Like `get()`, `get_mut()` returns an `Option`. If the word (`w`) is already a hash map key, the current count is wrapped in the `Some`, so the code increments the count; otherwise (that is, if the `Option` returns `None`), the word is inserted as a key with the value 1.

When all is done and said, the hash map is returned.

Getting the most frequently used words

The hash map of word frequencies has lots of useful information, but perhaps the most useful are the words that appear most often in the text. Extracting that information is the task of the `top_words()` function:

```
fn top_words(words: &[String], n: usize) -> Vec<(String, usize)> {
    let counts = word_counts(&words);

    let mut pairs: Vec<(String, usize)> = counts.into_iter().collect();
    pairs.sort_by(|a, b| b.1.cmp(&a.1));
    pairs.truncate(n);
    pairs
}
```

The `top_words()` function takes the word tokens as input (in the `words` argument) and the number of top words requested (in the `n` argument), and returns a vector of type `Vec<(String, usize)>`, which is a vector of tuples where the first element is the word and the second is the frequency.

The function first calls `word_counts()` to get the hash map of word frequencies, which is stored in the `counts` variable. Next the mutable variable `pairs` takes ownership of `counts` via `into_iter()`, and the `collect()` method transforms that iterator into the vector of tuples.

The function then uses `sort_by()` to sort the vector based on the frequencies (represented in the closure by the tuple values `a.1` and `b.1`). Finally, the vector is truncated to the first `n` elements and then returned.

Finding the longest word

This project's final analysis task is to find the longest word in the text. That's the job of the `longest_word()` function:

```
fn longest_word(words: &[String]) -> String {
    let longest = words.iter().max_by_key(|w| w.len());

    match longest {
        Some(word) => word.to_string(),
        None => "".to_string(),
    }
}
```

The `longest_word()` function takes the word tokens as input (in the `words` argument) and returns a `String` representing the longest word. To determine the longest word, the function turns to the `max_by_key()` method:


```
iterator.max_by_key(closure)
```

where *iterator* is the iterator to run through and *closure* is a key extraction function that derives a property from the iterator values and determines the maximum value based on those property values. In this case, the property is word length, so the following closure is used:

```
|w| w.len()
```

The `max_by_key()` method returns an `Option` where the `Some` result wraps the word that has the maximum length, which is what the function returns in the `match` statement's `Some` arm. If the `Option` returns `None`, for some reason, the `None` arm returns the empty string.

Putting it all together

Okay, now you can pull all these functions together in the project's `main()` function:

```
fn main() {
    // Here's the text to analyze
    let text = r" Miss Brooke had that kind of beauty
                which seems to be thrown into relief
                by poor dress.
                ...";

    println!("");
    println!("=====");
    println!("Word Wrangler");
    println!("=====");

    let words = tokenize(text);

    let top = top_words(&words, 10);
    println!("\nMost frequent words:");
    for (word, count) in top {
        println!("{word:>10}: {count}");
    }

    let longest = longest_word(&words);
    println!("\nLongest word:");
    println!("{longest} ({} letters)\n", longest.chars().count());
}
```

The text (only partially shown here) is stored as a raw string in the `text` variable. The project title is displayed, and then the `tokenize()` function

is called to get the tokens. The `top_words()` function is called, and the list of the ten most frequent words is displayed. Finally, the `longest_word()` function is called, and the longest word is displayed.

Here's the output:

```
=====
Word Wrangler
=====

Most frequent words:
    her: 76
    she: 55
    celia: 36
    you: 33
    dorothea: 33
    them: 29
    which: 25
    would: 23
    said: 19
    sister: 16

Longest word:
unenthusiastic (14 letters)
```

Chapter 7

Organizing Your Code

IN THIS CHAPTER

- » Traversing the module tree
 - » Organizing single-file projects
 - » Organizing multiple-file projects
 - » Making code public
 - » Importing external crates
-

I'm seeking peace and security amid a nightmare of chaos. I want to help programmers sleep well, worry less.

—GRAYDON HOARE (INVENTOR OF RUST)

If you're like most Rust coders, when you work on a significant Rust program, your project quickly morphs from the cute as a button “Hello, world!” default that Cargo creates for you to something that feels more like a semicolon-festooned Karl Ove Knausgaard novel. Functions appear randomly before, after, and even *inside* `main()`. Variable declarations are scattered all over the place. And it's starting to feel like you need to hire a private detective to locate that one struct you need to modify.

When your `main.rs` file starts to remind you of the bedroom of a messy teenager (I know: oxymoron), it means two things. First, it means that you're completely normal because trust me when I tell you that *every* Rust coder feels this way at some point in a project. Second, it means that you've come to this chapter in the nick of time. Over the next few pages, you discover the Rust features and tools that can help transform your disheveled, disorganized, and discombobulated code into a trim, tidy, and together project.

How Rust Treats Code Organization

Before getting into both the nitty and the gritty of organizing your code, it's a good idea to take a step back so that you can take in how Rust looks at code organization.

The big picture: Packages, crates, and modules

One of the hallmarks of a well-planned town is clearly defined neighborhoods, streets, and addresses. Rust Town is such a place because it defines three main code organization concepts:

- » **Crate:** A compilation unit, the chunk of code that Rust compiles together. A *binary* crate compiles into an executable program you can run. A *library* crate compiles into code that other crates can use. The crate itself is the source code; the executable or library is what gets produced from it. When you run `cargo build`, what you're building is a crate. A crate is the equivalent of an individual building on a block or in a neighborhood.
- » **Package:** A directory with a `Cargo.toml` file. When you run `cargo new my_project`, you're creating a package housed in the `my_project` directory. A package might contain a single crate, but packages can also contain multiple related crates, if needed. A package, then, is akin to a city block that can contain multiple buildings.
- » **Module:** A named container for related code. Unlike a function (which performs a single task), a module groups together multiple related items, such as functions, enums, structs, constants, and even other modules. A module, then, is like a room inside a building.

The module is where the bulk of your day-to-day code organization life happens, and it's where this chapter spends most of its time.

The module tree: Your code's family tree

Rust's code organization begins with an upside-down treelike structure for organizing modules. It's kind of like a family tree but for project

code. At the top of the tree is the *crate root module* (usually called just the *crate root*), which for applications is usually `main.rs` but for libraries is usually `lib.rs`. Every other module exists as a branch off this root. Rust also enables you to nest one module inside another, and these nested modules exist as sub-branches of the tree.

[Figure 7-1](#) shows what a module tree might look like conceptually for a hypothetical app. Each item in this tree is a module. As you'll see later, modules can exist as blocks of code within a file, or they can be split out into their own files.

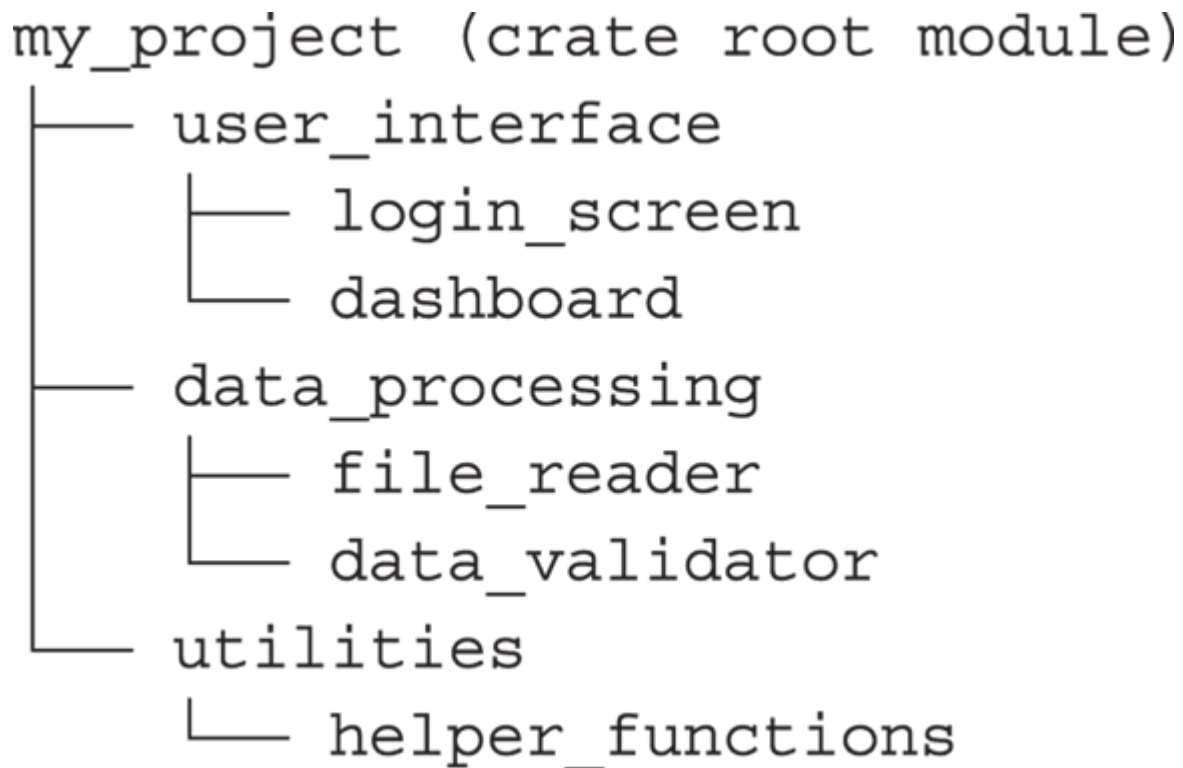


FIGURE 7-1: A module tree for a hypothetical app.

For example, the `login_screen` module might contain functions such as `display_login_form()` and `validate_credentials()`, along with a `LoginError` enum. The tree shows how these modules relate to each other hierarchically. For example, `login_screen` is a submodule of `user_interface`, which is a submodule of the crate root.

This tree structure isn't just a way of organizing code so that it's not so cluttered, although that's a bonus. No, this structure determines how you

access different parts of your program and controls which parts of your program can read and work with other parts.

Files and modules are not necessarily the same

One concept that flummoxes a lot of new Rust coders is that a file (such as `main.rs`) and a module aren't usually one and the same thing. I think the problem lies in the trivial case where a project consists of just `main.rs`. In that case, `main.rs` serves as both the project file and the crate root module.

But that one-file-one-module simplicity goes out the window in a hurry once you start organizing your code:

- » You can have multiple modules in a single file.
- » You can have a single module split across multiple files.



REMEMBER The key takeaway here is that, from Rust's perspective, files aren't important but modules are. A file is just a container. The organizing structure in your projects is the module.

Starting Small: Organizing Code in One File

When you're just getting started with Rust and your projects are still small, all your code might reside in `main()`. But once you start adding functions, structs, constants, and other code outside `main()`, it won't take long before you feel the need for organization. Step back and examine your code. Are there multiple related structures, such as multiple functions related to handling user input or a group of structs, constants, and functions related to mathematical operations? If the answer is yes, these related groups of code are your first organization targets. You organize them by corralling them into modules. And since your project

is still relatively simple at this stage, all this module-making can happen in `main.rs`.

The `main.rs` versus `lib.rs` decision

So far in this book, every project has stored its code in `main.rs`.

However, that doesn't mean that `main.rs` is your only option. Later in this chapter, you learn that your projects can have a bunch of files (refer to “[Growing Up: Moving to Multiple Files](#)”). For now, though, we're sticking with one file, which has been `main.rs` up to now because that's what you get when you run `cargo new`. This file is perfect for programs that you run directly. But sometimes you want to create a *library*, which is code that other people can use in their projects. Library code goes into a file named `lib.rs`, which you store in your project's `src` directory.

How do you decide?

Use `main.rs` when

- » You're building an application that people will run.
- » You have a clear “start here” entry point for the program.
- » You're making a CLI tool, web server, or any executable program.

When you have `main.rs`, running `cargo run` builds the executable file and runs it.

Use `lib.rs` when

- » You're building a library for other developers to use.
- » You want to expose functionality to other code.
- » You're planning to publish your crate on `crates.io`.

When you have `lib.rs`, running `cargo build` produces a library that other parts of your project (or even other people if you publish it) can use.

Happily, you also have the option of using both `main.rs` and `lib.rs` if you're building an application that also exposes library functionality. I

talk more about this later in the “[Growing Up: Moving to Multiple Files](#)” section.

Creating modules in main.rs

An *inline module* is a module declared directly in the code of a Rust file by using a `mod` block:

```
mod name {  
    // Module code goes here  
}
```

where *name* is the name of the module. The module code can be pretty much anything you’d put in `main()`, including functions, structs, enums, and constants.

Up to now, you’ve been writing functions directly in `main.rs` without wrapping them in modules, which works fine for small programs. But as your code grows, modules help you organize related code together and control which parts are accessible from where. When you put code inside a `mod` block, you’re creating a boundary: Code inside the module can freely use everything in that module, but code outside the module can access only items you’ve explicitly marked as `public`.

To ensure that your module code is visible to the rest of the code in `main.rs`, you need to precede the item’s definition with the keyword `pub`, for *public*. Here’s an example (`bk02ch07/example01/` in this book’s example files):

```
// main.rs  
mod greetings {  
    pub fn say_hello() {  
        println!("Hello from an inline module!");  
    }  
}
```

Now your code can call the module’s `say_hello()` function (`bk02ch07/example01/`, continued):

```
// main.rs  
fn main() {  
    greetings::say_hello();  
}
```


Output:

```
Hello from an inline module!
```

Note that you call a module function using the following syntax:

```
module::function();
```

where *module* is the name of the module block and *function* is the name of the function.

Within a `mod` block, declaring something with `pub` makes that something available to the rest of your code: `pub fn` for a function; `pub const` for a constant; `pub enum` for an enum; and `pub struct` for a struct. Note, however, that `pub struct` alone doesn't make the data fields within the struct public. You need to add the `pub` keyword to each field, as well:

```
mod models {  
    pub struct User {  
        pub name: String,  
        pub age: u8,  
    }  
}
```



REMEMBER Not everything inside a module has to be made public via the `pub` keyword. You'll often have data or functions inside a module that are used only by code in that module. In these cases, you can keep the data or function private by leaving off the `pub` keyword. For more on public versus private code, check out “[Controlling Access: Public versus Private](#),” later in this chapter.

Nesting one module inside another

If you just toss your shirts anywhere in your bedroom, can you confidently declare that your shirts are organized? I hope for the sake of the people you live with that the answer is “No”! Sure, it's best to keep your shirts in the bedroom, but to organize them you have to hang them up in a closet.

This room-within-a-room idea (depending, I guess, on the size of your closet) is also the idea behind nesting modules. That is, sometimes it's not enough to slot, say, functions A and B into one `mod` block and functions C and D into another. If those two modules are related, you should nest them within another `mod` block for better code organization.

For example, consider the following code (bk02ch07/example02/):

```
pub fn display_app_title() {
    println!("=====");
    println!("Dog Survey App");
    println!("=====");
}

mod input {
    pub fn get_user_name() -> String {
        // Actual function code goes here
        String::from("Chase")
    }

    pub fn get_favorite_activity() -> String {
        // Actual function code goes here
        String::from("Fetch")
    }
}

mod output {
    pub fn display_welcome(name: &str) {
        println!("Welcome, {name}! Are you ready to take the survey?");
    }

    pub fn display_entry(text: &str) {
        println!("You entered {text}!");
    }
}

fn main() {
    display_app_title();

    let name = input::get_user_name();
    output::display_welcome(&name);

    let fave_activity = input::get_favorite_activity();
    output::display_entry(&fave_activity);
}
```

This code is fine, but it could be better organized because the `display_app_title()` function and both the `input` module and the `output` module are related to the app's user interface. This means you can create a new inline module named, say, `user_interface`, which can hold the `display_app_title()` function and nest the `input` and `output` modules inside it, like so (bk02ch07/example03/):

```
mod user_interface {
  pub fn display_app_title() {
    println!("=====");
    println!("Dog Survey App");
    println!("=====");
  }

  pub mod input {
    pub fn get_user_name() -> String {
      // Actual function code goes here
      String::from("Chase")
    }

    pub fn get_favorite_activity() -> String {
      // Actual function code goes here
      String::from("Fetch")
    }
  }

  pub mod output {
    pub fn display_welcome(name: &str) {
      println!("Welcome, {name}! Are you ready to take the survey?");
    }

    pub fn display_entry(text: &str) {
      println!("You entered {text}!");
    }
  }
}

fn main() {
  user_interface::display_app_title();

  let name = user_interface::input::get_user_name();
  user_interface::output::display_welcome(&name);

  let fave_activity = user_interface::input::get_favorite_activity();
  user_interface::output::display_entry(&fave_activity);
}
```

Note, first, that `mod input` and `mod output` are now `pub mod input` and `pub mod output`, which makes these modules available to the rest of the code. Second, the path to each function now begins with the `user_interface` module. For example:

```
user_interface::output::display_welcome();
```

This path tells Rust to first look in the `user_interface` module, then look in the `output` submodule, and then find the `display_welcome()` function.

As your project grows, examine your modules and constantly ask yourself whether two or more modules are related in some logical way and therefore can be made submodules within a new module. Organizing your modules and submodules this way will make the move to multiple files (the topic of the next section) much easier.

Growing Up: Moving to Multiple Files

If you find that you spend great chunks of coding time scrolling vertically in search of a function or struct, that sense of scroll fatigue is a sign that it's time to split your project into multiple files.

Using `main.rs` and `lib.rs`

For many Rust projects, it makes a lot of sense to have *both* `main.rs` and `lib.rs`, where the code in `main.rs` can call into the module code defined in `lib.rs`. This project pattern is popular because

- » You can store and test most of your program's logic in the library.
- » You can reuse that logic in other projects.
- » Your `main.rs` stays small, usually just argument parsing and orchestrating the program flow.

Mapping rules for module files

The great thing about breaking out your modules into their own files is that, aside from moving the modules into their file, you don't have to change anything else about your code. In particular, all your module paths of the form `module::submodule::item` will still work because Rust maps that path to the module file tree.

Here are the mapping rules:

- » **Top-level modules:** A declaration such as `mod foo;` (note the semicolon instead of curly braces) tells the compiler to look for this module's code in a separate file. In this case, the compiler would look for `foo.rs` in the same directory.
- » **Submodules:** A declaration such as `mod bar; inside foo.rs` tells the compiler to look for the submodule's code in `foo/bar.rs` (that is, in a subdirectory named after the parent module).

The pattern is consistent in that the directory structure always mirrors the module hierarchy. For example, if `bar` had its own submodule `baz`, that code would go in `foo/bar/baz.rs`.

Creating your own module files

As an example, I'm going to take the Dog Survey code from the preceding section and divide it into multiple files. [Figure 7-2](#) shows the module tree I ended up with.

```
example04/
├─ Cargo.toml
└─ src/
    ├─ lib.rs           // the library crate root
    ├─ user_interface.rs // module file for 'mod user_interface;'
    ├─ user_interface/
    │   ├─ input.rs      // submodule file for 'pub mod input;'
    │   └─ output.rs     // submodule file for 'pub mod output;'
    └─ main.rs          // small binary using the library
```

```

example04/
├── Cargo.toml
└── src/
    ├── lib.rs           // the library crate root
    ├── user_interface.rs // module file for `mod user_interface;`
    └── user_interface/
        ├── input.rs      // submodule file for `pub mod input;`
        ├── output.rs     // submodule file for `pub mod output;`
        └── main.rs       // small binary using the library

```

FIGURE 7-2: A module tree for the Dog Survey code.

First, `lib.rs` just includes a reference to the `user_interface` module (all of the following files are in `bk02ch07/example04/`):

```

// lib.rs

pub mod user_interface

```

Now the compiler knows to look in `user_interface.rs` for the `user_interface` module:

```

// user_interface.rs

pub fn display_app_title() {
    println!("=====");
    println!("Dog Survey App");
    println!("=====");
}

pub mod input;
pub mod output;

```

For the `input` and `output` submodules, the compiler looks for `user_interface/input.rs` and `user_interface/output.rs`:

```

// user_interface/input.rs

pub fn get_user_name() -> String {
    // Actual function code goes here
    String::from("Chase")
}

pub fn get_favorite_activity() -> String {
    // Actual function code goes here
}

```

```

    String::from("Fetch")
}

// user_interface/output.rs

pub fn display_welcome(name: &str) {
    println!("Welcome, {name}! Are you ready to take the survey?");
}

pub fn display_entry(text: &str) {
    println!("You entered {text}!");
}

```

Finally, the only change required in `main.rs` is to import the `user_interface` module via the `use` keyword at the top of the file:

```

// main.rs
use example04::user_interface;

fn main() {
    user_interface::display_app_title();

    let name = user_interface::input::get_user_name();
    user_interface::output::display_welcome(&name);

    let fave_activity = user_interface::input::get_favorite_activity();
    user_interface::output::display_entry(&fave_activity);
}

```

Controlling Access: Public versus Private

When you're just getting your feet wet with Rust coding, all your programs will be internal in the sense that they'll exist only on your computer and only you will run them. Many Rust coders never get beyond this stage. They just have fun building programs for their own use. That's awesome!

At some point in your Rust adventure, however, you might get a hankering to share your code with others. The code could be a command-line utility, an API, a server, or a web app (all of which you learn to build later in the book). Maybe you want to share a library crate

that you think other Rust programmers might find useful. Whatever you create, that code will now be external in the sense that other people will be able to run it or access its data.

When you're ready to send code out into the world, you have to start thinking seriously about privacy and be extremely clear about what you want to expose publicly.

Understanding private by default

One of the attributes that makes Rust an admirable language is its sensible and safe defaults. For example, every bit of data you use in a program is immutable by default; to make some chunk of data mutable, you must deliberately add the `mut` keyword.

Another of Rust's sensible choices is that everything in a Rust program is private by default, which means that everything in a module can be accessed only by code within that same module. This privacy-first approach is a refreshing change from languages that take the opposite route and make *everything* public by default, which often leads to tightly coupled code that's hard to change. When everything is public, other code can depend on implementation details that you might want to change later. By making things private by default, Rust encourages you to think carefully about your module's interface — the public parts that other code can rely on — and keeps implementation details hidden so you can improve them without breaking anything.



REMEMBER Rust uses the terms *privacy* and *visibility* interchangeably. A public item is visible to code outside the module in which it's defined. A private item is not visible to external code.

Just like with data, where you must deliberately make things mutable by adding `mut`, with module code you must deliberately make things public by adding the `pub` keyword.

What private by default means, in practice, is that *you* decide what should be exposed publicly, knowing that everything else remains

private without any extra effort on your part. Check out this example (bk02ch07/example05/):

```
mod bakery {
  fn secret_recipe() -> String {
    // This function is private, so only code
    // inside the bakery module can call it
    "The secret ingredient is... love! \
    (And also butter. So much butter.)"
    .to_string()
  }

  pub fn make_cookies() -> String {
    // This function is public, so code
    // outside the bakery module can call it
    let recipe = secret_recipe();
    format!("Making cookies using the secret recipe. Shhh.")
  }
}

fn main() {
  println!("{}", bakery::make_cookies()); // This works!

  //println!("{}", bakery::secret_recipe()); // This wouldn't!
}
```

In the `bakery` module, the `make_cookies()` function has the `pub` keyword, so code that's outside the module can call it. That's why the first `println!` statement in `main()` works just fine.

However, the `secret_recipe()` function doesn't have the `pub` keyword, so it's private to the `bakery` module. Code outside the module doesn't even know that this function exists. If you were to uncomment the second `println!` statement in the `main()` function and try to run the program, the compile would fail with the following message:

[illegible]

Exposing only what's necessary when designing module interfaces

When you're creating a module interface, the main rule to follow is a straightforward one: Make public only that which other parts of your program (or external code sources) need to use. Everything else should stay private. Here's an example (bk02ch07/example06/):

```
mod user_account {
    // Public, but you can control visibility of individual fields
    pub struct User {
        pub username: String, // Public: Code outside the module can access
it
        email: String,        // Private: Only this module can access
directly
        password_hash: String, // Private: Definitely don't want this
exposed!
    }

    impl User {
        // Public: Code outside this module can call this function
        pub fn new(username: String, email: String, password: String) -> User
{
            User {
                username,
                email,
                password_hash: hash_password(password), // Calls private
method
            }
        }
        // Public: Code outside this module can call this method
        pub fn get_email(&self) -> &str {
            &self.email // This provides controlled access to a private field
        }
    }

    // Private: Only code in this module can call this function
    fn hash_password(password: String) -> String {
        format!("hashed_{}", password)
    }
}
```



REMEMBER When you're deciding what to expose and what to keep private, be sure to keep the following rules in mind:

- » When you make a struct public, its fields remain private. To expose a struct field, you must add the `pub` keyword to the field.
- » When you make an enum public, all its variants are automatically made public as well. Rust doesn't offer a way to control the visibility of individual enum variants.

Importing and Using Code

Now that you know how to organize code into modules and how to break out code into separate files when the time is right, you need to understand how to use your code once you've performed these organizational tasks. This is where the `use` keyword comes in because you use it to tell the compiler where to find your modules and their public items.

Importing code with the use keyword

The `use` keyword brings names into the current scope so that you don't have to write super-long module paths everywhere. For example, earlier (refer to `bk02ch07/example04/` in the “[Creating your own module files](#)” section) I broke up `main.rs` into multiple files. Here's one way I could have structured the code in `main()` to access the public items in the modules and submodules:

```
fn main() {
    example04::user_interface::display_app_title();

    //let name = user_interface::input::get_user_name();
    let name = example04::get_user_name();
    example04::user_interface::output::display_welcome(&name);

    let fave_activity =
        example04::user_interface::input::get_favorite_activity();
```

```
example04::user_interface::output::display_entry(&fave_activity);  
}
```

Wow, there are some crazy-long module paths in there! Since every path has `example04::user_interface` in common, what I did instead was bring the `user_interface` module into this scope with the `use` keyword:

```
use example04::user_interface;  
  
fn main() {  
    user_interface::display_app_title();  
  
    let name = user_interface::input::get_user_name();  
    user_interface::output::display_welcome(&name);  
  
    let fave_activity = user_interface::input::get_favorite_activity();  
    user_interface::output::display_entry(&fave_activity);  
}
```

Now the module paths can refer to `user_interface` directly, which shortens them a bit.

Shortening the module paths

In the preceding section, when I added `use example04::user_interface;`, I shortened the module paths, but they're still quite long. To save wear-and-tear on your typing fingers, here are a few ways you can shorten your module paths:

- » **Import the module with a shorter alias.** When you have a longish module name such as `user_interface`, you can import it with `root::module as alias`, where *alias* is a shorter name to use in your code:

```
use example04::user_interface as ui;  
  
fn main() {  
    ui::display_app_title();  
    let name = ui::input::get_user_name();  
    ui::output::display_welcome(&name);  
    let fave = ui::input::get_favorite_activity();  
    ui::output::display_entry(&fave);  
}
```

- » **Import the submodules.** When your code requires access to submodules, you can import them directly with `root::module::` {`sub1`, `sub2`, ...}, where `sub1`, `sub2`, and so on are the submodule names. You can also use `self` to refer to `module` if you want to specify an alias. For example:

```
use example04::user_interface::{self as ui, input, output};

fn main() {
    ui::display_app_title();
    let name = input::get_user_name();
    output::display_welcome(&name);
    let fave = input::get_favorite_activity();
    output::display_entry(&fave);
}
```

- » **Import the specific module and submodule elements.** You can get rid of module paths altogether by including in your `use` statement all the elements used by your code, such as all the functions that are called:

```
use example04::user_interface::{
    display_app_title,
    input::{get_user_name, get_favorite_activity},
    output::{display_welcome, display_entry},
};

fn main() {
    display_app_title();
    let name = get_user_name();
    display_welcome(&name);
    let fave = get_favorite_activity();
    display_entry(&fave);
}
```

In case you're wondering: yep, you can alias any element you import.



WARNING When you're importing, watch out for naming conflicts. When you import items with `use`, those names become part of your current scope. If you import two items with the same name from

different modules, you'll get a compiler error. For example, if both `audio::play()` and `video::play()` exist and you try to import both as just `play`, Rust won't know which one you mean. You can work around this problem in three ways:

- » Keep the module prefix: `audio::play()` and `video::play()`.
- » Use aliases, such as `use audio::play as play_audio;`.
- » Import only one directly and use the full path for the other.

Working with External Dependencies

In programming, a *dependency* is software that your program relies on to function. Without that software, your program would be incomplete or fail to run, so in a direct sense your program *depends on* that software.

An *external dependency* is a dependency that you didn't code yourself. Instead, it comes from some third-party source. Instead of wasting precious time reinventing software wheels, smart coders use external dependencies to handle tasks that would otherwise take hours, days, or even weeks to write from scratch.

In the world of Rust, dependencies come in the form of library crates — usually referred to as just *crates* — that contain prewritten functionality. You can search for and read about the roughly 200,000 available crates by asking a nearby web browser to take you to crates.io. Use the search box to find a particular crate, or browse the crates to see if anything strikes you as useful.

Adding a crate as a project dependency

Once you find a crate that you want to use as a dependency in your project, open a terminal window, `cd` into your project directory, and then run the following command:

```
cargo add crate
```

where *crate* is the name of the crate. For example, to add the `rand` crate, which is a popular crate for generating pseudorandom numbers, you'd run the following command in your project directory:

```
cargo add rand
```

A bunch of text flashes by quickly, which is Cargo installing not only `rand` but also the crates that `rand` itself depends on! When the smoke clears, open your project's `Cargo.toml` file in your text editor. That file will now include lines similar to the following:

```
[dependencies]
rand = "0.9.2"
```

This tells you that Cargo installed `rand` version 0.9.2 as a dependency in your project. Yes, it's as easy as that!

Importing the dependency

With the dependency installed and added to `Cargo.toml`, your next step is to bring the crate into your project's scope with our old pal, the `use` keyword. On crates.io, most crates have a link either to their documentation or to a repository that describes the crate. Either way, the crate documentation will tell you what's available for importing. Your `use` statement will look something like one of the following:

```
use crate::item;
use crate::{item1, item2,...};
use crate::*;
use crate::submodule::item;
use crate::submodule::{item1, item2,...};
use crate::submodule::*;
```

where *crate* is the name of the crate; *submodule* is the name of a public submodule within *crate*; and *item*, *item1*, *item2*, and so on are specific public elements what you want to import, such as functions. The global (*) pattern brings in everything, which saves typing, but it hides where names come from, increases the risk of name collisions (where two or more crates export the same name), and makes code fragile as crates evolve (because you never know when a crate might change its public

items). In most real-world Rust code, explicit imports are clearer, safer, and more maintainable.

For the `rand` example, we need to import the `Rng` trait, which offers all the functionality we'll need. Here's a project that uses `rand` to simulate rolling a six-sided die (bk02ch07/example07/):

```
use rand::Rng;

fn main() {
    // Create a random-number generator
    let mut rng = rand::rng();

    // Roll a six-sided die
    let roll: u8 = rng.random_range(1..=6);

    println!("You rolled a {roll}!");
}
```

Sample output:

```
You rolled a 6!
```

Real-World Organization Patterns

As your projects grow beyond simple examples, you'll need more sophisticated organization strategies. The next few sections take you through some patterns that work well for real applications.

Organizing larger applications

[Figure 7-3](#) shows a typical structure for a medium-sized Rust application (bk02ch07/my_app/).

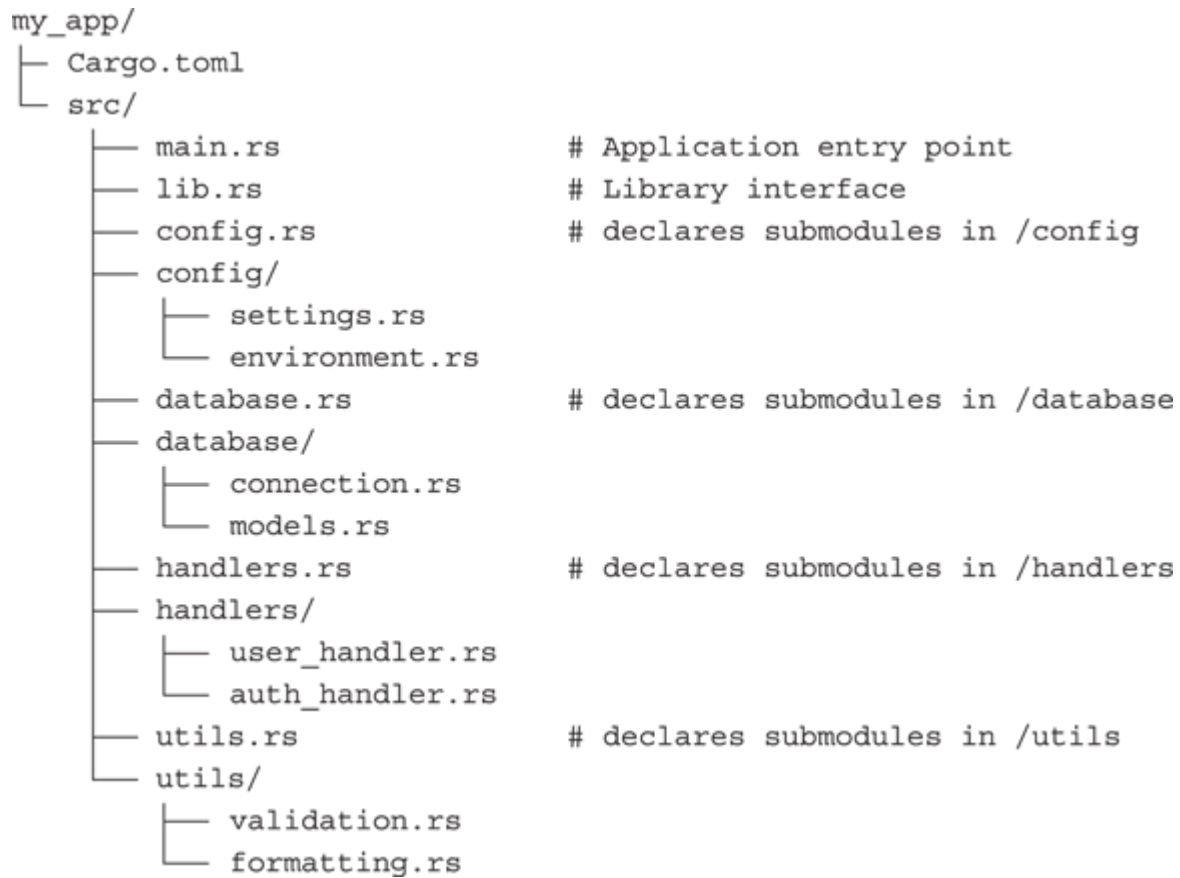


FIGURE 7-3: A typical directory structure for a medium-sized Rust app.

Each directory represents a major functional area of the program.

`lib.rs` contains the module declarations:

```
// src/lib.rs

pub mod config;
pub mod database;
pub mod handlers;
pub mod utils;
```

The four module files — `config.rs`, `database.rs`, `handlers.rs`, and `utils.rs` — act as the front doors to each area:

`config.rs` declares the submodules in the `/config` subdirectory:

```
// src/config.rs

pub mod settings;
pub mod environment;
```

`database.rs` declares the submodules in the `/database` subdirectory:

```
// src/database.rs

pub mod connection;
pub mod models;
```

`handlers.rs` declares the submodules in the `/handlers` subdirectory:

```
// src/handlers.rs

pub mod user_handler;
pub mod auth_handler;
```

`utils.rs` declares the submodules in the `/utils` subdirectory:

```
// src/utils.rs

pub mod validation;
pub mod formatting;
```

And `main.rs()` imports what it needs:

```
// src/main.rs

use my_app::config;
use my_app::database;
use my_app::handlers;
use my_app::utils;

fn main() {
    println!("Starting my_app...");

    config::settings::print_settings();
    database::connection::connect();
    handlers::user_handler::handle_user();
    utils::validation::validate();
}
```

Re-exports and crate interfaces

A *re-export* is when you publicly import (that is, declare with `pub use`) an item from a module or submodule at a higher level, such as the crate root. Re-exports are a powerful way to create clean, easy-to-use interfaces for your modules. Instead of forcing users to navigate deep

module hierarchies, you can bring important items to more convenient locations.

A common re-export location is `lib.rs` (bk02ch07/my_app2/):

```
// src/lib.rs

pub mod config;
pub mod database;
pub mod handlers;
pub mod utils;

// Re-export commonly used items at the crate root
pub use config::settings::print_settings;
pub use database::connection::connect;
pub use handlers::user_handler::handle_user;
pub use utils::validation::validate;
```

Now, in `main.rs`, you can import those items and call them directly:

```
// main.rs

use my_app2::{print_settings, connect, handle_user, validate};

fn main() {
    println!("Starting my_app...");

    print_settings();
    connect();
    handle_user();
    validate();
}
```

Splitting a project into multiple crates

Sometimes a single crate isn't enough. You might want to split your project into multiple crates when

- » Your project has reusable components that other projects could make good use of.
- » Your compile times are getting slow. When you split code into separate crates, Cargo recompiles only crates that have changed.
- » Your program has distinct functional areas that rarely interact.

- » You want to publish part of your code while keeping other parts private.

[Figure 7-4](#) shows how you might organize a larger project with multiple crates.

```
my_awesome_project/
├── Cargo.toml                # Workspace configuration
├── app/                      # Main application crate
│   ├── Cargo.toml
│   └── src/
│       └── main.rs
├── core/                    # Shared business logic
│   ├── Cargo.toml
│   └── src/
│       └── lib.rs
└── utilities/               # Common utilities
    ├── Cargo.toml
    └── src/
        └── lib.rs
```

FIGURE 7-4: A larger Rust project with multiple crates.

The workspace `Cargo.toml` file ties everything together:

```
[workspace]
members = ["app", "core", "utilities"]

[workspace.dependencies]
rand = "0.9.2"
```

Book 3

Deeper Rust Concepts

Contents at a Glance

Chapter 1: Traits: Shared Behaviors

[Traits Are Promises](#)

[When Your Types Won't Behave](#)

[Deriving: Traits in the Fast Lane](#)

[Behave Yourself: Implementing Traits for Your Own Types](#)

[Printing Prettier with the Display Trait](#)

[Setting Sensible Defaults with the Default Trait](#)

[Project: Building a Media Library with Traits](#)

Chapter 2: Generics: Code That Works for Almost Anything

[Why Write the Same Code Twice?](#)

[Introducing Generics](#)

[Type Bounds and Constraints](#)

[Traits and Generics = Chef's Kiss](#)

[Project: Build a Generic Logger That Works with Any Type](#)

Chapter 3: Lifetimes: How Long Things Live

[Introducing Lifetimes](#)

[When Rust Demands Lifetime Annotations](#)

[Adding Lifetime Annotations to Functions](#)

[Lifetimes in Structs: Keeping References Alive](#)

[Lifetime Elision: When the Compiler Does the Work](#)

[The 'static Lifetime: Data That Lives \(Sort of\) Forever](#)

[Project: A Zero-Copy Config Parser](#)

Chapter 4: Handling Errors Like a Pro

[Recoverable versus Unrecoverable: Know When to Panic!](#)

[Result<T, E>: Your Go-to Type for Errors](#)

[Saving Time with Helper Methods](#)

[Propagating Errors with the ? Operator](#)

[Grokking Rust's Error Types](#)

[Magically Creating a Result from an Option](#)

[Creating Your Own Error Types](#)

Chapter 5: Macros: When Code Writes Code

[Useful Macro Magic: println! and vec!](#)

[Using Macros in Your Own Code](#)

[Macro Syntax: Learning to Read the Hieroglyphics](#)

[Writing Macros](#)

[Project: Building a Timer Macro](#)

Chapter 6: Parallelism and Concurrency: Rust Multitasking

[Parallelism versus Concurrency: What's the Difference?](#)

[Spawning Threads: Your First Parallel Adventure](#)

[Passing Messages through Channels](#)

[Sharing State Safely with Arc and Mutex](#)

[Borrowing Data with Scoped Threads](#)

[Iterating in Parallel with Rayon](#)

[Async/Await: Concurrency without the Thread Overhead](#)

[Project: Collatz Conjecture Explorer](#)

Chapter 7: Testing, Debugging, and Documenting

[Testing Your Code](#)

[Debugging Your Code](#)

[Cleaning Your Code](#)

[Documenting Your Code](#)

Chapter 1

Traits: Shared Behaviors

IN THIS CHAPTER

- » Solving compiler errors by using traits
 - » Deriving traits automagically
 - » Implementing traits for custom types
 - » Making a media library with traits
-

Rust is uniquely positioned to give advantages that I can't get from other languages. It gives you multiple superpowers in one language.

—SHANE MILLER

In psychology, a *behavioral trait* is a pattern of behavior that remains consistent across different situations. Some people are always punctual, others can't seem to stop with the jokes, and still others are annoyingly organized. No matter what the circumstance, these traits rise to the surface.

In Rust, a *trait* is a defined behavior that's consistent across different types. That is, instead of describing how people behave, a Rust trait describes how *types* behave. Some types can be copied, others can be compared, and still others can be printed. These traits are behaviors shared by many different types, so they act as a kind of glue that binds the various parts of the Rust model. Remarkably, as you learn in this chapter, you can give your own types these same behaviors, so your custom types become full-fledged denizens in the Rust ecosystem.

Traits Are Promises

If you've programmed in other languages, you might know traits by another name: *interfaces*. Like interfaces in Java or C#, a Rust trait defines a kind of contract, a promise that any type implementing the trait will provide certain capabilities. The difference is that Rust traits are more flexible: You can add traits to types you didn't create, and traits can provide default implementations.

But you don't need to understand all that right now. The best way to understand traits is to see them in action, starting with some common situations where the Rust compiler tells you a trait is missing.

When Your Types Won't Behave

Back in Book 2, [Chapter 2](#), I introduce you to structs, which you use to build custom Rust types with the sweat of your own brow. Crafting custom types is an endlessly useful Rust technique, but often your types won't behave the way you might think they should. Here's an example:

```
struct Movie {  
    title: String,  
    year: u32,  
}
```

That definition is correct, but it's missing some crucial Rustian fairy dust that limits what you can do with it.

When your type won't display

After defining the `Movie` type, you create an instance and then try what seems like the most natural thing: printing the data to `stdout` (bk03ch01/example01/):

```
fn main() {  
    let movie = Movie {  
        title: "The Matrix".to_string(),  
        year: 1999,  
    };  
    println!("{}", movie);  
}
```

But the compiler refuses to participate:


```
error[E0277]: 'Movie' doesn't implement 'std::fmt::Display'
```

Hunh? A bit later in the error message, the compiler elaborates:

```
help: the trait 'std::fmt::Display' is not implemented for 'Movie'
```

What’s going on here is that the `println!` macro works with only types that have a specific behavior: the capability to format themselves as text for display. In Rust lingo, `println!` demands that each type it works with implement the `Display` trait. That trait is defined in the standard library’s `fmt` module, which contains utilities for formatting and printing strings.

The compiler is telling you that you haven’t implemented the `Display` trait for your custom `Movie` type, so the compiler doesn’t know how to print your instance. You learn how to implement the `Display` trait for your custom types a bit later in the “[Printing Prettier with the Display Trait](#)” section.

When your type won’t compare

As another example, one common technique with custom types is to create a clone so that you can work with two separate instances. Take a look (bk03ch01/example02/):

```
fn main() {
    let movie1 = Movie {
        title: "The Matrix".to_string(),
        year: 1999,
    };
    let movie2 = Movie {
        title: "Dark City".to_string(),
        year: 1998,
    };
    if movie1 == movie2 {
        println!("Yep, these are the same movie!");
    } else {
        println!("Nope, these aren't the same movie!");
    }
}
```

All that this code is trying to do is cheekily compare two `Movie` instances to see if they’re the same. The compiler doesn’t get the joke, however:

```

error[E0369]: binary operation '==' cannot be applied to type 'Movie'
--> src/main.rs:15:15
|
15 |         if movie1 == movie2 {
|             ^^^^^ ^^^ Movie
|             |
|             Movie
|
note: an implementation of 'PartialEq' might be missing for 'Movie'

```

What the compiler wants you to know here is that you can compare one type instance with another only if that type implements a trait called `PartialEq`, which is used for comparisons involving the equality (`==`) and nonequality (`!=`) operators. Helpfully, the compiler also tells you one way to implement the `PartialEq` trait:

```

help: consider annotating 'Movie' with '#[derive(PartialEq)]'

```

Head down to the “[Deriving: Traits in the Fast Lane](#)” section to learn what on Earth the compiler is talking about here.

When your type won't clone

Okay, one last example where the compiler produces a trait-specific error (bk03ch01/example03/):

```

fn main() {
    let movie1 = Movie {
        title: "The Matrix".to_string(),
        year: 1999,
    };
    let movie2 = movie1.clone();
    if movie1 == movie2 {
        println!("Yep, these are the same movie!");
    } else {
        println!("Nope, these aren't the same movie!");
    }
}

```

This code just tries to make a copy of the `Movie` instance using `clone()`, but the compiler takes out its red pencil:

```

error[E0599]: no method named 'clone' found for struct 'Movie' in the
current scope

```

And later:

```
= note: the following trait defines an item 'clone', perhaps you need
to implement it:
    candidate #1: 'Clone'
```

In this case, the compiler is telling you that your type doesn't implement a trait called `Clone`, which is used to create duplicates of things. I show you how to fix this later in the “[Deriving Clone for copying](#)” section.

The three behaviors discussed here — displaying, comparing, and cloning — are common Rust behaviors that are governed, respectively, by the `Display`, `PartialEq`, and `Clone` traits. These traits are implemented by default on many of Rust's standard library types, including `bool`, `char`, all the integer and floating-point types, and `String`. But, as shown in this section's code examples, these three traits are not automatically implemented on your custom types. Now it's time to learn how to fill that gap.

Deriving: Traits in the Fast Lane

As I describe in the section “[Behave Yourself: Implementing Traits for Your Own Types](#),” Rust offers a way to implement traits that are missing from your custom types. However, for common traits such as `Clone` and `PartialEq`, Rust offers an almost ridiculously easy way to add a trait to structs and enums: the `derive` attribute:

```
#[derive(trait1, trait2, ...)]
data_structure
```

where `trait1`, `trait2`, and so on are the names of the traits, and `data_structure` is the struct or enum for which you want to derive the traits.

The `#[...]` syntax is called an attribute in Rust. Unlike C preprocessor directives (which also start with `#`), Rust attributes are attached to specific items. That is, they must appear immediately before the struct, enum, or function they modify. Think of an attribute as a sticky note you place on your code to give the compiler special instructions about that particular item.

For example:

```
#derive[ (Debug, PartialEq) ]
struct my_type {
    // fields
}
```

This code tells Rust to automatically implement the `Debug` and `PartialEq` traits on the `my_type` struct.



REMEMBER Rust will derive a trait for a custom type only if that trait is derivable for every field in the custom type.

Rust can automatically derive a number of traits, but `Debug`, `PartialEq`, and `Clone` are the most common, so I cover them in this section.

Deriving Debug for display

Rust doesn't offer a way to automatically derive the `Display` trait (refer to the section "[Printing Prettier with the Display Trait](#)"), but while you're debugging your Rust programs, you can use the next best thing: the `Debug` trait:

```
#[derive (Debug) ]
```

When implemented on a custom type, `Debug` enables you to add `: ?` to a `println!()` placeholder to display the contents of any instance of the custom type. Here's an example (bk03ch01/example04/):

```
#[derive (Debug) ]
struct Movie {
    title: String,
    year: u32,
}

fn main() {
    let movie = Movie {
        title: "The Matrix".to_string(),
        year: 1999,
    };
    // This works now!
```

```
println!("{:?}", movie);  
}
```

Output:

```
Movie { title: "The Matrix", year: 1999 }
```

Deriving PartialEq for comparison

If you want to compare two custom type instances using the equality (`==`) or nonequality (`!=`) operator, you can derive the `PartialEq` trait:

```
#[derive(PartialEq)]
```

Here's an example (bk03ch01/example05/):

```
#[derive(PartialEq)]  
struct Movie {  
    title: String,  
    year: u32,  
}  
  
fn main() {  
    let movie1 = Movie {  
        title: "The Matrix".to_string(),  
        year: 1999,  
    };  
    let movie2 = Movie {  
        title: "Dark City".to_string(),  
        year: 1998,  
    };  
    // This works now!  
    if movie1 == movie2 {  
        println!("Yep, these are the same movie!");  
    } else {  
        println!("Nope, these aren't the same movie!");  
    }  
}
```

Output:

```
Nope, these aren't the same movie!
```

Deriving Clone for copying

If you want to copy a custom type instance using the `clone()` method, you can derive the `Clone` trait:

```
#[derive(Clone)]
```

Here's an example (bk03ch01/example06/):

```
#[derive(Clone, PartialEq)]

struct Movie {
    title: String,
    year: u32,
}

fn main() {
    let movie1 = Movie {
        title: "The Matrix".to_string(),
        year: 1999,
    };
    // This works now!
    let movie2 = movie1.clone();

    if movie1 == movie2 {
        println!("Yep, these are the same movie!");
    } else {
        println!("Nope, these aren't the same movie!");
    }
}
```

Output:

```
Yep, these are the same movie!
```

Note that this example requires deriving both `Clone` and `PartialEq` since the code both clones the instance and compares the original with the clone.

Behave Yourself: Implementing Traits for Your Own Types

Deriving traits, as I describe in the preceding section, requires almost no work on your part, which is pretty sweet. But `derive` doesn't help when you want your structs or enums to exhibit some custom behavior. For that you need to roll up your sleeves and build the required trait yourself.

To define a custom trait, you use the `trait` keyword:

```
trait Trait {  
    items  
}
```

where *Trait* is the name of the trait in PascalCase and *items* is one or more Rust items that specify the trait's behavior. These items are usually one or more methods and constants.

Here's a tiny little trait that models the behavior of speaking (bk03ch01/example07/):

```
trait Speak {  
    fn speak(&self) -> String;  
}
```

Note that although the trait includes the signature for a method named `speak`, it doesn't specify the function code. All the `trait` definition does is guarantee that any data structure that implements the `Speak` trait will have a `speak` method that can be called. The specifics of the `speak` method code are up to each data structure to define. This enables a single behavior such as speaking to be shared among multiple types.

Okay, so right about now you might be wondering why you should bother with traits. After all, couldn't you just add a `speak()` method directly to each struct? Here's why traits matter:

- » **Polymorphism:** Traits let you write functions that work with any type that has a certain behavior. A function that accepts `impl Speak` (as I describe shortly) can work with dogs, cats, ducks, or any future type you create, without changing the function.
- » **Shared contracts:** When multiple types implement the same trait, you're guaranteeing that they all have the same capability. Other code can rely on that guarantee.
- » **Extensibility:** You can implement traits on types you didn't create. Want your third-party library's types to be displayable? Implement `Display` for them.

The trait definition creates the contract; the `impl` block fulfills it. Let's see how this works by continuing with the example.

Next you define your types (bk03ch01/example07/, continued):

```
#[derive(Debug)]
struct Dog {
    name: String,
}

#[derive(Debug)]
struct Cat {
    name: String,
}

#[derive(Debug)]
struct Duck {
    name: String,
}
```

Now you can implement the trait on each type. Here's the general syntax:

```
impl Trait for Type {
    items
}
```

where *Trait* is the name of the trait you want to implement, *Type* is the name of the type, and *items* are the items from *Trait* that define the trait's behavior for this type.

For example (bk03ch01/example07/, continued):

```
impl Speak for Dog {
    fn speak(&self) -> String {
        format!("{}", says woof!", self.name)
    }
}

impl Speak for Cat {
    fn speak(&self) -> String {
        format!("{}", says meow!", self.name)
    }
}

impl Speak for Duck {
    fn speak(&self) -> String {
        format!("{}", says quack!", self.name)
    }
}
```


Each implementation provides its own code for the `speak()` method. You've created three custom types — `Dog`, `Cat`, and `Duck` — and now they all share the same behavior — speaking.

Now you can put it all together (bk03ch01/example07/, continued):

```
fn main() {
    let dog = Dog {
        name: "Chase".to_string(),
    };
    let cat = Cat {
        name: "Walter".to_string(),
    };
    let duck = Duck {
        name: "Daffy".to_string(),
    };

    println!("{}", dog.speak());
    println!("{}", cat.speak());
    println!("{}", duck.speak());
}
```

Output:

```
Chase says woof!
Walter says meow!
Daffy says quack!
```

Printing Prettier with the Display Trait

Implementing the `Debug` trait enables you to get a quick look inside a data structure by using `{:?}` in `println!` or a similar macro. The output is basic, which is all you need when you're testing or debugging your code, but it's not very readable for the people using your program. Programmers say that such text isn't *pretty*.

To get output suitable for end users — to *pretty-print*, in the vernacular — you must implement the `Display` trait and use it to teach Rust how to convert your type into human-readable text.

You begin by importing the `fmt` module:

```
use std::fmt;
```

Now here's the general syntax to use to implement the `Display` trait:

```
impl fmt::Display for Type {  
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {  
        write!(f, "format_string", arguments)  
    }  
}
```

Okay, there's a lot going on here, so I'm going to break down each piece for you:

- » **`fmt::Display`:** This trait defines the behavior for user-friendly text formatting. Any type that implements `Display` can be used with `{}` in `println!` and similar macros.
- » **`Type`:** This is the custom type for which you want to implement `Display`.
- » **`fmt()`:** This method is called when you use `Display` on your type.
- » **`fmt::Formatter<'_>`:** The `fmt` module's `Formatter` is Rust's text-formatting engine, which you use to format your text. The formatter is stored in the `f` variable as a mutable reference so that you can write text to it. The weird `<'_>` part is a lifetime parameter, which you learn about in Book 3, [Chapter 3](#), so you can safely ignore it here.
- » **`fmt::Result`:** This alias for `Result<(), fmt::Error>` tells you that the `fmt()` method returns a `Result` enum. If the formatting operation succeeds, the method returns the unit type as `Ok(())`. If the operation fails, the method returns `Err(fmt::Error)`.
- » **`write!`:** This macro writes formatted text into the formatter. The first argument is always the formatter (`f`, in this case). Then comes `format_string`, which defines the string you want to output, including text and `{}` placeholders. Finally, there's `arguments`, which is a comma-separated list of the values you want to insert into the

placeholders in the format string. The macro automatically returns `fmt::Result`, which is exactly what the `fmt()` method needs to return.

Here's an example (bk03ch01/example08/):

```
struct Movie {
    title: String,
    year: u32,
}

impl fmt::Display for Movie {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "The movie {} was released way back in {}. ",
               self.title, self.year)
    }
}

fn main() {
    let movie = Movie {
        title: "The Matrix".to_string(),
        year: 1999,
    };
    // This works now!
    println!("{}", movie);
}
```

Output:

```
The movie The Matrix was released way back in 1999.
```

The magic happens like this:

1. Your code calls `println!("{}", movie)`.
2. Rust sees the `{}` placeholder and looks for a `Display` implementation on `Movie`.
3. The `println!` macro calls your `fmt()` method, passing it a formatter.
4. Your `write!` macro formats the text and puts it in the formatter.
5. Rust prints the formatted text.

The awesomeness of implementing `Display` on a custom type is that now your type works with everything that expects displayable text. It works with the `println!` macro, as you just saw, but it also works with string formatting, error messages, and any library that needs to convert your type to text.

Setting Sensible Defaults with the Default Trait

When you create an instance of a data structure, it's common to apply a few sensible default values to the fields. Rust uses the following defaults:

- » Integers: `0`
- » Floating-point numbers: `0.0`
- » Strings: `""`
- » Booleans: `false`
- » Collections: `empty`

Rather than specifying those defaults manually, you can make Rust do all the work by using the `Default` trait, which applies the preceding defaults to your structure's fields. The even better news is that you can derive the `Default` trait:

```
#[derive(Default)]
```

Then you just run the `default()` method on your data structure:

```
Name::default()
```

where `Name` is the name of your data structure. Here's an example (bk03ch01/example09/):

```
#[derive(Debug, Default)]
struct Settings {
    username: String,
```

```

    volume: u32,
    gamma: f64,
    wifi_enabled: bool,
}

fn main() {
    let settings = Settings::default();
    println!("{:?}", settings);
}

```

Output:

```
Settings { username: "", volume: 0, gamma: 0.0, wifi_enabled: false }
```

You can also implement the `Default` trait manually if you prefer to set your own defaults ([bk03ch01/example10/](#)):

```

#[derive(Debug)]
struct Settings {
    username: String,
    volume: u32,
    gamma: f64,
    wifi_enabled: bool,
}

impl Default for Settings {
    fn default() -> Self {
        Settings {
            username: "guest".to_string(),
            volume: 50,
            gamma: 1.0,
            wifi_enabled: true,
        }
    }
}

fn main() {
    let settings = Settings::default();
    println!("{:?}", settings);
}

```

Output:

```
Settings { username: "guest", volume: 50, gamma: 1.0, wifi_enabled: true }
```

Project: Building a Media Library with Traits

To help bring this chapter's concepts together, I close with a project that creates a library that stores data related to various types of media, enhanced with several traits. Unfortunately, at more than 530 lines, the `main.rs` file ended up too big to fit in this chapter. So what follows is a kind of highlight reel that takes you through the main trait-related aspects of the project. The full file (`bk03ch01/media-library/`) is sprinkled liberally with comments, so you shouldn't have any trouble following along.

The traits

You can implement many traits for media, but in this project I settled on just three:

- » **Rateable:** The media can be rated on a numeric scale.
- » **Playable:** The media can be played.
- » **Searchable:** The media text can be searched.

Here's the `Rateable` trait definition:

```
trait Rateable {  
    fn rate(&mut self, score: u8) -> Result<(), String>;  
    fn get_rating(&self) -> Option<f32>;  
    fn get_review_count(&self) -> u32;  
}
```



REMEMBER When you implement a trait, you must provide implementations for *all* the methods defined in that trait (unless the trait provides default implementations, which these don't). If you forget one, the compiler will refuse to compile your code. This is the contract aspect of traits at work. The trait definition promises that any type

with this trait will have these capabilities, and the compiler enforces that promise.

The `rate()` method is used to rate the media item. This method makes changes to the item's `ratings` vector, so the `self` reference has to be mutable. This trait also contains a method for getting the average rating of the current media item and a method for returning the number of ratings for the item.

Here's the `Playable` trait definition:

```
trait Playable {  
    fn play(&self) -> String;  
    fn get_duration(&self) -> u32;  
}
```

In this project, the `play()` method doesn't play the media; it just returns a string that says the media is playing. The `get_duration()` method returns the duration of the media (although note that the project doesn't use this method because we're not using media files here).

And, finally, here's the `Searchable` trait definition:

```
trait Searchable {  
    fn matches_query(&self, query: &str) -> bool;  
    fn get_searchable_text(&self) -> String;  
}
```

The `matches_query()` method accepts a string slice and returns `true` if the `query` text is found in the item's data. The `get_searchable_text()` method returns the item data as a `String` that can be searched.

The media

This project works with three kinds of media: movies, books, and songs. Since the implementations of each media type are similar, I take you through just the movie code to save space.

Here's the `Movie` struct:

```
#[derive(Debug)]  
struct Movie {  
    title: String,  
    director: String,
```

```

    year: u32,
    duration_minutes: u32,
    ratings: Vec<u8>,
}

```

This data structure stores some basic metadata for each movie: title, director, year, and duration. The movie's ratings are stored in the ratings vector. The structs for the `Book` and `Song` types are similar.

Here's the `Movie` constructor:

```

impl Movie {
    fn new(title: String,
           director: String,
           year: u32,
           duration_minutes: u32) -> Self {
        Movie {
            title,
            director,
            year,
            duration_minutes,
            ratings: Vec::new(),
        }
    }
}

```

Implementing the traits

For each media type, the code implements the `Rateable` and `Searchable` traits. Since (non-audio) books aren't playable, only the `Movie` and `Song` types implement the `Playable` trait.

Here's the implementation of the `Rateable` trait for a `Movie` item:

```

impl Rateable for Movie {

    fn rate(&mut self, score: u8) -> Result<(), String> {
        if score < 1 || score > 10 {
            return Err("Movie rating must be between 1 and 10".to_string());
        }
        self.ratings.push(score);
        Ok(())
    }

    // Return this item's average rating
    fn get_rating(&self) -> Option<f32> {

```



```

        // Do we have any ratings yet?
        if self.ratings.is_empty() {
            // If not, return None
            None
        } else {
            // Otherwise, sum the ratings
            let sum: u32 = self.ratings.iter().map(|&r| r as u32).sum();
            // Return the average in the Some
            Some(sum as f32 / self.ratings.len() as f32)
        }
    }

    // Return the number of reviews
    fn get_review_count(&self) -> u32 {
        self.ratings.len() as u32
    }
}

```

Here's a quick summary of what each method does:

- » **rate()**: Rates the item, returning a `Result`. This method checks that the rating is legal (from 1 to 10). If it's not, the method returns an `Err` string; if it is, the method pushes the rating to the `ratings` vector and returns the unit type `()`.
- » **get_rating()**: Returns the item's average rating using an `Option`. The method checks whether the media item has any ratings. If not, `None` is returned. Otherwise, the `map()` iterator returns the sum of the rating, which is then divided by the number of ratings to get the average; the average is returned wrapped in the `Some`.
- » **get_review_count()**: Returns the number of reviews.

Here's the implementation of the `Playable` trait for a `Movie` item:

```

impl Playable for Movie {
    fn play(&self) -> String {
        format!("Now playing: \"{}\", \"{}\" directed by {} ({}).",
            self.title, self.director, self.year)
    }
    fn get_duration(&self) -> u32 {
        self.duration_minutes * 60
    }
}

```

The `play()` method uses `format!()` to return a *Now playing* string that includes the movie title, director, and year. The `get_duration()` method (again, not used in this project) returns the duration of the movie, in seconds.

And here's the implementation of the `Searchable` trait for a `Movie` item:

```
impl Searchable for Movie {

    fn matches_query(&self, query: &str) -> bool {
        let query = query.to_lowercase();
        let searchable = self.get_searchable_text().to_lowercase();
        searchable.contains(&query)
    }

    fn get_searchable_text(&self) -> String {
        format!("{}", {} {}, self.title, self.director, self.year)
    }
}
```

The `matches_query()` method takes a string slice as the `query` variable and converts it to lowercase. The method then calls `get_searchable_text()`, which uses the `format!` macro to bundle the title, director, and year into a `String`. Then the `String` is stored in the `searchable` variable and converted to lowercase. The method returns the result of the `contains()` method: `true` if `searchable` contains `query` or `false` otherwise.

Finally, the project also implements the `Display` trait for a `Movie` item:

```
impl fmt::Display for Movie {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}{}" ({})) directed by {}",
            self.title, self.year, self.director)
    }
}
```

Chapter 2

Generics: Code That Works for Almost Anything

IN THIS CHAPTER

- » Understanding the need for generics
 - » Making functions, structs, and enums generic
 - » Constraining generics with type bounds
 - » Combining traits and generics
 - » Building a generic logger
-

Writing Rust code feels very wholesome.

—JOHN CARMACK

The programmer Eric Raymond once wrote that “Good programmers know what to write. Great ones know what to reuse.” What he’s pointing to here is one of the secrets to programming productivity: Don’t repeat yourself. In fact, this maxim is important enough to have been elevated to an acronym: DRY. As a fundamental principle of software design, DRY means that every piece of logic or knowledge in a program should be expressed in just *one* place.

Implementing DRY in your own code generally involves reusing existing code as much as possible. I discuss this a bit back in Book 1, [Chapter 5](#) where I talk about loops, and in Book 1, [Chapter 7](#) where I talk about functions. In this chapter, you enter the professional ranks of code reuse by exploring one of Rust’s most useful and most powerful features: generics.

Why Write the Same Code Twice?

Here's a scenario you'll come upon time and time again when coding. You're, say, crafting a function or building a struct, and suddenly Yogi Berra is there in your head saying, "It's déjà vu all over again." That is, you get this uncanny feeling that you've written the same code before, perhaps multiple times!

For example, suppose a while back you wrote the `make_i32_tuple()` function to take two `i32` values and bundle them into a tuple (refer to `bk03ch02/example01/` in this book's example files):

```
fn make_i32_tuple(a: i32, b: i32) -> (i32, i32) {
    (a, b)
}

fn main() {
    let a = 5;
    let b = 10;
    let pair = make_i32_tuple(a, b);
    println!("{:?}", pair);
}
```

Output:

```
(5, 10)
```

Now, just this morning, you wrote the `make_string_tuple()` function to take two `String` values and bundle them into a tuple:

```
fn make_string_tuple(a: String, b: String) -> (String, String) {
    (a, b)
}

fn main() {
    let a = "Hello".to_string();
    let b = "world!".to_string();
    let pair = make_string_tuple(a, b);
    println!("{:?}", pair);
}
```

Output:

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

Here comes the déjà vu! These two functions have identical bodies — the expression `(a, b)` — and the signatures are the same, just with different types specified. Sure, these are small examples, but imagine if you later need to also bundle `f64` values into tuples, and `u8` values, and `bool`s, and string slices! Making multiple versions of essentially the same code is a clear violation of DRY.

But DRY violations don't just mean extra wear and tear on your typing fingers:

- » **Maintenance headaches:** With multiple variations of the same code all over the place, you must update every variation when you need to change the logic.
- » **Bug multiplication:** If you create your variations by copying and pasting them and the original version contains a bug, you've just cloned that bug into every variation!
- » **Cognitive overload:** Someone reading your code must now mentally juggle all these function variations to understand your code.

This is exactly the kind of problem that generics were born to solve. With generics, you can write just *one* function that handles different types in a more abstract, or *generic* (wink, wink) way.

Introducing Generics

In Rust, a *generic* is a placeholder for a type. Whether you're dealing with a function signature or the definition of a struct or an enum, anywhere that you specify a type, you can instead specify a generic, as long as it makes sense to do so, particularly with function logic.

Making something generic in Rust involves inserting after the name of the function, struct, or enum a *type parameter*, which is a short, capitalized name that acts as a kind of type placeholder. In RustWorld, the conventional type parameter name is `T` (for, of course, *type*), which goes inside a pair of angle brackets: `<T>`. (`T` is the convention, but you're not bound by it. Use any identifier that makes your code more readable.)

The type parameter says to Rust, “Sorry to interrupt, but this function (or struct or enum) uses a generic type that I’m going to write as `T` from here on, okay?”

Making a function generic

To make this concrete, I’ll fix the problem from the preceding section by taking the two functions that bundle `i32` values and `String` values into tuples and converting them into a single generic function that can bundle any type into a tuple (bk03ch02/example03/):

```
fn make_tuple<T>(a: T, b: T) -> (T, T) {
    (a, b)
}

fn main() {
    let a = 5;
    let b = 10;
    let pair = make_tuple(a, b);
    println!("{:?}", pair);

    let a = "Hello".to_string();
    let b = "world!".to_string();
    let pair = make_tuple(a, b);
    println!("{:?}", pair);
}
```

Output:

```
(5, 10)
("Hello", "world!")
```

Yay, it works!

Just to make things extra clear, let’s take a closer look at the magical, new `make_tuple()` function:

```
fn make_tuple<T>(a: T, b: T) -> (T, T) {
    (a, b)
}
```

First, note that the function contains the type parameter `<T>` right after the function name. The angle brackets `< >` are Rust’s way of saying “Here come the generic parameters.” Think of them as a separate parameter list. That is, just as parentheses `( )` hold a function’s value

parameters, angle brackets hold a function's *type* parameters. This syntax is common across many programming languages, so if you've seen `List<String>` in Java or `vector<int>` in C++, you're looking at the same idea.

Now you can handle the rest of the function signature:

- » In the arguments list, for each argument that you want to make generic, replace the specific type with `T`. Note that although in my example I replaced every argument type with `T`, you can replace only what makes sense for your function. If a particular argument must be, say, a Boolean, leave that argument's type as `bool`.



WARNING Since I specified `T` for both `a` and `b`, the Rust compiler will expect you to send the *same* data type to both arguments. Send each argument a `u8` and all will be well; send the `a` argument a `u8` and the `b` argument an `f32`, and the compiler will call you out on it.

- » In the return type, for each type that you want to make generic, replace that type with `T`. Again, let your code guide you here. If your function always returns, say, an owned string, leave the return type as `String`.

Now Rust knows that it doesn't matter what type you send to the function. The values can be not only `i32` integers or `Strings` but `f64` floating-point numbers, `bool` values, string slices, or whatever. Now one function can handle all your tuple-making needs. Now *that* is DRY!



REMEMBER In Rust lingo, you describe the `make_tuple()` function as being *generic over type T*.



TECHNICAL
STUFF

Okay (you might be wondering), if `make_tuple<T>` can accept any type, how does Rust know what code to generate? The answer is *monomorphization*, a fancy-schmancy word for a simple idea. When you compile your program, Rust looks at every place you use `make_tuple` and generates a specialized version for each type. So, if you call `make_tuple(5, 10)` and `make_tuple("Hello", "world")`, Rust creates two functions behind the scenes: one for integers and one for strings. You get the convenience of writing one generic function, but at runtime you get the same performance as if you'd written specialized functions yourself.

Making a struct generic

The flexibility and efficiency that generics bring to functions is available also for your own types via structs and enums. I cover structs here; refer to “Making an enum generic,” next, for the enum version.

You make a struct generic by adding a type parameter after the struct name:

```
struct Name<T> {  
    // fields, one or more of which can be generic  
}
```

For example, suppose you compose a daily report and use two structs for the task, one for numeric data and one for text data:

```
struct ReportNumeric {  
    title: String,  
    data: i32,  
}  
  
struct ReportText {  
    title: String,  
    data: String,  
}
```

Can't you just *smell* the inefficiency here? Let's DRY this by making the struct generic:


```
struct Report<T> {  
    title: String,  
    data: T,  
}
```

Since the report title is always text, I've kept the `title` field's type as `String`. But now the `data` field's type is `T`, which tells Rust to accept any data type in that field. Here's an example that puts our new generic struct to the test (bk03ch02/example04/):

```
struct Report<T> {  
    title: String,  
    data: T,  
}  
  
fn main() {  
    // Generate a report where the data is numeric  
    let financial_report = Report {  
        title: String::from("Q2 Sales"),  
        data: 1_492_500,  
    };  
  
    // Generate a report where the data is text  
    let event_report = Report {  
        title: String::from("Conference Summary"),  
        data: String::from("The conference was a rousing success..."),  
    };  
}
```

Making an enum generic

You make an enum generic by adding a type parameter after the enum name:

```
enum Name<T> {  
    // variants, one or more of which can be generic  
}
```

You might not have realized it, but you've read about and likely been using a couple of generic enums multiple times already: `Option` and `Result`. If you don't believe me, first check out the full definition of an `Option` direct from Rust's standard library:

```
pub enum Option<T> {  
    None,
```

```
    Some(T),  
}
```

Why, look at that! This definition includes the type parameter `<T>` right after the enum name, so `Option` is generic over type `T`, and the `Some` variant holds a value of type `T`.

Now here's the definition of `Result` from the standard library:

```
pub enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

Okay, interesting! In this case, you see that the `Result` enum is generic over *two* types, `T` and `E`. This is normal, and I'll explain it in more detail in the "[Specifying multiple generic type parameters](#)" section that follows. For now, you can gather from the definition that in the two variants, `Ok` holds a value of type `T` and `Err` holds a value of type `E`.

But you can also make your own enums generic, which comes in handy when you have an idea that can work with different types of data. For example, let's say you have a program that tracks measurements, some of which will be exact and some of which will be estimated. For example, you might know that the temperature in the lecture hall is exactly 21.4 degrees Celsius, but you might estimate that 250 are attending the lecture.

You can model these two ways of measuring things with a generic enum called `Measurement`:

```
enum Measurement<T> {  
    Exact(T),  
    Estimated(T),  
}
```

`T` is a placeholder for whatever type of data you want to measure. The `Measurement` enum can hold either an `Exact` value of type `T` or an `Estimated` value of type `T`. You could have a measurement of an integer, a floating-point number, or even a `String`. Here's an example (bk03ch02/example05/):

```

#[derive(Debug)]
enum Measurement<T> {
    Exact(T),
    Estimated(T),
}

fn main() {
    // A measurement of an integer (e.g., people in a lecture hall)
    let attendee_count = Measurement::Estimated(250);
    // A measurement of a float (e.g., temperature in Celsius)
    let temperature = Measurement::Exact(21.4);

    println!("{:?}", attendee_count);
    println!("{:?}", temperature);
}

```

Output:

```

Estimated(250)
Exact(21.4)

```

Specifying multiple generic type parameters

Sometimes one generic type just isn't enough. That's okay, because Rust can work with as many generics as you can throw at it. Recall the definition of `Result` from the standard library:

```

pub enum Result<T, E> {
    Ok(T),
    Err(E),
}

```

The `<T, E>` part is called a *generic parameter list*, and it tells Rust that `Result` is generic over both `T` and `E`.

Generalizing from this, you can specify multiple generic type parameters in your own functions, structs, and enums:

```

fn function_name<T, U, ...>
struct StructName<T, U, ...>
enum EnumName<T, U, ...>

```

By convention, the first generic type parameter is designated as `T`, the second as `U`, the third would be `V`, and so on. (That said, if you find yourself using more than three generics in a structure, you might want to rethink your approach because having so many generics floating around

can make your code more complex and less readable. Consider breaking up your data structures or functions into smaller units.)

As an example, it's possible to expand the `Report` struct from the “[Making a struct generic](#)” section to include an `id` field that's sometimes a number and sometimes text:

```
struct Report<T, U> {  
    title: String,  
    data: T,  
    id: U,  
}
```

Here's an example that puts this revised struct to work (bk03ch02/example06/):

```
struct Report<T, U> {  
    title: String,  
    data: T,  
    id: U,  
}  
  
fn main() {  
    // Generate a report where the data is numeric  
    let financial_report = Report {  
        title: String::from("Q2 Sales"),  
        data: 1_492_500,  
        id: 2026.2,  
    };  
  
    // Generate a report where the data is text  
    let event_report = Report {  
        title: String::from("Conference Summary"),  
        data: String::from("The conference was a rousing success..."),  
        id: "conference-summary",  
    };  
}
```

Type Bounds and Constraints

In the code examples I've used so far in this chapter, I replaced one or more specific type annotations with a generic type parameter (usually `T`). But this simple substitution approach doesn't always work. For example,

suppose you have a function called `array_contains()` that takes a reference to an array and a value, and returns `true` if the value is in the array (bk03ch02/example07/):

```
fn array_contains(slice: &[i32], item: i32) -> bool {
    for value in slice {
        if *value == item {
            return true;
        }
    }
    false
}

fn main() {
    let primes = [2, 3, 5, 7, 11];
    println!("{}", array_contains(&primes, 3));
}
```

Output:

```
true
```

If you find yourself working with arrays of `f64` values, `String` values, or `&str` values, it makes sense to make your function generic. So, you try this (bk03ch02/example08/):

```
fn array_contains<T>(slice: &[T], item: T) -> bool {
    for value in slice {
        if *value == item {
            return true;
        }
    }
    false
}

fn main() {
    let primes = [2, 3, 5, 7, 11];
    println!("{}", array_contains(&primes, 3));
}
```

Everything seems to be in order, but the Rust compiler disagrees:

```
error[E0369]: binary operation '==' cannot be applied to type 'T'
```

The problem is that using the generic parameter `T` is a promise to the compiler that the function's logic will work with *any* type `T`. But in this

case, the compiler can't guarantee that every possible type can be compared with the `==` operator.

That's the bad news. The good news is that the compiler also tells you how to fix the problem:

```
help: consider restricting type parameter 'T' with trait 'PartialEq'
```

The compiler is suggesting that you add a trait bound to the function signature. A *trait bound* is a constraint placed on a generic type parameter to guarantee that it has the behavior required by your code.

Say, what? When you write `<T>` with no constraints, you're promising the compiler that your code will work with literally *any* type: integers, strings, file handles, database connections, whatever. Ah, but here's the rub: Most code can't work with any type! The `array_contains()` function, for example, needs to compare values with `==`, but not every type supports that operation.

Not to worry, though, because a trait bound narrows that promise. When you write `<T: PartialEq>`, you're promising that your function works with any type *that can be compared for equality*. The compiler can then verify that your function uses only operations that `PartialEq` types support, and it will reject calls with types that don't implement `PartialEq`.

Adding a trait bound to a generic parameter

You add a trait bound by following the type parameter with a colon, a space, and then the name of the trait:

```
fn Name<T: trait>
```

To constrain the generic parameter `T` to those types that support `==` (or `!=`), you can add the `PartialEq` trait bound to the `array_contains()` function (bk03ch02/example09/):

```
fn array_contains<T: PartialEq>(slice: &[T], item: T) -> bool {
    for value in slice {
        if *value == item {
            return true;
        }
    }
}
```

```

    }
    false
}

fn main() {
    let primes = [2, 3, 5, 7, 11];
    println!("{}", array_contains(&primes, 3));

    let fruit = ["apple", "banana", "grape", "peach", "pear"];
    println!("{}", array_contains(&fruit, "cherry"));
}

```

Output:

```

true
false

```

Woo-hoo! It works!

Working with multiple trait bounds

Trait bounds are a way to work *with* the compiler rather than against it, so you'll find yourself turning to these constraints often to keep your code on the straight and narrow. Sometimes keeping the compiler happy means adding multiple constraints to a generic:

```
fn Name<T: trait1 + trait2>(arg: T) -> T { }
```

where *trait1* and *trait2* are the traits you want to use as constraints on the generic *T*.

For example, a useful bit of code to have around is one that restricts a value to a range defined by minimum and maximum values. This is called a *clamp*, and it's perfect as a generic function because you'll need it with many different types. However, the logic for a clamp requires comparison expressions using *<* and *>*, so that requires the `PartialOrd` trait. Also, the value being compared is referenced multiple times in the code, so the `Copy` trait is required to prevent the value from being moved the first time it's used. Here's the function ([bk03ch02/example10/](#)):

```

fn clamp<T: PartialOrd + Copy>(value: T, min: T, max: T) -> T {
    if value < min {return min}
    else if value > max {return max}
    value
}

```

```
fn main() {
    let i = 5;
    let f = 1.6;
    let c = "d";
    println!("{}", clamp(i, 5, 10));
    println!("{}", clamp(f, 0.0, 1.0));
    println!("{}", clamp(c, "a", "m"));
}
```

Output:

```
5
1.0
d
```

Making complex constraints more readable

Trait bounds can, in a sense, become too much of a good thing, especially when you have multiple bounds on multiple generics. For example:

```
fn handle_data<T: Display + Clone + PartialOrd, U: Display + Default>(
    first: T,
    second: U
) -> String {
    format!("{}", and {}, first, second)
}
```

This function signature has got a *lot* going on! In this example, the constraints are called *inline bounds* because they appear inside the function signature.

When runaway inline bounds cause your function signatures to become longer than your function body, it's time to call for help in the form of the `where` clause, which separates the function signature and the trait bounds to make your code more readable:

```
fn Name<T, U>(arg1: T, arg2: U) -> T
where
    T: trait1 + trait2,
    U: trait3 + trait4,
{
    // function body
}
```


You insert the `where` clause on separate lines right after the function signature, and you use the clause to move the trait bounds to their own lines. Here's the cleaned-up version of the previous function (bk03ch02/example11/):

```
fn handle_data<T, U>(first: T, second: U) -> String
where
    T: Display + Clone + PartialOrd,
    U: Display + Default,
{
    format!("{}", first, second)
}
```

Much better!

Traits and Generics = Chef's Kiss

Traits, as I describe in Book 3, [Chapter 1](#), define shared behaviors. Generics, as I describe in this chapter, define code that's abstract about the types it works with. Put traits and generics together, and now you can write code that can apply shared behaviors broadly, which is a powerful combination.

The most common way to combine traits and generics is to define a generic function that uses a custom trait bound. For example, suppose you're coding a health app and one task the app needs to perform is to log the energy value (that is, the calories) of different foods. However, the way you calculate calories for, say, an apple, is different from how'd you'd calculate them for an energy bar. So, this part of the code needs two things:

- » A trait for the shared behavior of food that provides energy in the form of calories.
- » A function to log the calories of different foods, generic over the provides energy trait.

First, here's the `ProvidesEnergy` trait that defines the shared behavior required by the app (bk03ch02/example12/):

```
trait ProvidesEnergy {  
    fn calories(&self) -> u32;  
}
```

Now you can define the structs and implement the shared behavior on each:

```
// The calories in an apple depend on its weight  
struct Apple {  
    weight_grams: f32,  
}  
  
// Implement ProvidesEnergy on Apple  
impl ProvidesEnergy for Apple {  
    fn calories(&self) -> u32 {  
        // Apples have about 52 calories per 100g  
        (self.weight_grams * 0.52) as u32  
    }  
}  
  
// The calories in an energy bar are written on the  
// package, so they don't require a calculation  
struct EnergyBar {  
    calories_per_bar: u32,  
}  
  
// Implement ProvidesEnergy on EnergyBar  
impl ProvidesEnergy for EnergyBar {  
    fn calories(&self) -> u32 {  
        // Just return the calories  
        self.calories_per_bar  
    }  
}
```

Now you can write a single function that's generic over any type that implements ProvidesEnergy:

```
// This function is generic over T,  
// but constrains T to implement ProvidesEnergy  
fn log_food_intake<T: ProvidesEnergy>(food: &T) {  
    println!("Logging food with {} calories.", food.calories());  
}
```

Give it a whirl:

```
fn main() {  
    let apple = Apple { weight_grams: 182.4 };  
}
```

```
let bar = EnergyBar { calories_per_bar: 255 };

// You can pass both types to the same function!
log_food_intake(&apple);
log_food_intake(&bar);
}
```

Output:

```
Logging food with 94 calories.
Logging food with 255 calories.
```

Project: Build a Generic Logger That Works with Any Type

To bring this chapter's concepts together, this section takes you through the construction of a simple logger that can print messages to the console. The twist is that the logger will be generic, so it can accept and print a message of any type that can be displayed.

In a terminal session, navigate to your Rust projects directory and run the following command:

```
cargo new generic-logger
```

Load that project's `main.rs` into your editor, delete the default `println!` statement, and you're ready to go.

You need to define simple `Logger` struct that holds a prefix string that will help you identify messages from this logger (bk03ch02/generic-logger/):

```
use std::fmt::Display;

struct Logger {
    prefix: String,
}
```

Later you see that the code requires a `Display` trait bound, hence the need to import `std::fmt::Display` at the top of the file.

Next, you implement a couple of `Logger` methods:

```
impl Logger {

    // Constructor method
    fn new(prefix: &str) -> Self {
        Self {
            prefix: prefix.to_string(),
        }
    }

    // This method is generic over any type T that implements Display
    fn log<T: Display>(&self, message: T) {
        println!("{}", self.prefix, message);
    }
}
```

The `new()` method is the constructor. The `log()` method is the meat and potatoes here: a method that's generic over any type `T` that implements the `Display` trait, which enables the use of `{}` placeholders in the `println!` macro.

Now you can put the logger to good use:

```
fn main() {

    // Make a logger
    let app_logger = Logger::new("APP_INFO");

    // Log a string slice
    app_logger.log("App has started.");

    // Log a String
    let warning = String::from("Disk space is running low!");
    app_logger.log(warning);
    // Log an integer
    app_logger.log(404);

    // Log a floating-point number
    app_logger.log(98.6);
}
```

Output:

```
APP_INFO: App has started.
APP_INFO: Disk space is running low!
APP_INFO: 404
APP_INFO: 98.6
```

Chapter 3

Lifetimes: How Long Things Live

IN THIS CHAPTER

- » Getting your head around lifetimes
 - » Using lifetimes in functions and structs
 - » Learning the secret of eternal lifetimes
 - » Building a config parser with lifetimes
-

Rust is not just C++ with a funny syntax and more static checking. It's even more different than that. Rust is another way of thinking.

—ERIK HEDVALL

Way back in Book 2, [Chapter 1](#), I mention one of the Rust borrow checker's most important rules: *A reference can't outlive the value it points to*. I then offer a simple example to — fingers crossed! — help you understand what this rule means. But perhaps that curious use of the word *outlive* has had you wondering if you really understood what was going on.

Outlive is a curious word in a programming context, but it's not just a metaphor. An entire branch of Rust is devoted to the idea of how long things live when a program is running. That branch is called lifetimes and it's the subject of this chapter. Don't worry if what follows seems arcane and even a bit weird at first. Even experienced programmers scratch their heads when introduced to lifetimes since it's a concept that's unique to Rust. But it's a concept that sits right at the very heart of memory safety in Rust.

Introducing Lifetimes

As you learn back in Book 2, [Chapter 1](#), a reference (`&T` or `&mut T`) borrows access to some value; it doesn't own that value. The reference is a pointer on the stack, so it's great for speed, and it doesn't duplicate data on the heap, so it reduces memory use. But lurking in the shadows behind every reference is a scary presence:

What if the referenced value disappears while I'm still holding the reference?

If that dire situation came to pass, you'd be left holding a useless bit of data called a dangling pointer, which has been the scourge of robust and safe programming for decades. Why? Because many languages treat the possibility of a referenced value disappearing out from underneath the reference as a “cross your fingers and hope for the best” situation!

Fortunately, Rust knows that you can't take hope to the bank, so Rust refuses to hope. Instead, before it will compile any program, it must prove decisively that every reference stays valid for exactly as long as it's used. The window during which the reference is valid is called its *lifetime*.

You need to know three main facts about lifetimes:

- » **When you're dealing with lifetimes, you're dealing with the lifetimes of references.** This is the foundational idea in this chapter. A lifetime is a way to track how long a borrow (a reference) remains valid as a way of ensuring that it never tries to access data that no longer exists.
- » **A reference's lifetime begins at the moment it's initialized.** A reference is a variable, so its “life” begins when it's created. When you write, say, `let borrowed_value = &initial_value;`, the lifetime of the `borrowed_value` reference begins at that semicolon.
- » **A reference's lifetime ends either when it's last used or when its scope ends.** That is, the Rust compiler keeps track of two lifetime

types:

- *Lexical lifetime*: A reference is valid from the moment it's initialized until the end of the code block in which that initialization occurred. (In this context, *lexical* means “relating to the structure of the code.”)
- *Non-lexical lifetime*: A reference is valid from the moment it's initialized until the end of the statement in which the reference is used for the last time.

Most of the time, your references will have non-lexical lifetimes. However, for things like certain function return values, closures that outlive the references they use, and references stored in data structures such as tuples or structs, the references will have lexical lifetimes.

Here's an example where a non-lexical lifetime comes in handy (bk03ch03/example01/ in this book's example files):

```
fn main() {  
    let mut original = String::from("Original string");  
    let ref1 = &original; // The ref1 lifetime starts here  
    println!("{}", ref1); // ref1 is not used after this, so its lifetime ends  
    let ref2 = &mut original; // No problem  
}
```

The `ref1` variable is an immutable borrow of `original`, so its lifetime begins at the end of its `let` statement. `ref1` is used in the `println!` macro that comes next, but then `ref1` is never used again, so its lifetime ends there. That's good news, because the next line declares `ref2` as a mutable borrow of `original`, which the borrow checker would disallow if the `ref1` borrow were still alive.

Here's an example where a reference requires a lexical lifetime (bk03ch03/example02/):

```
fn main() {  
    let s = String::from("hey");  
    let r = &s; // The lifetime of r starts here  
    let shout_it = || println!("{}", r.to_uppercase()); // Last use of r  
    shout_it(); // Doesn't matter, though: r must live until the closure is
```

```
done  
}
```

The reference `r` last gets used in the `shout_it()` closure, but the closure is used in the next line, so it outlives the non-lexical lifetime of `r`.

Therefore, the compiler considers `r` to have a lexical lifetime (down to the closing brace of the `main()` function) and so the code compiles without a protest from the borrow checker.

Lifetimes show up in your code in two ways:

- » **Implicitly:** When you pass around references and everything is obviously safe, the compiler deals with the lifetimes for you behind the scenes.
- » **Explicitly:** When it's not clear that a particular reference lives at least as long as its referenced value, the compiler will ask for your help to spell out that lifetime.

Explicit lifetimes are what this chapter is mostly all about, so let's get to it.

When Rust Demands Lifetime Annotations

Most of the time, the Rust compiler can figure out the lifetimes of references on its own. Inside a function, for example, the compiler can usually see where the data is created and where it's destroyed and, from that, track all the local references and their lifetimes.

Things get noticeably hairier when you start passing references into and out of functions. Consider the following function as an example (bk03ch03/example03/):

```
fn get_longer(text1: &str, text2: &str) -> &str {  
    if text1.len() > text2.len() {  
        return text1  
    }  
    text2  
}
```



```

}

fn main() {
    let str1 = "It was the best of times";
    let str2 = "It was the worst of times";
    let longer = get_longer(&str1, &str2);
    println!("The longer of the two is '{longer}'.");
}

```

If you try to run this program, the compiler will shake its head mournfully and deny your request. Why? Two reasons:

- » Whenever you return a reference from a function, Rust assumes that the reference must be connected to one of the arguments. (It assumes this even if the return value is in no way connected to the arguments!) This is spelled out in part of the error message:

```

this function's return type contains a borrowed
value, but the signature does not say whether
it is borrowed from 'text1' or 'text2'

```

- » Rust can't figure out how long the returned reference is supposed to be valid. That is, should it be valid as long as `text1` is valid? Or should it be valid as long as `text2` is valid? In other words, Rust doesn't know whether the returned reference should have the same lifetime as `text1` or `text2`. Since the function could return either one, the compiler won't guess and, instead, urges you to add a lifetime annotation:

```

error[E0106]: missing lifetime specifier
  |
1 | fn get_longer(text1: &str, text2: &str) -> &str {
  |               ----      ----          ^ expected named
lifetime parameter
  |
help: consider introducing a named lifetime parameter

```

Fortunately for you, the next section tells you exactly how to do that.

Adding Lifetime Annotations to Functions

Before getting into the (admittedly convoluted) specifics of lifetimes, it might help to approach the problem sideways, sort of, by coming up with an intuitive solution for why the Rust compiler demands a lifetime parameter for the following function:

```
fn get_longer(text1: &str, text2: &str) -> &str {  
    if text1.len() > text2.len() {  
        return text1  
    }  
    text2  
}
```

The compiler's main concern here is that the `&str` return value must be valid, but it doesn't know if that value will be borrowed from `text1` or `text2`. Depending on the inputs, one of these will live longer than the other. But here's a crucial fact: *The compiler doesn't care which value lives longer!*

What the compiler cares about is memory safety, which in this case means that it wants to guarantee that the returned reference isn't pointing to nothing. How can it possibly guarantee that?

By ensuring that the returned reference is valid for at least as long as both the inputs are valid

That is, there will be some point in the code where both `text1` and `text2` are valid, and then the code will diverge and one will become invalid. The compiler doesn't care which one becomes invalid or how long the other remains valid because as long as the return value is valid up to the point of divergence — where, crucially, *both* `text1` and `text2` are still valid — the reference is guaranteed to be valid.

A lifetime annotation, then, is a Rust mechanism that formalizes this guarantee.

A first look at lifetime annotations

In Book 3, [Chapter 2](#), I talk about using a generic parameter declaration `<T>`. When used after a function name, this declaration supplies the

generic type with a name ($\mathbb{T}$) that you can use as the type annotation for the function's arguments and return value.

A lifetime annotation is similar because it's a way of giving a lifetime a name so that you can connect the lifetimes of different values, which helps the compiler resolve lifetime ambiguities. Fair warning: Most people find the syntax of a lifetime annotation to be headache-inducingly weird at first read. But the strangeness is only on the surface. Once you get used to them, you'll see that lifetime annotations are straightforward.

A lifetime annotation consists of an apostrophe ('), followed by a short, lowercase name. By convention in Rust, the first annotation is always 'a, the second is 'b, and so on. (Just so you know, most Rust folks pronounce 'a as "tick a," but I also like "lifetime a" because it drives the this-is-a-lifetime message home.)

Okay, but what *is* 'a, exactly? Does it represent a period of time, as you might expect? Nope. It's really just a *label* for a lifetime. You don't know what that lifetime is, but that doesn't matter. It has a label, and the crux of the matter is that by using the same label on multiple references, you're letting Rust know that all those references are connected in the sense that they must share a common lifetime.

Annotating a function with a lifetime

Okay, enough jawing. Let's finally get to it by annotating the problematic function from the previous section (bk03ch03/example04/):

```
fn get_longer<'a>(text1: &'a str, text2: &'a str) -> &'a str {
    if text1.len() > text2.len() {
        return text1
    }
    text2
}

fn main() {
    let str1 = "It was the best of times".to_string();
    let str2 = "It was the worst of times".to_string();
    let longer = get_longer(&str1, &str2);
    println!("The longer of the two is '{longer}'.");
}
```

Yes, I know: That revised signature of the `get_longer()` function is a sight to *create* sore eyes! But it's not as awful as it seems at first. Let's go through it piece by piece:

```
fn get_longer<'a>(text1: &'a str, text2: &'a str) -> &'a str {
```

First up is `<'a>`. This is the generic lifetime parameter declaration, and it consists of a lifetime annotation surrounded by angle brackets. It always appears inserted snugly between the function name and the argument list. This declaration introduces a new lifetime label, `'a`, that can be used in the function signature.

From here, it helps to break down the rest of the signature into two parts:

- » **The input:** `text1: &'a str, text2: &'a str`. This part of the signature acts as a kind of promise that you're making to the function that the compiler can treat both `text1` and `text2` as living at least as long as some shared lifetime, which you're calling `'a`. What is that lifetime? It's the last point in the code where both references are valid. This is the compiler's way of ensuring that both pieces of data exist for the same minimum amount of time. In practice, `'a` is no longer than the shortest lifetime of the two inputs, which ensures memory safety.
- » **The output:** `-> &'a str`. This part of the signature also acts as a kind of promise, but in this case it's a promise that the function makes to the statement that calls it. The promise is that the function will return a reference to a string slice that will be valid for exactly the same lifetime, `'a`, as the two input references. This part is the most crucial. The `'a` on the returned reference is a guarantee. It tells the compiler and anyone using the function that the lifetime of the returned reference is tied to, and limited by, the shortest of the two input lifetimes.

The Rust compiler will then look inside your function's code to make sure that these two promises are upheld.

Typing a lifetime to a single input

In the example in the previous section, the code could return one of two references, so the lifetime annotation had to be applied to both. But sometimes it's clear that a returned reference requires the same lifetime as just one input. Here's an example (bk03ch03/example05/):

```
fn split_before<'a>(text: &'a str, marker: &str) -> &'a str {
    if let Some(index) = text.find(marker) {
        &text[..index]
    } else {
        text
    }
}

fn main() {
    let email = "user@example.com";
    let username = split_before(email, "@");
    println!("{}", username);
}
```

Output:

```
user
```

The `split_before()` function takes a string literal as the `text` argument and a character as the `marker` argument. It searches for `marker` in `text` and, if it's found, it returns the substring of `text` up to that point; otherwise, it returns the original `text`. The function returns an `&str` reference, so Rust requires a lifetime annotation. But it's clear that the return reference needs to be tied to the lifetime of only the `text` input (since `marker` is only ever used as a parameter in the `find()` method), so only `text` gets the `'a` annotation.

Adding multiple lifetime annotations

Sometimes your function inputs have distinctly different lifetimes, so you need separate lifetime annotations to keep the borrow checker in good spirits. Here's an example (bk03ch03/example06/):

```
fn get_both<'a, 'b>(name: &'a str, title: &'b str) -> (&'a str, &'b str) {
    (name, title)
}

fn main() {
    let employee_name = "Alice".to_string();
}
```

```

let name;

{
    let title;
    let temp_title = String::from("Manager"); // Shorter lifetime
    (name, title) = get_both(&employee_name, &temp_title);

    println!("Name: {}, Title: {}", name, title); // This works
} // temp_title gets dropped here, but employee_name lives on

println!("Name: {}", name);
}

```

In the `main()` function, `employee_name` is declared in the outer scope, while `temp_title` is declared in the inner code block. This means `temp_title` has a shorter lifetime than `employee_name`, so Rust requires that the `get_both()` function annotate these inputs with separate lifetimes:

```

fn get_both<'a, 'b>(name: &'a str, title: &'b str) -> (&'a str, &'b str) {

```

Lifetimes don't solve everything

Lifetime annotations aren't pixie dust that you can just sprinkle on your functions to make them safe. They're not a panacea for bad code decisions. If your function returns a reference and that reference points to data owned by the function, that's not going to pass muster with the compiler:

```

fn do_not_do_this<'a>(_s: &str) -> &'a str {
    let made = String::from("I'm not long for this world.");
    &made // error: cannot return reference to local variable 'made'
}

```

Lifetimes don't magically make unsafe designs safe. They make unsafe designs *not compile*.



TIP In case you're wondering, the `do_not_do_this()` function's `s` argument is preceded by an underscore (`_s`) to tell the compiler not only that `s` isn't used in the function but also that you *meant* for that

to happen. This prevents you getting the `unused variable: 's'` warning when you compile.

Lifetimes in Structs: Keeping References Alive

In all the structs I've sent your way so far in this book, each has owned its data. But in the real Rust world, it's common to build structs that hold one or more references to data owned by something else. This pattern is efficient because it enables you to work with data without having to copy it. (As proof, check out the project at the end of the chapter.)

But once you add a reference to a struct, you must make a promise that any instance of the struct will never outlive the data that it references. How do you make that promise? By annotating the struct with a lifetime.

Okay, but hold on a second. If Rust is so great at memory safety, why do you have to write these lifetime annotations? Why can't the compiler figure it out?

Inside a single function, the compiler can see all the code and track every reference. But structs are different. They're definitions that get used in many different places throughout your program. The compiler can't predict every possible way someone might use your struct, so it needs you to spell out the rules up front. Think of lifetime annotations as a contract you're writing where the fine print says, "Any code that uses this struct must follow these rules." Once you've written that contract, the compiler takes over and enforces it everywhere. You provide the intent; Rust provides the enforcement.

For example, suppose you're building a music player and part of that app is code that displays info about the currently playing song, with that info stored in a struct. It would be wasteful to copy all the song data from its original source into the struct, so you use references, instead:

```
struct NowPlaying {  
    artist: &str,  
    title: &str,
```

```
    album: &str,  
    duration_seconds: u32,    // Owned data is fine, too  
}
```

Ah, but the borrow checker is already cracking its knuckles in anticipation of typing a gentle rebuke. Why? Because there's no guarantee here that any struct instance won't outlive the data it references. The borrow checker requires that you annotate the struct with a lifetime parameter:

```
struct NowPlaying<'a> {  
    artist: &'a str,  
    title: &'a str,  
    album: &'a str,  
    duration_seconds: u32,    // Owned data is fine, too  
}
```

The struct name is followed immediately by `<'a>`, the generic lifetime parameter declaration. Then `'a` is used for each reference in the struct fields. This tells Rust that every instance of `NowPlaying` has a lifetime `'a` associated with it, and the references held by each of those instances must be valid for that entire lifetime. That is, any `NowPlaying<'a>` instance can't outlive any `&'a str` it was built from.

Your struct is now armored with a promise — that any instance of it will never outlive any of its references — and the borrow checker will now ruthlessly enforce that promise everywhere you use the `NowPlaying` struct (bk03ch03/example07/):

```
fn main() {  
    let artist_name = String::from("The Beatles");  
    let song_title = String::from("Here Comes the Sun");  
    let album_name = String::from("Abbey Road");  
  
    let current = NowPlaying {  
        artist: &artist_name,  
        title: &song_title,  
        album: &album_name,  
        duration_seconds: 185,  
    };  
}
```

You can also annotate struct methods with the struct lifetime:


```
impl<'a> NowPlaying<'a> {  
    fn full_description(&self) -> String {  
        format!("{}", by {} (from {})", self.title, self.artist, self.album)  
    }  
}
```

The struct's lifetime parameter declaration (<'a>) appears right after the `impl` keyword and the method name. This positioning tells Rust that you're implementing methods for a `NowPlaying` instance with any lifetime 'a.

Lifetime Elision: When the Compiler Does the Work

Right about now you might be trembling in your boots thinking you have to splatter 'a, 'b, and 'c all over your code. Not to worry! The Rust developers understood that adding lifetime annotations is no one's idea of fun, so they took pity on the rest of us by baking into the compiler a set of rules that enable you to omit — they prefer the term *elide* — lifetime annotations in cases where the correct annotations are obvious. These are the *lifetime elision rules*.

Rule 1: Every input reference gets its own lifetime parameter

The first rule is that every input reference gets its own lifetime parameter. This rule means that the compiler always assigns an inferred and unique lifetime parameter to every reference in a function's list of input arguments. So, for example, when you have a function such as this:

```
fn print_two_things(a: &str, b: &i32) {  
    println!("The first thing is: {}", a);  
    println!("The second thing is: {}", b);  
}
```

The compiler treats the function as though it was written like this:

```
fn print_two_things<'a, 'b>(a: &'a str, b: &'b i32) {  
    println!("The first thing is: {}", a);  
}
```

```
println!("The second thing is: {}", b);  
}
```

This rule mostly concerns how the compiler thinks about inputs. Since this function doesn't return any references, there are no lifetimes to connect, and the compiler is happy.

Rule 2: If there's just one input reference, any output reference gets the same lifetime

The second rule is that if there's just one input reference, any output reference gets the same lifetime. This rule is probably the most common because many functions have exactly one input reference. In such cases, the compiler automatically assigns the lifetime of that input reference to all the output references (if any).

So, for example, when you have a function like this:

```
fn get_first_byte(data: &[u8]) -> &u8 {  
    // Return a reference to the first byte of the slice  
    &data[0]  
}
```

The compiler sees one input (`data: &[u8]`) and one output (`-> &u8`). It knows the returned reference must be borrowing from `data`, so there's no ambiguity. It applies this rule and treats the function as if you had written it like this:

```
fn get_first_byte<'a>(data: &'a [u8]) -> &'a u8 {  
    // Return a reference to the first byte of the slice  
    &data[0]  
}
```

Rule 3: If &self is an input, its lifetime is used for the output

The third rule is that if `&self` is an input, its lifetime is used for the output. If your function is a method on a struct that takes `&self` or `&mut self` as a parameter, the compiler assumes that any returned references are tied to the lifetime of `self`. For example, consider a `Book` struct that has a method to return its title:

```

struct Book {
    title: String,
}

impl Book {
    fn get_title(&self) -> &str {
        &self.title
    }
}

fn main() {
    let my_book = Book { title: String::from("A Tale of Two Cities") };
    let title = my_book.get_title();
    println!("The book's title is: {}", title);
}

```

The `get_title()` method has one input reference (`&self`), so the compiler applies this third rule. It's a safe bet that if you're calling a method on an object and getting a reference back, that reference probably points to data inside the object itself. The compiler treats the method as if you had written it like this:

```

impl Book {
    fn get_title<'a>(&'a self) -> &'a str {
        &self.title
    }
}

```

The 'static Lifetime: Data That Lives (Sort of) Forever

Rust offers a special lifetime called `'static` (pronounced “tick static” or “lifetime static”) that’s reserved for data that lives for the duration of your program. By far, the most common example of `'static` data is the string literal. *Any* string literal you write in your code automatically has a `'static` lifetime:

```

let methuselah: &'static str = "Live and let live";

```

As I explain in Book 2, [Chapter 5](#), when the Rust compiler comes upon a string literal, it embeds that string directly into the compiled executable file. When the program runs, the string gets parked in a special read-only section of memory, where it resides, undisturbed, until the program stops

executing. This always-there quality of a string literal means that any reference to it will always be valid and the reference, too, will have a 'static lifetime.

Outside of string literals, the 'static lifetime is rare, so you won't come across it very often. For those occasional times when you do need a non-string-literal variable to live for the duration of your program, you can declare that variable using the `static` keyword (instead of `let` or `const`):

```
static VAR_NAME: type = value;
```

where `VAR_NAME` is the variable name (written in what's known in the programming trade as `SCREAMING_SNAKE_CASE`), `type` is the data type, and `value` is the initial value. I know: This looks just like a constant! But there's a big difference:

- » With a constant, Rust performs a find-and-replace during the compile. That is, everywhere in the code where the constant name shows up, Rust replaces that name with the declared value of the constant.
- » With a static, Rust stores the data at a fixed location in memory that has a known address. This means you can create a reference to it. Constants, by contrast, don't have a guaranteed fixed address. The compiler is free to inline the value wherever it's used, so there's no single location to point to.

Here's an example of a static variable in action (bk03ch03/example08/):

```
// A simple struct to hold the application's configuration
struct AppConfig {
    timeout_ms: u32,
    max_retries: u8,
}
// This static variable holds the config and is always available
static CONFIG: AppConfig = AppConfig {
    timeout_ms: 5000,
    max_retries: 3,
};
fn main() {
```

```

    // The static is available here, of course
    println!("Starting app with {} retries...", CONFIG.max_retries);
    connect_to_server();
}
fn connect_to_server() {
    // But it's also available here
    println!("Setting server timeout to {}ms", CONFIG.timeout_ms);
}

```

This code defines the `AppConfig` struct to hold some application-wide data, and then declares `CONFIG` as a `static` instance of that struct. `CONFIG` has a `'static` lifetime, so it's available everywhere in the program for as long as the program runs.

Project: A Zero-Copy Config Parser

A common real-world requirement is to parse a configuration string with `KEY=value` pairs, one on each line, something like this:

```

URL = https://example.com
API_KEY = abc123xyz
PORT = 8080
ENVIRONMENT = production

```

To complete this chapter, in this project you build a parser to convert such a string into data your program can use. But this won't be any old parser. No, this will be a *zero-copy* parser, which means that it gives the program access to the key-value pairs without copying any string data. This approach makes the parser incredibly fast and efficient and is possible only because lifetimes enable you to implement everything safely.

Crank up a terminal session, navigate into your Rust project folder, and type the following command:

```
cargo new zero-copy-parser
```

Load this project's `main.rs` into your text editor, bid a fond farewell to the default `println!` statement, and get ready to code.

The first order of coding business is to fashion a struct to hold the key-value data. A Rust hash map is the perfect choice here. For the zero-

copy approach, the hash map needs to store `&str` keys and `&str` values. These string slices are references, so the struct requires a lifetime parameter:

```
use std::collections::HashMap;

#[derive(Debug)]
struct Config<'a> {
    data: HashMap<&'a str, &'a str>,
}
```

The `Config` struct is now tied to the lifetime `'a` of the original data string. I feel safer already.

Now it's time to implement a method on the `Config` struct for parsing the configuration string:

```
impl<'a> Config<'a> {
    // This method parses the configuration string and creates a Config
    fn parse(config_string: &'a str) -> Self {
        // Create a new, mutable hash map
        let mut data = HashMap::new();

        // Iterate over each line in the configuration string
        for line in config_string.lines() {
            // Use split_once to divide the line at the first '='
            if let Some((key, value)) = line.split_once('=') {
                // Insert the trimmed slices into hash map as key-value pairs
                data.insert(key.trim(), value.trim());
            }
        }

        // Return a new Config instance containing the parsed data
        Self { data }
    }
}
```

The `parse()` method takes a reference to the configuration string (with a lifetime `'a`) and returns a new `Config` instance (with the same lifetime `'a`). After forging a new, mutable hash map named `data`, the method iterates over each line in the configuration string. For each line, the function uses `split_once()` to split the line at the first equals sign (`=`) and then inserts the data into the hash map:

- » The part of the line to the left of `=` is the key.
- » The part of the line to the right of `=` is the value.

Once the iteration is complete, `parse()` returns a new `Config` instance loaded with the parsed data.

Okay, parsing the key-value pairs is one thing, but your program must be able to retrieve those values when it needs them. So, in the same `impl<'a> Config<'a>` block that you just used to implement the `parse()` method, add the following `get_value()` method:

```
fn get_value(&self, key: &str) -> Option<&'a str> {  
    self.data.get(key).copied()  
}
```

The `get_value()` method takes a key string slice as input and returns the corresponding value, if it exists. Just in case it doesn't, the returned value will be an `Option` where the `Some` will wrap a string slice with the lifetime `'a`.

One aspect of the `get_value()` method that likely has you scratching your head is the addition of the `copied()` method in the return expression. This method is normally used to convert an `Option<&T>` to an `Option<T>` by copying the contents of `Option`. In our function, the hash map's `get()` method returns a reference to the value, but in this project the value is already a reference, so `get()` returns `&&str`. The `Config` struct's `get_value()` method has to return `&str`, so the `copied()` method gets you there by converting `Option<&&str>` to `Option<&str>`.

Now we have everything we need to do some zero-copy parsing! Here's the `main()` function that pulls it all together:

```
fn main() {  
    // Here's the configuration string,  
    // which is a String that owns the data  
    let config_string = String::from(  
        "  
        URL = https://example.com  
        API_KEY = abc123xyz  
        PORT = 8080  
    )  
}
```

```

        ENVIRONMENT = production
    },
);

// Create a Config instance that borrows the configuration string
let config = Config::parse(&config_string);

// Get the API key
let api_key = config.get_value("API_KEY").unwrap_or("not found");
println!("API Key: {api_key}");

// Get the port
let port = config.get_value("PORT").unwrap_or("not found");
println!("Port: {port}");

// Debug print everything
println!("{:?}", config);
}

```

Output:

```

API Key: abc123xyz
Port: 8080
Config { data: {"PORT": "8080", "URL": "https://example.com",
[CA]"ENVIRONMENT": "production", "API_KEY": "abc123xyz"} }

```

Nice work! You've successfully crafted a configuration string parser that's efficient because it doesn't copy the string and safe because it uses lifetimes. You told the compiler exactly how long your data lives, and in return, it has guaranteed that your program is free from dangling references.

Chapter 4

Handling Errors Like a Pro

IN THIS CHAPTER

- » Understanding Rust's panic attacks
 - » Getting reacquainted with `Result`
 - » Taking advantage of Rust's error-related helper methods
 - » Making your own error types
-

Rust makes programming feel more like solving a puzzle than any other language I've used. I've never trusted a compiler more.

—JACOB LEWALLEN

Sleep apnea is a disorder in which a person unintentionally stops breathing for 10, 20, or even 30 seconds at a time during sleep. A lesser-known malady is *build apnea*, in which a programmer unconsciously stops breathing while their Rust code is compiling. Running `cargo build` is a tense time, for sure, but you can't beat that small but measurable squirt of dopamine that bathes your frontal cortex when your code compiles without an error.

But just because your code compiles, it doesn't follow that it will always run flawlessly. When your program faces the real world of distracted users, unreliable connections, and poorly documented APIs, bad things can happen to good code. You need to follow Rust's lead here and *assume* that your program will generate errors. Fortunately, error handling is baked into Rust like chocolate chips in a cookie, and in this chapter, you get a taste of the sweetness of a properly handled error.

Recoverable versus Unrecoverable: Know When to Panic!

When you use a vending machine, most of the time everything works fine: The machine takes your bill, and your chosen snack drops perfectly into the bin. But occasionally things don't go that smoothly:

- » Sometimes the machine rejects your bill or the snack doesn't quite release. That's annoying, but you can usually recover either by smoothing out the bill (or taking out a different one) and trying again, or by giving the machine a strategic shake to release the stuck snack.
- » Sometimes the machine takes your money, does nothing, and refuses to return your cash. There's no recovering from this, so you either take the loss or call the number on the machine so you have someone to yell at.

Rust, too, views errors as either recoverable or unrecoverable.

Recoverable errors: Handle gracefully mode

A *recoverable error* is one that you can not only reasonably anticipate but also handle right in your code. If you want the user to enter a number between 1 and 100, your code can look for numbers outside that range and ask the user to try again. In Rust, you handle recoverable errors by using the `Result` enum that I introduce in Book 2, [Chapter 3](#) (and that you learn much more about later in this chapter).

Unrecoverable errors: Panic mode

An *unrecoverable error* occurs when your program comes upon a situation or condition that's so dire the program can't reasonably or safely continue. In such cases, Rust pulls a hissy fit in the guise of the `panic!` macro, which stops your program dead in its tracks, scrubs the stack clean, prints an error message, and exits. It's the programming equivalent of flipping the table and storming out of the room. It's Rust at

its most dramatic, but a panic happens only when Rust has no other option. Here's an example (bk03ch04/example01/ in this book's example files):

```
fn main() {  
    let primes = vec![2, 3, 5, 7, 11];  
    println!("The fifth prime number is {}", primes[5]);  
}
```

This code defines a five-element vector named `primes`, and then tries to print the last item in the vector by accessing `primes[5]`. But the index values for a five-element vector run from 0 to 4, so `primes[5]` is invalid (an example of the infamous *off-by-one* errors that have plagued programmers from day one — or maybe from day zero!) and Rust has no choice but to panic:

```
thread 'main' panicked at src/main.rs:3:52:  
index out of bounds: the len is 5 but the index is 5
```

This error message tells you two things:

- » **Where the error occurred:** In this example, the error occurred in the `main` function (that's what `thread 'main'` is referring to) of the `main.rs` file, on line 3, column 52 (just prior to the problematic `[5]` reference).
- » **What went wrong:** In this example, the code generated an `index out of bounds` error, specifically where the `len` is 5 (that is, the maximum legal index value is 4), but the `index` is 5 (an invalid value).

Interestingly, Rust also enables you to handle these kinds of catastrophic errors yourself by calling the `panic!` macro with a custom error message. For example (bk03ch04/example02/):

```
fn get_first_element(vec: &[i32]) -> i32 {  
    if vec.is_empty() {  
        panic!("Return the first element of an empty vector? Get real.");  
    }  
    vec[0]  
}
```

```
fn main() {
    let primes = vec![];
    let first = get_first_element(&primes);    // Panic time!
    println!("The first element is {first}");
}
```

This code defines an empty vector and passes a reference to the vector to the `get_first_element()` function, which returns the first element but not before checking whether the vector is empty. If it is, the code calls the `panic!` macro with a custom error. Here's the result:

```
thread 'main' panicked at src/main.rs:3:9:
Return the first element of an empty vector? Get real.
```

In this example, passing a reference to an empty vector to a function that expects a non-empty vector is pure programmer error. Bugs are what the `panic!` macro is designed to handle, so you'll use `panic!` a lot when you're testing and prototyping your code.

But most of your error handling will be in the service of dealing with recoverable errors that you can readily expect might happen at runtime, such as user input errors. Handling recoverable errors is the province of the `Result` enum, discussed next.

Result<T, E>: Your Go-to Type for Errors

If `panic!` is the Danish battle-axe of Rust error handling, `Result` is the chef's knife: versatile, precise, and endlessly useful for day-to-day coding. I introduce you to `Result` back in Book 2, [Chapter 3](#), but it's time to zoom in a little closer on this crucial Rust tool.

Result: A review

The `Result` type is an enum with two variants:

```
enum Result<T, E> {
    Ok(T),    // Everything went according to plan. Here's your value of type T.
```

```
Err(E), // Doh! Something went wrong. Here's an error of type E, instead.
}
```

`Result` is a kind of package delivery service for some data. Either you get `Ok(here's_your_package)` or you get `Err(reason_for_non-delivery)`. The important thing is that Rust forces you to check which of these you received. You can't just assume the package arrived and start using it.

Here's an example (bk03ch04/example03/):

```
fn price_per_unit(price: f64, units: f64) -> Result<f64, String> {
    if units == 0.0 {
        Err(String::from("Hey, whoa, you can't divide by zero!"))
    } else {
        Ok(price / units)
    }
}

fn main() {
    let result1 = price_per_unit(18.95, 50.0);
    println!("{:?}", result1);

    let result2 = price_per_unit(32.95, 0.0);
    println!("{:?}", result2);
}
```

Output:

```
Ok(0.379)
Err("Hey, whoa, you can't divide by zero!")
```

The `price_per_unit()` function takes a `price` and the number of `units` and calculates the price for a single unit (handy for comparison shopping). The return value type is `Result<f64, String>`. In the function, an `if` statement checks whether the `units` value is `0.0`. If it is, the function returns the `Err` variant with an error message; otherwise, the function returns the result of `price / units` in the `Ok` variant.

Pattern matching on Result

The standard way to handle the `Result` type is with pattern matching with the `match` keyword, which I introduce way back in Book 1, [Chapter 6](#). Although explicitly handling both variants takes a little extra effort on

your part, you're making your code safer and easier to understand. Here's an example (bk03ch04/example04/):

```
fn check_password(password: &str) -> Result<(), String> {
    if password.len() < 8 {
        Err(String::from("Password too short!"))
    } else if password.contains(' ') {
        Err(String::from("Password can't contain spaces!"))
    } else {
        Ok(())
    }
}

fn main() {
    let password = "abc123";
    match check_password(password) {
        Ok(()) => println!("Password accepted!"),
        Err(problem) => println!("Password rejected: {problem}"),
    }
}
```

Output:

```
Password rejected: Password too short!
```

Besides noting how the `match` expression handles both the `Ok` and `Err` variants, there are a couple of other important things to notice in this code:

- » In cases where your code doesn't need to bundle a value with the `Ok` variant, return the empty tuple `()` inside `Ok`.
- » It's perfectly fine, even common, to handle multiple error conditions in your code.

Saving Time with Helper Methods

Pattern matching on the `Result` type, as I describe in the preceding section, is the way to go when you want full control of the `Result` or when you want your code to be as explicit as possible. However, for simple examples, writing out both variants is often overkill and you really just want to get on with your life. Rust has your back here by

offering a few useful helper methods that make it faster and easier to handle `Result` types in your code.

Living dangerously: `unwrap()` and `expect()`

Every now and then, you have the luxury of being lead-pipe-cinch certain that a particular `Result` will be `Ok`. Perhaps you've supplied the data yourself or some other logic in your code guarantees success. For these rare occasions, you can extract the `Ok` value directly by using the `unwrap()` method. If the `Result` really is `Ok`, `unwrap()` will return the value. But if the `Result` is `Err`, instead, your program will panic.

Rust also offers the `expect()` method, which does the same thing as `unwrap()`, but lets you specify a custom error message. The name `expect` reflects the intent: You're saying, "I expect this to succeed, but it doesn't, here's why." It's a way of documenting your assumption right where you make it. Here's an example ([bk03ch04/example05/](#)):

```
fn main() {
    let good_input = "42";
    let bad_input = "forty-two";

    // This is fine
    let number = good_input.parse::<u32>().unwrap();
    println!("The number is *drumroll*: {number}!");

    // This will panic if you uncomment it
    //let uhoh = bad_input.parse::<u32>().expect("Failed to parse number!");
}
```

Output:

```
The number is *drumroll*: 42!
```

If you were to uncomment the final statement and run the code, the program would panic and you'd see your custom error message:

```
thread 'main' panicked at src/main.rs:10:44:
Failed to parse number!: ParseIntError { kind: InvalidDigit }
```

You really, really, *really* want to exercise caution when using `unwrap()` or `expect()`. Using them and having an `Err` crop up anyway is a common source of panics in Rust programs. These methods are fine for

prototyping and testing, but your production code needs to be using one of the helper methods I discuss over the next few sections. (If you're wondering what the heck a `ParseIntError` is, refer to the later section [“Grokking Rust’s Error Types.”](#))

Living alternatively I: `unwrap_or()`

Rather than winging it with either `unwrap()` or `expect()`, a much safer approach is to provide an alternative, just in case an error crops up. The `unwrap_or()` method gives you the value if the `Result` is `Ok` or an alternative value you provide if the `Result` is `Err`.

The following example puts `unwrap_or()` to the test (bk03ch04/example06/):

```
fn set_temperature(temp: f64) -> Result<f64, ()> {
    if temp < 0.0 || temp > 100.0 {
        return Err(());
    }
    Ok(temp)
}

fn main() {
    println!("Setting the temperature...");

    // Example 1: unwrap_or() with Ok value
    let temp1 = 66.0;
    let result1 = set_temperature(temp1);
    let value1 = result1.unwrap_or(72.0);
    println!(
        "You asked for {temp1} degrees. \
        Temperature set to {value1} degrees."
    );

    // Example 2: unwrap_or() with Err value
    println!();
    println!("Setting the temperature...");
    let temp2 = 105.0;
    let result2 = set_temperature(temp2);
    let value2 = result2.unwrap_or(72.0);
    println!(
        "You asked for {temp2} degrees. \
        Temperature set to {value2} degrees."
    );
}
```


Output:

```
Setting the temperature...
You asked for 66 degrees. Temperature set to 66 degrees.

Setting the temperature...
You asked for 105 degrees. Temperature set to 72 degrees.
```

The `set_temperature()` function accepts a temperature value (`temp: f64`) and returns `Result<f64, ()>`, which means a floating-point value in `Ok`, or nothing — that is, `()` — in `Err`. The function checks whether `temp` is in the range 0 to 100. If it's not, the function returns `Err(())`; if it is, the function returns `Ok(temp)`.

In `main()`, the first example passes a legit temperature (`66.0`) to the function, so it gets that temperature back by unwrapping `Ok` via `unwrap_or()`. The second example passes an illegal value (`105.0`) to the function, so it gets back an `Err`. In this case, the error is ignored, and the resulting value is the alternative value of `72.0` defined in `unwrap_or(72.0)`.

Living alternatively II: unwrap_or_else()

The `unwrap_or_else()` method is similar to `unwrap_or()` but uses a closure to define what happens when `Result` is `Err`. Wrapping the default in a closure is useful if your alternative value is expensive to compute, since the closure runs only if there's an error. More commonly, `unwrap_or_else()` is useful also when you need access to the error message returned with `Err`. Here's an example ([bk03ch04/example07/](#)):

```
fn set_temperature(temp: f64) -> Result<f64, &'static str> {
    if temp < 0.0 || temp > 100.0 {
        return Err("The temperature must be between 0 and 100!");
    }
    Ok(temp)
}

fn main() {
    println!("Setting the temperature...");

    // Example 1: unwrap_or_else() with Ok value
    let temp1 = 66.0;
```

```

    let result1 = set_temperature(temp1);
    let value1 = result1.unwrap_or_else(|err| {
        println!("{err}");
        72.0
    });
    println!(
        "You asked for {temp1} degrees. \
        Temperature set to {value1} degrees."
    );

    // Example 2: unwrap_or() with Err value
    println!();
    println!("Setting the temperature...");
    let temp2 = 105.0;
    let result2 = set_temperature(temp2);
    let value2 = result2.unwrap_or_else(|err| {
        println!("{err}");
        72.0
    });
    println!(
        "You asked for {temp2} degrees. \
        Temperature set to {value2} degrees."
    );
}

```

Output:

```

Setting the temperature...
You asked for 66 degrees. Temperature set to 66 degrees.

Setting the temperature...
The temperature must be between 0 and 100!
You asked for 105 degrees. Temperature set to 72 degrees.

```

This code performs the same task as the code in the preceding section, but with the following differences:

- » The `set_temperature()` function returns `Result<f64, &'static str>`, where the `Err` result wraps a string literal error message.
- » In `main()`, both examples handle the function `Result` with `unwrap_or_else()`.
- » Example 1 passes a legal temperature (`66.0`) to the function, so it gets that temperature back by unwrapping `Ok` via `unwrap_or_else()`.

- » Example 2 passes an illegal temperature (105.0) to the function, so it gets back an `Err`. In this case, the error is handled with a closure that uses the error value (`err`) as the closure argument. The closure executes two statements:

```
println!("{err}");  
72.0
```

The first line prints the error string, and the second line specifies the return value (72.0).

Living sensibly: unwrap_or_default()

The `unwrap_or()` method, which I describe earlier, is a welcome addition to your Rust error-handling toolkit, but you might find yourself setting the same alternative values over and over, such as 0 for integers.

To handle situations where Rust's predefined type defaults are just the ticket, use the `unwrap_or_default()` method, which, when `Result` is the `Err` variant, returns the default value for the `Result` type:

- » 0 for an integer
- » 0.0 for a floating-point type
- » `false` for a Boolean
- » The empty string ("") for a string

Technically, Rust defines a default value for any type that implements the `Default` trait.

Here's an example (bk03ch04/example08/):

```
fn read_stock_count(item_name: &str) -> Result<i32, String> {  
    match item_name {  
        "apples" => Ok(50),  
        "bananas" => Ok(25),  
        _ => Err("Item not found in inventory".to_string()),  
    }  
}  
  
fn main() {
```

```
let apple_stock = read_stock_count("apples").unwrap_or_default();
let orange_stock = read_stock_count("oranges").unwrap_or_default();
println!("Apples in stock: {}", apple_stock);
println!("Oranges in stock: {}", orange_stock);
}
```

Output:

```
Apples in stock: 50
Oranges in stock: 0
```

The `read_stock_count()` function mimics reading the products in stock from a file and returns `Result`. Depending on the item name, the stock amount is wrapped in `Ok`, but if the item name has no match, `Err` gets returned. In `main()` the two statements that call the `read_stock_count()` function both use `unwrap_or_default()` to handle `Result`. In the second case, no oranges are in stock, so the code sensibly defaults to 0 for the count.

Propagating Errors with the ? Operator

One of the problems with handling errors is that it can clutter your code, making your programs harder to read and harder to troubleshoot. One way to clean up your code is to tell Rust to send any errors that arise in a function back to the caller. This way, you keep the function logic clean and all the `Ok` and `Err` matching happens in one place. Here's an example (bk03ch04/example09/):

```
use ::std::num::ParseIntError;

fn parse_age(text: &str) -> Result<u8, ParseIntError> {
    let n = text.trim().parse::<u8>()?;
    Ok(n)
}

fn main() {
    match parse_age("42") {
        Ok(age) => println!("Age is {age}."),
        Err(e) => println!("Couldn't parse age: {e}."),
    }
}
```

```
match parse_age("forty-two") {  
    Ok(age) => println!("Age is {age}."),  
    Err(e) => println!("Couldn't parse age: {e}."),  
}  
}
```

Output:

```
Age is 42.  
Couldn't parse age: invalid digit found in string.
```

The `parse_age()` function takes a string slice (`text`) and tries to use `parse()` to convert the string to a number. That conversion works like a charm when `text` is something `parse()` knows how to handle, such as the string `"42"`. However, that conversion will fail if `text` isn't something `parse()` knows how to convert, such as the string `"forty-two"`, so `parse()` returns `Result`. Note that `parse_age()` returns `Result<u8, ParseIntError>`. That return type is the promise that lets `?` work:

- » If `parse()` returns `Err`, the `?` returns that `Err` from `parse_age()` immediately.
- » Otherwise, if `parse()` returns `Ok(n)`, `?` yields `n` and the function returns normally.

The key here is that the caller — `main()` in this case — decides what to do. In this example, it performs a `match` on `Ok` or `Err`.

Grokking Rust's Error Types

You've now seen types such as `ParseIntError` a few times in this chapter, and you might be wondering what they're all about. Why does Rust have these specific error types instead of just returning a simple string that says "Ahem, something went wrong"?

The answer is that these error types are structured data. A type such `ParseIntError` is just a regular struct whose job is to be a precise,

machine-readable messenger that carries specific details about a failure. Think of it this way:

- » A string error is like your spouse telling you, “The car won’t start!” You know there’s a problem, but you don’t know why.
- » A specific error type is like a mechanic handing you a diagnostic report that says, “The battery is dead,” “The fuel tank is empty,” or “There’s a squirrel in the engine.” You know there’s a problem *and* you know why.

Because the error is a specific type, your code can inspect it and make smart decisions. `ParseIntError`, for example, can tell you why a string couldn’t be parsed. Was it empty? Did it contain an invalid digit? Was the number too big to fit in the target type? This approach enables you to give much more helpful feedback to the user (bk03ch04/example10/):

```
use std::num::ParseIntError;
use std::io;

fn demonstrate_parse_error() -> Result<i32, ParseIntError> {
    "definitely_not_a_number".parse::<i32>()
}

fn demonstrate_io_error() -> Result<String, io::Error> {
    std::fs::read_to_string("probably_doesnt_exist.txt")
}

fn main() {
    match demonstrate_parse_error() {
        Ok(n) => println!("Got number: {}", n),
        Err(e) => {
            println!("Parse failed: {}", e);
            println!("Error kind: {:?}", e.kind());
        }
    }

    match demonstrate_io_error() {
        Ok(contents) => println!("File contents: {}", contents),
        Err(e) => {
            println!("I/O operation failed: {}", e);
            println!("Error kind: {:?}", e.kind());
        }
    }
}
```

```
}  
}
```

Output:

```
Parse failed: invalid digit found in string  
Error kind: InvalidDigit  
I/O operation failed: No such file or directory (os error 2)  
Error kind: NotFound
```

The `demonstrate_parse_error()` function returns `ParseIntError`, while the `demonstrate_io_error()` function returns `io::Error`. In both cases, `Err` includes a specific error message, but note that you can also call methods on each error, such as the `kind()` method demonstrated here.

What all of Rust’s built-in error types have in common is that they implement the standard `std::error::Error` trait. This trait is what allows these error types to work seamlessly with tools such as the `?` operator and other error-handling libraries. When you forge your own error enums (as I describe a bit later, in the “[Creating Your Own Error Types](#)” section), you implement this same trait to make them first-class citizens in Rust’s error-handling world.

Magically Creating a Result from an Option

Every now and then (more now than then, actually) you’ll find yourself in a situation where some Rust method or other code has provided you with an `Option<T>`, but the function you’re working in needs to return a `Result<T, E>`. This often happens when a failure needs more context. `None` tells you (or the user) that nothing was found but doesn’t explain why. Was the database down? Was the item not there?

To bridge this gap, Rust provides a couple of handy methods that act like magic wands, transforming `Option` into `Result`.

Converting None with ok_or()

The `ok_or()` method is your go-to tool for a simple conversion from `Option` to `Result`. It works like this:

- » If `Option` is `Some(value)`, it transforms into `Ok(value)`.
- » If `Option` is `None`, it transforms into `Err(error_value)`, where `error_value` is an error value that you provide.

For example, suppose your code gets a user's ID from a database (which I represent in the following code by a hash map). The `.get()` method returns `Option`. If the user isn't found, `Option` returns `None`, but it would be better if your function returned a helpful `Err` string instead. Here's how (bk03ch04/example11/):

```
use std::collections::HashMap;

fn get_user_id(users: &HashMap<&str, u32>, name: &str) -> Result<u32, String>
{
    // .get() returns an Option<&u32>, .copied() turns it into Option<u32>
    let user_id_option = users.get(name).copied();

    // Convert the Option to a Result. If it's None, use the provided string.
    let user_id_result = user_id_option.ok_or(String::from("User not
found"));

    user_id_result
}

fn main() {
    let mut users: HashMap<&'static str, u32> = HashMap::new();
    users.insert("alice", 101);

    println!("{:?}", get_user_id(&users, "alice"));
    println!("{:?}", get_user_id(&users, "bob"));
}
```

Output:

```
Ok(101)
Err("User not found")
```

The `get_user_id()` function returns `Result<u32, String>`, but the function's `.get()` hash map method returns `Option<&u32>`. (Why `&u32`?

Because `.get()` returns a reference to the value corresponding to the provided key, and the `users` hash map is defined with `u32` values.) You don't want a reference here, so the `.copied()` method is tacked on to convert `Option<u32>` to `Optionu32`. The next line uses `ok_or()`, which either transforms the `Some(id)` into `Ok(id)` if the passed `name` was found, as it does in the first example call, or transforms the (in this case) unhelpful `None` into `Err("User not found")`, as it does in the second example call.



TIP The `ok_or()` method is especially powerful when combined with the `?` operator because it enables you to find a value and propagate an error in a single, clean line:

```
let user_id = users.get(name).copied().ok_or(String::from("User not found"))?;
```

Lazily converting with `ok_or_else()`

In Rust, function arguments are evaluated before the function itself is called. So, in the `get_user_id()` function from the preceding section, the `String::from("User not found")` expression is run and a new `String` is allocated on the heap every single time, regardless of whether `Option` is `Some` or `None`. If the user is found and `Option` is `Some`, that newly created `String` is immediately discarded without being used, which is a small but real inefficiency. In Rust lingo, the `ok_or()` method is said to be *eager* because it always takes a value that has already been created.

The `ok_or_else()` method is the slightly more advanced sibling of `ok_or()`. Instead of taking an error value directly (eager evaluation), it takes a closure that it will run *only* if `Option` is `None`. In the Rust business, this is known, unjudgementally, as *lazy* evaluation.

As with the `unwrap_or_else()` method that I discuss earlier, you use `ok_or_else()` when your error value is more computationally expensive

or you need something a bit more elaborate than a simple string. A common example is when you need to build a complex error string that includes the input that failed. You don't want to waste time building that string unless there's an error.

Here's another version of the code from the preceding section that takes advantage of `ok_or_else()` to lazily include the failed username in the error message (bk03ch04/example12/):

```
use std::collections::HashMap;

fn get_user_id(users: &HashMap<&str, u32>, name: &str) -> Result<u32, String>
{
    users.get(name).copied().ok_or_else(|| {
        // This code runs only if the user was not found
        println!("Generating error message for '{name}'...");
        format!("User '{name}' does not exist.")
    })
}

fn main() {
    let mut users: HashMap<&'static str, u32> = HashMap::new();
    users.insert("alice", 101);

    println!("{:?}", get_user_id(&users, "alice"));
    println!("{:?}", get_user_id(&users, "bob"));
}
```

Output:

```
Ok(101)
Generating error message for 'bob'...
Err("User 'bob' does not exist.")
```

Creating Your Own Error Types

For small programs, returning `Err(String)` when things go awry in some way is almost always good enough. But as your programs grow and become more complex and mature, you'll want to provide your users with errors that are more descriptive than just a simple `String`. Welcome, then to the world of custom error types, which enable you to

categorize and handle different failure modes in ways that makes sense for your app.

Bolstering your program with custom error types is usually a four-step operation:

1. Create an enum in which the variants are your custom error types.
2. Implement the `Display` trait for your error enum so that you can define the error messages.
3. Implement the `Error` trait to make your custom errors work with Rust's error ecosystem.
4. Implement the `From` trait so that the `?` operator can automatically convert from a standard library error type to one of your custom error types.

Here's a program that loads (or, really, mimics loading) server configuration data and implements several custom error types (bk03ch04/example13/):

```
use std::error::Error;
use std::fmt;
use std::num::ParseIntError;

// 1. Custom error enum with different variants
#[derive(Debug)]
enum ConfigError {
    Missing(String),           // Variant with context
    InvalidPort(ParseIntError), // Wraps another error type
    IllegalPort(String),       // Another variant with context
}

// 2. Implement Display for human-friendly messages
impl fmt::Display for ConfigError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            ConfigError::Missing(s) => write!(f, "Missing config: {}", s),
            ConfigError::InvalidPort(e) => write!(f, "Invalid port #: {}",
e),
            ConfigError::IllegalPort(s) => write!(f, "Illegal port #: {}",
s),
        }
    }
}
```

```

    }
}

// 3. Implement Error trait
impl Error for ConfigError {}

// 4. Implement From for automatic conversion with ?
impl From<ParseIntError> for ConfigError {
    fn from(error: ParseIntError) -> Self {
        ConfigError::InvalidPort(error)
    }
}

// Using the custom error type
fn load_server_config(port_str: &str) -> Result<u16, ConfigError> {
    if port_str.is_empty() {
        return Err(ConfigError::Missing("port".to_string()));
    }

    let port = port_str.parse::<u16>()?;

    if port < 1024 {
        return Err(ConfigError::IllegalPort("port must be >=
1024".to_string()));
    }

    Ok(port)
}

fn main() {
    let test_ports = vec!["3000", "", "8o8o", "80"];

    for port_str in test_ports {
        match load_server_config(port_str) {
            Ok(port) => println!("{}", port_str, port),
            Err(e) => println!("{}", port_str, e),
        }
    }
}

```

Output:

```

'3000' -> Port 3000
'' -> Error: Missing config: port
'8o8o' -> Error: Invalid port #: invalid digit found in string
'80' -> Error: Illegal port #: port must be >= 1024

```

The first step is covered by the `ConfigError` enum, which defines three custom error types as its variants: `Missing`, `InvalidPort`, and `IllegalPort`. The `Missing` and `IllegalPort` variants each wrap a `String` value that will define extra error context for the user; the `InvalidPort` variant wraps the standard library's `ParseIntError` error, which later the code converts to `ConfigError` when it implements the `From` trait.

The second step is to implement the `Display` trait for `ConfigError`. For each variant, the `Display` trait defines a human-readable error message.

The third step is to implement the `Error` trait for `ConfigError`. Implementing the `std::error::Error` trait is how you tell the Rust compiler that your custom enum is an official error type. This marker trait acts like a contract, allowing your type to interoperate with the wider Rust error-handling ecosystem. You might not need all that right now, but it's a one-liner that should be part of your custom error code.

The fourth and final step is to implement the `From` trait for `ConfigError`. In this example, the goal is to convert the standard library's `ParseIntError` error type to the custom `ConfigError` type, which enables the code to use `?` to propagate the error.

The code defines a `load_server_config()` function that takes a port (as a string slice) and returns `Result<u16, ConfigError>`:

- » If the string is empty, the function returns `ConfigError::Missing` in `Err`.
- » If the port string contains an invalid value (such as the letter `o` in `8o8o`), the function uses `?` to convert `ParseIntError` to `ConfigError` and propagate `Err` back to the caller.
- » If the parsed port value is less than 1024, the function returns `ConfigError::IllegalPort` in `Err`.
- » Otherwise, the function just returns the parsed port number in `Ok`.

Chapter 5

Macros: When Code Writes Code

IN THIS CHAPTER

- » Getting reacquainted with some old macro friends
 - » Peeking under the macro hood
 - » Writing your own macros
 - » Making a useful macro for timing code
-

Any sufficiently advanced macro is indistinguishable from magic.

—BAROSL LEE

In Book 1, [Chapter 1](#), I mention that Rust has been voted Stack Overflow’s most admired programming language a billion or so years in a row. That unflagging admiration is generated by many aspects of the language, but one of the major reasons is that the Rust team really seems to care about programmers. That caring is seen in the astonishing quality of the compiler’s messages, but it also comes up in the many ways that Rust tries to make a programmer’s life easier by providing shorthand methods for common tasks. The `for` loop, for example, is a shorthand for a common iterator task; `if let` statements hide some underlying pattern-matching machinery; and the `?` operator hides some behind-the-scenes `match` complexity. These shorthands not only help you code faster but also make your code more readable and more understandable.

In this chapter, you explore yet another programmer-friendly shorthand: macros. You learn what they are and even how to create your own!

Useful Macro Magic: *println!* and *vec!*

In Rust, a *macro* is code that writes other code. (Computer science uber-nerds refer to this as *metaprogramming*, but just ignore them.) When you use a macro such as `println!` in your program, the Rust compiler expands it into regular Rust code before it tackles the full compile. Why not just have you write that regular Rust code in the first place? Because every macro is designed to hide some complexity that would make your code harder to write and harder to read.

Take your (by now) old friend `println!` (bk03ch05/example01/ in this book's example files):

```
fn main() {  
    let name = "Alice";  
    println!("Hello, {name}!");  
}
```

Internally, the compiler expands this code to the following:

```
fn main() {  
    let name = "Alice";  
    {  
        ::std::io::_print(format_args!("Hello, {0}!\n", name));  
    };  
}
```

The `::std::io::_print()` function is used internally to print to `stdout`, and `format_args!` is an internal Rust macro that makes sure the format string can be printed. This expanded version is more complex and harder to read than the simple `println!` syntax. And, technically, the compiler won't even let you use the expanded code directly in your program because `_print()` is marked as for-internal-use only. (The `format_args!` macro is public, but it's designed to work with the formatting machinery, not to be used on its own.)

The `vec!` macro tells a similar tale (bk03ch05/example02/):

```
fn main() {
    let names = vec!["Alice", "Bob", "Carol"];
}
```

Here’s what the compiler turns this into:

```
fn main() {
    let names = <[_]>::into_vec(::alloc::boxed::box_new(["Alice", "Bob",
"Carol"]));
}
```

Hunh!? This looks like gibberish, but it reveals exactly how `vec!` builds a vector at runtime. There are three main parts:

```
["Alice", "Bob", "Carol"]
```

This part is a fixed-size array of string slices (`&str`), allocated on the stack.

```
::alloc::boxed::box_new(...)
```

This call allocates that array on the heap and returns a `Box<[&str; 3]>` type. `Box<T>` is a *smart pointer* that owns heap memory. (Smart pointers are an advanced Rust topic, so I don’t cover them much in this book, except for a brief mention in Book 3, [Chapter 6](#).) So, at this point, the array’s contents have moved from the stack into heap storage.

```
<[_]>::into_vec(...)
```

The `<[_]>` syntax means “the slice type.” The underscore (`_`) is Rust’s way of saying to the compiler, “Hey, figure out the element type for me, would you?” That is, it acts as a placeholder that tells the compiler to infer the type from context. So, `<[_]>` effectively means `<[&str]>` here, since the array contains string slices. Calling the slice’s associated function `into_vec()` converts a boxed slice (`Box<[T]>`) into a `Vec<T>`.

That’s a lot of machinery! It’s a pain to type and, more importantly, it’s unsafe if you get the allocation or pointer conversions wrong. That’s not a worry, though, because the `vec!` macro hides all that complexity and does everything safely and economically.

Using Macros in Your Own Code

Rust offers prefab macros such as `println!` and `vec!` to make your coding life easier and safer, which is very nice of Rust. If that was the entirety of what macros can do, they would still be very much worth understanding (and this would be the shortest chapter in the history of technical books!). But Rust's internal macros aren't the full story. The really exciting part is that you can craft your *own* macros, which can help reduce some of the repetitive aspects of your code and also enable you to perform some tasks that you just can't do with, say, a regular Rust function.

For example, a classic limitation of a Rust function is that the number of arguments is fixed. If you want to add two numbers, you define a function that accepts two inputs. If you want to add three numbers, well, you need a different function that accepts three inputs:

```
// Add two numbers? You got it:
fn add_two(a: i32, b: i32) -> i32 {
    a + b
}

// Add three numbers? You need another function:
fn add_three(a: i32, b: i32, c: i32) -> i32 {
    a + b + c
}

// Add four numbers? This gets tedious in a hurry.
```

Ah, but this is a perfect use case for a macro. You can write, say, a single `sum!` macro that can take *any* number of inputs and add them. Being able to handle a variable number of arguments at the call site is one of the main reasons to reach for a macro. Yes, sure, you can pass a slice or an iterator to a function, but then your callers have to write `sum(&[1, 2, 3])` instead of just `sum!(1, 2, 3)`. Macros let you write cleaner, more natural-looking code.

Macro Syntax: Learning to Read the Hieroglyphics

Okay, it's time to learn how to write your own macros. Be warned, though: Examining any non-trivial macro for the first time might make you think the writer's cat walked on their keyboard! Macros can get a bit cryptic, for sure, but rest assured there's a satisfying amount of method behind the on-the-surface madness.

The most common type of macro is a declarative macro, defined with `macro_rules!`. It's called *declarative* because you declare patterns and what they should expand to, rather than writing code that manipulates the syntax tree directly. (Rust also has procedural macros that do exactly that, but they're an advanced topic beyond the scope of this book.) Here's the general syntax:

```
macro_rules! macro_name {  
    (matcher) => {  
        expansion  
    };  
}
```

where `macro_name` is the name you provide for the macro; `matcher` defines the pattern that the macro input is compared against; and `expansion` is a kind of template for the code that the compiler will generate whenever the input matches the pattern.

The simplest pattern is straight text (bk03ch05/example03/):

```
macro_rules! toggle {  
    (on) => {  
        true  
    };  
    (off) => {  
        false  
    };  
}  
  
fn main() {  
    let switch1 = toggle!(on);  
    let switch2 = toggle!(off);  
    println!("switch1 = {switch1}, switch2 = {switch2}");  
}
```

Output:

```
switch1 = true, switch2 = false
```

Here's an example with an input (bk03ch05/example04/):

```
macro_rules! shout {
    ($word:expr) => {
        println!("{}", $word.to_uppercase());
    };
}

fn main() {
    shout!("help");
}
```

Output:

```
HELP!!!
```

The pattern `$word:expr` defines a *metavariable* as follows:

- » **\$:** What follows is a variable.
- » **word:** The name of the variable.
- » **:expr:** Match any input expression (anything that evaluates to a value).

In this example, the "help" input string gets bound to the `$word` variable, which is used in the `println!` statement.

The power of macros really starts to show when you create repetition patterns. When you include a plus sign (+) in your pattern, you're telling the compiler that the pattern can repeat one or more times. Similarly, when you include an asterisk (*), you're saying that the pattern can repeat zero or more times. Here's an example (bk03ch05/example05/):

```
macro_rules! print_em {
    ( $( $x:expr ),* ) => {
        $(
            print!("{}", $x);
        )*
    };
}

fn main() {
    print_em!(1, 2, 3, 'a', 'b', 'c');
    println!();
}
```

```
print_em!(10, 20, 30);  
println!();  
print_em!(2, 3, 5, 7, 11, 13, 17, 19);  
println!();  
}
```

Output:

```
123abc  
102030  
235711131719
```

Here's what's going on with the pattern `$ ( $x:expr ), * ):`

- » `$ ()`: Tells the compiler to temporarily save — or *capture* — whatever matches the pattern that appears between the parentheses.
- » `$x:expr`: Matches any expression and binds it to the variable `$x`.
- » `,`: Specifies a literal separator.
- » `*`: Specifies that the pattern can repeat zero or more times.

In other words, the `$ (...), *` syntax tells the compiler to capture whatever matches the pattern inside the parentheses zero or more times, each separated by a comma. This pattern means that you can send 0, 1, 2, 3, or any number of inputs to the `print_em!` macro.

In the macro's expansion part, a `print!` statement that uses the `$x` variable is enclosed in `$ (...)*`, which tells the compiler to repeat the code between the parentheses for every `$x` captured in the pattern.

Writing Macros

You always begin writing a declarative macro with `macro_rules!`, which is a built-in macro that Rust uses internally to create macros, followed by the macro name (without the exclamation point) and braces:

```
macro_rules! sum {  
}
```

Next you define your matcher, which always resides within parentheses — `(` and `)` — and is followed by a fat arrow (`=>`) and a code block:

```
( matcher ) => {  
};
```

For a matcher that requires a pattern, you use `$()` to tell the compiler that you want it to capture a pattern:

```
( $( pattern ) => {  
};
```

Next you specify what you want to capture. This is usually one of the following:

- » **An expression:** Use `$var:expr`, where `var` is the variable name you want to use for each captured expression.
- » **A literal:** Use `$var:literal`, where `var` is the variable name you want to use for each captured literal (such as a string literal).
- » **A code block:** Use `$var:block`, where `var` is the variable name you want to use for each captured block.
- » **An identifier:** Use `$var:ident`, where `var` is the variable name you want to use for each captured identifier (such as a variable name or a function name).
- » **A type:** Use `$var:type`, where `var` is the variable name you want to use for each captured type.

You can also include a literal value either within the pattern (if, say, your pattern includes two captured items, separated by a colon) or outside the pattern (if, say, each captured item is separated by a comma). And you can specify the repetition by following the pattern with one of the following:

- » *****: The pattern occurs zero or more times.
- » **+**: The pattern occurs one or more times.

Your matcher can also take advantage of the `?` symbol, which matches a pattern that occurs zero times or one time.

Finally, in the code block, you specify the expansion, which appears in its own code block:

```
( $( pattern ) => {  
    {  
        $(  
            // Expansion code with captured elements  
            // Possible return expression  
        )*|+  
        // Possible return expression  
    }  
};
```

Within `$()`, type the code that uses the values captured in the matcher, including the same repetition specifier (`*` or `+`) that you used in the matcher. You complete the expansion code block by specifying the macro's return value:

- » If your expansion code computes the return value, use an expression (that is, a statement without a closing semicolon) as the final line of the expansion code.
- » Otherwise, use an expression immediately after the expansion code.

Okay, I know that's all annoyingly vague and placeholder-y. The next two sections take you through concrete examples that I hope will make everything clear.

Example 1: Adding any number of values

Earlier (back in the “[Using Macros in Your Own Code](#)” section), I mention that regular Rust functions can't handle a variable number of arguments. But, as I describe in the preceding section, when you use `*` as part of a macro pattern, your macro can take any number of inputs, even none. (If you use `+` in the pattern, instead, your macro can take any number of inputs but must take at least one.)

Here's a macro that sums any number of inputs (bk03ch05/example06/):

```
macro_rules! sum {  
    ( $( $x:expr ),* ) => {  
        {  
            0 $(+ $x)*
```

```

    }
};
}

fn main() {
    let total5 = sum!(1, 2, 3, 4, 5);
    println!("The 5-item total is: {}", total5);

    let total3 = sum!(1, 2, 3);
    println!("The 3-item total is: {}", total3);

    let total1 = sum!(1);
    println!("The 1-item total is: {}", total1);

    let total0 = sum!();
    println!("The 0-item total is: {}", total0);
}

```

Output:

```

The 5-item total is: 15
The 3-item total is: 6
The 1-item total is: 1
The 0-item total is: 0

```

The `$( $x:expr ), *` pattern is the same as the one I explained in detail earlier in the “[Macro Syntax: Learning to Read the Hieroglyphics](#)” section. In this example, each captured `$x` is a number.

In the expansion part of the macro, the template that generates the code is `0 $(+ $x)*`. It starts with a literal `0`; the rest tells the compiler that, for every `$x` it captured, generate a plus sign(+), a space, and then the expression `$x`.

For example, a call such as `sum!(1, 2, 3, 4, 5)` gets expanded into the expression `0 + 1 + 2 + 3 + 4 + 5`, and the value of that expression is returned. Magic!

Example 2: Converting a list of key-value pairs to a vector

Next up is a slightly more advanced example than the one in the preceding section. I show you the code for a macro that takes a comma-separated list of key-value pairs — specifically pairs of the form `key:`

value, where *key* is a string literal and *value* is any value — and converts them to a vector of (*key*, *value*) tuples. Here's the code (bk03ch05/example07/):

```
macro_rules! pairs {
  ( $( $k:literal : $v:expr ),* $(,)? ) => {
    {
      let mut tmp = Vec::new();
      $(
        tmp.push(($k, $v));
      )*
      tmp
    }
  };
}

fn main() {
  let settings = pairs!(
    "port": 3000,
    "timeout": 5000,
    "retries": 3,
  );

  println!("{:?}", settings);
}
```

Output:

```
[("port", 3000), ("timeout", 5000), ("retries", 3)]
```

The `pairs` macro's matcher has two patterns:

- » The first pattern — `$( $k:literal : $v:expr ),*` — matches two inputs, the first (`$k`) is a literal and the second (`$v`) is an expression, separated by a colon (`:`).
- » The second pattern — `$(,)?` — matches either a single trailing comma or no comma after the final input pair.

The expansion code creates a new, mutable vector named `tmp`, and then uses `$(...) *` to run through every captured `$k/$v` pair, pushing each as a tuple to the `tmp` vector. When all that's done, the `tmp` vector is returned.

Project: Building a Timer Macro

As a practical example of what a macro can do, in this project you learn how to build a simple timer macro that can wrap any block of code, execute it, and then print how long that block of code took to run. Being able to time blocks of code is endlessly useful when building programs with performance in mind. At a terminal window, navigate to your Rust projects folder and run the following command:

```
cargo new timer-macro
```

Load this project's `main.rs` file into your favorite text editor, delete the default `println!` statement, and get ready to code.

The workhorse here will be an internal struct called `Instant`, which you can use to measure how much time elapses between the start of the code and the end. Here's how it works:

```
use std::time::Instant;

// Start the timer
let start = Instant::now();

// Code to be timed goes here

// Stop the clock!
let duration = start.elapsed();
```

This is the heart of the `timer!` macro (bk03ch05/timer-macro/):

```
use std::time::Instant;

macro_rules! timer {
    ( $block:block ) => {
        {
            // Start the timer
            let start = Instant::now();

            // Execute the block of code
            $block

            // Stop the clock!
            let duration = start.elapsed();
```

```

        println!("Code block execution time: {:.2?}", duration);
    }
};
}

```

The matcher here is `$block:block`, a pattern that matches on a block of code, which gets stored in the `$block` variable. The expansion uses `Instant::now()` to start the timer, executes the block of code using the `$block` expression, and then stops the timer with `start.elapsed()` and prints the time it took the block to run.

This project uses the `timer!` macro to compare the performance of two functions that perform the same task, but with different algorithms:

```

// Calculate sum using a for loop
fn sum_iterative(n: u32) -> u64 {
    let mut total = 0;
    for i in 1..=n {
        total += i as u64;
    }
    total
}

// Calculate sum using the formula: n * (n + 1) / 2
fn sum_formula(n: u32) -> u64 {
    let n = n as u64;
    n * (n + 1) / 2
}

```

The `sum_iterative()` function calculates the sum of the numbers from 1 to the argument `n` using a `for` loop. The `sum_formula()` function performs the same calculation, but instead of a loop it uses the formula $n(n+1)/2$.

The `main()` function puts the `timer!` macro to use:

```

fn main() {
    timer!({
        let result = sum_iterative(1_000_000);
        println!("Sum using loop: {}", result);
    });

    timer!({
        let result2 = sum_formula(1_000_000);
        println!("\nSum using formula: {}", result2);
    });
}

```

```
});  
}
```

Output:

```
Sum using loop: 500000500000  
Code block execution time: 10.17ms  
  
Sum using formula: 500000500000  
Code block execution time: 2.13µs
```

For the `sum_iterative()` function, the execution time was 10.17ms, which is pretty fast. However, for the `sum_formula()` function, the execution time was a mere 2.137µs (microseconds), or approximately 5,000 times faster!

Chapter 6

Parallelism and Concurrency: Rust Multitasking

IN THIS CHAPTER

- » Understanding parallelism versus concurrency
 - » Writing multithreaded code
 - » Sharing data between threads
 - » Writing asynchronous code
 - » Exploring a famous unsolved math problem
-

At Mozilla, there is a sign on the wall with a dark horizontal line, below which is the text, “You must be this tall to write multithreaded code.” The line is roughly nine feet off the ground. We created Rust to allow us to lower that sign.

—JIM BLANDY

All the Rust programs you’ve coded so far can perform the digital equivalent of either walking or chewing gum. What those programs have *not* been able to do is perform both tasks simultaneously. That lack is about to change because in this chapter you explore the marvelous world of Rust multitasking, where you make your programs faster and more responsive by configuring them to do multiple tasks at once.

If multitasking sounds like a recipe for bugs that are hard to find and even more difficult to fix, you can relax. This is Rust, after all, which comes with a not-so-secret weapon: the compiler. Rust’s robust ownership and type systems are designed from the ground up to eliminate entire categories of multitasking bugs at compile time.

This chapter introduces you to the different tools Rust provides for running code in parallel and concurrently. By the time you're done, you'll understand why Rust's approach to multitasking is famously called *fearless concurrency*.

Parallelism versus Concurrency: What's the Difference?

Multitasking has its own jargon and terminology (much of which is offputtingly abstruse, so apologies in advance), so in this chapter you learn quite a few new terms. But two terms that I use throughout this chapter are crucial for you to get straight right off the bat:

- » **Parallelism:** Executing multiple tasks at the same time. Parallelism is about hardware because it requires multiple central processing units (CPUs) such as you find in the multi-core CPUs that run all modern computers. Each of those cores can run a different task at the same time.
- » **Concurrency:** Managing multiple tasks at the same time. Concurrency is about program structure because it deals with multiple tasks that can overlap in time, particularly when your program must wait for one or more of those tasks (such as retrieving data or making a network connection).

Imagine your program as the kitchen of a restaurant.

A concurrent kitchen would be one where you have a single chef making, say, a soup and a salad. The chef can start the soup simmering, chop vegetables for the salad, then stir the soup, and then toss the salad. The chef is managing two tasks at once, switching between them.

A parallel kitchen would be one that has both a chef and a sous chef. In this case, the chef might make the soup while the assistant makes the salad, each person executing their respective task at the same time.

Why two multitasking systems? Because each is designed to handle decidedly different scenarios:

- » Many program operations are said to be *CPU-bound* because they feature heavy computation workloads that require raw processing power. Think scientific simulations, image processing, massive dataset searches. Parallelism is designed for these types of workloads because you can speed things up significantly by splitting the tasks across multiple cores.
- » Other real-world program operations are said to be *I/O-bound* because they spend much of their time waiting for something. Think user input, database queries, network connections. Concurrency is designed for these types of operations because while one task is waiting, other tasks can go about their business.

This difference between CPU-bound and I/O-bound operations makes it straightforward to choose a multitasking route for your code. When your program has to wait for something (such as a file, a user, or a connection), go with concurrency; when your program needs processing power for a task (such as parsing, compressing, or heavy-duty math), go with parallelism.

Spawning Threads: Your First Parallel Adventure

The most straightforward way to run code in parallel is by creating — or *spawning* in multitasking lingo — a thread. A *thread* is separate path of execution that runs independently from your main program. The operating system's scheduler decides when each thread gets CPU time and which core it runs on. You don't control that directly, and a thread might run on different cores at different times. You can spawn a new thread to handle a long-running calculation, which leaves your main thread free to handle other tasks and keeps your application feeling responsive.

Creating and joining a thread

To spawn a thread in Rust, you need to import the `thread` module:

```
use std::thread;
```

With that done, you can spawn a new thread by running the `thread::spawn()` function, giving it a closure with the code you want the thread to execute. `thread::spawn()` returns a `JoinHandle`, which you can use to wait for the thread to finish its work.

Here's an example (bk03ch06/example01/ in this book's example files):

```
use std::thread;
use std::time::Duration;

fn main() {
    println!("Starting main thread.");

    // Spawn a new thread to do some "heavy" work
    let handle = thread::spawn(|| {
        println!("Starting spawned thread.");

        // Sleep for two seconds to simulate work
        thread::sleep(Duration::from_secs(2));

        // This statement doesn't run until two seconds later
        println!("Spawned thread finished.");

        // Return a value from the thread
        42
    });

    // The main thread does other stuff while the worker thread is busy
    println!("Main thread is doing other things.");

    // Sleep for 2500ms to simulate work
    thread::sleep(Duration::from_millis(2500));

    // This statement doesn't run until 2500ms later
    println!("Main thread is ready to wait.");

    // Wait for the spawned thread to finish
    if let Ok(result) = handle.join() {
        println!("Worker thread returned: {result}");
    }
}
```

Output:

```
Starting main thread.  
Main thread is doing other things.  
Starting spawned thread.  
Spawned thread finished.  
Main thread is ready to wait.  
Spawned thread returned: 42
```

Here's the workflow for this program:

1. The main thread starts and displays `Starting main thread.`
2. Just prior to doing some work, the main thread displays `Main thread is doing other things.`
3. The main thread uses `Duration::from_millis(2500)` to sleep for 2500ms.
4. While the main thread is working, `handle` is called and the new thread is spawned using `thread::spawn()`. Within the closure, the new thread displays `Starting spawned thread.`
5. Within the closure, the spawned thread uses `Duration::from_secs(2)` to sleep for 2 seconds (2000ms).
6. Still within the closure, the spawned thread completes its work, displays `Spawned thread finished`, and returns an arbitrary value of 42 to complete the closure.
7. The main thread completes its work and displays `Main thread is ready to wait.`
8. The main thread uses `join()` to get the `JoinHandle` result from the spawned thread.
9. The value returned by the spawned thread is in a `Result`, so `Ok(result)` gets the value and the final `println!` displays `Worker thread returned: 42.`

So, you use `thread::spawn()` with a closure to spawn a new thread and define its code, and then, in the main thread, you use `join()` to wait for

the spawned thread to finish.

In this example, the work done by the main thread is longer than the work done by the spawned thread, so the `join()` function runs immediately after the main thread wakes up. In the main thread, try changing the `sleep` value to, say, 500 or 1000, so that the main thread finishes *before* the spawned thread. Here's the output:

```
Starting main thread.  
Main thread is doing other things.  
Starting spawned thread.  
Main thread is ready to wait.  
Spawned thread finished.  
Spawned thread returned: 42
```

Note the different order. When you run the code, you'll also notice a significant lag after the `Main thread is ready to wait` message. That lag happens because `join()` pauses — or *blocks*, in the vernacular — the main thread execution to wait for the spawned thread to complete its task. If it didn't pause, the main thread could be done before the spawned thread had a chance to report back.

Moving data into threads

In Book 2, [Chapter 6](#), I mention that one of the great advantages of using a closure over a function is that a closure can capture local variables that would otherwise have to be passed to a function. Sharing local data with a spawned thread's closure via capture is a bit tricky, however. Check out this attempt to share a variable with a spawned thread (bk03ch06/example02/):

```
use std::thread;  
  
fn main() {  
    let message = String::from("Greetings, thread!");  
    let handle = thread::spawn(|| {  
        println!("Got message: {}", message);  
    });  
    handle.join().unwrap();  
}
```

Output:

```
error[E0373]: closure may outlive the current function, but it borrows
'message', which is owned by the current function
```

Because a thread might outlive the lifetime in which it was spawned, Rust gives the thread closure a `'static` lifetime, meaning the lifetime of the entire program. Since at compile time Rust can't be sure when the thread will end, it doesn't allow you to borrow a value that might not live as long as the thread's closure.

You could work around the problem, at least in this example, by declaring the variable as a string literal with a static lifetime:

```
static MESSAGE: &str = "Greetings, thread!";
```

However, the compiler offers a different solution:

```
help: to force the closure to take ownership of 'message'
(and any other referenced variables), use the 'move' keyword
|
6 |     let handle = thread::spawn(move || {
|                                   +++++
```

The compiler is suggesting that you use the `move` keyword to move ownership of any captured variable into the closure (bk03ch06/example03/):

```
use std::thread;

fn main() {
    let message = String::from("Greetings, thread!");

    let handle = thread::spawn(move || {
        println!("Got message: {}", message);
    });

    handle.join().unwrap();
}
```

Output:

```
Got message: Greetings, thread!
```

It works!

That's all good, but what if you still need to use the captured variable later in the main thread? Once you move the variable into the thread

closure, that variable is no longer available to the main thread. The solution in this case is to clone the variable, and then move the clone into the closure (bk03ch06/example04/):

```
use std::thread;

fn main() {
    let message = String::from("Greetings, thread!");
    let message_clone = message.clone();

    let handle = thread::spawn(move || {
        println!("Got message: {}", message_clone);
    });

    handle.join().unwrap();
    println!("Still available: {}", message);
}
```

And if you need to use the same data in multiple threads, you have no choice but to clone it (bk03ch06/example05/):

```
use std::thread;

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let mut handles = vec![];

    for i in 0..3 {
        let numbers_clone = numbers.clone();
        let handle = thread::spawn(move || {
            let sum: i32 = numbers_clone
                .iter()
                .map(|n| n * i)
                .sum();
            println!("Thread {i} calculated sum: {sum}");
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

Sample output:

```
Thread 3 calculated sum: 45
Thread 1 calculated sum: 15
Thread 2 calculated sum: 30
```

Interestingly, the threads didn't finish in the same order they were spawned! Even more interestingly, that order will often change each time you run the program! If you're curious about why this happens, read the sidebar "[Why threads don't always finish in the same order they were spawned.](#)"



TECHNICAL
STUFF

WHY THREADS DON'T ALWAYS FINISH IN THE SAME ORDER THEY WERE SPAWNED

The fact that threads don't always finish in the same order they were spawned comes down to a bunch of factors:

- **Thread scheduling:** The operating system's thread scheduler determines when each thread gets CPU time. Thread scheduling is non-deterministic and depends on factors such as the current system load, what other processes are running, CPU availability, and scheduler algorithm priorities.
- **Concurrency versus parallelism:** All three threads are running concurrently (and potentially in parallel on multi-core systems), but even though they're spawned in order (0, 1, 2), there's no guarantee they'll start executing immediately or in that same order. The operating system can switch between threads at any point.
- **No synchronization:** The code doesn't use any of Rust's synchronization mechanisms (such as mutexes or channels, discussed later), so each thread operates independently and prints as soon as it finishes its calculation. There's no coordination between the threads to ensure ordered output.
- **Minimal workload:** The calculation inside the closure is very lightweight (it's just arithmetic), so the execution time differences between threads are minimal, meaning that small variations in scheduling can easily change the completion order.
- **Race condition in output:** Multiple threads may be trying to print to `stdout` simultaneously. While `println!` is thread-safe in Rust, the printing order depends on which thread reaches the print statement first. (*Thread-safe* refers

to code that can be safely executed by multiple threads concurrently without causing data races, corruption, or undefined behavior.)

This non-deterministic behavior sounds like a bug, but it's actually a feature of concurrent programming because it enables the system to optimize resource usage and maximize parallelism.

Passing Messages through Channels

When your code spawns multiple threads, much of the time those threads go about their business independent of each other, and the main thread hears back from a spawned thread only when the latter ends and returns a value. However, in many situations it's extremely useful for threads to communicate. For example, rather than just returning a final result to the main thread, it might be useful for a spawned thread to send periodic reports about its progress or its current state.

One of the safest and most common ways for threads to communicate in Rust is through *message passing*, where threads send data to each other through channels.

A *channel* is like a one-way walkie-talkie because it offers a *transmitter* for one thread to send messages and a *receiver* for another thread to pick them up. A key rule in Rust is that a channel enforces ownership: once you send a value down the channel, you can't use it anymore.

Passing messages between a sender and a receiver

As an example, let's mock up a temperature sensor that uses a spawned thread to read the current temperature and uses a channel to report that reading back to the main thread (bk03ch06/example06):

```
use rand::Rng;
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    // Create a new channel
```

```

let (sender, receiver) = mpsc::channel();

// Spawn a "sensor" thread that generates data
thread::spawn(move || {
    // Use a for loop to mimic regular sensor updates
    for _ in 1..=5 {
        // Generate a random floating-point value between 20.0 and 21.0
        let temp: f64 = 20.0 + rand::rng().random:<f64>();

        // Take the "reading"
        let reading = format!("{:.2}°C", temp);

        println!("Sensor: sending '{}'", reading);

        // Send it to the receiver
        sender.send(reading).unwrap();

        // Sleep for a second
        thread::sleep(Duration::from_millis(1000));
    }
});

// The main thread acts as the receiver
println!("Monitor: waiting for readings...");

// A for loop over the receiver waits for messages
for received in receiver {
    println!("Monitor: received reading -> '{}'", received);
}

// Channel is closed at this point
println!("Monitor: channel closed, shutting down.");
}

```

Sample output:

```

Monitor: waiting for readings...
Sensor: sending '20.97°C'
Monitor: received reading -> '20.97°C'
Sensor: sending '20.17°C'
Monitor: received reading -> '20.17°C'
Sensor: sending '20.39°C'
Monitor: received reading -> '20.39°C'
Sensor: sending '20.48°C'
Monitor: received reading -> '20.48°C'
Sensor: sending '20.26°C'
Monitor: received reading -> '20.26°C'
Monitor: channel closed, shutting down.

```

The `mpsc::channel()` function returns a tuple: a sender and a receiver, where the sender transmits and the receiver listens. In the spawned thread, the `send()` method ships data through the channel. The main thread takes advantage of the fact that the `receiver` is an iterator, so it uses a `for` loop to iterate over `receiver` to wait for and retrieve any messages that come in.

That `for` loop is the standard Rust method for consuming messages sent through a channel, but it hides some of the details that make the receiving portion of the show understandable. Here's the explicit way to listen for incoming messages:

```
loop {
    match receiver.recv() {
        Ok(received) => println!("Monitor: received reading -> '{}'",
                                received),
        Err(_) => break, // Channel closed
    }
}
```

The key here is the `receiver.recv()` method, which waits for incoming messages on `receiver`. These come in wrapped in `Result`, which is processed by the `match` statement, where `Ok()` holds the message and `Err()` indicates that the channel has closed.

Multiple producers, single consumer

The `mpsc` in `mpsc::channel` stands for *multiple producers, single consumer*, which means multiple threads can send messages through the same channel to a receiver. Here's an example (bk03ch06/example07/):

```
use rand::Rng;
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (sender, receiver) = mpsc::channel();

    // Spawn a "temperature sensor" thread
    let sender_clone = sender.clone();
    thread::spawn(move || {
        for _ in 1..=3 {
```

```

        let temp: f64 = 20.0 + rand::rng().random::<f64>();
        let reading = format!("Current temperature: {:.2}°C", temp);
        println!("Sensor: sending '{}'", reading);
        sender_clone.send(reading).unwrap(); // Send the data
        thread::sleep(Duration::from_millis(500));
    }
});

// Spawn a "humidity sensor" thread
let sender_clone2 = sender.clone();
thread::spawn(move || {
    for _ in 1..=3 {
        let humidity = 65 + rand::rng().random_range(-5..=5);
        let reading = format!("Current humidity {:.1}%", humidity);
        println!("Sensor: sending '{}'", reading);
        sender_clone2.send(reading).unwrap(); // Send the data
        thread::sleep(Duration::from_millis(500));
    }
});

// Drop the original sender so the receiver will
// close when all spawned threads finish
drop(sender);

println!("Monitor: waiting for readings...");
for received in receiver {
    println!("Monitor: got reading -> '{}'", received);
}
println!("Monitor: channel closed, shutting down.");
}

```

Sample output:

```

Monitor: waiting for readings...
Sensor: sending 'Current humidity 67%'
Sensor: sending 'Current temperature: 20.90°C'
Monitor: got reading -> 'Current humidity 67%'
Monitor: got reading -> 'Current temperature: 20.90°C'
Sensor: sending 'Current humidity 64%'
Sensor: sending 'Current temperature: 20.79°C'
Monitor: got reading -> 'Current humidity 64%'
Monitor: got reading -> 'Current temperature: 20.79°C'
Sensor: sending 'Current humidity 68%'
Sensor: sending 'Current temperature: 20.71°C'
Monitor: got reading -> 'Current humidity 68%'
Monitor: got reading -> 'Current temperature: 20.71°C'
Monitor: channel closed, shutting down.

```


This code spawns two sensor threads, one for temperature and one for humidity. Each worker thread gets a clone of the sender and both send messages to the main thread receiver. Note the addition of `drop(sender)`, which runs when all the spawned threads are done so that the receiver knows when to shut down.

Sharing State Safely with Arc and Mutex

Message passing, as I describe in the preceding section, is a useful tool, but it's not always the right tool. You will often have program data that multiple threads need to read or update. This is where most languages hand you a loaded footgun. Rust hands you safety equipment, instead, by offering tools that enable your programs to share their state safely across multiple threads.

Sharing ownership with Arc

If you have some data that you want multiple threads to use, you could clone that data for each thread that needs it. However, all that cloning can get expensive, especially if the data is large. A more efficient alternative is to create a pointer to the original data and then clone just the pointer.

That alternative is exactly what you get with `Arc` (atomic reference counting), a Rust type that creates a pointer to some data. Technically, `Arc` is a *smart pointer*, which means that it not only stores a memory address (pointing to the data) but also provides extra features. In the case of `Arc`, those extras include the following:

- » **Shared ownership across threads:** Multiple `Arc` pointers can all own the same data at once, and the data isn't dropped until the last `Arc` is gone.
- » **Thread-safe reference counting:** The internal counter that tracks how many `Arc`s exist for a single memory address uses *atomic*

operations, meaning Rust performs the operation in one unbreakable step, so no two threads can mess up each other's update.

Here's an example that demonstrates `Arc` at work (bk03ch06/example08/):

```
use std::sync::Arc;
use std::thread;

fn main() {
    let original = vec![1, 2, 3, 4, 5];

    // Move the vector into a new Arc
    let data = Arc::new(original);
    println!(
        "Original Arc reference count: {}\n",
        Arc::strong_count(&data)
    );

    // Clone the Arc safely to share the vector across threads
    println!("Cloning Arc pointer...");
    let clone1 = Arc::clone(&data);
    println!("Reference count is now: {}\n", Arc::strong_count(&data));

    println!("Cloning Arc pointer...");
    let clone2 = Arc::clone(&data);
    println!("Reference count is now: {}\n", Arc::strong_count(&data));

    // Spawn thread #1
    let handle1 = thread::spawn(move || {
        let product: i32 = clone1.iter().product();
        println!("Thread #1 using data to calculate product: {product}");
    });

    // Close thread #1
    handle1.join().unwrap();
    println!("Thread #1 complete.");
    println!("Reference count is now: {}\n", Arc::strong_count(&data));

    // Spawn thread #2
    let handle2 = thread::spawn(move || {
        let sum: i32 = clone2.iter().sum();
        println!("Thread #2 using data to calculate sum: {sum}");
    });

    // Close thread #2
```

```

    handle2.join().unwrap();
    println!("Thread #2 complete.");
    println!("Reference count is now: {}\n", Arc::strong_count(&data));
}

```

Output:

```

Original Arc reference count: 1

Cloning Arc pointer...
Reference count is now: 2

Cloning Arc pointer...
Reference count is now: 3

Thread #1 using data to calculate product: 120
Thread #1 complete.
Reference count is now: 2

Thread #2 using data to calculate sum: 15
Thread #2 complete.
Reference count is now: 1

```

After importing `Arc` with `std::sync::Arc`, this program declares a vector and immediately moves that vector into a new `Arc` with `Arc::new()`. The program creates a couple of `Arc` clones with `Arc::clone()`. It's important to remember that they are clones of *pointers*, not the underlying heap data (that is, the vector). All through the program, the current reference count is displayed with `Arc::strong_count()`.

Modifying shared data safely with Mutex

`Arc`, as I describe in the preceding section, enables you to share ownership across threads, but none of those threads can modify the shared data. If your program's threads need to make changes to shared data, you need to bring in `Mutex` (mutual exclusion), which enables a thread to put a lock on the data. Once that lock is in effect, no other thread can access the data.

Here's an example (bk03ch06/example09/):

```

use std::sync::{Arc, Mutex};
use std::thread;

```

```

fn main() {
    // Create an Arc smart pointer to some heap data
    // and protect it with a Mutex
    let shared_log = Arc::new(Mutex::new(vec![]));

    // Create a vector to hold the spawned thread handles
    let mut handles = vec![];

    // Loop to spawn some threads
    for i in 0..3 {
        // Clone the Arc to give shared ownership to the upcoming thread
        let log_clone = Arc::clone(&shared_log);

        // Spawn a thread
        let handle = thread::spawn(move || {
            // Acquire the lock to get mutable access to the data
            let mut log = log_clone.lock().unwrap();
            log.push(format!("Event from thread {i}"));
            // Lock automatically released here when 'log' goes out of scope
        });
        handles.push(handle);
    }

    // Wait for all the threads to finish
    for handle in handles {
        handle.join().unwrap();
    }

    // Acquire the lock to print the final shared log
    println!("Final log:\n{:?}", shared_log.lock().unwrap());
}

```

Output:

```

Final log:
["Event from thread 0", "Event from thread 1", "Event from thread 2"]

```

After importing both `Arc` and `Mutex` with `std::sync::{Arc, Mutex};`, this program creates a new `Arc` for an empty vector and protects it with a new `Mutex`. Within each thread, the code uses `lock()` to give the `log` variable exclusive mutable access to the data and pushes `String` to the vector. The lock is released automatically when the `log` variable goes out of scope.

Borrowing Data with Scoped Threads

In the previous sections, I describe a few ways to share data among threads, including using the `move` keyword, cloning data, and using `Arc`. But let's be honest: Doesn't it all feel, well, a bit clunky? All you want to do is share some data, so what's the big deal?

The big deal is that, so far, every thread you've spawned could potentially outlive the main thread. That's why Rust insists you either move ownership into the thread or use `Arc` or `Mutex`. But what if you could promise Rust that all your threads would finish before the borrowed data goes away? That's exactly what scoped threads do.

The idea is that you first create a scope using the `thread` module's `scope()` function. Then within that scope, every thread you spawn is guaranteed to be joined (completed) before the scope ends. Such a thread is called a *scoped thread*. The guarantee enables a scoped thread's closure to safely borrow data directly from the parent thread's stack, removing the need for moving, cloning, or using `Arc` or `Mutex` in many situations.

Here's an example (bk03ch06/example10/):

```
use std::thread;

fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5, 6];

    println!("Before processing: {:?}", numbers);

    // Create a scope for our threads
    thread::scope(|s| {
        // You can borrow 'numbers' from the main thread here
        let mid = numbers.len() / 2;

        // Split 'numbers' into two parts
        let (part1, part2) = numbers.split_at_mut(mid);

        // Spawn a thread to process the first part
```

```

s.spawn(|| {
    println!("Worker #1 started.");
    for num in part1 {
        *num *= 2; // Double each number
    }
    println!("Worked #1 finished.")
});

// Spawn another thread to process the second part
s.spawn(|| {
    println!("Worker #2 started.");
    for num in part2 {
        *num += 10; // Add 10 to each number
    }
    println!("Worked #2 finished.")
});

// The scope automatically waits here for both threads to complete
});
println!("After processing: {:?}", numbers);
}

```

Output:

```

Before processing: [1, 2, 3, 4, 5, 6]
Worker #1 started.
Worked #1 finished.
Worker #2 started.
Worked #2 finished.
After processing: [2, 4, 6, 14, 15, 16]

```

The main thread declares the `numbers` variable, which is a mutable vector of six numbers. Then `thread::scope()` is called to create a scope for the threads. The `scope()` function provides a `Scope` object, which here is the closure's `s` variable. To give the spawned threads some work to do, the code calls `numbers.split_at_mut(mid)`, which creates two mutable references: one to the first half of `numbers` (`part1`) and one to the second half of `numbers` (`part2`).

The code then uses `s.spawn()` to spawn two scoped threads. The first thread doubles each number in `part1`, and the second thread adds 10 to each number in `part2`. The scope waits for both threads to finish, which is how the scope can guarantee to the compiler that neither thread will outlive the main thread.

Iterating in Parallel with Rayon

Sometimes you just want to make your loops faster without thinking about threads or channels or ownership or any of that complexity. Enter Rayon, the “make it parallel” magic wand.

First, run `cargo add rayon` in your project directory to add Rayon as a dependency. With Rayon added, parallelizing iterations is almost embarrassingly easy (bk03ch06/example11/):

```
use rayon::prelude::*;

fn main() {
    let numbers: Vec<u64> = (1..=1_000_000).collect();

    // Sequential sum
    let sequential_sum: u64 = numbers.iter().sum();
    println!("Sequential sum is {}", sequential_sum);

    // Parallel sum
    let parallel_sum: u64 = numbers.par_iter().sum();
    println!("Parallel sum is {}", parallel_sum);
}
```

Output:

```
Sequential sum is 500000500000
Parallel sum is 500000500000
```

No, your eyes aren’t deceiving you. Just add `par_` to your iterator, and Rayon automatically distributes the work across your CPU cores. The beauty of Rayon is that it manages the thread pool for you. It creates a number of worker threads based on your CPU cores and efficiently distributes work among them. You get parallelism without the complexity of manual thread management. Win, win!

Async/Await: Concurrency without the Thread Overhead

When your program needs to do some CPU-based heavy lifting, threads are the way to go. But what about when your code is doing I/O-based work? The key characteristic of I/O-bound workloads is that they do a lot of waiting: They wait for a network request to be fulfilled, a file to be read from a disk, or for a user to enter some input. Wait, wait, and then wait some more. Spawning a separate thread only to have that thread spend most of its time waiting is an inefficient use of resources.

Instead of using multithreaded programming for I/O work, you can use *asynchronous programming*, which enables tasks to run independently from the main program flow. With async code (as the cool nerds refer to it), your program can continue doing other work while it waits for a slow operation (such as an I/O workload) to complete. Async programming enables a single thread to manage not just a single task but hundreds or even thousands of concurrent tasks.

In Rust, you make some code asynchronous by preceding it with the `async` keyword, which can appear before a function, a closure, or a code block:

```
async fn fetch_data() -> String {  
    // Simulate an I/O workload  
    String::from("On the internet, no one knows you're a dog.")  
}
```

An async function returns `Future`, a value that will be ready at some point.

By itself, a future doesn't do very much. To make a future useful, you tell your code to watch out for when that value finally becomes available by using the `await` keyword:

```
async fn process_data() {  
    let data = fetch_data().await;  
    println!("Got: {}", data);  
}
```

When the runtime encounters `await`, it pauses the current task and allows other tasks to run. When the awaited future's result becomes ready, the runtime resumes the paused task. This process is incredibly

complex under the hood because the code must manage all of the following:

- » **Scheduling:** Deciding which async task to run next
- » **Polling futures:** Checking whether an async operation (that is, a future) has finished or needs to wait
- » **Waking tasks:** Resuming paused futures when their results are ready
- » **Integrating with I/O:** Coordinating non-blocking file, network, or timer events with the operating system

Because of this complexity, the management of async code isn't baked into the Rust standard library. Instead, you need to use an external crate, such as Tokio, which is known in Rust World as an *async runtime*. In your project directory, open your project's `Cargo.toml` and add the following:

```
[dependencies]
tokio = { version = "1.48.0", features = ["full"] }
```

Here's an example (bk03ch06/example12/):

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

async fn check_website(url: &str) -> Result<bool, String> {
    // Simulating a web request
    println!("Checking {}...", url);
    sleep(Duration::from_millis(500)).await;
    Ok(true)
}

#[tokio::main]
async fn main() {
    let sites = vec![
        "https://rust-lang.org",
        "https://crates.io",
        "https://docs.rs",
    ];
    let site1_check = check_website(sites[0]);
    let site2_check = check_website(sites[1]);
    let site3_check = check_website(sites[2]);
```

```
// Wait for all checks to complete concurrently
let (r1, r2, r3) = join!(site1_check, site2_check, site3_check);

// Print results
for (site, result) in [(sites[0], r1), (sites[1], r2), (sites[2], r3)] {
    match result {
        Ok(true) => println!("{}", site),
        _ => println!("{}", site),
    }
}
}
```

Output:

```
Checking https://rust-lang.org...
Checking https://crates.io...
Checking https://docs.rs...
https://rust-lang.org is up!
https://crates.io is up!
https://docs.rs is up!
```

This code defines an `async` function `check_website()` that simulates a web request given a URL by waiting for a Tokio `sleep()` function for 500ms. The code enables the Tokio runtime with `#[tokio::main]`, and then uses Tokio's `join!` macro to run all three site checks concurrently, returning the function `Result` values in a tuple.

Project: Collatz Conjecture Explorer

Consider the following rules:

- » **Even rule:** If a positive integer n is even, divide it by 2.
- » **Odd rule:** If a positive integer n is odd, multiply it by 3 and then add 1.

Pretty simple, right? Ah, but there are surprising depths to be found here. For example, examine what happens when these two rules are applied to the first ten integers:

```
1
2 -> 1
3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
```

```

4 -> 2 -> 1
5 -> 16 -> 8 -> 4 -> 2 -> 1
6 -> 3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
7 -> 22 -> 11 -> 34 -> 17 -> 52 -> 26 -> 13 -> 40 -> 20 -> 10 -> 5 -> 16 -> 8
[CA]-> 4 -> 2 -> 1
8 -> 4 -> 2 -> 1
9 -> 28 -> 14 -> 7 -> 22 -> 11 -> 34 -> 17 -> 52 -> 26 -> 13 -> 40 -> 20 ->
10
[CA]-> 5 -> 16 -> 8 -> 4 -> 2 -> 1
10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

```

Did you see what happened there? That's right: every sequence eventually ended up at 1!

Welcome to the Collatz Conjecture, which states that when you start with any positive integer and apply the two preceding rules, the resulting chain of numbers always ends up at 1. Note that it's the Collatz *Conjecture*, not the Collatz *Theorem*, because no one — not even some of the brightest mathematical minds of the past nearly 100 years (the sequence was discovered by Lothar Collatz in 1937) have been able to prove it. It has been shown to be true for every positive integer up to 2.36×10^{21} , but so far there has been no proof, which makes the Collatz Conjecture one of the most famous unsolved problems in mathematics.

One of the most interesting aspects of these Collatz sequences is how long they can run before eventually settling on 1. For example, here's the full sequence using 27 as the starting number:

```

27 -> 82 -> 41 -> 124 -> 62 -> 31 -> 94 -> 47 -> 142 -> 71 -> 214 -> 107
-> 322 -> 161 -> 484 -> 242 -> 121 -> 364 -> 182 -> 91 -> 274 -> 137 -> 412
-> 206 -> 103 -> 310 -> 155 -> 466 -> 233 -> 700 -> 350 -> 175 -> 526 -> 263
-> 790 -> 395 -> 1186 -> 593 -> 1780 -> 890 -> 445 -> 1336 -> 668 -> 334
-> 167 -> 502 -> 251 -> 754 -> 377 -> 1132 -> 566 -> 283 -> 850 -> 425 ->
1276
-> 638 -> 319 -> 958 -> 479 -> 1438 -> 719 -> 2158 -> 1079 -> 3238 -> 1619
-> 4858 -> 2429 -> 7288 -> 3644 -> 1822 -> 911 -> 2734 -> 1367 -> 4102 ->
2051
-> 6154 -> 3077 -> 9232 -> 4616 -> 2308 -> 1154 -> 577 -> 1732 -> 866 -> 433
-> 1300 -> 650 -> 325 -> 976 -> 488 -> 244 -> 122 -> 61 -> 184 -> 92 -> 46
-> 23 -> 70 -> 35 -> 106 -> 53 -> 160 -> 80 -> 40 -> 20 -> 10 -> 5 -> 16
-> 8 -> 4 -> 2 -> 1

```

That's 111 steps!

So, an interesting question to ask is: Given some limit, what integer starting value produces the longest Collatz chain? That's the question behind the chapter's project. The aim here is to not only find the answer given various limits but also to show how much faster the computation is when you use parallelism. This might seem surprising since the Collatz rules are just simple arithmetic. But once you use starting values in the hundreds of thousands, a lot of number-crunching takes place, and all that crunching can be handled much faster using a multithreaded approach.

To get started, use a terminal to navigate to the main Rust projects directory, and then run the following commands:

```
cargo new collatz-explorer
cd collatz-explorer
cargo add rayon
```

Now open this project's `main.rs` in your text editor, delete the default code, and get ready to code.

Begin by adding the following import statements (bk03ch06/collatz-explorer/):

```
use rayon::prelude::*;
use std::time::Instant;
```

Here's a function that, given a positive integer `n`, applies the two rules and returns the number of steps in that integer's Collatz chain:

```
fn collatz_length(mut n: u64) -> u32 {
    let mut steps = 0;

    while n != 1 {
        if n % 2 == 0 {
            n = n / 2;
        } else {
            n = n * 3 + 1;
        }
        steps += 1;
    }
    steps
}
```

Next is the `find_longest_sequential()` function, which uses a sequential (that is, single-threaded) approach to finding the integer in a specified range (given by the `start` and `end` arguments) that produces the longest Collatz chain:

```
fn find_longest_sequential(start: u64, end: u64) -> (u64, u32) {
    let mut longest_n = start;
    let mut longest_length = 0;

    for n in start..=end {
        let length = collatz_length(n);
        if length > longest_length {
            longest_length = length;
            longest_n = n;
        }
    }
    (longest_n, longest_length)
}
```

This function uses a sequential `for` loop to run through the entire range, a process that positively *begs* to be parallelized. You do that by converting the `for` loop to an iteration chain, beginning with `into_par_iter()` to get Rayon to handle the threads for you:

```
fn find_longest_parallel(start: u64, end: u64) -> (u64, u32) {
    (start..=end)
        .into_par_iter()
        .map(|n| (n, collatz_length(n)))
        .max_by_key(|&(_, length)| length)
        .unwrap()
}
```

Now all that's left is to put these functions to the test inside the `main()` function:

```
// Test with increasingly large ranges
let ranges = vec![(1, 10_000), (10_000, 100_000), (100_000, 1_000_000)];

for (start, end) in ranges {
    println!("Range: {} to {}", start, end.to_string().as_str());
    println!("{}", "-".repeat(30));

    // Sequential
    let start_time = Instant::now();
    let (longest_seq, length_seq) = find_longest_sequential(start, end);
```

```

let time_seq = start_time.elapsed();

println!("Sequential:");
println!(
    "    The number {} produces the longest chain ({} steps)",
    longest_seq, length_seq
);
println!("    Time: {:.3}s", time_seq.as_secs_f64());

// Parallel
let start_time = Instant::now();
let (longest_par, length_par) = find_longest_parallel(start, end);
let time_par = start_time.elapsed();

println!("Parallel:");
println!(
    "    The number {} produces the longest chain ({} steps)",
    longest_par, length_par
);
println!("    Time: {:.3}s", time_par.as_secs_f64());

println!(
    "    Speedup: {:.2}x\n",
    time_seq.as_secs_f64() / time_par.as_secs_f64()
);
}

```

Sample output:

```

Range: 1 to 10000
-----
Sequential:
    The number 6171 produces the longest chain (261 steps)
    Time: 0.003s
Parallel:
    The number 6171 produces the longest chain (261 steps)
    Time: 0.001s
    Speedup: 2.79x

Range: 10000 to 100000
-----
Sequential:
    The number 77031 produces the longest chain (350 steps)
    Time: 0.038s
Parallel:
    The number 77031 produces the longest chain (350 steps)
    Time: 0.008s
    Speedup: 4.74x

```

```
Range: 100000 to 1000000
-----
Sequential:
  The number 837799 produces the longest chain (524 steps)
  Time: 0.445s
Parallel:
  The number 837799 produces the longest chain (524 steps)
  Time: 0.072s
  Speedup: 6.19x
```

As these results show, the higher the range, the greater the speedup produced by the parallel approach.

Chapter 7

Testing, Debugging, and Documenting

IN THIS CHAPTER

- » Putting your code to the test
 - » Fumigating your code
 - » Formatting your code like a grown-up
 - » Explaining your code
-

This is the first programming language I've learned that makes it so easy to make test cases! It's actually a pleasure to implement them.

—0XMB

The way you've coded all your Rust program so far has probably been characterized by a kind of get-it-done casualness: You type in the code as quickly as possible, make up some sample data, perhaps add a comment or two, and then run the code to see if it works. This workflow is fine when you're just getting started. But as your programs grow and become more ambitious, the inherent unruliness of this method will accumulate into full anarchy, where your code becomes opaque even to you and hard to make changes to without breaking it.

The solution to this problem is to build a safety net made out of four coding best practices: testing, debugging, formatting, and documenting. These practices are what separate a casual coder from a professional developer. They're the tools that let you write robust, reliable, and understandable code that you (and others) can maintain for years to

come. And the best part? Rust makes this process not just easy but genuinely enjoyable.

Testing Your Code

In Rust, the compiler is great at catching problems such as syntax errors, type mismatches, and ownership and borrowing violations. Most of the time, the compiler will also show you how to fix the problem (or at least steer you in the right direction).

The compiler is one of Rust's most powerful and helpful features, but it can't fix logic errors. If you type `>` instead of `>=`, use `1..10` instead of `1..=10`, or write `||` (OR) instead of `&&` (AND), the compiler can't help you. Even worse, as long as the compiler has nothing to complain about, it will dutifully build your code and you, feeling buoyant after a successful compile, might believe that all is well with your code.

To know when your code is zigging when it's supposed to be zagging, you need to put it through its paces. And by that I don't just mean doing a `cargo run` using the vector of made-up values that you wrote when you were starting to code the program. No, you need to be systematic, and in Rust being systematic about running your code means writing tests.

A *test* is a small, self-contained piece of code that has one job: to check whether another piece of code is behaving the way you expect it to. A test essentially does two things:

- » It calls a particular function or method using a specified input.
- » It verifies that the output of the function or method matches what you expect, given the known input.

A test is like a little robot that runs your code for you and checks whether the code does what you think it should. If the robot finds a mismatch between what *should* happen and what *actually* happens, it raises a red flag to tell you that you've got a logic bug in that function or method.

I know what you're thinking: *Groan! It takes me long enough just to write my program, now I have to write more code just to check my code?*

I hear you, but here's a capital-T programming Truth to tattoo onto your frontal cortex: Yes, tests take a bit of extra time to write, so they create more work in the short term, but by catching logic bugs that would otherwise take you great gobs of time to track down, tests *save* you lots of time (and lower your stress levels) in the long term.

And tests are a great development safety net because when you make changes to one part of your program, you can just run your tests again to find out if you accidentally broke some other part. If the changes pass, you can keep coding; if they fail, you know your new code has a problem that needs fixing. Just think of all the great sleeps you're going to have!

Writing your first tests

Let's start with something simple. Suppose you're writing a function that calculates the estimated reading time for a blog post based on word count. Assuming people read 200 words per minute, the following function looks like it will get the job done:

```
fn estimate_reading_time(word_count: u32) -> u32 {
    let words_per_minute = 200;

    // Round up to the nearest minute
    word_count.div_ceil(words_per_minute)
}
```

`dividend.div_ceil(divisor)` is a numeric type (`u32`, in this case) method that performs the division `dividend/divisor` and rounds the result towards positive infinity.

The `div_ceil()` method panics if `divisor` is 0, but in this case that can never happen because the function sets the `words_per_minute` divisor to 200 right off the bat. So, you can check this function off your coding to-do list now, right? Not so fast. Do you really know that it works correctly?

It probably does, but to know for sure you must put the function to the test (bk03ch07/example01/ in this book's example files):

```
fn estimate_reading_time(word_count: u32) -> u32 {
    let words_per_minute = 200;

    // Round up to the nearest minute
    word_count.div_ceil(words_per_minute)
}

#[test]
fn test_exact_minutes() {
    // 400 words at 200 wpm should be exactly 2 minutes
    assert_eq!(estimate_reading_time(400), 2);
}
```

There are three things to note here:

- » **#[test]**: This attribute tells Rust that the function that follows is a test, not regular code.
- » **test_exact_minutes()**: This function performs the test of the `estimate_reading_time()` function.
- » **assert_eq!**: This macro takes two values as input and checks whether they're equal. If they are, the test passes; if they're not, the test fails, the program panics, and Rust tells you what went wrong.

In this example, you're testing the case of an exact (that is, non-rounded) result where the words per minute (wpm) value divides evenly into the word count. It's clear that a 400-word post at 200 wpm should return a value of 2. So, when you write `assert_eq!`

`(estimate_reading_time(400), 2)`, you're saying (asserting) that the values `estimate_reading_time(400)` and `2` are equal. Let's see!

To run the test, you have two choices. The first choice is to open a terminal window, navigate to your project directory, and run the following command:

```
cargo test
```

The second choice is to open your code file in VS Code. Assuming you have the `rust-analyzer` extension installed (I show you how in Book 1, [Chapter 2](#)), you see a Run Test button just above the `#[test]` attribute, as pointed out in [Figure 7-1](#). Click Run Test to do just that. (Note, as well, the Run Tests button at the top of the file. When you have multiple tests in a crate, which you get to shortly, you can click that button to run all the tests at once.)

Either way, Cargo runs the test and then displays the following output:

```
running 1 test
test test_exact_minutes ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

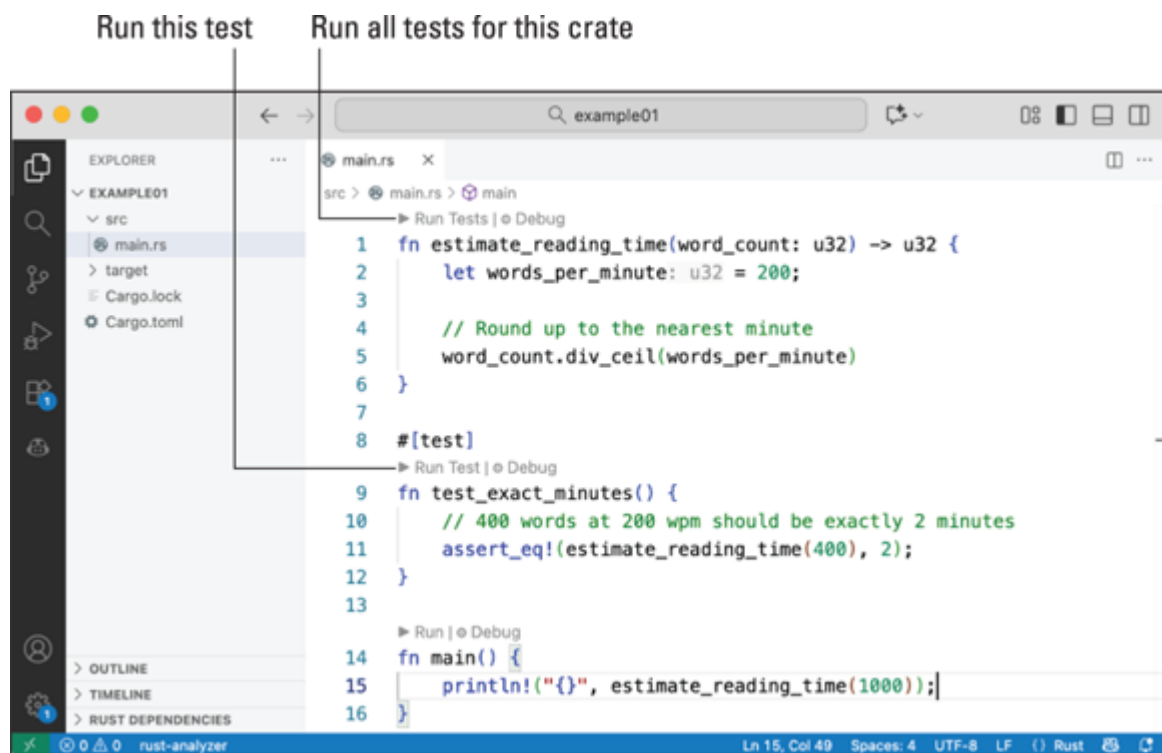


FIGURE 7-1: With `rust-analyzer` installed, you can click Run Test in VS Code to run your test.

Success! That's nice, but so far you've checked only one scenario: when the wpm value divides the word count evenly. To be really sure that your function works, you need to try other scenarios, such as what happens

when the code rounds up. You should also try edge cases, such as when the word count is 1 word and 0 words (bk03ch07/example02/):

```
fn estimate_reading_time(word_count: u32) -> u32 {
    let words_per_minute = 200;

    // Round up to the nearest minute
    word_count.div_ceil(words_per_minute)
}

#[test]
fn test_exact_minutes() {
    // 400 words at 200 wpm should be exactly 2 minutes
    assert_eq!(estimate_reading_time(400), 2);
}

#[test]
fn test_rounds_up() {
    // 450 words should round up to 3 minutes
    assert_eq!(estimate_reading_time(450), 3);
}

#[test]
fn test_very_short_text() {
    // Even 1 word should show as 1 minute
    assert_eq!(estimate_reading_time(1), 1);
}

#[test]
fn test_zero_words() {
    // Zero words should take no time to read!
    assert_eq!(estimate_reading_time(0), 0);
}
```

Run `cargo test` (or click Run Tests in VS Code). You should see these results:

```
running 4 tests
test test_very_short_text ... ok
test test_rounds_up ... ok
test test_zero_words ... ok
test test_exact_minutes ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

All is well!



REMEMBER When you test one small, isolated unit of your code, such as a single function as in the two examples so far, you're doing *unit testing*, which is the foundation for any good testing strategy.



TIP If you find that one test is quite slow or is interfering with your testing in some other way, you can tell Cargo to ignore that test by adding the `#[ignore reason = "Reason"]` attribute, where you replace *Reason* with a short reason why this test is being ignored. (The reason is optional, so you can skip it and just use `#[ignore]`, instead.) Here's an example:

```
#[test]
#[ignore reason = "Too slow!"]
fn very_slow_test() {
    // Too slow code lives here
}
```

Meeting the rest of the assertion family

The heart of any test is its assertion macro, which lets you know whether the test passes or fails. There are three main assertion macros:

- » **`assert_eq!(expr1, expr2)`**: Asserts that the expressions *expr1* and *expr2* are equal
- » **`assert_ne!(expr1, expr2)`**: Asserts that the expressions *expr1* and *expr2* are not equal
- » **`assert!(expr)`**: Asserts that the Boolean expression *expr* is true

All three of these macros have a second syntax that enables you to display a custom message when the test fails and the code panics:

```
assert_eq!(estimate_reading_time(0), 0, "Testing zero words.");
```

Testing functions that return Result

If you have a function that returns `Result`, you still test it using an assertion macro, but in this case you want to test both variants: the `Ok` result and the `Err` result. Here's an example (bk03ch07/example03/):

```
fn parse_tablespoons(input: &str) -> Result<f64, String> {
    let trimmed = input.trim();
    if trimmed.is_empty() {
        return Err("Input cannot be empty".to_string());
    }
    trimmed
        .parse::<f64>()
        .map_err(|_| format!("{}", {} ' is not a valid number", trimmed))
}

#[test]
fn test_valid_tablespoons() {
    assert_eq!(parse_tablespoons("2.5"), Ok(2.5));
    assert_eq!(parse_tablespoons(" 3 "), Ok(3.0));
}

#[test]
fn test_invalid_tablespoons() {
    assert!(parse_tablespoons("two").is_err());
    assert!(parse_tablespoons("").is_err());
}

#[test]
fn test_error_messages() {
    match parse_tablespoons("xyz") {
        Err(msg) => assert!(msg.contains("not a valid number")),
        Ok(_) => panic!("Expected an error but got Ok"),
    }
}
```

The function to be tested here is `parse_tablespoons()`, which takes a string slice input and returns `Result`: The number of tablespoons as an `f64` wrapped in the `Ok` if the input could be parsed, or an error message wrapped in the `Err` if not. The tests include two valid values (where the `Ok` is checked directly using the `assert_eq!` macro), two invalid values (checked using the `assert!` macro on the Boolean result of the `is_err()` method), plus a test that checks the error handling.

Testing panics

Sometimes you want to verify that your code panics in certain situations (such as when given invalid input that it can't handle gracefully). Rust has you covered (bk03ch07/example04/):

```
fn divide_cookies(cookies: u32, people: u32) -> u32 {
    if people == 0 {
        panic!("Can't divide cookies among zero people!");
    }
    cookies / people
}

#[test]
#[should_panic]
fn test_panic_with_zero_people() {
    divide_cookies(10, 0); // This should panic
}

#[test]
#[should_panic(expected = "zero people")]
fn test_panic_message() {
    divide_cookies(10, 0); // Panic message must contain "zero people"
}
```

The `#[should_panic]` attribute lets Rust know that the test passes if the code panics. As shown in the second test, you can even check that the panic message contains specific text.

Creating a module for your tests

So far, all the tests I've shown you have been mixed in with the regular program code. There's nothing wrong with that for small programs, but as your programs — and therefore your tests — grow, that test and code fraternization can lead to problems:

- » The tests get compiled along with the rest of your code. They get run only when you invoke `cargo test`, but they still live inside your program's binary file, taking up space.
- » Unless you're careful about organizing your tests, they can make your program code harder to read and navigate.

For all but the most trivial programs, it's far better to provide a home for all the unit tests associated with a particular file. That home is a special module, usually named `tests`, that resides just after the `#[cfg(test)]` attribute:

```
#[cfg(test)]
mod tests {
    use super::*;

    // All your tests go here
}
```

where

- » **`#[cfg(test)]`**: Tells Rust to compile the following module only when running tests, not when building the regular program. This approach keeps your tests out of your compiled binary.
- » **`mod tests {...}`**: Provides all your tests for the current file. As before, you precede each test with the `#[test]` attribute.
- » **`use super::*`**: Imports everything from the parent module (that is, everything in the current Rust file), which gives your tests access to the code they need to test.

Here's an example (bk03ch07/example05/):

```
pub struct PasswordValidator {
    min_length: usize,
    require_uppercase: bool,
    require_number: bool,
}

impl PasswordValidator {
    pub fn new(min_length: usize) -> Self {
        PasswordValidator {
            min_length,
            require_uppercase: true,
            require_number: true,
        }
    }

    pub fn validate(&self, password: &str) -> Result<(), String> {
        if password.len() < self.min_length {
            return Err(format!("Password must be at least {} characters",
```

```

        self.min_length));
    }
    if self.require_uppercase && !password.chars().any(|c|
c.is_uppercase()) {
        return Err("Password must contain an uppercase
letter".to_string());
    }
    if self.require_number && !password.chars().any(|c| c.is_numeric()) {
        return Err("Password must contain a number".to_string());
    }
    Ok(())
}
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_valid_password() {
        let validator = PasswordValidator::new(8);
        assert!(validator.validate("Secret123").is_ok());
    }
    #[test]
    fn test_too_short() {
        let validator = PasswordValidator::new(8);
        let result = validator.validate("Hi1");
        assert!(result.is_err());
        assert!(result.unwrap_err().contains("at least 8"));
    }
    #[test]
    fn test_missing_uppercase() {
        let validator = PasswordValidator::new(8);
        let result = validator.validate("secret123");
        assert!(result.is_err());
        assert!(result.unwrap_err().contains("uppercase"));
    }
    #[test]
    fn test_missing_number() {
        let validator = PasswordValidator::new(8);
        let result = validator.validate("SecretPassword");
        assert!(result.is_err());
        assert!(result.unwrap_err().contains("number"));
    }
}

```

This code implements a simple password validator that checks whether the password is a minimum length, contains at least one uppercase letter,

and contains at least one number. All the tests live inside the `tests` module.

Creating a library automatically includes a test module

Here's something cool: You already know that when you run `cargo new` to fire up a new Rust project, the `main.rs` file includes the `main.rs()` function with a `println!` macro. But you can also use `cargo new` to create a fresh library:

```
cargo new my-library --lib
```

In this case, Cargo adds the `lib.rs` file to the `src` directory. Opening `lib.rs` reveals the following boilerplate code:

```
pub fn add(left: u64, right: u64) -> u64 {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

Why, look at that: It's a tiny API complete with its own test module!

Integration tests: Testing how things work together

Unit tests are great for checking the small details, but they don't tell you if all the pieces of your program work together correctly. For that, you need *integration tests*, which check that multiple parts of your program (or crate) work correctly together through the public API. That is, integration tests approach your code from the outside, just like an

external user would (as opposed to unit tests, which test your code from the inside).

To create integration tests, you first need to create a directory called `tests` at the top level of your project (that is, on the same directory level as the `src` directory). The basic structure looks something like this:

```
my-crate/  
├─ Cargo.toml  
├─ src/  
│   └─ lib.rs  
└─ tests/  
    └─ integration.rs
```

Any `.rs` file you put inside the `tests` directory will be compiled as a separate crate that depends on your main library.

To show you how to set this up, I'll first make a minimal crate that computes a simple loyalty score (bk03ch07/example06/):

```
// src/lib.rs  
  
pub fn loyalty_score(purchases: u32, returns: u32) -> u32 {  
    let raw = purchases.saturating_sub(returns);  
    // Bonus if the customer made at least one purchase and no returns  
    if purchases > 0 && returns == 0 {  
        raw + 1  
    } else {  
        raw  
    }  
}
```

This public function returns a simple loyalty score based on the number of purchases minus the number of returns. This is your first look at the `saturating_sub()` method, so let me take you through what it does. In this example, `purchases` and `returns` are both `u32` (unsigned 32-bit integers), which means they can't be negative. If a customer has more returns than purchases (for example, `purchases = 5`, `returns = 6`), a standard subtraction (`purchases - returns`) would result in a negative number. For unsigned integers, this result would cause an underflow, and the program would panic in debug builds or wrap to a very large number in release builds. `purchases.saturating_sub(returns)` prevents both

of these unattractive options by absorbing the underflow and returning 0 if the result of the subtraction is less than 0. (And, just in case you're wondering, yes there's a `saturating_add()` method that absorbs overflow.)

Now you create the integration tests in a new file in the `tests` directory:

```
// tests/loyalty.rs

use example06::loyalty_score;

#[test]
fn loyal_without_returns_gets_bonus() {
    assert_eq!(loyalty_score(3, 0), 4);
}

#[test]
fn returns_reduce_score_but_not_below_zero() {
    assert_eq!(loyalty_score(1, 2), 0);
}
```

Run `cargo test` (or click Run Tests if you have `loyalty.rs` open in VS Code) and Cargo automatically runs whatever tests you have stashed in the `tests` directory.

Debugging Your Code

Since *debugging* is the process of removing errors from your code, let me coin the term *inbugging* for the process of introducing bugs into your code. However, since research has shown that it's nearly impossible to write bug-free programs, the terms *coding* and *inbugging* are essentially synonymous. When you write code, you create bugs, and that truism applies not just to the greenest of rookie coders but also to the grizzliest of veteran programmers.

Whether the Rust compiler is complaining about something in your code, or your unit and integration tests have turned up an anomaly, you need to put on your virtual deerstalker hat and investigate the problem like a true detective: Gather clues, form theories, and then test those theories until you crack the case.

The dbg! macro: The Watson to your Holmes

When gathering data on a debugging case, some of the most useful clues come from the current values of variables and expressions at particular points in the program. This debugging technique is so useful and popular that Rust offers a tool for it in the form of the `dbg!` macro:

```
dbg!(expression)
```

The idea is that you wrap the `dbg!` macro around some *expression* already in your code, such as a variable name, an arithmetic or logical expression, or a method or function call. Rust executes *expression* normally, but then `dbg!` goes to work and prints the following to the console:

- » The path and name of the current file, followed by the line number of the `dbg!` call
- » The expression in the `dbg!` parentheses
- » The value of that expression



REMEMBER You can specify two or more expressions in the `dbg!` macro. Just separate each expression with a comma.

Super-handly, right? Here's an example (bk03ch07/example07/):

```
fn calculate_discount(price: f64, customer_age: u32) -> f64 {
    let senior_age = 65;
    let senior_discount = 0.15;
    let regular_discount = 0.05;

    let discount_rate = if dbg!(customer_age) >= senior_age {
        dbg!(senior_discount)
    } else {
        dbg!(regular_discount)
    };

    let discount_amount = dbg!(price * discount_rate);
    price - discount_amount
}
```

```
fn main() {  
    let final_price = calculate_discount(99.95, 70);  
    println!("Final price: ${:.2}", final_price);  
}
```

Output:

```
[src/main.rs:6:28] customer_age = 70  
[src/main.rs:7:9] senior_discount = 0.15  
[src/main.rs:12:27] price * discount_rate = 15.0  
Final price: $84.96
```



WARNING The `dbg!` macro is great, but it can be a *greedy* little thing. For types that implement the `Copy` trait (integers, Booleans, string slices, characters), Rust just copies the value for use in the `dbg!` macro, so no problem there. However, for types that live on the heap and don't implement `Copy`, such as `String`, `Vec`, or your custom structs, `dbg!` *takes ownership* of the expression, and then discards it once it's done! If you still need the value after the `dbg!` statement, borrow the value:

```
let s = String::from("Watson");           // Declare a String  
dbg!(&s);                                 // Borrow s for dbg!  
println!("Helpful assistant: {}", s);     // s is still valid here
```

Debugging in VS Code

Using the `dbg!` macro and `println!` macro (covered next) to peer inside your code execution are handy debugging techniques, but they require adding code to your code, and then removing it later when it's no longer needed. And, if you're not careful, handing a non-`Copy` value to the greedy `dbg!` macro can introduce *new* bugs to your program!

Rust pros use `dbg!` and `println!` for quick-and-dirty debugging, but when the going gets tough, the pros turn to an actual debugger to investigate their code. If you're a VS Code user, you have at your disposal one such actual debugger called CodeLLDB. (There are others, but CodeLLDB is the debugger extension used by most serious Rust

folks.) So, to begin, go to the Extensions tab in VS Code, search for CodeLLDB, and install it.

Now open the Rust file you want to debug. Note, just above `fn main()`, that a Debug button appears to the right of the Run button, as pointed out in [Figure 7-2](#). This button appears courtesy of the `rust-analyzer` extension, but it only does anything useful after you install a debugger, such as CodeLLDB. Before clicking Debug, you must tell the debugger where you want the program to pause so you can inspect what's happening. This action is known in the trade as *setting a breakpoint*, and you do it by clicking in the gutter (the space to the left of the line numbers) beside the line where you want to pause program execution. In [Figure 7-2](#), I set a breakpoint on line 6, which is indicated by the dot in the gutter (the dot is red in real life).

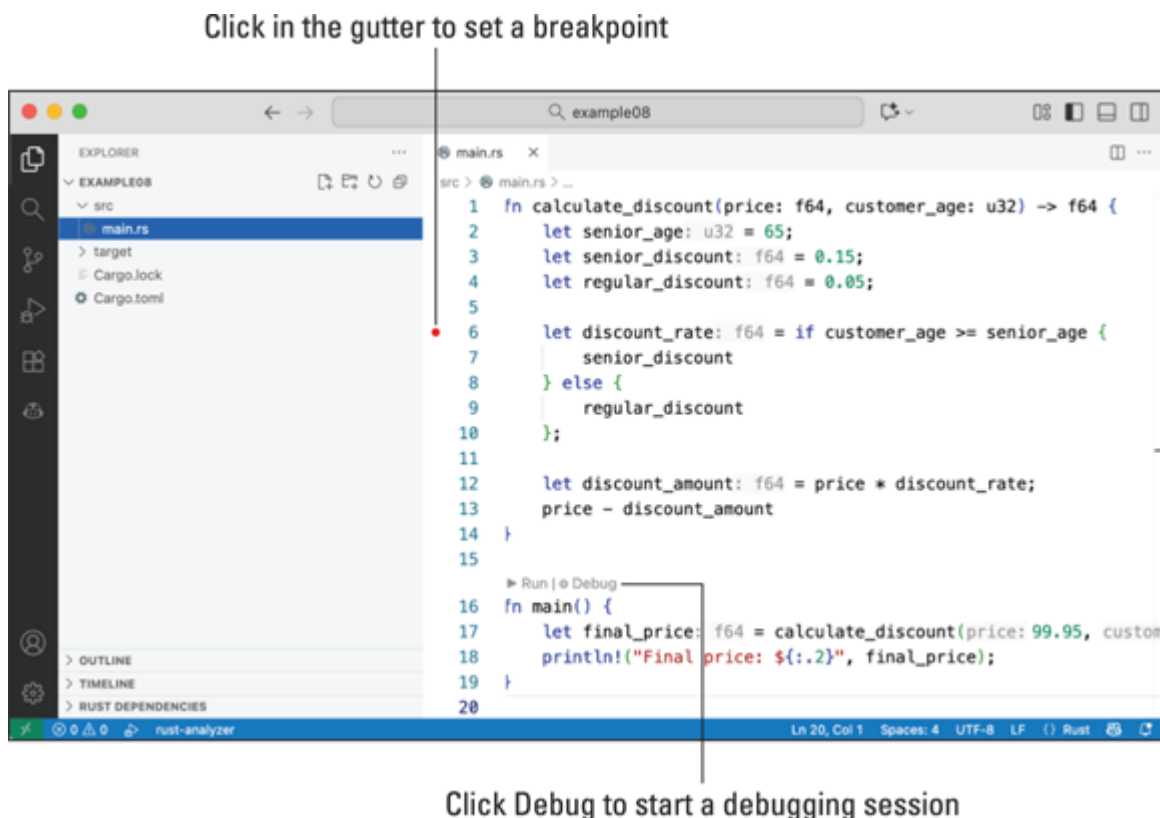


FIGURE 7-2: Set a breakpoint before starting your debugging session.

Click Debug and VS Code launches a debugging session. Your Rust code executes normally until it reaches your breakpoint, and then it

pauses execution — this is called *break mode* — and displays three important debugging features (all pointed out in [Figure 7-3](#)):

» **Run and Debug pane:** This pane is divided into several sections, of which the following four are the most important:

- *Variables*: Displays the current value of all your code’s local variables. This section also shows the values of your code’s static and global variables, if any.
- *Watch*: Enables you to monitor one or more variables or expressions. When you move your mouse pointer into this area, the add expression (+) icon appears. Click +, type the expression you want to monitor, then press Enter or Return.
- *Call Stack*: Displays the stack frames that Rust has called up to the breakpoint, with the most recent stack frame at the top.
- *Breakpoints*: Displays the breakpoints set on the code. Use the checkboxes to toggle particular breakpoints as needed.

The Run and Debug pane appears

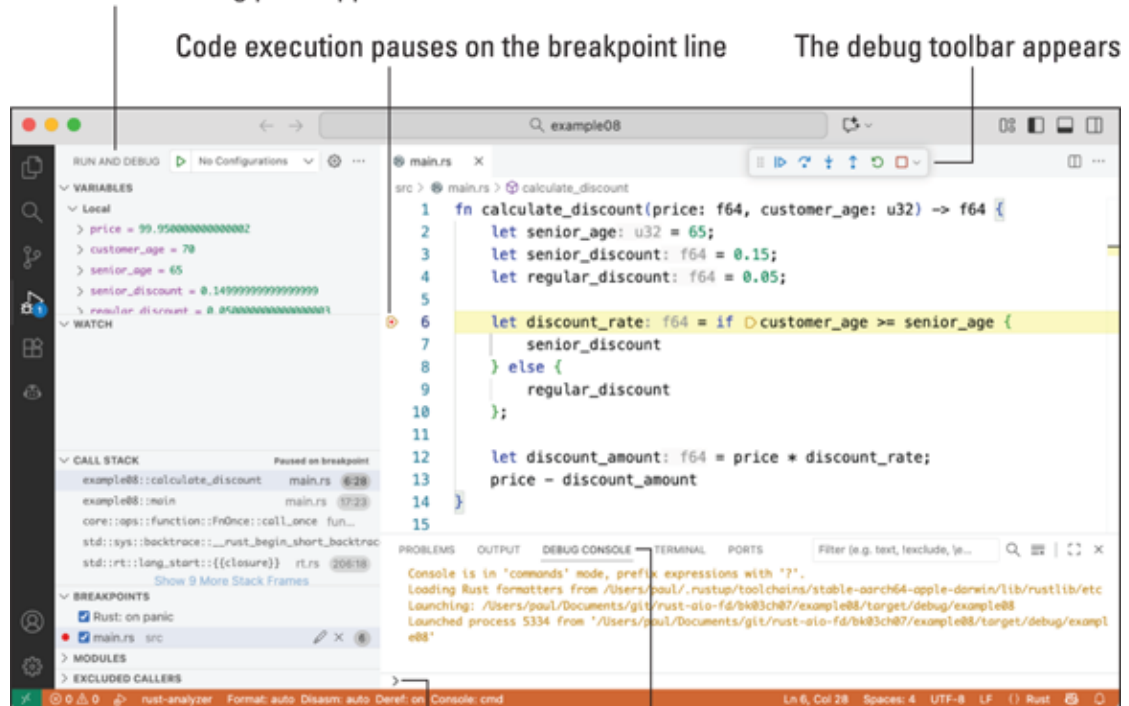


FIGURE 7-3: A Rust program in break mode.

» **Debug toolbar:** This toolbar enables you to use the following icons to control code execution:

-  *Continue:* Resumes code execution. The code runs until either the next breakpoint is reached or until the program exits. Shortcut: F5.
-  *Step Over:* Executes the following function or method, and then resumes break mode at the next statement immediately after the function or method call. Shortcut: F10.
-  *Step Into:* Calls the following function or method call, and then resumes break mode at the first statement of the function or method. Shortcut: F11.
-  *Step Out:* Executes the remainder of the current function or method, and then resumes break mode at the next statement immediately after the function or method call. Shortcut: Shift+F11.
-  *Restart:* Launches the program from the beginning. Shortcut: Ctrl+Shift+F5 (⌘+Shift+F5 on macOS).
-  *Stop:* Ends program execution. Shortcut: Shift+F5.

» **Debug console:** This area enables you to call up the value of any variable or expression. At the prompt, type ? followed by the variable name or expression, and then press Enter or Return to display the value in the Debug console.

Using println! debugging

The `dbg!` macro works great for showing the location and value of one or more variables or expressions. However, sometimes debugging requires a bit more context, so it would be nice to augment the output with some text explaining what you're doing or why you're doing it.

Alas, you can't do that with the `dbg!` macro, but you most certainly can do it with the `println!` macro. This solution is affectionately known as `println!` debugging. Yes, it feels, well, a bit simplistic, but every Rust programmer on the planet does it — heck every programmer, period, does it — so there's no shame in it. It's an easy-peasy to get a quick look at the state of your program.

Here's an example (bk03ch07/example09/):

```
fn process_inventory(items: Vec<(&str, i32)>) -> i32 {
    let mut total = 0;

    for (item, count) in items {
        println!("Processing {} (count: {})", item, count);

        if count < 0 {
            println!("    WARNING: Negative count for {}", item);
            continue;
        }

        total += count;
        println!("    Running total: {}", total)
    }
    total
}

fn main() {
    let inventory = vec![
        ("apples", 5),
        ("bananas", -2), // Oops!
        ("oranges", 3),
    ];

    let total = process_inventory(inventory);
    println!("Final total: {}", total);
}
```

Output:

```
Processing apples (count: 5)
    Running total: 5
Processing bananas (count: -2)
    WARNING: Negative count for bananas
Processing oranges (count: 3)
    Running total: 8
```

Note that I use indents to make the output a little easier to read.

Cleaning Your Code

It doesn't matter to me whether in the real world you're a messy person or a neat person. Maybe you thrive on disarray, or maybe a single paper out of place drives you crazy. Whatever works for you. It's all good.

But I'm far less sanguine when it comes to messy code. Why? It's simple:

Messy code breeds (and hides) bugs.

For example, you might not think that indenting your code consistently (or at all) could possibly make a difference, but if you can't tell where, say, a function or a `for` loop ends, you might include an unnecessary statement in, or exclude a crucial statement from, that function or loop. Bam! A bug is born.

Okay, I get it: You just want to bang out your code without having to worry about indentations or blank lines or trailing commas. That's fine. Go for it. But at some point, you must bring order to all that chaos. The great news is that doing this is nowhere near as much work as you probably think because Rust comes with two amazing tools that act like a cleaning service for your code.

rustfmt: Your automatic code cleaner-upper

Rust maintains an official style guide that you can peruse online at <https://doc.rust-lang.org/nightly/style-guide/>. But you don't need to memorize that style guide because Rust offers a command-line tool that automatically formats your code to match all those fussy Rust styles. That tool is called `rustfmt` and it comes with every Rust installation.

To get a feel for what `rustfmt` can do, here's some messy code (bk03ch07/example10/):

```
fn messy_function(x:i32,y:i32)->i32{
  let result=if x>y{
    x*2
  }else{
    y*2
  };
  return result;
}

fn main(){
  let a=5;let b=3;
  let answer=messy_function(a,b);
  println!("Answer: {}",answer);
}
```

Eww. But, not to worry; `rustfmt` can clean this up real nice. In a terminal window, navigate to the project directory, and then run the following command:

```
cargo fmt
```

Here's the result:

```
fn messy_function(x: i32, y: i32) -> i32 {
    let result = if x > y { x * 2 } else { y * 2 };
    return result;
}

fn main() {
    let a = 5;
    let b = 3;
    let answer = messy_function(a, b);
    println!("Answer: {}", answer);
}
```

Ah, better! `rustfmt` enforces many style guidelines, the following are the major ones:

- » Spacing around operators and punctuation
- » Indentation (four spaces)
- » Line breaks
- » Brace placement

Clippy: Your friendly code reviewer

As you've seen throughout this book, Rust has a preferred way of doing many standard programming tasks. These methods might sometimes seem idiosyncratic, but to the Rust community they're known as *idiomatic* Rust. It's just Rust being Rust. The community also overwhelmingly prefers efficient code and also avoids unnecessary complexity.

That's all fine, but how can you be sure you're writing Rust code that's idiomatic, efficient, and straightforward? You can always ask a more experienced Rust developer to review your code, but Rust gives you the next best thing. It's called Clippy, a command-line tool that reviews your program looking for mistakes, inefficiencies, unneeded complexities, and non-idiomatic code.

First, make sure Clippy is installed by firing up a terminal and entering the following command:

```
rustup component add clippy
```

With that done, `cd` into your project directory, and then run Clippy on your code:

```
cargo clippy
```

Here's an example program (bk03ch07/example11/):

```
fn process_scores(scores: &Vec<i32>) -> Vec<i32> {
    let mut results = Vec::new();

    for score in scores.iter() {
        if score > &80 {
            results.push(score.clone());
        }
    }

    return results;
}

fn main() {
    let scores = vec![75, 82, 69, 90, 78, 56, 72, 66, 85, 77];
    let high_scores = process_scores(&scores);
}
```

```
println!("{:?}", high_scores);
}
```

Here's (a lightly edited version of) what Clippy has to say about it:

```
warning: unneeded 'return' statement
--> src/main.rs:10:5
  |
10 |     return results;
  |     ^^^^^^^^^^^^^^^
  |
help: remove 'return'
  |
10 -     return results;
10 +     results
  |

warning: writing '&Vec' instead of '&[_]' involves a new object where a slice will do
--> src/main.rs:1:27
  |
1 | fn process_scores(scores: &Vec<i32>) -> Vec<i32> {
  |                                ^^^^^^^^^ help: change this to: '&[i32]'
  |

warning: using 'clone' on type 'i32' which implements the 'Copy' trait
--> src/main.rs:6:26
  |
6 |         results.push(score.clone());
  |                        ^^^^^^^^^ help: try dereferencing it:
  | '*score'
  |
```

Clippy provides three warnings here (which I've bolded), and in each case it describes the problem, specifies its location, and suggests a fix. Here's the clean version that compiles flawlessly after I implemented Clippy's suggestions:

```
fn process_scores(scores: &[i32]) -> Vec<i32> {
    let mut results = Vec::new();

    for score in scores.iter() {
        if score > &80 {
            results.push(*score);
        }
    }
    results
}
```

```
fn main() {  
    let scores = vec![75, 82, 69, 90, 78, 56, 72, 66, 85, 77];  
    let high_scores = process_scores(&scores);  
    println!("{:?}", high_scores);  
}
```

Documenting Your Code

The underlying theme to this chapter can be summarized with a single word: *quality*. Test your code to make sure it works as advertised; get rid of program bugs; and tidy your code to make it clean, efficient, and idiomatic. A coder who performs all these steps is a coder who cares about quality.

For the quality-conscious, there's one more step to take before you can say your program is ready for the world: documenting your code. Good documentation explains what your code does, how to use it, and why it was designed the way it was. Documentation is a gift to your future self. When you return to this code six months from now, you'll thank yourself for having taken the time to explain your code. And if someone else ever needs to work with your code, good documentation can transform confusion into comprehension.

Adding documentation comments

The main way that Rust programmers document their code is with *documentation comments*, which differ from regular Rust comments in a few ways:

- » **Module-level documentation:** A comment that describes the entirety of the file or module in which it appears. Module-level documentation (sometimes called *inner documentation*) lines begin with `//!` and usually appear at the top of the file or module:

```
//! Utilities for working with user accounts.  
//!  
//! This module defines functions and types for user authentication,  
//! password hashing, and session management.  
  
pub fn authenticate(...) { ... }
```

» **Item-level documentation:** A comment that describes the item that follows, such as a function, a struct, or a module. Each line in an item-level documentation begins with `///` and appears immediately before the item it describes:

```
/// Returns the number of whole words in 'text'.  
///  
/// Words are sequences of alphabetic characters.  
/// Numbers and symbols are ignored.  
pub fn count_words(text: &str) -> usize {...}
```

» **Markdown support:** Doc comments support Markdown, a simple markup language for formatting text. Markdown is useful later when you use Cargo to build those docs (refer to the section “[Building your program’s docs](#)”). All Markdown is supported, but you’ll use the following codes most often:

- `#` for a main heading (you can also use `##` for a subheading).
- `text` for bold text and `text` for italic text.
- `- item` for an item in a bulleted list. (For a numbered list, precede each item with `1.`; Markdown will automatically replace each `1.` with the correct numeric sequence.)
- `'code'` for inline code.
- `'''statements'''` for code blocks.

Here’s an example (bk03ch07/example12/):

```
//! Utilities for working with financial data.  
//!  
//! This module defines functions and types for loan payments,  
//! investments, future values, and other financial calculations.  
///  
/// Calculates the monthly payment for a loan.  
///  
/// # Arguments  
///  
/// - 'principal' - The loan amount in dollars  
/// - 'rate' - Annual interest rate as a decimal (0.05 for 5%)  
/// - 'years' - Length of the loan in years  
///
```

```

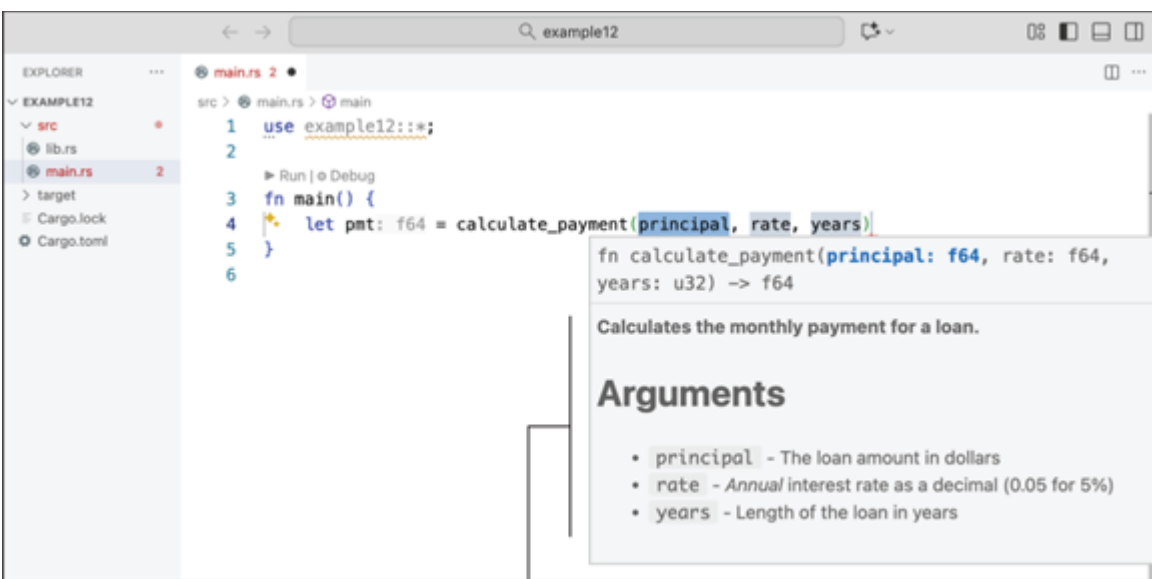
/// # Returns
///
/// The monthly payment amount, rounded to 2 decimal places
/// # Example
///
/// '''
/// let payment = example12::calculate_payment(200000.0, 0.045, 30);
/// assert_eq!(payment, 1013.37);
/// '''

pub fn calculate_payment(principal: f64, rate: f64, years: u32) -> f64 {
    let months = years * 12;
    let monthly_rate = rate / 12.0;
    let payment = principal *
        (monthly_rate * (1.0 + monthly_rate).powi(months as i32)) /
        ((1.0 + monthly_rate).powi(months as i32) - 1.0);
    (payment * 100.0).round() / 100.0
}

```

Special sections such as `# Arguments`, `# Returns`, and `# Examples` are conventions that make your documentation consistent and easy to read.

Interestingly, if you use VS Code with the `rust-analyzer` extension installed, that extension will pick up your documentation comments and display them as a code hint when you include a documented function in your code. [Figure 7-4](#) shows an example. Notice that `rust-analyzer` uses the Markdown codes when it renders the text in the hint.



This text comes from the documentation comments in `lib.rs`

FIGURE 7-4: In VS Code, the `rust-analyzer` extension uses your documentation comments as code hints.

Running documentation examples as tests

Including an example or two in your documentation comments tells others how to use your code, which is awesome. However, as a bonus, when you run `cargo test` on your code, Rust will run those documentation examples as tests! Running a documentation example as a test is called a *doctest*, and it's a great way to make sure that your documentation examples stay accurate.



TIP In VS Code, you see a button named Run Doctest above any documentation code that includes an example. Click that button to run the example as a test.

Building your program's docs

Documentation comments are a major aspect of writing quality software, but their usefulness — and their contribution to the quality of your software — doesn't end there. You can convert your comments into professional-looking documentation site by running the following command in your project directory:

```
cargo doc --open
```

This command builds the HTML documentation (the code resides in your project's `target/doc/` directory) and then opens it in your default web browser. [Figure 7-5](#) shows part of the result. With no effort on your part, you've now got a pro-grade website (modeled on the Rust documentation site) for your code, generated from just a few simple comments and a sprinkling of Markdown.

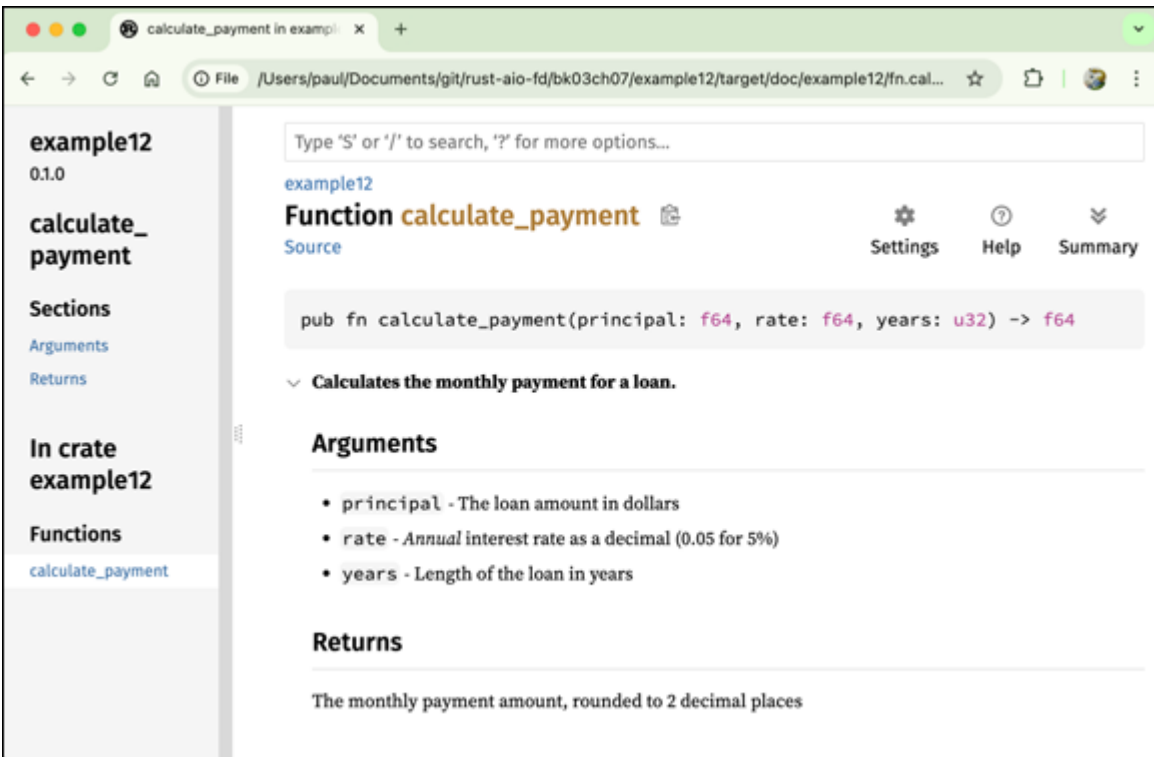


FIGURE 7-5: Cargo uses your doc comments to build a rather nice website for your code.

Book 4

Building Command-Line Tools

Contents at a Glance

Chapter 1: Coding Your First Tools

[Understanding What Makes a Good CLI Tool](#)

[Touring a Typical CLI](#)

[Parsing Command-Line Arguments](#)

[Reading from Standard Input](#)

[Basic File Operations for CLI Tools](#)

[Project: A Word Replacer Tool](#)

Chapter 2: File Processing and Configuration

[Learning File System Foundations](#)

[Reading and Writing Files Safely](#)

[Serializing and Deserializing Data](#)

[Project: Creating a File Organizer](#)

Chapter 3: Polishing Your Command-Line Tools

[Organizing Your Tools with Subcommands](#)

[Fancifying Output with Colors](#)

[Implementing Progress Bars](#)

[Project: Building a Multicommand File Organizer](#)

Chapter 1

Coding Your First Tools

IN THIS CHAPTER

- » Learning some CLI best practices
 - » Parsing arguments like a pro
 - » Messing with files
 - » Building a word replacer CLI tool
-

There should be a huge red warning on Rust's package, saying it is addictive. How am I supposed to use any other language now?

—MICHAL VANER

You've been using command-line tools all through this book, from the impenetrable `curl` command you use in Book 1, [Chapter 2](#) to install Rust, to the `cargo new`, `cargo build`, and `cargo run` commands that you use to create, compile, and launch your Rust programs. Along the way, you've likely also made good use of shell commands such as `cd`, `ls`, and `mkdir` to perform various command-line chores.

Now, here in [Book 4](#), you graduate from merely *using* command-line tools to *making* your own. Will it be glamorous? Well, no, not really. But I promise that it will be useful because every developer needs little custom utilities to rename files, process text, automate tedious tasks, and perform whatever chores need to get done. The command line is the perfect home for these utilities and Rust is the perfect language in which to code them. This chapter gets you started with your first CLI (command-line interface) tools, and then the next two add both spit and polish to your command-line creations.

Understanding What Makes a Good CLI Tool

Before diving into the code, it's a good idea to take a short pause to talk about what makes a good command-line tool. Even if you're relatively new to the command line, you've probably used quite a few command-line tools by now: `ls` to list files, `cp` to copy them, perhaps even `grep` to search through them. What do the good ones have in common?

The best CLI tools follow what I call the Three Cs of Command-Line Comfort:

- » **Clarity:** The tool does one thing well, and it's clear what that thing is.
- » **Composability:** The tool plays nicely with other tools through pipes and redirects.
- » **Courtesy:** The tool respects the user with helpful errors and sensible defaults.

For example, when you run `ls`, it lists files; that's it. There's your clarity. When you run `ls -R | grep .rs`, the output of `ls -R` (a recursive listing of files in subdirectories) flows into `grep`, which looks for files with names containing `.rs`. There's your composability. And when you run `ls --#%^&#@` (because, say, your cat jumped on the keyboard just as you were pressing Enter) you get helpful information instead of a chastising error message. There's your courtesy.

Your Rust CLI tools will follow these same principles. However, since you're building these tools in Rust, they'll also be fast, safe, and reliable because that's just how Rust rolls.

Touring a Typical CLI

A standard CLI tool is a kind of story, with a beginning, a middle, and an end. The elements of that story arc correspond to the three essential parts of a CLI:

- » **Arguments, flags, and options:** Characters you type after the program name (or after `cargo run`) when you run the tool. These could be a name or a file path, a switch such as `--verbose`, or an option such as `--limit 10`.
- » **Input/output streams:** How your program reads and writes text while it's running. There are three main streams:
 - *stdin* (*standard input*): Where your program reads text either typed by the user or piped from another program.
 - *stdout* (*standard output*): Where your program writes normal results, particularly messages to the user. Your code uses the `println!` macro for *stdout* messages.
 - *stderr* (*standard error*): Where your program writes warnings and error messages. Your code uses the `eprintln!` macro for *stderr* messages. Check out the “Proving Feedback” section for more.
- » **Exit code:** A number your program returns when it ends. By longstanding convention, an exit code of 0 means success (that is, your program terminated normally) and any non-zero exit code means failure (that is, your program terminated abnormally because of an error). For the latter, the exit code value reflects the error, and other tools pay attention to this number. For example, an exit code of 1 might refer to a general error, while an exit code of 2 might refer to a missing command-line argument.

Okay, enough theory. It's time to start building some CLI tools!

Parsing Command-Line Arguments

Your CLI's first point of contact with your users is what they type after the program name when they run the program at the command line (or after you type `cargo run` when you're developing the program). These words and symbols are the tool's *arguments*. When you type `cd ..` to navigate to the parent directory, `..` is an argument. When you launch a new Rust project with `cargo new my-project`, both `new` and `my-project` are arguments.

Getting the arguments

To learn which arguments the user passed to your program, you include the `std::env` module and then call the function `env::args()`, which returns an iterator that contains all the arguments.

Here's an example (bk04ch01/example01/ in this book's example files):

```
use std::env;

fn main() {
    let arguments: Vec<String> = env::args().collect();

    for (index, arg) in arguments.iter().enumerate() {
        println!("Argument {index}: {arg}")
    }
}
```

Let's say you run this program with the following command:

```
cargo run argy bargy
```

Here's the output:

```
Argument 0: target/debug/example01
Argument 1: argy
Argument 2: bargy
```

Interesting! The first argument isn't `argy`, as you might expect, but the path to the program file itself. This path is rarely useful, so when you're parsing arguments, the first one is typically ignored, and the real parsing begins with the second argument.

Parsing arguments manually

Getting the program's arguments is the easy part. The real fun begins when your code has to parse them! Parsing is often laborious and finicky work, so Rust developers almost always farm it out to a specialized crate, which I introduce you to in the next section. For now, it's a good idea to get a sense of what's involved in parsing CLI arguments, so here's a quick example that should help (bk04ch01/example02/):

```
use std::env;
use std::process;

fn print_usage() {
    eprintln!("Usage: example02 [--upper] [--times N] <words...>");
}

fn main() {
    // Get the arguments
    let mut args = env::args();

    // Ignore the program name (first argument)
    let _program = args.next();

    // Program configuration
    let mut upper = false;
    let mut times: u32 = 1;
    let mut words: Vec<String> = Vec::new();

    // Iterate through the rest of the arguments
    while let Some(arg) = args.next() {
        match arg.as_str() {
            "--upper" => upper = true,
            "--times" => {
                // Check for no number specified with --times
                let n = args.next().unwrap_or_else(|| {
                    eprintln!("example02: --times requires a number.");
                    process::exit(2);
                });
                // Parse the number of times
                times = n.parse().unwrap_or_else(|_| {
                    eprintln!("example02: invalid number for --times: {n}");
                    process::exit(2);
                });
            }
        }
        // Check for unknown option
        if s.starts_with('-') => {
            eprintln!("example02: unknown option: {s}");
            print_usage();
        }
    }
}
```

```

        process::exit(2);
    }
    // Push the word(s) to the vector
    word => words.push(word.to_string()),
}

// Check for no words specified
if words.is_empty() {
    eprintln!("example02: No word(s) specified");
    print_usage();
    process::exit(2);
}

// Compose and send the message to stdout
let message = words.join(" ");
let output = if upper {
    message.to_uppercase()
} else {
    message
};
for _ in 0..times {
    println!("{output}");
}
}

```

For example, running `cargo run -- --upper --times 3 Hello World` produces the following output:

```

HELLO WORLD
HELLO WORLD
HELLO WORLD

```



REMEMBER Note that the `--` after `cargo run` tells Rust that everything that follows comprises the program's arguments.

This code imports both `std::env` for access to the program's arguments and `std::process` so that it can use the `process::exit(code)` function to terminate the program with an exit *code*. The program stores the argument iterator in the `args` variable.

This code uses the `next()` method on the iterator. This is the first time I've shown you this handy iterator method, so here's how it works:

1. Create the iterator.

The iterator contains an internal pointer positioned just before the first item.

2. Call `next()`.

The pointer moves to the next item and returns an `Option` that wraps the item's value. For example, when you call `next()` for the first time, the pointer advances to the first item and returns `Some(value)`, where `value` is that item's value.

3. Repeat Step 2.

The pointer continues advancing and returning values until it reaches the end of the iterator, at which point it returns `None`.

This procedure explains why the `args` variable must be mutable because calling `next()` caused the iterator's internal state to change.

The bulk of the rest of the code iterates over `args` using a `while let Some(arg) = args.next()` loop, which uses `match` to look for the argument patterns, such as `--upper` and `--times`. The `match` arms look for errors, such as a missing or invalid numeric value for the `--times` argument. If an error crops up, the program displays a message to `stderr` using `eprintln!` and terminates using `process::exit(2)`.

The preceding code demonstrates the three main flavors of command-line arguments:

- » **Positional argument:** Depends on the order that it's given in the command. In the preceding code, `words` is a positional argument because it must come at the end of the command.
- » **Flag:** Turns a program feature on or off. Flags are preceded with `--` or `-`. In the preceding code, `--upper` is a flag.

- » **Option:** Turns a program feature on or off, like a flag but with a value. In the preceding code, `--times N` is an option, where `N` specifies the number of times to repeat the `words` argument.

There's nothing difficult about this code, but wow does it ever feel like a ton of effort for what seems like the relatively simple task of parsing command-line arguments. Fortunately, Rust offers a solution that's not only cleaner and simpler but is pure idiomatic goodness.

Parsing arguments idiomatically

Handling command-line arguments the idiomatic way in Rust requires two essential changes to the way I handled things in the preceding section:

- » Store command-line arguments, flags, and options in a struct.
- » Use a parsing library such as `clap` or `argh` to handle the parsing details.

To use `clap`, for example, run the following commands in your project directory:

```
cargo add clap --features derive
cargo update
```

I needed to run `cargo update` to solve some dependency conflicts.

With `clap` installed, the code from the preceding section can be refactored as follows (bk04ch01/example03/):

```
use clap::Parser;

/// A simple program to print words
#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Make the output uppercase
    #[arg(long)]
    upper: bool,

    /// The number of times to print
    #[arg(long, default_value_t = 1)]
```

```

    times: u32,

    /// The word(s) to print
    #[arg(required = true)]
    words: Vec<String>,
}

fn main() {
    let args = Args::parse();

    // Compose and send the message to stdout
    let message = args.words.join(" ");
    let output = if args.upper {
        message.to_uppercase()
    } else {
        message
    };
    for _ in 0..args.times {
        println!("{}", output);
    }
}

```

There are two main differences between this code and the code from the preceding section:

- » The arguments are stored in the `Args` struct instead of in program variables. The `#[command(...)]` attribute configures metadata for your CLI tool. The `author`, `version`, and `about` fields are pulled automatically from your `Cargo.toml` file, while `long_about = None` tells `clap` not to generate extended help text.
- » All the argument parsing is handled by a single line:

```
let args = Args::parse();
```

This code does exactly what the previous code does, but with half the number of lines because `clap` automatically handles all the parsing and error handling. As a bonus, `clap` also automatically builds a `--help` argument that, when you execute `cargo run -- --help`, displays the following text, which it builds from your struct comments:

```

A simple program to print words

Usage: example03 [OPTIONS] <WORDS>...

```

```
Arguments:
  <WORDS>...  The word(s) to print

Options:
  --upper           Make the output uppercase
  --times <TIMES>  The number of times to print [default: 1]
  -h, --help       Print help
  -V, --version     Print version
```

Nice!

Reading from Standard Input

Reading from standard input (`stdin`) generally means reading the text that comes in when the user types something at the command line and presses Enter or Return. Rust makes reading `stdin` a straightforward proposition by offering the `std::io` library's `stdin()` function, which creates a handle that gives your code access to the input stream.

Here's a simple program that reads from standard input (bk04ch01/example04/):

```
use std::io;

fn main() {
    let mut input = String::new();
    println!("Type some text and then press Enter or Return:");
    io::stdin().read_line(&mut input).unwrap();
    println!("{}", input.trim());
}
```

This code reads from `stdin` by using the `read_line()` method, which wraps the text in `Result`. Note the use of the `trim()` method on `input` `String`, which removes the trailing newline character that is added when the user presses Enter or Return.

Reading buffered input

For more complex `stdin` reads, your code needs to import the `BufRead` trait. Why? Because the basic `stdin` handle doesn't include the `lines()` method, which enables your code to iterate over multiple input lines. It's

defined in `BufRead`, so importing that trait brings that method into scope. Here's an example (bk04ch01/example05/):

```
use std::io::{self, BufRead};

fn main() {
    let stdin = io::stdin();
    for line in stdin.lock().lines() {
        println!("{}", line.unwrap());
    }
}
```

The `lock()` method acquires exclusive access to `stdin` and returns a buffered reader. The lock ensures that if multiple threads tried to read from `stdin` simultaneously, they wouldn't scramble each other's input. The buffered reader, in turn, enables efficient line-by-line reading through the chained `lines()` method. The `lines()` method returns `Result`, which wraps each line of text. When you run this code, type each line and press Enter or Return to see it echoed onscreen. When you're done, press Ctrl+D to shut down the program.

Reading redirected input

A bonus you get when using a buffered read is that you can also use `stdin` to read text redirected to your program from another file using the `<` input redirection operator. For example, you can use the program from the preceding section without modification to read the contents of a redirected file:

```
cargo run example05 < src/jabberwocky.txt
```

Running this command displays the contents of the `jabberwocky.txt` file to the console.

Basic File Operations for CLI Tools

File operations are so common with CLI tools that I devote all of Book 4, [Chapter 2](#) to handling files in various ways. For now, consider this section and the project (next) as a kind of coding *amuse bouche* for the filesystem.

Rust's standard library provides the `std::fs` (filesystem) and `std::path::Path` modules to handle file and directory tasks, so you need to import them:

```
use std::fs;
use std::path::Path;
```

This section and the next use the following methods:

- » **`fs::copy(from, to)`**: Copies the contents of the file specified by the path *from* to the file given by the path *to*, overwriting *to* if it exists. Returns `Result: Ok` if the copy was successful or `Err` if not.
- » **`path::exists(path)`**: Determines whether the specified *path* points to an existing directory or file. Returns `true` if *path* points to an existing entity; returns `false` if *path* is legal but doesn't point to an existing entity.
- » **`fs::read_to_string(path)`**: Reads the contents of the file specified by *path* into a string. Returns `Result: the string wrapped in Ok` if the operation was successful; `Err` if not.
- » **`fs::write(path, contents)`**: Writes the text specified by *contents* as the entire contents of the file specified by *path*, creating the file if it doesn't exist or overwriting its contents if it does. Returns `Result: Ok` if the write was successful or `Err` if not.

Note that for most of the methods, the path must be a `Path` type, which you can work with directly because you were smart enough to import the `std::path::Path` module. You can use `new()` to create the path to the resource you want to work with:

```
let filepath = Path::new("src/hello.txt");
```

Here's an example program that reads the contents of a file (bk04ch01/example06/):

```
use std::fs;
use std::path::Path;

fn main() {
```

```

// Get a Path for the file
let filepath = Path::new("src/hello.txt");

// Use display() to safely print the path
println!("Attempting to read from {}...", filepath.display());

// Use fs::read_to_string() to read the contents of the file
match fs::read_to_string(filepath) {
    // Process the Result
    Ok(contents) => {
        // Success! We have the file contents in a String
        println!("--- File Contents ---");
        println!("{}", contents);
    }
    Err(e) => {
        // Failure! Print a helpful error to stderr
        eprintln!("Oh no! We had a problem reading the file:");
        eprintln!("{}", e);
    }
}
}

```

Output (if the file exists):

```

Attempting to read from src/hello.txt...
--- File Contents ---
You say, "Yes", I say, "No"
You say, "Stop" and I say, "Go, go, go"
Oh no

You say, "Goodbye" and I say, "Hello, hello, hello"
I don't know why you say, "Goodbye", I say, "Hello, hello, hello"
I don't know why you say, "Goodbye", I say, "Hello"

```

Output (if the file doesn't exist):

```

Attempting to read from src/hello.txt...
Oh no! We had a problem reading the file:
No such file or directory (os error 2)

```

Project: A Word Replacer Tool

To close this chapter, I take you through a project that replaces one word with another in a specified file. The tool is called `word-replacer`, so navigate to your Rust projects folder and run the following command:

```
cargo new word-replacer
```

Open `Cargo.toml` and include `clap` as a dependency:

```
[dependencies]
clap = { version = "4.5.51", features = ["derive"] }
```

To help you follow the code, here's the help text that `clap` generates from the finished code:

```
A CLI tool to replace text in a file

Usage: word-replacer [OPTIONS] <FROM> <TO> <FILEPATH>

Arguments:
  <FROM>      The string to replace
  <TO>        The string to replace it with
  <FILEPATH>  The file to process

Options:
  -p, --preview  Preview changes without writing to file
  -h, --help     Print help
  -V, --version  Print version
```

Open `main.rs`, remove the default code, and then start adding the code. To begin, you need some imports (`bk04ch01/word-replacer/`):

```
use clap::Parser;
use std::fs;
use std::io;
use std::path::Path;
```

Now create the `Args` struct for `clap` to use to store the command-line arguments:

```
/// A CLI tool to replace text in a file
#[derive(Parser)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Preview changes without writing to file
    #[arg(short, long)]
    preview: bool,

    /// The string to replace
    from: String,

    /// The string to replace it with
```

```

    to: String,

    /// The file to process
    filepath: std::path::PathBuf,
}

```

Next you need a helper function that performs the text replacement:

```

fn replace_in_text(text: &str, from: &str, to: &str) -> String {
    text.replace(from, to)
}

```

Now here's where the main action occurs:

```

fn process_file(filepath: &Path,
                from: &str,
                to: &str,
                preview: bool) -> io::Result<usize> {
    // Read the file
    let contents = fs::read_to_string(filepath)?;

    // Count the number of occurrences to be replaced
    let occurrences = contents.matches(from).count();

    if occurrences > 0 {
        let replaced_contents = replace_in_text(&contents, from, to);

        // Is this just a preview?
        if preview {
            eprintln!(
                "This will replace {} occurrences in {:?}",
                occurrences, filepath
            );
        } else {
            // Create a backup copy of the file
            let backup_path = format!("{}.backup", filepath.display());
            fs::copy(filepath, &backup_path)?;

            // Write the replaced contents
            fs::write(filepath, replaced_contents)?;

            eprintln!(
                "Replaced {} occurrences in {:?} (backup: {})",
                occurrences, filepath, backup_path
            );
        }
    } else {
        eprintln!("No occurrences found in {:?}", filepath);
    }
}

```

```

    }

    Ok(occurrences)
}

```

The `process_file()` function performs the heavy file lifting here:

- » It uses `fs::read_to_string()` to get the contents of the file.
- » Before replacing the text, it uses `fs::copy()` to make a backup copy of the file.
- » It uses `fs::write()` to write the replaced text to the original file.

Finally, the `main()` function brings everything together:

```

fn main() -> io::Result<()> {
    let args = Args::parse();

    let filepath = &args.filepath;

    // Check if file exists and is actually a file
    if !filepath.exists() {
        eprintln!("Error: {:?} does not exist.", filepath);
        std::process::exit(1);
    }

    if !filepath.is_file() {
        eprintln!("Error: {:?} is not a file.", filepath);
        std::process::exit(1);
    }

    // Attempt the replacement
    match process_file(filepath, &args.from, &args.to, args.preview) {
        Ok(count) => {
            if count > 0 {
                println!("\nSummary: {} replacements in {:?}",
                    count, filepath);
            }
        }
        Err(error) => {
            eprintln!("Error processing {:?}: {}", filepath, error);
        }
    }

    Ok(())
}

```

Here's how to use the tool with a preview (no changes made to the file):

```
target/debug/word-replacer --preview "old" "new" "src/test1.txt"
```

Here's the preview output:

```
This will replace 7 occurrences in "src/test1.txt"
```

```
Summary: 7 replacements in "src/test1.txt"
```

Chapter 2

File Processing and Configuration

IN THIS CHAPTER

- » Working with paths and the filesystem
 - » Reading and writing files
 - » Handling CSV, JSON, and TOML files
 - » Building a file organizer tool
-

For your ambitions, Rust is the only reasonable choice in a world filled with compromises.

—BRIAN ANDERSON

It's axiomatic that programs require data, and that applies to command-line programs as well. CLI tools can use internal variables and user input as data, but it's common for command-line utilities to get their data from external files and, when done, to write data back to one or more files.

But it's also common for CLI tools to use external files themselves as data to perform tasks such as creating, copying, renaming, and deleting files, setting file permissions, and reading file metadata.

In this chapter, you delve into these and many more file-related techniques. Along the way, you learn that Rust offers powerful and easy-to-use tools that take all the drudgery and complexity out of working with files, while still maintaining the speed and safety that has made Rust the toast of systems programmers the world over.

Learning File System Foundations

Within the context of `main.rs` (or `lib.rs`), your programs lead a peaceful, Zen-like existence of nothing but pure and predictable Rust code. But the filesystem is where your programs meet the messiness of the real world: objects that may or may not exist; files you don't have permission to access; directories full of documents, logs, photos, and files with names such as `final_v7_ACTUALLY_FINAL_THIS_TIME_MAYBE.pdf`. Yuck.

Rust feels your pain and offers several libraries that make dealing with files and other recalcitrant objects easier and tidier:

- » **`std::path`**: Work with paths in a platform-agnostic way.
- » **`std::fs`**: Create, read, write, rename, and delete files and directories.
- » **`std::io`**: Read and write streams, such as `stdin` and `stdout`.
- » **`std::env`**: Work with environment variables, the current directory, and program arguments.

I discuss `std::io` and `std::env` in Book 4, [Chapter 1](#), so in this chapter the focus is on `std::path` and `std::fs`.

Manipulating paths without breaking a sweat

Imagine a house with three roommates who can't agree on which way the toilet paper should hang. Silly, right? Yet the computing world has a similar domestic dispute: the three big operating systems — Windows, macOS, and Linux — can't agree on which character should separate directories in a file path. Windows prefers using a backslash (`\`, as in `\Users\Bob\Documents`), while macOS and Linux are adamant that the forward slash is the only civilized choice (`/`, as in `/Users/Alice/Documents`).

Fortunately, your Rust programs don't have to take sides in this slash skirmish. The `std::path` module provides powerful tools for working with paths that let you manipulate directories and files safely across platforms. You never have to worry about which slash to use because

Rust's path types — and the operating system itself — handle those details behind the scenes.

But while `std::path` makes your path-related code cross-platform safe, that's not the headline feature. The big benefit is that you can work with paths as *structured data*, not as strings. That is, instead of manually concatenating strings with slashes or splitting them apart to find directory names, Rust gives you path-aware methods that handle all that safely and correctly.

Rust provides two types for working with paths: `Path` (immutable, borrowed) and `PathBuf` (mutable, owned). These are akin to `&str` and `String`, meaning that `&Path` is a view into path data, while `PathBuf` owns the path data.

Creating a new path

If you already know the path you want to work with, you can use a string literal, like so:

```
let user_path_borrowed = Path::new("/Users/Alice");
let user_path_owned = PathBuf::from("/Users/Bob");
```

Building a path

The great news with Rust is that you no longer build a file path by concatenating strings and then worrying about the resulting string having doubled or missing slashes. Forget that. Instead, Rust offers methods that not only build paths for you but also ensure that the resulting paths are well-formed. How you build a longer path from a base path depends on whether you want to modify the base path.

If you don't want the base path modified, initialize it as `&Path` and use the `join()` method to build your path from there. Here's an example (bk04ch02/example01/ in this book's example files):

```
use std::path::Path;

fn main() {

    // base is an &Path
    let base = Path::new("/etc");
```

```

// join() creates a brand new PathBuf
let nginx_conf = base.join("nginx/nginx.conf");

// base is unchanged
println!("Base path: {}", base.display());

// nginx_conf is an owned PathBuf, so you can use it freely
println!("Joined path: {}", nginx_conf.display());
}

```

Output:

```

Base path: /etc
Joined path: /etc/nginx/nginx.conf

```

Alternatively, if you're fine with having the base path modified, initialize it as `PathBuf` and use the `push()` method to add to the path (bk04ch02/example02/):

```

use std::path::PathBuf;

fn main() {
    // config_path is an owned, mutable PathBuf
    let mut config_path = PathBuf::from("/home/user");

    println!("Initial path: {}", config_path.display());

    // push() modifies config_path directly
    config_path.push(".config");
    println!("After push 1: {}", config_path.display());

    config_path.push("app/settings.json");
    println!("After push 2: {}", config_path.display());
}

```

Output:

```

Initial path: /home/user
After push 1: /home/user/.config
After push 2: /home/user/.config/app/settings.json

```

Getting path components

Rust simplifies path handling by enabling coders to let go of string concatenation as a path-building method (as I describe in the preceding section). But Rust also does programmers a favor by enabling them to

say goodbye to string splitting as a path-deconstruction method. Instead, `std::path` offers methods you can use to get at a path's parts:

- » **`path.parent()`**: Returns `Option` containing the directory that contains `path` (that is, the parent directory).
- » **`path.file_name()`**: Returns `Option` containing the final component of `path`, which will be either a file name or a directory name, depending on how `path` ends.
- » **`path.extension()`**: Returns `Option` containing the file extension portion of `path.file_name()` (or `None` if there's no extension).
- » **`path.ancestors()`**: Returns an `Ancestors` iterator over the path's ancestors as `&Path` items (but returning `path` as the first item).
- » **`path.components()`**: Returns a `Components` iterator over the path's `std::Component` items, which can be any of the following:
 - *RootDir*: The root directory (such as `/` or `C:\`)
 - *Prefix*: A Windows drive prefix (such as `C:`)
 - *Normal("item")*: A directory or file with the name *item*
 - *CurDir*: The `.` (current) directory
 - *ParentDir*: The `..` (parent) directory

Here's an example that tries out these methods (bk04ch02/example03/):

```
use std::path::{Component, Path};

fn main() {
    let settings = Path::new("/home/user/.config/app/settings.json");

    let parent = settings.parent().unwrap();
    println!("The parent directory is {:?}.", parent);

    let filename = settings.file_name().unwrap();
    println!("The filename is {:?}.", filename);

    let extension = settings.extension().unwrap();
    println!("The extension is {:?}.", extension);
}
```

```

    for component in settings.components() {
        match component {
            Component::Normal(name) => {
                println!("Path component: {:?}", name.to_string_lossy());
            }
            _ => {}
        }
    }
    for ancestor in settings.ancestors() {
        println!("Next ancestor: {:?}", ancestor);
    }
}

```

Output:

```

The parent directory is "/home/user/.config/app".
The filename is "settings.json".
The extension is "json".
Path component: "home"
Path component: "user"
Path component: ".config"
Path component: "app"
Path component: "settings.json"
Next ancestor: "/home/user/.config/app/settings.json"
Next ancestor: "/home/user/.config/app"
Next ancestor: "/home/user/.config"
Next ancestor: "/home/user"
Next ancestor: "/home"
Next ancestor: "/"

```

Note that the `to_string_lossy()` method converts the path component to a string, replacing any non-Unicode characters with a placeholder. It's lossy because you might lose some information, but it works great for most practical purposes.

Fiddling with the filesystem

Rust's standard library provides the `std::fs` (filesystem) module to handle any type of file operation you can think of. Here's a sampling of the most useful methods that become available when you import

`std::fs`:

- » **`fs::copy(from, to)`**: Copies the contents of the file specified by the path *from* to the file given by the path *to*, overwriting *to* if it exists.

Returns `Result: Ok` if the copy was successful or `Err` if not.

- » **`fs::create_dir_all(path)`**: Creates a directory at the specified *path* as well as all its parent components if they don't exist. Returns `Result: Ok` if the operation was successful or `Err` if not.
- » **`fs::create_dir(path)`**: Creates a new, empty directory at the specified *path*. Returns `Result: Ok` if the operation was successful or `Err` if not.
- » **`fs::exists(path)`**: Determines whether the specified *path* points to an existing directory or file. Returns `Ok(true)` if *path* points to an existing entity; returns `Ok(false)` if *path* is legal but doesn't point to an existing entity; returns `Err` if the *path* is illegal (say, because your program doesn't have the needed permissions).
- » **`fs::read(path)`**: Reads the contents of the file specified by *path* into a bytes vector. Returns `Result: the vector wrapped in Ok` if the operation was successful or `Err` if not.
- » **`fs::read_dir(path)`**: Returns an iterator over the entries in the directory specified by *path*. Returns `Result: Ok` if the operation was successful or `Err` if not.
- » **`fs::read_to_string(path)`**: Reads the contents of the file specified by *path* into a string. Returns `Result: the string wrapped in Ok` if the operation was successful or `Err` if not.
- » **`fs::remove_dir(path)`**: Removes the directory specified by *path*, which must be empty. Returns `Result: Ok` if the removal was successful or `Err` if not.
- » **`fs::remove_dir_all(path)`**: Removes the contents of the directory specified by *path*, and then removes the directory. Returns `Result: Ok` if everything was removed successfully or `Err` if not.
- » **`fs::remove_file(path)`**: Removes the file specified by *path*. Returns `Result: Ok` if the removal was successful or `Err` if not.

- » `fs::rename(from, to)`: Renames the file specified by the path *from* to the name given by *to*, overwriting *to* if it exists. Returns `Result: Ok` if the rename was successful or `Err` if not.
- » `fs::set_permissions(path)`: Changes the permissions on the file or directory specified by *path*. Returns `Result: Ok` if the rename was successful or `Err` if not.
- » `fs::write(path, contents)`: Writes the text specified by *contents* as the entire contents of the file specified by *path*, creating the file if it doesn't exist or overwriting its contents if it does. Returns `Result: Ok` if the write was successful or `Err` if not.

Note that all these methods return `Result`. In your code, you'll usually take advantage of the `?` operator (refer to Book 3, [Chapter 4](#)) to bubble up errors and then use `match` or helper methods to handle them. I provide examples of many of these methods in the rest of this chapter.

Reading and Writing Files Safely

I briefly discuss file reads and writes in Book 4, [Chapter 1](#), but these are important command-line techniques, so it's worth revisiting them in more detail. Alas, reading and writing files is where things can go wrong in a variety of ugly ways. A file or directory might not exist, your program might not have permission to read a file, or the disk might be full. Rust provides a safety net by forcing you to handle the possibility that an error might occur.

Reading files in one fell swoop

For small files that fit comfortably in memory, Rust provides some convenient methods, especially `read_to_string()`. Here's an example (bk04ch02/example04/):

```
use std::fs;

fn read_shopping_list() -> std::io::Result<String> {
    // Read entire file into a String
}
```

```

    let contents = fs::read_to_string("shopping-list.txt")?;
    // Print the contents line by line
    println!("Shopping list:");
    for line in contents.lines() {
        println!("  □ {}", line);
    }
    Ok(())
}

fn main() {
    match read_shopping_list() {
        Ok(_) => println!("\nDon't forget your reusable bags!"),
        Err(e) => println!("Couldn't read list: {}. Guess we're ordering
pizza.", e),
    }
}

```

The `read_shopping_list()` function uses `fs::read_to_string()` to read the contents of `shopping-list.txt` to `String`. (Note the `?` operator that appears at the end of the line.) It then uses `contents.lines()` to iterate through each line, printing each one as it goes. The `main()` function calls `read_shopping_list()` and uses `match` to handle the returned `Result`.

Reading files incrementally

The preceding section's `read_to_string()` function is perfect when you have a small file that your program can comfortably swallow whole. But what if the file is huge? Reading, say, a 10GB log file into memory all at once is like trying to eat an entire buffet in a single sitting. Technically possible (for a computer with a lot of RAM or a person with an enormous stomach and a lot of time) but not recommended!

For larger files, you'll want to process them chunk by chunk. That's where the `std::io` module's `BufReader` comes in. Without buffering, every call to read a line would trigger a separate system call to the operating system, which is expensive and slow. `BufReader` wraps the file and reads larger chunks (8KB by default) into memory at once, and then hands you lines from that buffer. This *buffered reading* process dramatically reduces the number of system calls, which enables you to process large files more efficiently. As a bonus, with this method your

program uses a constant amount of memory, no matter how large the file.

Here's an example (bk04ch02/example05/):

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn count_words_in_file(filename: &str) -> std::io::Result<usize> {
    let file = File::open(filename)?;
    let reader = BufReader::new(file);

    let mut total_words = 0;

    for line in reader.lines() {
        let line = line?; // Each line could fail to read
        total_words += line.split_whitespace().count();
    }
    Ok(total_words)
}

fn main() {
    match count_words_in_file("book.md") {
        Ok(count) => println!("Your book has {} words. Keep writing!",
count),
        Err(e) => println!("Couldn't count words: {}", e),
    }
}
```

Note the use of `std::fs::File`, which gives you a file handle you can work with directly. Unlike `fs::read_to_string()`, which reads the entire file in one shot, `File::open()` just opens the file and returns a handle. You can then decide *how* to read it: all at once, line by line, or in custom-sized chunks. You can also wrap it with `BufReader` or use it with other I/O adaptors. In this example, `reader` is a new `BufReader` for the file, so when the code iterates through the file's contents (lines) with `for line in reader.lines()`, the lines are buffered in as needed. Note, too, that the file is closed automatically as soon as the `file` variable goes out of scope.

Writing files safely

When it comes to writing data to files, the safest pattern is the following four-step procedure:

1. Read the content of the original file.
2. Transform the content in some way.
3. Write the transformed content to a new, temporary file.
4. Rename the temporary file to the name of the original file.

This procedure protects the user from half-written files if something goes wrong. In particular, Rust performs the rename in Step 4 *atomically* (that is, in a single, indivisible step), so that at no point can another process or thread see a partially written or missing file. The switch happens instantaneously and safely.

Here's an example that implements the four steps (bk04ch02/example06/):

```
use std::fs::{self, File};
use std::io::{self, Read, Write};
use std::path::Path;

fn replace_in_file(src: &Path,
                  target: &str,
                  replacement: &str) -> io::Result<()> {
    // 1. Read the original content
    let mut input = String::new();
    File::open(src)?.read_to_string(&mut input)?;

    // 2. Transform the content
    let output = input.replace(target, replacement);

    // 3. Write the transformed content to a temporary file
    // tmp is an owned PathBuf that with "txt" replaced by "tmp"
    let tmp = src.with_extension("tmp");
    { // This creates a scope for the temporary file

        // Create the temporary file
        let mut f = File::create(&tmp)?;

        // Write everything to the file
        // This operation is buffered, therefore...
        f.write_all(output.as_bytes())?;
    }
}
```

```

        // ...make sure everything in the buffer is written to disk
        f.flush()?;
    } // The scope ends here, so the temporary file is automatically closed

    // 4. Rename the temporary file to the original
    fs::rename(&tmp, src)?;

    Ok(())
}

fn main() -> io::Result<()> {
    // Get an &Path to the source file
    let src = Path::new("sexist_idioms.txt");

    // Try the replacement, propagate the error if it fails
    replace_in_file(src, "man", "person")?;

    println!("Replacement successful!");
    Ok(())
}

```

The `replace_in_file()` function takes `&Path` and two string slices and replaces all instances of the `target` string with the `replacement` string. Of particular interest here are the substeps used to implement Step 3 of the procedure:

- a. Use the `with_extension("tmp")` method to create an owned `PathBuf`. Since the original path is `sexist_idioms.txt`, the `tmp` path is `sexist_idioms.tmp`.
- b. Use braces to create a new scope.
- c. Use `File::create(&tmp)` to create the temporary file.
- d. Use `write_all()` to write the transformed content to the temporary file.
- e. Writing to a file is buffered, so call `flush()` to ensure that everything gets to the file safely.
- f. When the block scope ends, the temporary file is automatically closed, which means it's now safe to rename the temporary file in Step 4.

Appending text to a file

Rather than overwriting a file's contents completely, there will be plenty of times when you just want to add some text to the end of a file's existing contents. You can append text by importing

`std::fs::OpenOptions`, which enables you to set various options that configure how a file should be opened. You use `OpenOptions::new()` to start a fresh set of options, and then you set one or more flags. The most useful of these flags are

- » **`create()`**: Create the file if it doesn't exist; open it otherwise.
- » **`create_new()`**: Create the file if it doesn't exist; return an error otherwise.
- » **`read()`**: Enable the file to be read.
- » **`write()`**: Enable the file to be written to.
- » **`append()`**: Enable writes to be appended to the end of the file.

Finally, you call the `open()` method to open the file with the options that have been set. Here's an example that appends text to an existing file (bk04ch02/example07/):

```
use chrono;
use std::fs::OpenOptions;
use std::io::{self, Write};

fn add_journal_entry(entry: &str) -> std::io::Result<()> {
    let mut file = OpenOptions::new()
        .create(true) // Create if it doesn't exist
        .append(true) // Add to the end, don't overwrite
        .open("journal.txt")?; // Open file with options

    writeln!(
        file,
        "[{}]\n{}\n",
        chrono::Local::now().format("%Y-%m-%d %H:%M"),
        entry
    )?;
    Ok(())
}

fn main() -> io::Result<()> {
```

```
add_journal_entry("This is a test entry.")?;  
    println!("Journal entry added.");  
    Ok(())  
}
```

This example uses the `chrono` crate for timestamps. (Run `cargo add chrono` to add it as a dependency.) The `chrono::Local::now()` call gets the current local time, and `format()` formats it as a string using a pattern where `%Y` is the four-digit year, `%m` is the month, `%d` is the day, `%H` is the hour (24-hour), and `%M` is the minute.

Reading and writing CSV files

Plain text files — in whatever flavor they come in: `.txt`, `.md`, `.html`, even `.rs` — are straightforward to work with because they consist of one or more lines of text. But in the real world, lots of data has *structure* of some kind.

I discuss structured files in the JSON and TOML formats in the next two sections. For now, another common structured format is CSV (comma-separated values), a plain text file (`.csv`) where each line represents a record and fields are separated by commas, with text fields optionally enclosed in quotation marks when they contain commas, quotation marks, or line breaks. Here's a partial example, where the first line contains the field names:

```
name,subject,grade  
Harper Jackson,Chemistry,76  
Oliver Anderson,English,72  
Emma Anderson,Science,80  
James Harris,English,68
```

If the idea of parsing a CSV file manually has you shaking in your boots (are there quotation marks or not? what if a field contains a comma?), I hear you. The Rust community also hears you because it offers a better way to process CSV files: the aptly named `csv` crate. Run the following command in your project directory to install it as a dependency:

```
cargo add csv
```

Here's an example that reads data from a CSV file (bk04ch02/example08/):

```

use csv::Reader;
use std::error::Error;

fn analyze_grades(filename: &str) -> Result<(), Box<dyn Error>> {
    // Create a new csv Reader on the file
    let mut reader = Reader::from_path(filename)?;

    // Initialize the analysis variables
    let mut total = 0.0;
    let mut count = 0;

    // Iterate through the records
    for result in reader.records() {
        let record = result?;

        // This CSV has three columns: name, subject, and grade
        let name = &record[0];
        let subject = &record[1];
        let grade: f64 = record[2].parse()?;

        println!("{}", got {} in {}", name, grade, subject);
        total += grade;
        count += 1;
    }

    // Display the average grade
    if count > 0 {
        println!("\nClass average: {:.1}", total / count as f64);
    }

    Ok(())
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
    analyze_grades("grades.csv")
}

```

I'll let the comments explain this code, but this is the first time you've seen a reference to `Box`, so I provide a brief explanation in the sidebar, "[Box: A pointer with smarts](#)." If you elect not to read that sidebar, let me just say that `Box<dyn Error>` refers to `Box` containing any type that implements Rust's `Error` trait. In this example, it allows the function to return different kinds of errors (I/O errors, parsing errors, and so on) through one common return type. (The `dyn` keyword means dynamic

dispatch, where, at runtime, Rust keeps track of which specific error type the box contains.)



TECHNICAL
STUFF

BOX: A POINTER WITH SMARTS

In Rust, `Box<T>`, usually just called `Box`, is the language's most basic smart pointer. Its one and only job is to store data on the heap instead of the stack. Recall that the stack memory area is like a small, tidy desk. It's extremely fast and organized, but you can put things on it only if you know their exact size when you write your code. By contrast, the heap is like a giant, slightly messy storage warehouse. It's slower to access, but you can store things of any size or whose size you won't know until the program is running. To find your stuff in the warehouse, you need a claim ticket. `Box` is that claim ticket.

When you write `let b = Box::new(42);`, you're telling Rust that you want the value `42` stored on the heap instead of its default location, the stack. The variable `b` (`Box`) acts as your small, fixed-size claim ticket that lives on your fast stack desk, but it points to the actual data (the `42`) living in the heap warehouse.

There are two main scenarios when you'd use `Box`:

- **Your data has an unknown size.** If you have a type whose size Rust can't know at compile time (such as a recursive data structure), you must use `Box`. Rust can't put an unknown-sized item on the stack, but it can put the fixed-size claim ticket there.
- **Your data is very large.** If you have a massive chunk of data, it's faster to move the tiny claim ticket around than to copy the entire massive item.

The "smart" part of this smart pointer is that `Box` *owns* the data it points to. When your `Box` variable goes out of scope, Rust automatically takes the claim ticket, goes to the warehouse, and cleans up (deallocates) the data for you, preventing memory leaks.

Writing data to a CSV file is just as straightforward
(bk04ch02/example09/):

```
use csv::Writer;
use std::error::Error;

fn save_inventory(filename: &str) -> Result<(), Box<dyn Error>> {
    // Create a new csv Writer on the file
    let mut writer = Writer::from_path(filename)?;
```

```

// Write the field headings
writer.write_record(&["Item", "Quantity", "Price"])?;

// Write the data
writer.write_record(&["Rubber Duck", "42", "5.99"])?;
writer.write_record(&["Keyboard", "3", "89.99"])?;
writer.write_record(&["Coffee Mug", "17", "12.50"])?;

// Make sure the buffer is cleared and everything is written to the file
writer.flush()?;

println!("Inventory saved!");
Ok(())
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
    save_inventory("src/inventory.csv")
}

```

Serializing and Deserializing Data

Dealing with structured data in Rust often means shifting between the internal and the external. *Internal* data refers to in-memory Rust structures, such as hash maps, vectors, and custom types such as structs and enums. *External* data refers to non-Rust structures, such as the JSON, TOML, and YAML formats. Shifting between these representations involves two processes:

- » **Serializing:** Converting an internal Rust data structure (such as a struct or collection) into an external format that can be easily stored or transmitted (such as JSON or TOML).
- » **Deserializing:** Converting an external data format (such as JSON or TOML) into an internal Rust data structure (such as a struct or collection).

Implementing these processes sure sounds like a lot of work, but once again the Rust crate community rides to the rescue with a library called `serde` (it rhymes with *wordy*), which handles all the messy serialization and deserialization complexity behind the scenes. The result is that your

Rust programs can move smoothly between internal and external data representations with just a few lines of code!

To use `serde`, navigate to your project directory and run the following command:

```
cargo add serde --features derive
```

This chapter's project (refer to the next section) uses `serde` to handle TOML data, so here I show you how to handle the internet's favorite file format: JSON (JavaScript Object Notation), a plain text file (`.json`) that stores data in a structured, hierarchical format made up of key–value pairs and arrays, where objects are enclosed in curly braces (`{}`), arrays in square brackets (`[]`), and keys and string values in double quotation marks (`"`). Here's an example:

```
{
  "player_name": "Hero",
  "level": 42,
  "score": 1337,
  "items": [
    "Sword of Debugging",
    "Shield of Error Handling"
  ]
}
```

To make working with JSON super-easy in your code, add a crate named `serde_json` as a dependency by running the following command in your project directory:

```
cargo add serde_json
```

Here's an example that uses `serde` and `serde_json` to read and write JSON data ([bk04ch02/example10/](#)):

```
use serde::{Deserialize, Serialize};
use std::fs;

// Need to derive Serialize and Deserialize for a custom struct
#[derive(Debug, Serialize, Deserialize)]
struct GameState {
    player_name: String,
    level: u32,
    score: u64,
```



```

        items: Vec<String>,
    }

    // Save game state to a JSON file
    fn save_game(save: &GameState) -> Result<(), Box<dyn std::error::Error>> {
        // Convert the game state to a JSON string
        let json = serde_json::to_string_pretty(save)?;

        // Write the JSON string to a file
        fs::write("savegame.json", json)?;
        println!("Game saved!");
        Ok(())
    }

    // Load a saved game state into a struct
    fn load_game() -> Result<GameState, Box<dyn std::error::Error>> {
        // Read the JSON data as a String
        let json = fs::read_to_string("savegame.json")?;

        // Convert the JSON string to a GameState struct
        let save = serde_json::from_str(&json)?;

        // Return the GameState
        Ok(save)
    }

    fn main() {
        // Initialize the player's game state
        let game_state = GameState {
            player_name: String::from("Hero"),
            level: 42,
            score: 1337,
            items: vec![
                String::from("Sword of Debugging"),
                String::from("Shield of Error Handling"),
            ],
        };

        // Save the game state
        save_game(&game_state).expect("Failed to save");

        // Load and pattern-match the game state
        match load_game() {
            Ok(save) => println!(
                "Welcome back, {}! You're on level {}",
                save.player_name, save.level
            ),
            Err(e) => println!("Couldn't load save: {}", e),
        }
    }
}

```

```
}  
}
```

Output:

```
Game saved!  
Welcome back, Hero! You're on level 42
```

This code stores a player's current game state by using a custom `GameState` struct. Note that the code needs to derive the `serde` traits `Serialize` and `Deserialize` for the custom struct.

The `save_game()` function uses the `serde_json` method `to_string_pretty()` to save the `GameState` data to a JSON string, which is then saved to a file with `fs::write("savegame.json")`.

The `load_game()` function reads the saved JSON file with `fs::read_to_string("savegame.json")`, and then uses the `serde_json` method `from_str()` to load the JSON string to a `GameState` struct, which is then returned wrapped in `Ok`.

Project: Creating a File Organizer

To close, this section brings many of this chapter's file-handling techniques together by showing you how to build a tool called File Organizer that takes all the files in a messy directory (I'm looking at *you*, Downloads directory!) and neatly moves them into subdirectories organized by file type. To get started, `cd` into your Rust projects folder and run the following commands:

```
cargo new file-organizer  
cd file-organizer  
cargo add toml  
cargo add serde --features derive
```

First, create `config.toml` in your project's root:

```
# Rules for the file organizer  
[rules]  
jpg = "images"  
png = "images"  
gif = "images"
```

```
csv = "data"
json = "data"
pdf = "documents"
docx = "documents"
zip = "archives"
mp3 = "music"
```

Each rule specifies a file extension followed by a subdirectory name. File Organizer will traverse a specified directory and, when it comes across a file with one of the rule extensions, move that file into the specified subdirectory, creating the subdirectory if it doesn't exist.

Open `main.rs` in your code editor and then add the imports (`bk04ch02/file-organizer/`):

```
use serde::Deserialize;
use std::collections::HashMap;
use std::fs;
use std::path::Path;
```

Next you need a struct to hold the configuration rules from the TOML file:

```
#[derive(Deserialize)]
struct Config {
    // Use a HashMap to store the rules
    rules: HashMap<String, String>,
}
```

Then fill in the `main()` function:

```
fn main() {
    // Load the TOML configuration file into a String
    let config_contents = fs::read_to_string("config.toml")
        .unwrap_or_else(|e| {
            eprintln!("Error reading config file: {}", e);
            std::process::exit(1);
        });

    // Convert the TOML String to a Config
    let config: Config = toml::from_str(&config_contents).unwrap_or_else(|e| {
        eprintln!("Error parsing config file: {}", e);
        std::process::exit(1);
    });

    println!("Config loaded: {} rules found.", config.rules.len());
}
```

```

// Get the contents of the downloads directory
let current_dir = Path::new("downloads");
let entries = fs::read_dir(current_dir).unwrap_or_else(|e| {
    eprintln!("Error reading current directory: {}", e);
    std::process::exit(1);
});

// Collect the entries into a Vec
let entries_to_process: Vec<_> = entries
    .filter_map(Result::ok)
    .filter(|e| e.path().is_file())
    .collect();

// Track the number of files actually moved
let mut moved_count = 0;

// Iterate through the entries
for entry in &entries_to_process {
    // Get the path
    let path = entry.path();

    // Get the file extension
    if let Some(ext) = path.extension().and_then(|s| s.to_str()) {
        // Check if there is a rule for this extension
        if let Some(dest_folder) = config.rules.get(ext) {
            // If so, set up the destination directory
            let dest_path = Path::new("downloads").join(dest_folder);

            // Create the destination directory if it doesn't exist
            if !dest_path.exists() {
                fs::create_dir_all(&dest_path).ok();
            }

            // Build the full new path for the file
            let new_file_path =
dest_path.join(path.file_name().unwrap());

            // Move the file
            if fs::rename(&path, &new_file_path).is_ok() {
                println!("Moved: {} -> {}",
                    path.display(), new_file_path.display());
                moved_count += 1;
            } else {
                println!("Failed to move: {}", path.display());
            }
        }
    }
}

```

```
}  
println!(  
    "Organization complete! {} files moved.", moved_count  
);  
}
```

Here's a review of what the `main()` function does:

- » **Loads the configuration:** Reads `config.toml` from the current working directory and parses it as TOML into the `Config` struct. If reading or parsing fails, it prints an error and bails out of the function.
- » **Scans the files to organize:** Reads the `downloads` directory, keeps only entries that are files, and collects them into the `entries_to_process` vector.
- » **Moves the matching files:** For each file, it
 - Extracts the file extension.
 - Looks up that extension in `config.rules`.
 - If a rule exists, it ensures the target folder `downloads/dest_path` exists (and creates it if needed).
 - Builds the new path and moves the file there using `fs::rename`.
 - Prints whether the move succeeded. If the move was successful, the `moved_count` variable is incremented.
- » **Prints a completion message:** Outputs how many files were moved.

Chapter 3

Polishing Your Command-Line Tools

IN THIS CHAPTER

- » Making tools flexible with subcommands
 - » Formatting output with colors
 - » Showing progress bars
 - » Building a file organizing tool
-

Rust...is more restrictive, indeed. But only in the sense that a car with seatbelts is more restrictive than one without: both reach the same top speed, but only one of them will save you on a bad day.

—FÉLIX FISCHER

You might have read the verb *polishing* in this chapter's title and figured you were about to read the shortest chapter in the history of technical books. What could possibly need polishing in a command-line tool? Isn't it enough that the program does what it's supposed to do and does it well?

That's a legit argument. You could also argue that a bicycle has no need for a bell, a comfy seat, and a basket for your snacks. Both bikes will get you where you want to go, but only one makes the journey a lot more pleasant.

Making things pleasant for the people who use your command-line tools is what this chapter is all about. Here you learn to build subcommands (because a versatile tool is better than a rigid one), add colors to your output (because life's too short for boring text), and even show progress bars (because nobody likes being kept in the dark).

Organizing Your Tools with Subcommands

You've used Cargo a lot by this point, which means you've used subcommands. Instead of having one giant executable that does everything through a maze of flags, Cargo organizes functionality into clear subcommands: `cargo new`, `cargo build`, `cargo run`, and so on. Each subcommand is like a mini program with its own set of options and arguments. This is a common pattern in CLI tools. If you've used Git or Docker, for example, you've seen even more subcommands: `git commit`, `docker run`, and so on.

When you expand the capabilities of your CLI tool to accomplish more than one or two tasks, subcommands become essential for keeping everything organized and user-friendly.

Why not just use command-line arguments, like you learned in Book 4, [Chapter 1](#)? Okay, imagine a file management tool. Without subcommands, you might end up with a tool that operates something like this:

```
filetool --list
filetool --copy source dest
filetool --delete file
filetool --search term
```

Sure, that works, but it gets messy fast. What if `list` also needs options? What if `copy` needs flags? You end up with a tangled web of mutually exclusive arguments. With subcommands, it's much cleaner:

```
filetool list --detailed
filetool copy source dest --overwrite
filetool delete file --recursive
filetool search term --case-sensitive
```

Each subcommand comes first right after the tool name, followed by the subcommand's options and flags. Users can immediately understand what the tool does just by looking at the available subcommands.

Adding subcommands

The `clap` crate (which you met back in Book 4, [Chapter 1](#)) makes building subcommand-based CLIs straightforward. You can use the `clap` `derive` API, which enables you to define your CLI command structure using enums and structs. To use `clap`, run the following command in your project directory:

```
cargo add clap --features derive
```

As an example, here's the skeleton of a basic note-taking tool that uses subcommands (`bk04ch03/example01/`):

```
use clap::{Parser, Subcommand};

#[derive(Parser)]
#[command(name = "notes")]
#[command(about = "A simple note-taking CLI", long_about = None)]
struct Cli {
    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand)]
enum Commands {
    /// Add a new note
    Add {
        /// The note content
        text: String,

        /// Optional tag for categorization
        #[arg(short, long)]
        tag: Option<String>,
    },

    /// List all notes
    List {
        /// Filter by tag
        #[arg(short, long)]
        tag: Option<String>,
    },

    /// Delete a note by its ID
    Delete {
        /// Note ID to delete
        id: usize,
    },
}
```



```

    },
}

fn main() {
    // Parse the command line
    let cli = Cli::parse();

    // Pattern-match the subcommand
    match cli.command {
        // Add a note
        Commands::Add { text, tag } => {
            println!("Adding note: {}", text);
            if let Some(t) = tag {
                println!("  Tag: {}", t);
            }
        }
        // List the notes
        Commands::List { tag } => {
            println!("Listing notes");
            if let Some(t) = tag {
                println!("  Filtered by tag: {}", t);
            }
        }
        // Delete a note
        Commands::Delete { id } => {
            println!("Deleting note with ID: {}", id);
        }
    }
}

```

Here's a breakdown of what's happening here:

- » **The `cli` struct represents your entire application.** It contains a single field named `command` that represents the tool's subcommands, so this struct field tells the program what action to take.
- » **The `Commands` enum defines all the available subcommands.** Each variant of the enum — `Add`, `List`, and `Delete` — represents one subcommand, and each variant can have its own fields representing that subcommand's arguments and options.
- » **In `main()`, after `clap` parses the command line, a `match` statement pattern-matches on the subcommand.** Each `match` arm enables you to handle each command with its specific arguments.

Here are some ways to invoke this tool:

```
cargo run -- add "Buy groceries"
cargo run -- add "Meeting at 3pm" --tag work
cargo run -- list
cargo run -- list --tag work
cargo run -- delete 1
cargo run -- --help
```

Of course, none of these subcommands does anything other than run a `println!` macro. In a production tool, each `match` arm would call a function that implements the subcommand's full code.

Working with nested subcommands

A single level of subcommands is often enough, but there will be times when you need multiple subcommand levels. For example, a database tool might have a `user` subcommand that requires the second-level subcommands `create` and `delete`: `db user create` and `db user delete`.

Here's an example (bk04ch03/example02/):

```
use clap::{Parser, Subcommand};

#[derive(Parser)]
#[command(name = "db")]
#[command(about = "A simple database CLI", long_about = None)]
struct Cli {
    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand)]
enum Commands {
    /// User management commands
    User {
        #[command(subcommand)]
        action: UserCommands,
    },

    /// Database backup commands
    Backup {
        #[command(subcommand)]
        action: BackupCommands,
    },
}
```

```

}

#[derive(Subcommand)]
enum UserCommands {
    /// Create a new user
    Create {
        username: String,
        #[arg(short, long)]
        email: String,
    },

    /// Delete an existing user
    Delete { username: String },
}

#[derive(Subcommand)]
enum BackupCommands {
    /// Create a database backup
    Create {
        #[arg(short, long, default_value = "./backup")]
        output: String,
    },

    /// Restore from a backup
    Restore { backup_file: String },
}

fn main() {
    // Parse the command line
    let cli = Cli::parse();

    // Pattern-match the subcommands
    match cli.command {
        // Pattern-match the User second-level subcommands
        Commands::User { action } => match action {
            // Create a new user
            UserCommands::Create { username, email } => {
                println!("Creating user: {} ({})", username, email);
            }
            // Delete a user
            UserCommands::Delete { username } => {
                println!("Deleting user: {}", username);
            }
        },
        // Pattern-match the Backup second-level subcommands
        Commands::Backup { action } => match action {
            // Create a backup
            BackupCommands::Create { output } => {

```

```

        println!("Creating backup at: {}", output);
    }
    // Restore from a backup
    BackupCommands::Restore { backup_file } => {
        println!("Restoring from: {}", backup_file);
    }
    },
}
}

```

This code shows that you build multiple levels of subcommands by nesting subcommand enums. In this code, there's a main enum named `Commands` where each variant is a subcommand: `User` and `Backup`. Each of those variants points to another enum that holds the second-level subcommands. For example, within `Commands`, the variant named `User` points to another enum named `UserCommands` that has variants named `Create` and `Delete`.

Here are some example command-line calls for this tool:

```

cargo run -- user create alice --email alice@example.com
cargo run -- user delete bob
cargo run -- backup create --output /backups/daily
cargo run -- backup restore /backups/daily/backup.sql

```

Fancifying Output with Colors

No rule says that your CLI tool output must be white text on a black background (or black text on a white background, depending on your preferred look and Alpha Geek status). If you're a Git user, for example, running `git status` displays your repo's staged files in green and unstaged files in red. These colors aren't there just for show. Rather, they're part of a functional design that helps you quickly understand the current state of your repository.

Adding a splash or two of color to your own CLI tools is a win-win-win-win situation:

- » **Improved scanability:** When a user is faced with 100 lines of log output, yellow warnings and red errors jump out immediately. A user

can spot problems without close-reading every line.

- » **Enhanced user experience:** A CLI tool that uses formatting well feels professional and polished. It shows you care more about UX (user experience) than DX (developer experience).
- » **Better accessibility:** Used thoughtfully, text formatting can make output more accessible. (Used carelessly, text formatting can cause problems for users who are color-blind or have low vision.)
- » **Added emphasis:** Bold text, underlines, and colors enable you to gently but inexorably guide the user's attention to what matters most.

You *could* implement colors and other formatting by hand using cryptic codes called *ANSI escape sequences*, but trust me, it's a nightmare. Thankfully, you don't have to bother with any of that because, once again, the Rust ecosystem has your back in the form of a wonderful little crate called `colored`.

Navigate to your project directory and use the following command to add `colored` to your `Cargo.toml`:

```
cargo add colored
```

With that done, import the crate by adding `use colored::Colorize;` at the top of your module. Here's an example (bk04ch03/example03/):

```
use colored::Colorize;

fn print_status(message: &str, status: &str) {
    match status {
        "success" => {
            // We can chain methods: green and bold.
            println!("{}", "Success:".green().bold(), message);
        }
        "error" => {
            // Use eprint! for errors, and make them red.
            eprintln!("{}", "Error:".red().bold(), message);
        }
        "warning" => {
            println!("{}", "Warning:".yellow(), message);
        }
        _ => {
            println!("{}", message);
        }
    }
}
```

```

    }
}

fn main() {
    print_status("File saved successfully.", "success");
    print_status("Config file not found.", "warning");
    print_status("Permission denied.", "error");
}

```

As demonstrated by the `print_status()` function, you use the `colored` crate by chaining methods for colors, such as `red()`, `green()`, and `yellow()`, and styles, such as `bold()`, `italic()`, and `underline()`, on `String` or `string slice`.

Implementing Progress Bars

When your CLI tool is performing some long-running task, the last thing you want is for your users to be staring at a blank terminal, not knowing whether that blankness means the program has crashed or is still doing its thing. To resolve this user anxiety, you can display a progress bar that shows the user not only that your program is still running but also how much of the task remains to be completed.

That sounds like a major amount of work, but it's actually quite easy thanks to the `indicatif` crate, which makes it a breeze to implement progress bars. To add it to your project's `Cargo.toml`, `cd` into your project directory and then run the following command:

```
cargo add indicatif
```

Here's an example (`bk04ch03/example04/`):

```

use indicatif::{ProgressBar, ProgressStyle};
use std::thread;
use std::time::Duration;

fn main() {
    let items = 1000;
    let pb = ProgressBar::new(items);

    // Customize the appearance
    pb.set_style(

```

```

        ProgressStyle::default_bar()
            .template(
                "{spinner:.green} [{elapsed_precise}] [{bar:40.cyan/blue}]
[CA]{pos}/{len} ({eta})",
            )
            .unwrap()
            .progress_chars("#>-"),
    );

    for _ in 0..items {
        thread::sleep(Duration::from_millis(10));
        pb.inc(1);
    }

    pb.finish_with_message("Complete!");
}

```

This code uses `ProgressBar::new()` to forge a fresh progress bar, and then uses `ProgressStyle::default_bar()` to format it using the following components in the template string:

- » **{spinner:.green}**: A spinning animation in green.
- » **[{elapsed_precise}]**: How long the operation has been running.
- » **[{bar:40.cyan/blue}]**: The progress bar, defined in this case to be 40 characters wide with cyan and blue as the foreground and background colors, respectively.
- » **{pos}/{len}**: The current position and total (for example, 260/1000).
- » **({eta})**: The estimated time remaining.

`ProgressStyle` also defines `progress_chars("#>-")`, which are the characters that make up the bar:

- » **#**: Indicates each completed portion of the task.
- » **>**: Appears just ahead of the # markers (so it acts like the front of the progress).
- » **-**: Indicates each remaining portion of the task.

Here's an example of this progress bar midtask:

```
[00:00:02] [#####>-----] 260/1000 (8s)
```

Project: Building a Multicommand File Organizer

I close this chapter (and [Book 4](#)) by showing you how to build a complete CLI tool that uses all the concepts from this chapter. In this section you create `tidy`, a small but surprisingly useful CLI that helps users organize and tidy up directories.

It has three subcommands:

- » **scan:** List all files in a directory, with colored output by file type. Sample usage:

```
tidy scan ~/Downloads
```

- » **archive --days N:** Move files that are older than *N* days into an archive folder, with a progress bar display. Sample usage:

```
tidy archive ~/Downloads --days 30
```

- » **stats:** Display summary stats, including the number of files, the total size, and the largest file. Sample usage:

```
tidy stats ~/Downloads
```

Here's a sample output from the tool:

```
Scanning /Users/paul/Downloads
notes.txt
photo.png
archive.zip
...
Found 47 files, total size 382.41 MB
Archiving files [████████████████████████████████████████] 47/47 Done!
```

To get started, navigate to your main Rust projects directory, and then run the following commands:


```
cargo new tidy
cd tidy
cargo add clap --features derive
cargo add colored
cargo add indicatif
cargo add walkdir
```

The `walkdir` crate makes it easier for the code to traverse directories recursively.

Now open `main.rs` in your editor, remove the default code, and begin with the imports (`bk04ch03/tidy/`):

```
use clap::{Parser, Subcommand};
use colored::*;
use indicatif::{ProgressBar, ProgressStyle};
use std::error::Error;
use std::fs;
use std::path::PathBuf;
use std::time::{Duration, SystemTime};
use walkdir::WalkDir;
```

Building the command structure

Now you define the command-line interface structure:

```
/// Tidy - a small multi-command tool for cleaning and inspecting
directories.
#[derive(Parser)]
#[command(version, about = "Keep your folders clean and organized")]
struct Cli {
    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand)]
enum Commands {
    /// Scan a directory and list all files
    Scan { path: PathBuf },

    /// Archive files older than N days
    Archive {
        path: PathBuf,
        #[arg(long)]
        days: u64,
    },
}
```

```

    /// Display directory statistics
    Stats { path: PathBuf },
}

```

This code shows a common CLI tool pattern: an enum that holds the defined commands and a struct that holds the enum.

The `#[command(...)]` attributes configure metadata for your CLI. The `name` field sets the program name shown in help text; `about` provides a brief description that appears when users run `--help`; and `long_about = None` tells `clap` not to generate extended help text (you could set this to a longer string if you wanted a more detailed description).

Implementing main()

The `main()` function only parses the command-line arguments and then uses `match` to route the code to the appropriate command:

```

fn main() -> Result<(), Box<dyn Error>> {
    // Parse the command-line arguments
    let cli = Cli::parse();

    // Route the command to the appropriate function
    match cli.command {
        Commands::Scan { path } => scan_dir(path)?,
        Commands::Archive { path, days } => archive_old(path, days)?,
        Commands::Stats { path } => stats(path)?,
    }

    Ok(())
}

```

Now all that remains is to implement the functions for each of the three subcommands.

Implementing the scan command

Here's the code for the `scan` subcommand:

```

fn scan_dir(path: PathBuf) -> Result<(), Box<dyn Error>> {
    println!("{}", format!("Scanning {}", path.display()).bold());

    // Iterate through a new WalkDir (recursive directory traversal)
    for entry in WalkDir::new(path).into_iter().filter_map(Result::ok) {
        // Only handle files
        if entry.file_type().is_file() {

```

```

        // Get the file name
        let file_name = entry.file_name().to_string_lossy();

        // Color the files by type
        let colorized = match entry.path()
            .extension()
            .and_then(|e| e.to_str()) {
            Some("rs") => file_name.green(),
            Some("jpg" | "png") => file_name.cyan(),
            Some("zip") => file_name.yellow(),
            _ => file_name.normal(),
        };
        println!("  {}", colorized);
    }
}
Ok(())
}

```

This code uses the `WalkDir` crate to traverse the specified path recursively and then, for each file, applies a color based on the file type. I included only a few file types here, but feel free to add as many as you want.

Implementing the archive command

Here's the code for the `archive` subcommand:

```

fn archive_old(path: PathBuf, days: u64) -> Result<(), Box<dyn Error>> {
    // Calculate the date before which you want to archive files
    let cutoff = SystemTime::now() - Duration::from_secs(days * 24 * 60 * 60);

    // Set up the archive directory
    let archive_dir = path.join("archive");
    fs::create_dir_all(&archive_dir)?;

    // Create a new WalkDir and store the files in a vector
    let files: Vec<_> = WalkDir::new(&path)
        .into_iter()
        .filter_map(Result::ok)
        .filter(|e| e.file_type().is_file())
        .collect();

    // Initialize the progress bar
    let pb = ProgressBar::new(files.len() as u64);
    pb.set_style(
        ProgressStyle::default_bar()
    );
}

```

```

        .template("{prefix} [{bar:40.cyan/blue}] {pos}/{len}")
        .unwrap()
        .progress_chars("█"),
    );
    pb.set_prefix("Archiving files");

    // Iterate through the files vector
    for entry in files {
        // Get the file's metadata
        let metadata = entry.metadata()?;

        // Find out when the file was last modified
        if let Ok(modified) = metadata.modified() {
            // If it was last modified prior to the cutoff date, archive it
            if modified < cutoff {
                let dest = archive_dir.join(entry.file_name());
                fs::rename(entry.path(), &dest)?;
            }
        }
        pb.inc(1);
    }
    pb.finish_with_message("Done!");
    Ok(())
}

```

This code first determines the cutoff date before which a file is considered old enough to be archived. After setting up an `archive` directory, the code uses `WalkDir` to get all the files, sets up a progress bar, and then iterates through the files. The loop gets the metadata for each file and checks the last modified date. If that date is older than the cutoff date, the file is archived.

Implementing the stats command

And, finally, here's the code for the `stats` subcommand:

```

fn stats(path: PathBuf) -> Result<(), Box<dyn Error>> {
    let mut count = 0;
    let mut total_size = 0;

    // Iterate through a new WalkDir
    for entry in WalkDir::new(path).into_iter().filter_map(Result::ok) {
        // Only handle files
        if entry.file_type().is_file() {
            // Update the file count and the total file size
            count += 1;

```

```

        total_size += entry.metadata()?.len();
    }
}

// Display the stats
println!(
    "{}",
    format!(
        "Found {} files, total size {:.2} MB",
        count,
        total_size as f64 / 1_000_000.0
    )
    .bright_yellow()
);

Ok(())
}

```

This code creates a `WalkDir` recursive traversal of the path, and then iterates through the found entries. For each entry that's a file, the code increments the file counts and updates the total file size. Once the iteration loop is complete, the code displays the number of files and their total size.

Book 5

WebAssembly: Rust in Your Browser

Contents at a Glance

Chapter 1: WebAssembly Basics and Setup

[Web What Now? Understanding WebAssembly](#)

[Why Rust and WebAssembly Are Best Friends](#)

[Setting Up Your Rust-to-WASM Toolchain](#)

[Running Your First Rust Function in the Browser](#)

[Running JavaScript Functions in Rust](#)

[Common WASM Build Commands](#)

[Project: Building a Rust-Powered Password Analyzer](#)

Chapter 2: Exchanging Data between Rust and JavaScript

[Understanding the wasm-bindgen Bridge](#)

[Exchanging Data through the Wall](#)

[Swapping Strings](#)

[Passing Arrays, Vectors, and Other List-Like Things](#)

[Project: Building a Remote Markdown Renderer](#)

Chapter 3: Building Interactive WebAssembly Apps

[Controlling DOM from Rust](#)

[Handling Browser Events](#)

[Managing Application State](#)

[Project: Populating DOM from Rust](#)

Chapter 1

WebAssembly Basics and Setup

IN THIS CHAPTER

- » Understanding WebAssembly
 - » Setting up your development environment for WASM
 - » Writing and running your first Rust code in a web browser
 - » Building a Rust-powered password analyzer
-

It feels absolutely insane that I can just compile this language that's basically the opposite of JavaScript and it's running in the browser.

—TOMAS SEDOVIC

The home of every Rust program you've built so far has been the terminal. Nothing wrong with that, but the biggest, most-used application platform is not the command line but the web browser.

Okay, fine, I hear you saying to yourself, but to get code to run in the browser I have to use JavaScript. Not so fast! Yep, JavaScript is by far the language of choice for standard-issue web work. But what if you want the browser to run high-performance graphics, complex physics simulations, or a heavy-duty data-processing task? And what if you want the browser to do it with the speed, safety, and power of Rust?

Forget JavaScript. Welcome, instead, to the remarkable world of WebAssembly. This chapter is your launching pad, taking your Rust skills from the terminal and putting them in front of billions of users on the web.

Web What Now? Understanding WebAssembly

JavaScript is a great language that can do tons of wonderful things, but many modern web apps strain JavaScript to its limits. Whether it's video editing, real-time collaboration, 3D graphics, or big-time data processing tasks, JavaScript just isn't designed to handle this level of complexity.

Enter WebAssembly, or *WASM* as the cool kids call it (pronounced “WAH-zum” or “WUZ-em,” depending on the kid). WebAssembly sounds like a web version of Assembly language, but the name is misleading. Assembly language is a low-level language that talks almost directly to a computer's hardware. WebAssembly is low-level in a similar sense, but it's not a language that you write directly. Instead, you write code in a higher-level language such as Rust (or C++ or Go) and a compiler translates your code to WebAssembly. The browser downloads this compiled code and runs it at near-native speed. WASM has been supported by all modern web browsers since 2017, so it's safe to use for any modern web project.

To understand where WebAssembly fits in, it helps to know how web applications work. When you visit a web page, the server sends your browser the page's code. Your browser then runs that code on your computer, not back on the server. That client-side execution is where both JavaScript and WebAssembly live. Running code in the browser means faster responses for the user (no round-trip to the server needed) and less work for the server.

So, is WebAssembly meant to replace JavaScript? Nope, not even close. Each has its own role:

- » JavaScript is the high-level manager of the web page. It's fantastic at handling user events, manipulating the Document Object Model (DOM), and orchestrating all things page-related.
- » WebAssembly is the behind-the-scenes specialist that JavaScript can call on to do the heavy lifting. Need to process a terabyte of data,

render 3D graphics, or run a complex physics engine? You don't want the manager doing that; you want to hand it off to the specialist who can do it ten times faster.

Here are some real-world applications that use WebAssembly:

- » **Adobe Photoshop:** The web version of Adobe's popular graphics app uses WebAssembly to handle computationally intensive graphics processing.
- » **Figma:** This popular design tool uses WASM to handle complex graphics operations.
- » **Google Earth:** This virtual globe app renders 3D terrain and imagery using WebAssembly.
- » **AutoCAD:** This 35-year-old desktop application for creating 2D and 3D drawings is on the web thanks to WASM.
- » **Games:** Many web-based games use WASM for game logic and physics.

The beauty of WebAssembly is that users don't know that it's there. They just know that their web apps feel faster and more responsive.

Why Rust and WebAssembly Are Best Friends

Several languages can compile to WebAssembly, so why use Rust? It turns out that Rust and WebAssembly are almost perfectly matched. Here's why:

- » **No garbage collector:** Many languages use a garbage collector to manage memory automatically. That's convenient, but garbage collectors add overhead and can cause unpredictable pauses. Rust's ownership system means you get safe, compile-time memory management without garbage collection. This matters in the browser because you want predictable, consistent performance.

- » **Small binary sizes:** Rust produces compact WebAssembly files because there's no runtime environment to include. A simple Rust WebAssembly module might be just a few kilobytes. Every kilobyte matters when users are downloading your code over potentially slow connections.
- » **Memory safety:** WebAssembly runs in a sandbox, but bugs such as accessing freed memory or writing past the end of a buffer can still cause crashes or unexpected behavior. Rust's compiler catches these kinds of bugs before your code even runs, which means fewer surprises in production.
- » **Great tooling:** The Rust ecosystem has first-class support for WebAssembly. Tools such as `wasm-bindgen` and `wasm-pack` (both explained in this chapter) make it pleasantly straightforward to build, test, and deploy Rust code that runs in browsers.
- » **Growing ecosystem:** The Rust WASM community is active and growing. When you need to integrate with web APIs or JavaScript libraries, chances are someone has already built a crate to help.

Even better, you've already learned Rust's trickiest concepts, including ownership, borrowing, and lifetimes. Using Rust for WebAssembly doesn't add much complexity beyond understanding how to build a bridge between Rust and JavaScript.

That said, Rust for WebAssembly isn't magic. You'll need to learn some new concepts, especially around how Rust code interacts with JavaScript. But if you've made it through the first four books, you have all the Rust knowledge you need. The rest is just learning the WASM-specific tools and patterns.

Setting Up Your Rust-to-WASM Toolchain

Before you can write Rust code that runs in a browser, you need to add two (or maybe three) new tools to your Rust workshop. The next two (or

maybe three) sections provide the details.

Installing the wasm32 target

First, you need to tell Rust's compiler that you want to compile code for WebAssembly. Rust uses the term *target* to describe different platforms it can compile for. The WebAssembly target is called `wasm32-unknown-unknown`. (Yep, that's really its name. The `32` means it uses a 32-bit memory address space. The two `unknown` parts mean *unknown operating system* and *unknown environment*, respectively. They're unknown because WebAssembly doesn't run on a traditional operating system. Instead, it runs inside a browser or other host, so there's no specific operating system or environment to target.)

Open your terminal and run the following command:

```
rustup target add wasm32-unknown-unknown
```

This command downloads the libraries Rust needs to compile code for WebAssembly. You need to run this command only once.

Installing wasm-pack

Next, install `wasm-pack`, which is the official tool for building, testing, and publishing Rust-generated WebAssembly. Think of it as Cargo's specialized cousin for web development.

On macOS and Linux, run:

```
curl https://rustwasm.github.io/wasm-pack/installer/init.sh -sSf | sh
```

On Windows, download the installer from the `wasm-pack` GitHub releases page:

```
https://github.com/rustwasm/wasm-pack/releases
```

After installation, verify that `wasm-pack` is available:

```
wasm-pack --version
```

You should see a version number such as `0.13.1` or higher.

Installing Node.js (optional but recommended)

Although not strictly required, having Node.js installed makes local development much easier. Node.js includes `npm`, which provides a simple local web server for testing your WASM projects.



TIP If you have Python installed, you can skip Node.js and use Python’s built-in web server, instead (as I explain in the “[Spinning up a local server for your page](#)” section).

Download Node.js from <https://nodejs.org>. Choose the LTS (Long-Term Support) version and follow the installation instructions for your platform. Verify the installation:

```
node --version
npm --version
```

Creating a test project

Verify that everything works by creating a minimal WebAssembly project:

```
cargo new --lib wasm_test
cd wasm_test
```

Now add the `wasm-bindgen` dependency:

```
cargo add wasm-bindgen
```

The `wasm-bindgen` dependency provides the glue between Rust and JavaScript (refer to Book 5, [Chapter 2](#) to learn more).

Open `Cargo.toml` and add the following just above the `[dependencies]` section:

```
[lib]
crate-type = ["cdylib"]
```

The `crate-type = ["cdylib"]` line tells Rust to create a dynamic library, which is the format WebAssembly needs.

Now try building for WebAssembly:

```
wasm-pack build --target web
```

If you see a message saying `Your wasm pkg is ready to publish`, congratulations! Your toolchain is set up correctly. If you encounter errors, double-check that you’ve installed the `wasm32` target and that `wasm-bindgen` is listed in your dependencies.

Running Your First Rust Function in the Browser

Now for the fun part: writing Rust code that runs in a web browser. I’m excited! I’m going to keep things simple and show you how to implement a variation of the classic “Hello, World!” example, except Rust will be saying hello to the web.

Getting your project off the ground

In a nearby terminal, navigate to your Rust projects directory and run the following commands:

```
cargo new --lib hello_wasm
cd hello_wasm
cargo add wasm-bindgen
```

Note that WASM projects are libraries that are called by a web app running in the browser, not binaries that run on their own. Open `Cargo.toml` and add the following above the `[dependencies]` section:

```
[lib]
crate-type = ["cdylib"]
```

Open `src/lib.rs` and replace the default code with the following (bk05ch01/hello_wasm/ in this book’s example files):

```
use wasm_bindgen::prelude::*;

/// A tiny greeting maker just to prove Rust can run in your browser
#[wasm_bindgen]
pub fn hello(name: &str) -> String {
    format!("Greetings and salutations from Rust, {name}!")
}
```

Here’s what’s going on in this code:

- » `use wasm_bindgen::prelude::*`: This line imports everything you need from the `wasm-bindgen` library, which is the bridge between Rust and JavaScript.
- » `#[wasm_bindgen]`: This attribute tells the compiler to make the following function available to JavaScript. Without this attribute, JavaScript wouldn't be able to see or call your Rust function. Think of it as a kind of export keyword for WebAssembly.
- » `pub fn hello()`: This standard-issue Rust function takes a string slice, creates a formatted greeting, and returns a new `String`. Note that the function is marked `pub`, which is required for any function you want to call from JavaScript.

Building your project

Now you can build your project:

```
wasm-pack build --target web
```

This command compiles your Rust code to WebAssembly and generates JavaScript code to load and interact with the WASM module. When the compilation finishes, check inside the `pkg` directory, where several new files now lurk:

- » `hello_wasm_bg.wasm`: Your compiled Rust code in WebAssembly format.
- » `hello_wasm.js`: JavaScript glue code that loads the WASM code and sets everything up for your Rust code to run successfully in the browser.
- » `hello_wasm.d.ts`: TypeScript definitions (useful if you use TypeScript).
- » `hello_wasm_bg.wasm.d.ts`: Another TypeScript declaration file. This one provides type definitions for the low-level, raw WebAssembly exports (such as memory and pointer-based functions) contained in the main `.wasm` binary.

Creating a web page

Okay, you have your WebAssembly binary, now how do you get it to run in the browser? First, you need a web page. Create a file named `index.html` in your project root (the parent directory of `pkg` and `src`). Open the HTML file in your editor and populate it with the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Hello, WASM World!</title>
</head>
<body>
  <h1>Rust WebAssembly Test Drive</h1>
  <div id="output"></div>

  <script type="module">
    // Import what we need from the JS file
    import init, { hello } from './pkg/hello_wasm.js';

    // Run the Rust code
    async function run() {
      // Initialize WASM
      await init();

      // Run the Rust function
      const message = hello("World!");

      // Output the returned string
      document.getElementById('output').textContent = message;
    }
    // Go, WASM, go!
    run();
  </script>
</body>
</html>
```

The HTML here is super-simple: just an `h1` heading and a `div` to display the WASM output. The HTML also includes the `<script type="module">` tag, which tells the browser to treat this JavaScript code as a module so that the `import` statement can do its thing. The

`import init, { hello } from './pkg/hello_wasm.js';` line imports two things from the generated JavaScript:

- » **init:** A function that loads and initializes your WebAssembly module
- » **hello:** Your Rust function, now callable from JavaScript

The `run()` function is marked `async` because loading WebAssembly is asynchronous. You must call `await init()` before you can use any Rust functions. This initialization step loads the WASM binary and sets up everything.

After initialization, `hello()` is now just a regular JavaScript function. You call it with a string argument, and it returns a string result. The fact that it's running compiled Rust code is invisible.

Spinning up a local server for your page

You can't just double-click the HTML file to open it in a browser. Modern browsers block loading WASM modules from the local file system for security reasons. You need a web server.

If you installed Node.js earlier, you can use `npm` to run a simple server. In your project directory, run the following command:

```
http-server
```

This starts a local web server and tells you the URL to visit, usually `http://127.0.0.1:8080`.

If you don't have Node.js, you can use Python's built-in web server. If you have Python 3, in your project directory run the following command:

```
python -m http.server 8000
```

(In some Python installs, notably on macOS, you need to change `python` to `python3`.) This command will serve files from the current directory on port 8000. You can then access these files in your web browser by navigating to `http://localhost:8000`.

Either way, open the server URL in your browser, and you should see your greeting message, as shown in [Figure 1-1](#).

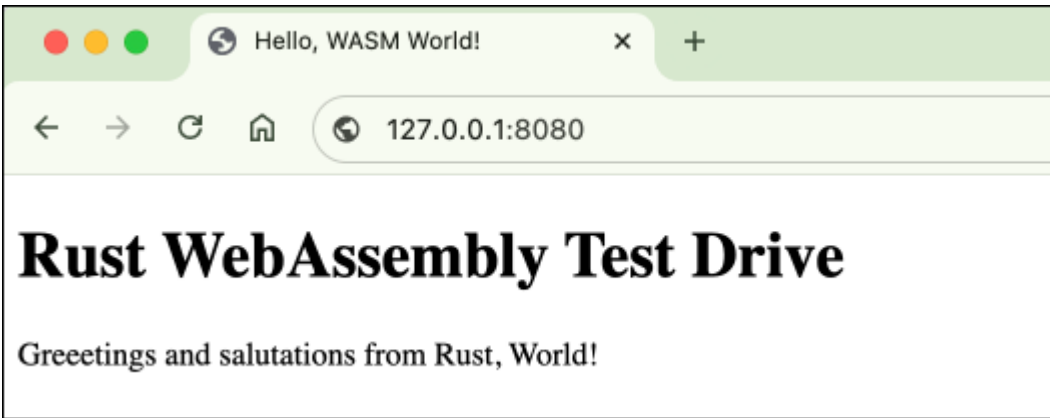


FIGURE 1-1: Honest-to-goodness Rust code running in a web browser!

What just happened?

I urge you to take a moment to appreciate what you just accomplished. You wrote some code in Rust, compiled it to WebAssembly, loaded it in a web browser, called it from JavaScript, and saw the results on a web page. Not only did the entire roundtrip from Rust to WASM to JavaScript to DOM run silky smooth, but that journey is everything important you need to know about WebAssembly.

And, from this simple example, you now know the full workflow you'll use for all your WebAssembly development:

1. Write Rust code with `#[wasm_bindgen]` attributes.
2. Build your project with `wasm-pack`.
3. Create HTML and JavaScript to load and use your WASM module.
4. Serve the page by using a local web server.
5. Test in your browser.
6. Repeat Steps 1 through 5 as needed.

Running JavaScript Functions in Rust

In the example I ran through in the preceding section, you sent some Rust code to JavaScript, which worked because `wasm-bindgen` handled both the nitty and the gritty details of bridging the gap between Rust and JavaScript. These two languages handle data differently, so `wasm-bindgen` generates the code that enables them to pass data back and forth. That bridge is where the `bindgen` part of the name comes from: It's a *binding generator*, meaning it generates the glue code — the *bindings* — between the Rust and JavaScript worlds. But those bindings are a two-way street, which means `wasm-bindgen` also enables you to use JavaScript code in your Rust WebAssembly projects!

Here's the general syntax for bringing in external JavaScript code:

```
[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = namespace)]
    fn method(args);
}
```

where *namespace* is a JavaScript namespace, *method* is a method in *namespace*, and *args* is a list of arguments for *method*.



TECHNICAL
STUFF

The `extern "C"` block tells Rust that the function (or the collection of functions; you can define two or more in a single block) you're defining doesn't live in Rust. Instead, it's a foreign function supplied by JavaScript. The `"C"` part specifies how Rust should set up the function calls, which includes tidbits such as how arguments are passed and how return values are handled. Even though C code is not involved, the `"C"` designation gives `wasm-bindgen` a well-known convention to work with when generating

the glue code that enables WebAssembly and JavaScript to communicate with each other.

[Table 1-1](#) lists a few common JavaScript namespaces that you can use as the `js_namespace` value.

TABLE 1-1 Common JavaScript Namespaces

Namespace	Rust Usage	What You Can Call
Math	# [wasm_bindgen(js_namespace Math.random(), Math.max(), and so on = Math)]	
console	# [wasm_bindgen(js_namespace console.log(), console.warn(), and so on = console)]	
JSON	# [wasm_bindgen(js_namespace JSON.stringify(), JSON.parse() = JSON)]	
Date	# [wasm_bindgen(js_namespace Date.now() and other static Date methods = Date)]	
Promise	# [wasm_bindgen(js_namespace = Promise)]	Promise.resolve(), Promise.all(), and so on
Object	# [wasm_bindgen(js_namespace = Object)]	Object.keys(), Object.entries(), and so on
window	# [wasm_bindgen(js_namespace = window)]	window.alert(), window.fetch(), and other window globals
document	# [wasm_bindgen(js_namespace = document)]	document.getElementById() and other DOM-related calls
navigator	# [wasm_bindgen(js_namespace = navigator)]	navigator.userAgent() and other browser/device info

Namespace	Rust Usage	What You Can Call
performance	<pre># [wasm_bindgen(js_namespace = performance)]</pre>	performance.now() and other performance-related info
URL	<pre># [wasm_bindgen(js_namespace = URL)]</pre>	URL.hostname(), URL.pathname, and other URL parsing and manipulation

For example, suppose you want to time an operation using `performance.now()` and write a message to the browser console using `console.log()`. You can bind both JavaScript methods to your Rust code as follows (bk05ch01/count_primes_wasm/):

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    // Use the browser's console.log function
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);

    // Use the browser's high-resolution timer: performance.now()
    #[wasm_bindgen(js_namespace = performance)]
    fn now() -> f64;
}
```

Now you can use the `log()` and `now()` functions like regular Rust functions:

```
// A simple (non-optimized) prime check
fn is_prime(n: u32) -> bool {
    if n < 2 {
        return false;
    }
    if n == 2 {
        return true;
    }
    if n % 2 == 0 {
        return false;
    }
    let mut d = 3;
    while d * d <= n {
        if n % d == 0 {
```

```

        return false;
    }
    d += 2;
}
true
}

#[wasm_bindgen]
pub fn count_primes(limit: u32) {
    let start = now();

    let mut count = 0;
    for n in 2..=limit {
        if is_prime(n) {
            count += 1;
        }
    }
    let end = now();
    let elapsed = end - start;

    let message = format!(
        "Found {count} primes up to {limit} in {elapsed:.3} ms
        (computed in Rust, logged from console.log).");
    log(&message);
}

```

This code defines the `is_prime()` function to return `true` if the passed number is prime or `false` otherwise. Using this in an HTML page is straightforward:

```

<script type="module">
    import init, { count_primes } from "../pkg/count_primes_wasm.js";

    async function run() {
        await init();
        // Let Rust do the work and log to the console.
        count_primes(1_000_000);
    }
    run();
</script>

```

I should mention here that some browser functions have global scope, so you can use them directly in your Rust code. Here's an example that uses `window.alert()` directly without having to bind the `window` namespace ([bk05ch01/alert_wasm/](#)):

```

use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn show_warning(value: i32) {
    if value > 100 {
        alert("Warning: Value exceeds maximum!");
    }
}

```

[Table 1-2](#) lists the most common global browser functions.

TABLE 1-2 Common Global Browser Functions

<i>Global</i>	<i>Rust Binding</i>	<i>What It Does</i>
<code>alert()</code>	<code>fn alert(s: &str);</code>	Shows a modal alert dialog
<code>confirm()</code>	<code>fn confirm(s: &str) -> bool;</code>	Shows a confirmation dialog; returns <code>true/false</code>
<code>prompt()</code>	<code>fn prompt(s: &str) -> Option<String>;</code>	Shows an input dialog box. May return <code>null</code>
<code>setTimeout()</code>	<code>fn setTimeout(closure: &Closure<...>, ms: i32);</code>	Sets a timer
<code>setInterval()</code>	<code>fn setInterval(closure: &Closure<...>, ms: i32);</code>	Sets a repeating timer
<code>clearTimeout()</code>	<code>fn clearTimeout(id: i32);</code>	Cancels a timeout set by <code>setTimeout()</code>
<code>clearInterval()</code>	<code>fn clearInterval(id: i32);</code>	Cancels a repeating timer
<code>fetch()</code>	<code>fn fetch(input: &str) -> Promise;</code>	Fetches a resource
<code>btoa()</code>	<code>fn btoa(s: &str) -> String;</code>	Encodes a string in base64
<code>atob()</code>	<code>fn atob(s: &str) -> String;</code>	Decodes a base64 string

Common WASM Build Commands

Here's a quick list of the `wasm-pack` commands you'll use most often:

- » **Build for the web:** This is what you've been doing so far. The `--target web` command creates output optimized for loading directly in a browser:

```
wasm-pack build --target web
```

This build is also known as the *release* build. Building for release is the default, but if you enjoy typing, you can also build explicitly for release by using the following variation:

```
wasm-pack build --target web --release
```

- » **Build for a bundler:** If you're using a JavaScript bundler such as Webpack or Vite, use this command, instead:

```
wasm-pack build --target bundler
```

- » **Build in development mode:** By default, `wasm-pack` builds in release mode with optimizations. For faster builds during development, add the `--dev` flag:

```
wasm-pack build --target web --dev
```

The downside is that the WASM file will be larger and slower. Use release mode when you need to test performance or before deploying.

- » **Clean the build artifacts:** If something seems wrong, try cleaning the build output, and then running a new build:

```
cargo clean  
wasm-pack build --target web
```

Project: Building a Rust-Powered Password Analyzer

Okay, it's time to bring everything together with a project that's slightly more ambitious than the examples I present earlier in this chapter. This project is a password analyzer that takes a password entered in a web

page text box and then provides a score for that password based on various security metrics, such as the number of characters and whether the password includes characters such as uppercase letters and numbers.

When working with WebAssembly, it's best to decide up front what you want JavaScript to handle and what you want to pass on to Rust. In this project, JavaScript will handle the front end of the web page, including the input field and the DOM updates, while Rust will handle the back end of analyzing the password and computing a score.

At this point, you might be wondering why you'd bother using Rust here. You could write a password-strength checker in plain JavaScript. So why involve Rust at all? A few reasons:

- » **Safer string handling:** Rust makes you think about characters instead of raw bytes, which helps avoid subtle Unicode bugs.
- » **Testable logic:** The password-scoring function is just normal Rust code. You can write unit tests for the function, run them with `cargo test`, and be confident it works before you ever plug it into a web page.
- » **Reusable core:** The same Rust library can be used in a command-line tool, a desktop app, and a WebAssembly module. JavaScript logic tends to be trapped in the browser.
- » **A stepping stone to heavier tasks:** Today it's password scoring; tomorrow it might be cryptographic checks, data parsing, or anything else that benefits from Rust's speed and safety. The integration pattern is the same.

The point of this chapter's project isn't that you must use Rust for password analysis. It's that you learn how to hand a piece of logic over to Rust and call it from JavaScript, which opens the door to far more demanding tasks later.

Okay, use a terminal to navigate to your main Rust projects directory, and then run the following commands:


```
cargo new --lib password_analyzer
cd password_analyzer
cargo add wasm-bindgen
```

Using your favorite text editor, open this project's `Cargo.toml` file and add the following just above the `[dependencies]` section:

```
[lib]
crate-type = ["cdylib"]
```

Open `src/lib.rs` and replace the boilerplate code with the following (bk05ch01/password_analyzer/):

```
use wasm_bindgen::prelude::*;

/// Analyzes a password and returns a short report.
///
/// This function is compiled to WebAssembly and called from JavaScript.
#[wasm_bindgen]
pub fn analyze_password(pw: &str) -> String {
    // Count Unicode scalar values
    // This is safer than using pw.len(), which returns the number of bytes
    let length = pw.chars().count();

    // Initialize the character categories
    let mut has_lower = false;
    let mut has_upper = false;
    let mut has_digit = false;
    let mut has_symbol = false;

    for c in pw.chars() {
        // Check for a lowercase letter
        if c.is_ascii_lowercase() {
            has_lower = true;
        }
        // Check for an uppercase letter
        } else if c.is_ascii_uppercase() {
            has_upper = true;
        }
        // Check for a digit
        } else if c.is_ascii_digit() {
            has_digit = true;
        }
        // Otherwise, it's a symbol
        } else {
            has_symbol = true;
        }
    }

    // Add up the character categories
```

```

let mut char_categories = 0u32;
if has_lower {
    char_categories += 1;
}
if has_upper {
    char_categories += 1;
}
if has_digit {
    char_categories += 1;
}
if has_symbol {
    char_categories += 1;
}

// Calculate the score as follows:
// - up to 10 characters, 5 points each (max 50 points)
// - each character category adds 10 points (max 40 points)
// - bonus for long + diverse passwords (10 points)
let mut score = 0u32;
score += (length as u32).min(10) * 5;
score += char_categories * 10;
if length > 10 && char_categories >= 3 {
    score += 10;
}
if score > 100 {
    score = 100;
}

// Rate the password based on the score
let rating = if score < 30 {
    "Very weak"
} else if score < 50 {
    "Weak"
} else if score < 70 {
    "Okay"
} else if score < 90 {
    "Strong"
} else {
    "Very strong"
};

// Return a report
format!(
    "Length: {length} characters\n\
    Character types: {char_categories}\n\
    Score: {score}/100 ({rating})"
)
}

```

The `analyze_password()` function analyzes the received password and generates a score as described in the comments. The function returns a report that displays the password length, character types, score, and rating.

The HTML code is minimal:

```
<h1>Rust-Powered Password Analyzer</h1>

<label for="pw">Enter a password:</label>
<input id="pw" type="password" autocomplete="off">

<div class="strength" id="strength-label">
Type a password to see its score.</div>
<pre id="report"></pre>

<script type="module" src="main.js"></script>
```

Note that I've put the JavaScript in a separate file named `main.js`, which resides in the same directory as the `index.html` file. Here's the code:

```
import init, { analyze_password } from "../pkg/password_analyzer.js";

const passwordInput = document.getElementById("pw");
const strengthLabel = document.getElementById("strength-label");
const reportPre = document.getElementById("report");

function chooseColor(score) {
  if (score < 30) return "red";
  if (score < 50) return "orangered";
  if (score < 70) return "orange";
  if (score < 90) return "green";
  return "darkgreen";
}

init().then(() => {
  passwordInput.addEventListener("input", () => {
    const pw = passwordInput.value;

    if (pw.length === 0) {
      strengthLabel.textContent = "Type a password to see its score.";
      strengthLabel.style.color = "inherit";
      reportPre.textContent = "";
      return;
    }
  })
})
```

```

const report = analyze_password(pw);

// Extract the "Score: NN/100 (...)" line for a quick label.
const scoreLine = report.split("\n").find(line =>
line.startsWith("Score:")) || "";
const scoreMatch = scoreLine.match(/Score:\s+(\d+)/);
const score = scoreMatch ? parseInt(scoreMatch[1], 10) : 0;

strengthLabel.textContent = scoreLine || "Score unavailable";
strengthLabel.style.color = chooseColor(score);
reportPre.textContent = report;
});
}).catch(err => {
  console.error("Error initializing WebAssembly module:", err);
  strengthLabel.textContent = "Error loading password checker.";
});

```

After the WASM initialization, this code listens for input on the password field. When a character is entered, the code sends the input to the `analyze_password()` function, and then displays the result. Once your code is set, run the following command and then launch your web server to check it out:

```
wasm-pack build --target web
```

[Figure 1-2](#) shows the password analyzer at work.

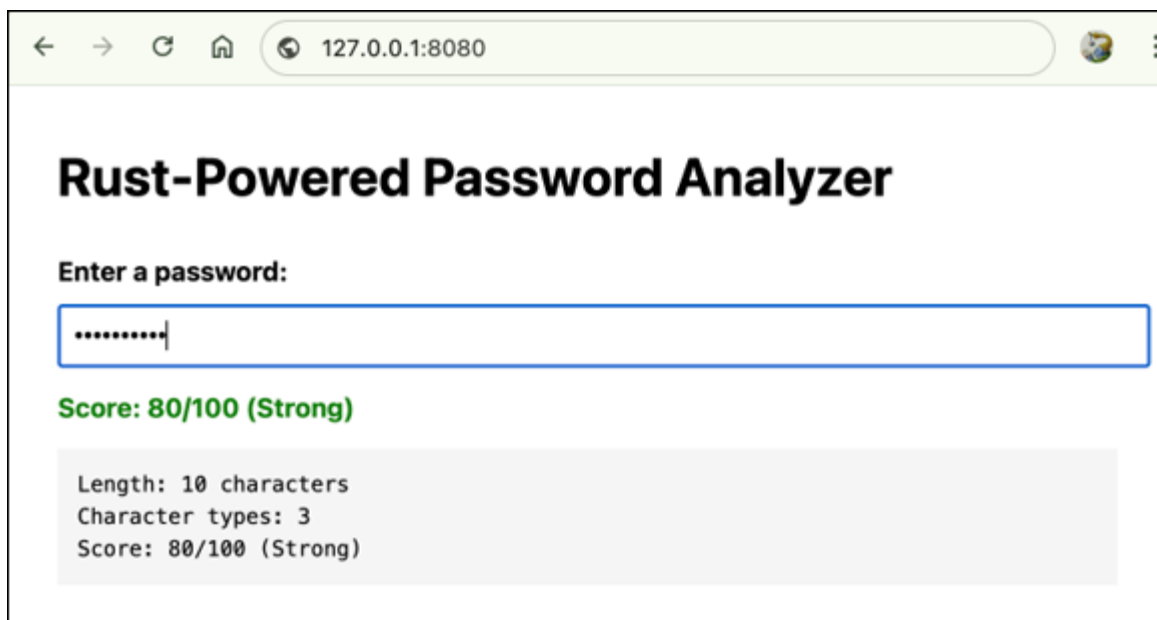


FIGURE 1-2: The password analyzer doing its thing.

Chapter 2

Exchanging Data between Rust and JavaScript

IN THIS CHAPTER

- » Understanding how Rust and JavaScript swap data
 - » Sending strings back and forth
 - » Exchanging more complex types
 - » Building a remote Markdown renderer
-

A Rustacean is a programmer that dislikes being told “yes” in situations where they’ll regret it later.

—PREDRAG GRUEVSKI

The secret to making WebAssembly a useful part of your Rust toolkit is becoming adept at exchanging data between Rust and JavaScript. That task sounds straightforward enough, but it’s actually a significant challenge because you’re talking about sharing stuff between two languages that differ down to their bones on how data should be handled. JavaScript is the lah-di-dah friend who views types as suggestions and thinks `undefined` is a reasonable value. Rust is the buttoned-down pal who gets upset when you call a `u32` an `i32` and thinks `null` is an unseemly way to live.

In this chapter, you learn how to bridge this cultural divide. You discover how to pass strings, numbers, arrays, and even complex data structures between Rust and JavaScript without either one having an existential crisis.

Understanding the wasm-bindgen Bridge

The `wasm-bindgen` library, which I introduce in Book 5, [Chapter 1](#), is doing a lot of work behind the scenes. To use WebAssembly effectively, you need to understand what it's doing and why.

WebAssembly has a very limited set of types it can work with directly. WASM only understands two types, in fact:

- » Integers (`i32`, `i64`)
- » Floating-point numbers (`f32`, `f64`)

That's it. No strings, no arrays, no structs, no enums. If you want to pass a string from JavaScript to Rust, someone needs to convert that JavaScript string into a format Rust understands, which involves copying bytes around in memory.

Similarly, JavaScript doesn't understand Rust's memory layout. If your Rust function returns a `Vec<u32>`, JavaScript has no idea what to do with that.

This is where `wasm-bindgen` slides in. It generates code on both the Rust side and the JavaScript side to handle these conversions automatically. When you call a Rust function from JavaScript, the generated code

1. Converts JavaScript types to WASM-compatible types
2. Calls your Rust function
3. Converts the Rust return value back to JavaScript types

All of this happens transparently, which is why, in Book 5, [Chapter 1](#), you could call the Rust function `hello("World")` in JavaScript and it just worked.

The `wasm-bindgen` library supports conversions for many common types:

- » **Basic types:** `bool`, integers, and floats all cross the boundary directly.
- » **Strings:** JavaScript strings convert to a Rust `String` or `&str`, and vice versa. This conversion copies the string data, so it's not free, but it's automatic.
- » **Numbers in collections:** JavaScript arrays of numbers can convert to Rust `Vec<i32>`, `Vec<f64>`, and similar types.
- » **Options:** Rust's `Option<T>` converts to JavaScript nullable values. For example, `Some(5)` becomes `5`, and `None` becomes `null`.
- » **Results:** With a bit of extra work (which you see later), Rust `Result` types can throw JavaScript exceptions.
- » **Borrowed vs owned:** For strings, you can use `&str` if you need to only read the value, or `String` if you need to modify it or return it. Using `&str` is more efficient because it avoids copying when possible.

In the rest of this chapter, you learn techniques for working with a few common data types.

Exchanging Data through the Wall

Imagine Rust and JavaScript are two people sitting in separate rooms. They can't see each other. They can't peer inside each other's memory. They can't access each other's variables directly. The only way they can communicate is by passing notes through a slot in the wall.

That slot is WebAssembly's *linear memory*, a single, contiguous block of bytes that a WASM module uses for all its data. It's basically a big array of bytes that both JavaScript and Rust can read from and write to.

When you call a function such as `add(1, 2)`, passing the data is straightforward. Rust writes the number `1` and the number `2` onto a note, slides it through the slot, and JavaScript reads it. Numbers are `Copy` types, so they're small, fixed size, and cheap to move. Other types that

implement `Copy`, such as `bool` and `char`, also get passed through the wall with next to no fuss.

But what happens when you want to pass non-`Copy` data through the wall, such as a string? Ah, that's where things get interesting, as the next couple of sections demonstrate.

Swapping Strings

On the surface, strings are a big problem because JavaScript and Rust have fundamentally different takes on what constitutes a string. JavaScript strings are sequences of UTF-16 code units. They're immutable, they live on the heap, and they're garbage collected. Rust has two string types: `String` (owned) and `&str` (borrowed). Both are UTF-8, so neither is directly compatible with JavaScript's UTF-16 approach, which means every string passed through the wall needs to be converted.

When you write a WASM function that takes a string parameter, it's best if that parameter is of type `&str`, as I demonstrate in the following example from Book 5, [Chapter 1](#):

```
#[wasm_bindgen]
pub fn hello(name: &str) -> String {
    format!("Greetings and salutations from Rust, {name}!")
}
```

Behind the scenes, `wasm-bindgen` handles the conversion. When JavaScript calls this function with a string, the JavaScript string gets converted to UTF-8, copied into WebAssembly memory, and passed to your Rust function as a string slice. When the function returns its `String`, that data gets converted back to UTF-16 and handed to JavaScript as a proper JavaScript string.

This conversion isn't free: It takes time and uses memory. But for most applications, it's fast enough that you won't notice. Just be aware that if you're processing huge strings or doing string operations in a tight loop, those conversions can add up.

One of the best use cases for Rust-in-the-browser is string processing that JavaScript handles inefficiently. Here's a function that removes all punctuation from text (bk05ch02/remove_punctuation/ in this book's example files):

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn remove_punctuation(text: &str) -> String {
    text.chars()
        .filter(|c| !c.is_ascii_punctuation())
        .collect()
}
```

This function takes a string slice, iterates through its characters, filters out anything that's ASCII punctuation, and collects the remaining characters into a new `String`, which is what gets sent back to the browser.

Another useful example is a function that capitalizes the first letter of each word (bk05ch02/title_case/):

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn title_case(text: &str) -> String {
    text.split_whitespace()
        .map(|word| {
            let mut chars = word.chars();
            match chars.next() {
                None => String::new(),
                Some(first) => {
                    first.to_uppercase().collect::<String>()
                        + &chars.as_str().to_lowercase()
                }
            }
        })
        .collect::<Vec<_>>()
        .join(" ")
}
```

This splits the text into words, capitalizes the first character of each word, lowercases the rest, and joins everything back together with spaces. It's more robust than JavaScript's typical approach because it

properly handles Unicode characters. (For example, using `charAt()` and string indexing in JavaScript can break multibyte characters.)

Passing Arrays, Vectors, and Other List-Like Things

JavaScript is famous for a few things: flexible arrays, creative debugging sessions, and the uncanny capability to turn everything into `undefined`. Rust, meanwhile, prefers its data well-typed, well-behaved, and preferably stored in a `Vec<T>` where it can't cause any mischief.

When Rust code meets JavaScript code through WebAssembly, these two worlds need a translation guide, especially when you want to pass collections such as arrays, vectors, typed arrays, lists of strings, or even custom objects.

In this section, you learn three useful patterns for shuttling list-type data through the wall separating Rust and JavaScript. Each pattern has its own sweet spot and its own mental model. By the end, you'll know which pattern to choose and how to implement it.

The three essential patterns are

- » **Numeric arrays using typed arrays:** The high-performance path for numbers, bytes, audio samples, images, and anything that looks like numbers in a row
- » **Lists of strings using `js_sys::Array`:** The reliable way to pass a list of text values
- » **Complex structs using Serde:** The JSON-flavored way to safely pass complex data through the wall

Over the next few sections, I take you through each pattern step-by-step, with examples from both sides of the wall.

Passing numeric vectors and arrays

List-passing is at its most straightforward when the lists in question contain only numeric values. Numeric vectors and arrays are the easiest type of list to share between the two languages because both sides agree on how to represent a sequence of numbers.

For example, JavaScript has typed arrays such as `Uint8Array`, `Float32Array`, and `Int32Array`. Meanwhile, Rust has slices (`&[T]`) and owned vectors (`Vec<T>`). Thanks to the behind-the-scenes efforts of `wasm-bindgen`, these types map straightforwardly.

Sending a Rust vector to JavaScript

Suppose your Rust code generates a bunch of floating-point values. They could be the values that define a sine wave, sensor data, financial data, the secret codes to health, wealth, and happiness, whatever. To return this list to JavaScript, you do two things:

- » Store the data in a `Vec[T]`, where the `T` is the appropriate Rust numeric type, such as `Vec<f32>`.
- » Convert that vector into a JavaScript typed array, such as a `Float32Array`.

How does that conversion work? For that you need a crate called `js-sys`, which handily provides bindings for all JavaScript global objects and functions. To use this crate, navigate to your project directory and run the following command to add the dependency to `Cargo.toml`:

```
cargo add js-sys
```

Then, in your code, import the JavaScript type you want to use:

```
use js_sys::Float32Array;
```

Now you can use `Float32Array` just as though it was a native Rust type! Here's an example (`bk05ch02/generate_wave/`):

```
use js_sys::Float32Array;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn generate_wave(samples: u32) -> Float32Array {
```

```

let mut data = Vec::with_capacity(samples as usize);
let frequency = 440.0; // A4 note

for i in 0..samples {
    let t = i as f32 / samples as f32;
    // Calculate sine values in radians (note: TAU = 2π)
    let value = (t * frequency * std::f32::consts::TAU).sin();
    data.push(value);
}

// Copy the Vec<f32> data from WASM memory into a JavaScript Float32Array
// This allows JS to access the samples in its own typed array format
Float32Array::from(data.as_slice())
}

```

This code generates 1 second of audio samples using the `generate_wave()` function. The resulting `Vec<f32>` is converted to a JavaScript-friendly `Float32Array` and stored in memory as a slice to allow JavaScript to access it. The web page plays this sine wave through the Web Audio API:

```

import init, { generate_wave } from './pkg/generate_wave.js';

async function run() {
    // Initialize the WASM module
    await init();

    const playButton = document.getElementById('playButton');
    const status = document.getElementById('status');

    playButton.addEventListener('click', async () => {
        try {
            // Create audio context
            const audioContext = new (window.AudioContext ||
window.webkitAudioContext)();

            // Generate 44100 samples (1s at 44.1kHz sample rate)
            const sampleRate = audioContext.sampleRate;
            const duration = 1; // seconds
            const samples = sampleRate * duration;

            status.textContent = `Generating ${samples} samples...`;

            // Call WASM function to generate wave data
            const waveData = generate_wave(samples);

```

```

        // Create audio buffer
        const audioBuffer = audioContext.createBuffer(1, samples,
            sampleRate);
        const channelData = audioBuffer.getChannelData(0);

        // Copy WASM-generated data into audio buffer
        channelData.set(waveData);

        // Create and play audio
        const source = audioContext.createBufferSource();
        source.buffer = audioBuffer;
        source.connect(audioContext.destination);
        source.start();

        status.textContent = 'Playing sine wave (${samples}
            samples at ${sampleRate}Hz)';

        source.onended = () => {
            status.textContent = 'Playback finished. Click to
play
            again.';
        };
    } catch (error) {
        status.textContent = 'Error: ${error.message}';
        console.error(error);
    }
    });
}
run();

```

Sending a JavaScript array to Rust

Passing a JavaScript array to Rust is straightforward. JavaScript hands Rust a typed array, and Rust receives it as either a slice (`&[T]`) or a vector (`Vec<T>`).

Here's an example where Rust calculates the average brightness of an image pixel buffer (`bk05ch02/average_brightness/`):

```

use wasm_bindgen::prelude::*;

#[wasm_bindgen]
// Receive the JS data as a slice: &[u8]
pub fn average_brightness(pixels: &[u8]) -> f32 {
    if pixels.is_empty() {
        return 0.0;
    }
}

```

```

    let sum: u32 = pixels.iter().map(|&b| b as u32).sum();
    sum as f32 / pixels.len() as f32
}

```

And here's how JavaScript calls the function:

```

import init, { average_brightness } from "../pkg/average_brightness.js";

await init();

const pixels = new Uint8Array([0, 128, 255, 64, 192]);
console.log('The average pixel brightness is
            ${average_brightness(pixels).toFixed(2)}');

```

Note that in the Rust code, the passed JavaScript array is handled as an `& [u8]` slice.

Passing string vectors and arrays

There is no typed array of strings, so you use a general `Array` on the JavaScript side, which is constructed on the Rust side using `js_sys::Array`. But what to put in that object? For these situations, `wasm-bindgen` provides the `JsValue` type, which is essentially a handle to any JavaScript value. In this case, you use it to store strings, as shown in the next section.

Sending a vector of strings to JavaScript

Here's an example where Rust string slices are sent to JavaScript as `JsValue` types in a `js_sys::Array` ([bk05ch02/split_words/](#)):

```

use js_sys::Array;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn split_words(text: &str) -> Array {
    let arr = Array::new();

    for word in text.split_whitespace() {
        // Push the slice to the Array as a JsValue
        arr.push(&JsValue::from(word));
    }

    arr
}

```

Here's the JavaScript:

```
import init, { split_words } from "../pkg/split_words.js";

await init();

const words = split_words("Rust to JavaScript");
console.log(words);
console.log(`${words[2]} ${words[1]} ${words[0]}`);
```

Sending an array of strings to Rust

The process of accepting a JavaScript array of strings in Rust is similar to what I demonstrate in the preceding section, except that instead of sending `js_sys::Array` to JavaScript, your Rust code accepts `js_sys::Array` as a Rust function argument.

Here's an example (bk05ch02/join_strings/):

```
use js_sys::Array;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn join_words(arr: &Array, separator: &str) -> String {
    let mut pieces = Vec::new();

    for value in arr.iter() {
        if let Some(s) = value.as_string() {
            pieces.push(s);
        }
    }

    pieces.join(separator)
}
```

The `join_words()` function takes a borrowed `&Array` and a string slice specifying the separator, pushes the strings to a vector, and then returns the joined strings.

Here's the JavaScript:

```
import init, { join_words } from "../pkg/join_words.js";

await init();
```

```
const words = ["JavaScript", "to", "Rust"];
console.log(join_words(words, " "));
```

Passing structs and objects

When you need to pass more complex data such as Rust structs or JavaScript objects, you run into problems because Rust structs are laid out in WebAssembly's linear memory, while JavaScript objects float in JavaScript's own heap. These layouts are fundamentally incompatible. So, you're stuck, right?

Happily, no. The way out is to use Serde (the handy crate I talk about in Book 4, [Chapter 2](#)) to serialize and deserialize Rust structs and JavaScript objects. The conversion works because JavaScript has JSON-like objects, Rust has Serde, and `wasm-bindgen` uses `JsValue` as the carrier between them.

To let the compiler know you want to use `wasm-bindgen` with Serde support, include the `serde-wasm-bindgen` crate in your project:

```
cargo add serde wasm-bindgen serde-wasm-bindgen --features serde/derive
```

Sending a Rust struct to JavaScript

Imagine a Rust function that returns basic user profile info: username, age, and whether they're an admin. On the JavaScript side, you want an ordinary object such as this:

```
{ username: "Alice", age: 42, isAdmin: true }
```

Here's an example that sends a Rust user profile struct to JavaScript as such an object (`bk05ch02/get_user_profile/`):

```
use serde::Serialize;
use wasm_bindgen::prelude::*;

#[derive(Serialize)]
pub struct UserProfile {
    username: String,
    age: u8,
    is_admin: bool,
}

#[wasm_bindgen]
```



```
pub fn get_user_profile() -> JsValue {
    let profile = UserProfile {
        username: "Alice".to_string(),
        age: 42,
        is_admin: true,
    };

    // Convert the struct into a JsValue using serde-wasm-bindgen
    serde_wasm_bindgen::to_value(&profile).unwrap()
}
```

The `get_user_profile()` function creates and initializes an instance of the `UserProfile` struct, Serde serializes it, and `wasm-bindgen` hands JavaScript a plain object that looks like this:

```
{
  username: "Alice",
  age: 42,
  is_admin: true
}
```

The `serde_wasm_bindgen::to_value()` function converts this Rust struct into a JavaScript object by

- » Converting the `String` field `username` into a JavaScript string
- » Converting the `u8` field `age` into a JavaScript number
- » Converting the `bool` field `is_admin` into a JavaScript Boolean
- » Using the struct field names as JavaScript object property names (in `snake_case`)

Here's what's happening on the JavaScript side:

```
import init, { get_user_profile } from "../pkg/get_user_profile.js";

async function main() {
    await init();

    // Get the user profile as a JavaScript object
    const profile = get_user_profile();

    // Access the object's properties
    console.log(profile.username);
    console.log(profile.age);
}
```

```
        console.log(profile.is_admin);
    }
    main()
```

Sending a JavaScript object to Rust

Sending a Rust struct to JavaScript involves serializing the data into a JavaScript object, so you might not be surprised to learn that passing a JavaScript object to Rust involves deserializing the data into a Rust struct.

For example, suppose you have a JavaScript object that holds app settings:

```
const settings = {
  theme: "dark",
  show_tips: true,
  items_per_page: 20,
};
```

You want Rust to apply or validate the settings, which ideally means getting that JavaScript data into a Rust struct with the proper types. Here's an example (bk05ch02/apply-settings/):

```
use serde::Deserialize;
use wasm_bindgen::prelude::*;

#[derive(Debug, Deserialize)]
pub struct AppSettings {
    theme: String,
    show_tips: bool,
    items_per_page: u32,
}

#[wasm_bindgen]
pub fn apply_settings(settings_js: &JsValue) -> String {
    // Try to deserialize the incoming JS object into the Rust struct
    let settings: AppSettings = match
    serde_wasm_bindgen::from_value(settings_js.clone()) {
        Ok(s) => s,
        Err(err) => {
            return format!("Failed to parse settings: {err}");
        }
    };

    // Here you'd normally store the settings, update global state, etc.
    // For the demo, just return a summary string
```

```
format!(
    "Applied settings: theme = {}, show_tips = {}, items_per_page = {}",
    settings.theme, settings.show_tips, settings.items_per_page
)
}
```

Here, the `apply_settings()` function gets the JavaScript object as a `JsValue`, and then `serde_wasm_bindgen::from_value()` attempts to deserialize the object into an `AppSettings` struct.

Here's the story from JavaScript's point of view:

```
import init, { apply_settings } from "./pkg/apply_settings.js";

async function main() {
    await init();

    const settings = {
        theme: "dark",
        show_tips: true,
        items_per_page: 20,
    };

    const result = apply_settings(settings);
    console.log(result);
}

main();
```

Here, `settings` is declared as a standard-issue JavaScript object, which is then passed to the `apply_settings()` function.

Project: Building a Remote Markdown Renderer

Okay, it's time to combine everything you've learned about passing data back and forth into a handy utility: fetching remote Markdown code and rendering it as HTML right in the browser! Specifically, this project is a tool that takes the address of a remote Markdown file (such as the `README.md` file from a GitHub repository) and renders it as HTML right in the browser.

To fetch remote data, the code needs to handle asynchronous tasks. In JavaScript, you use `Promises`. In Rust, you use `Futures`. Fortunately, the `wasm-bindgen-futures` crate lets you bridge the two easily.

This project requires a few dependencies:

- » **web_sys:** Provides raw Rust bindings to practically all the Web APIs implemented by browsers. In this project, instead of writing JavaScript that uses `fetch()` to asynchronously retrieve remote data, you use the Rust types provided by the `web_sys` crate.
- » **wasm_bindgen_futures:** Acts as the translator between how JavaScript handles asynchronous tasks (using `Promises`) and how Rust handles them (using `Futures`). This crate provides a `JsFuture` type that wraps a JavaScript `Promise` and turns it into a Rust `Future`, allowing you to use `await` in your Rust code just like you would in JavaScript.
- » **pulldown-cmark:** Parses Markdown and renders it as HTML.

Navigate to your main Rust projects folder, and then run the following commands:

```
cargo new --lib render_markdown
cd render_markdown
cargo add pulldown-cmark wasm-bindgen wasm-bindgen-futures
cargo add web-sys --features Window,Request,RequestInit,RequestMode,Response
```

Open `Cargo.toml` and add the following above the `[dependencies]` section:

```
[lib]
crate-type = ["cdylib"]
```

Open `src/lib.rs`, remove the boilerplate code, and then add the imports at the top of the file (`bk05ch02/render_markdown/`):

```
use pulldown_cmark::{Parser, html};
use wasm_bindgen::prelude::*;
use wasm_bindgen_futures::JsFuture;
use web_sys::{Request, RequestInit, RequestMode, Response};
```

Now add the function that converts Markdown to HTML:

```
#[wasm_bindgen]
pub fn render_markdown(markdown_input: &str) -> String {
    // 1. Create a parser for the Markdown
    let parser = Parser::new(markdown_input);

    // 2. Create a buffer to hold the HTML output
    let mut html_output = String::new();

    // 3. Push the parsed HTML into the buffer
    html::push_html(&mut html_output, parser);

    // 4. Return the HTML string
    html_output
}
```

This code puts `pulldown_cmark` to work by creating a new `Parser` for Markdown, creating a buffer for the output, and then pushing the parsed HTML into that buffer. The function then returns a `String` containing the HTML. What could be easier?

Next up is the code for asynchronously fetching the remote Markdown file:

```
#[wasm_bindgen]
pub async fn fetch_and_render(url: &str) -> Result<String, JsValue> {
    // 1. Set up the fetch request with CORS mode
    let opts = RequestInit::new();
    opts.set_method("GET");
    opts.set_mode(RequestMode::Cors);

    // 2. Create the request object
    let request = Request::new_with_str_and_init(url, &opts)?;

    // 3. Get the global window object and call fetch
    let window = web_sys::window().ok_or("No window object")?;
    let resp_value =
        JsFuture::from(window.fetch_with_request(&request)).await?;

    // 4. Convert the response to a Response object
    let resp: Response = resp_value.dyn_into()?;

    // 5. Check if the response was successful
    if !resp.ok() {
        return Err(JsValue::from_str(&format!(
```

```

        "HTTP error: {} {}",
        resp.status(),
        resp.status_text()
    ));
}

// 6. Get the response body as text
let text = JsFuture::from(resp.text()).await?;
let markdown = text.as_string().ok_or("Response was not valid text")?;

// 7. Render the Markdown to HTML and return
Ok(render_markdown(&markdown))
}

```

The `fetch_and_render()` function is a great example of the `web_sys` and `wasm-bindgen-futures` crates working together to perform a network request. Here's what's happening under the hood:

- » **Setting up the request (`web_sys`):** `web_sys::RequestInit` and `web_sys::Request` build the HTTP request. This request is the Rust equivalent of creating the options object you would normally pass to a JavaScript `fetch()` call. In this case, the code sets the method to GET and the mode to `Cors`.
- » **Getting the Window (`web_sys`):** In the browser, `fetch()` lives on the global `window` object. The code uses `web_sys::window()` to grab a handle to that global object so that it can call methods on it.
- » **Bridging the async gap (`wasm_bindgen_futures`):** Calling `window.fetch_with_request()` means the browser returns a JavaScript `Promise`. Rust can't wait for that natively, however, so the code uses `JsFuture::from()` to wrap that `Promise`. The `from()` method converts `Promise` into something Rust understands, enabling the code to use the `await` keyword. `await` pauses the function until the network request finishes.
- » **Handling the response (`web_sys`):** Once the request returns, the function casts the result into a `web_sys::Response` object. This action gives the code access to standard web properties such as `ok()`,

`status()`, and `status_text()` so the code can check if the request was successful (for example, that there wasn't a 404 error).

» **Reading the body (`wasm_bindgen_futures`):** Just as in JavaScript, reading the text body of a response is also an asynchronous operation that returns `Promise`. The code uses `JsFuture` one more time to `await resp.text()`, which provides the final Markdown string that is then passed to the `render_markdown()` function.

Build your WASM code:

```
wasm-pack build --target web
```

The `index.html` file is quite long, so I won't repeat the code here.

[Figure 2-1](#) shows the page, which provides a dropdown list box populated with some GitHub README.md examples, plus the option to enter a custom URL. The box on the left shows the Markdown (which is editable) while the box on the right shows the rendered HTML, straight from Rust!

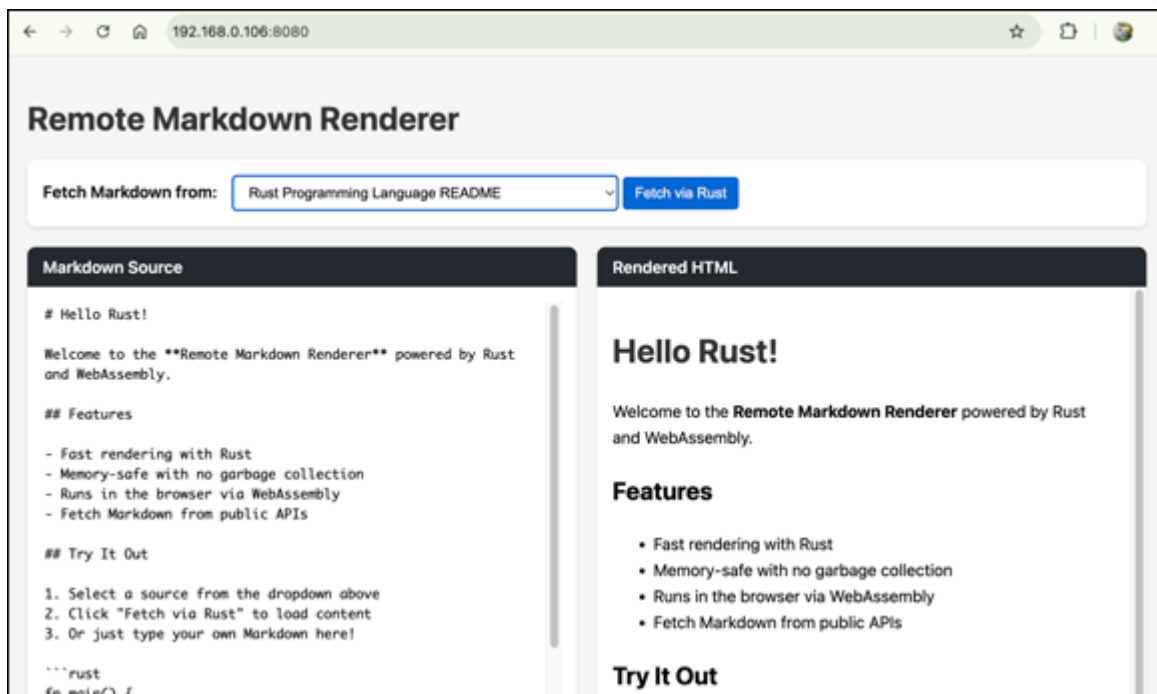


FIGURE 2-1: The remote markdown renderer running in the browser.

Here's a partial listing of the JavaScript that handles the WASM code:

```

import init, { render_markdown, fetch_and_render }
  from './pkg/render_markdown.js';

async function run() {
  await init();

  const input = document.getElementById('markdown-input');
  const output = document.getElementById('html-output');
  const sourceSelect = document.getElementById('source-select');
  const fetchBtn = document.getElementById('fetch-btn');

  // Render markdown from textarea
  const update = () => {
    const html = render_markdown(input.value);
    output.innerHTML = html;
  };

  // Get the current URL (from dropdown or custom input)
  const getUrl = () => {
    if (sourceSelect.value === 'custom') {
      return customUrlInput.value.trim();
    }
    return sourceSelect.value;
  };

  // Fetch markdown via Rust/WebAssembly
  fetchBtn.addEventListener('click', async () => {
    const url = getUrl();
    if (!url) return;

    fetchBtn.disabled = true;

    try {
      // Call Rust to fetch and render the Markdown
      const html = await fetch_and_render(url);
      output.innerHTML = html;
    } catch (error) {
      showStatus('Error: ' + error, 'error');
    } finally {
      fetchBtn.disabled = false;
    }
  });

  input.addEventListener('input', update);
  update();
}

run();

```


Note that the `render_markdown()` WASM function is used in the `update()` function, which is called once at the end of the `run()` function and each time the `input` text area is changed.

Chapter 3

Building Interactive WebAssembly Apps

IN THIS CHAPTER

- » Messing with DOM, Rust-style
 - » Handling clicks and keypresses
 - » Managing JavaScript state — from Rust!
 - » Using Rust to populate a page with JSON data
-

Rust does best when we're ambitious.

—NIKO MATSAKIS

Exchanging data between Rust and JavaScript, as I describe in the first two chapters here in [Book 5](#), is legit awesome. Or, I should say, it's awesome the first few times you try it. After that, you might feel a little, well, *dissatisfied*. Why? Because in all the WebAssembly examples I've presented so far, Rust has been the passive partner waiting around for JavaScript to tell it what to do.

It's time to change that. In this chapter, you learn how to build WebAssembly apps that actively engage with the browser, respond to user input, and manage their own state. And you close your WASM education by building a project that uses Rust to populate a web page with data from a JSON file.

Controlling DOM from Rust

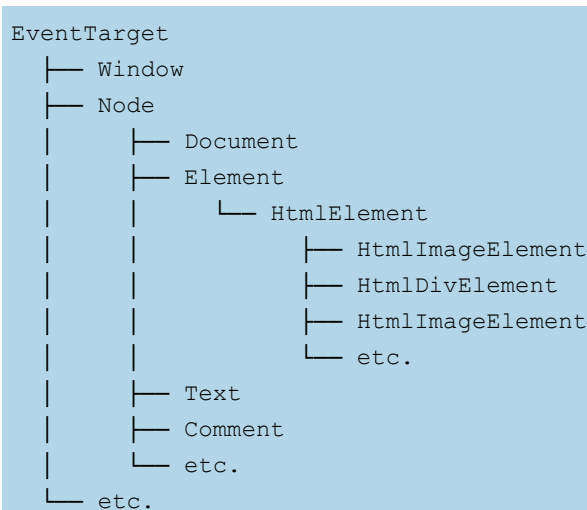
The scaffolding holding up every web page is the Document Object Model (DOM), a Web API that represents the page as a hierarchy of

nodes and objects. Think of DOM as a family tree for your webpage. The `html` element is the top-level ancestor, and its children are the `head` and `body` elements. These have their own children, those children have children, and so on down the tree. Every element, every piece of text, every attribute lives somewhere in this tree.



REMEMBER I'm assuming that you're familiar with DOM and how to manipulate it with JavaScript. If you're not, you can find out everything you need to know in my book *HTML, CSS, & JavaScript All-in-One For Dummies*.

In Book 5, [Chapter 2](#), I introduce the `web_sys` crate, which provides Rust bindings to most Web APIs including, crucially, DOM. For a web page DOM, `web-sys` creates the following hierarchy:



The root is called `EventTarget` because it's the base for any object that can receive events, including all of the following:

» **Window:** The browser window or tab. As you learn shortly, running `web_sys::window()` is your entry point to the entire DOM because it gives your Rust code access to the browser's global `window` object, from which you can access all kinds of good things:

- `window.document()`: Returns the document object associated with `window`
- `window.location()`: Returns the current URL of `window`
- `window.local_storage()`: Provides access to the browser's Storage object for the `window`

Docs: https://wasm-bindgen.github.io/wasm-bindgen/api/web_sys/struct.Window.html

» **Node:** The base interface for all DOM objects. It represents any node in the document tree, including elements, text, comments, and the document itself. It provides fundamental tree-traversal and manipulation methods, including the following:

- `parent.append_child(child)`: Appends `child` to `parent`
- `parent.remove_child(child)`: Removes `child` from `parent`
- `child.parent_node()`: Returns the parent node of `child`
- `parent.child_nodes()`: Returns all the child nodes of `parent`
- `node.set_text_content(str)`: Sets the text content of `node` to `str`, which must be an `Option<&str>`

Docs: https://wasm-bindgen.github.io/wasm-bindgen/api/web_sys/struct.Node.html

» **Document:** The loaded HTML document. It provides methods to find and create elements, including the following:

- `document.get_element_by_id(id)`: Returns the element that has the `id` attribute value given by `id`
- `document.query_selector(selector)`: Returns the first element that matches the specified CSS `selector`
- `document.query_selector_all(selector)`: Returns all the elements that match the specified CSS `selector`
- `document.create_element(element)`: Creates the element specified by `element`

Docs: https://wasm-bindgen.github.io/wasm-bindgen/api/web_sys/struct.Document.html

» **Element:** The base interface for all generic HTML elements in DOM, extending `Node` with element-specific functionality. It provides methods for working with elements, including the following:

- `element.set_attribute(attribute, value)`: Sets *attribute* equal to *value* for the specified *element*
- `element.get_attribute(attribute)`: Gets the current value of *attribute* for the specified *element*
- `element.set_class_name(class)`: Adds the class named *class* to the specified *element*
- `element.class_list()`: Returns all the class names currently set on *element*

Docs: https://wasm-bindgen.github.io/wasm-bindgen/api/web_sys/struct.Element.html

» **HtmlElement:** The base interface for HTML elements, such as `p`, `div`, and `main`. These bindings extend `Element` with functionality that's specific to HTML elements, such as the following methods:

- `element.focus()`: Sets the focus on the specified *element*
- `element.blur()`: Removes the focus on the specified *element*
- `element.click()`: Simulates a mouse click on the specified *element*

Docs: https://wasm-bindgen.github.io/wasm-bindgen/api/web_sys/struct.HtmlElement.html

There are also bindings for specific element types, such as `HtmlButtonElement`, `HtmlInputElement`, and `HtmlCanvasElement`, which give you access to methods specific to that subtype.

For each binding you want to use in your code, you need to include the binding name as a feature in the `dependencies` section of your project's `Cargo.toml`:

```
web-sys = { version = "0.3.83", features = [
    "Document",
    "Element",
    "Node",
    "Window",
] }
```

The easiest way to do this is to run the following command in your project's directory:

```
cargo add web-sys --features Document,Element,Node,Window
```



TECHNICAL
STUFF

DOM INHERITANCE WITH Deref COERCION

Even though a particular method is defined on a particular binding, it doesn't mean that the method is available only with that binding. `web-sys` defines a kind of inheritance that enables you to call certain methods on objects lower in the hierarchy. This inheritance is modeled using Rust's `Deref` trait and `deref coercion`, which I discuss in Book 2, [Chapter 5](#). (As a quick refresher, `deref coercion` is where the compiler automatically converts a reference to one type into a reference to another type. The conversion kicks in when a type implements the `Deref` trait.)

For example, the `Element` struct implements the `Deref` trait with `Target = Node`. What that means is that a reference to `Element` can automatically coerce into a reference to `Node`, which gives that element access to `Node` methods. Consider the following statements:

```
let p = document.create_element("p").ok_or("Couldn't create p")?;
p.set_text_content(Some("Hello"));
```

The `p` variable holds `Element`, but you can scour the `web-sys` docs from stem to stern and you won't find the `set_text_content()` method implemented for the `Element` struct. The preceding code works thanks to *automatic dereferencing*: When you call a method on `Element` that doesn't exist on `Element` itself, Rust automatically looks for it on `Node` (since that's the `DeRef` target of `Element`). Rust finds the `set_text_content()` method

there, breathes a sigh of relief, and calls the method. This happens at compile time, so there's no runtime cost.

Downcasting elements

DOM elements inherit methods from bindings higher in the hierarchy (for the technical details on this, refer to the sidebar “[DOM inheritance with deref coercion](#)”). But what if you need to go in the other direction? That is, what if you need to use a method that’s defined on only a binding that’s *lower* in the hierarchy?

For example, the following code returns the element with `id="username"`:

```
let input = document
  .get_element_by_id("username")
  .ok_or("No element with id 'username'");
```

Assuming the `get_element_by_id()` method is successful, the `Ok()` variant contains the element with `id="username"` as `Element`. If you then want to set the focus on this element, you can’t because the `Element` struct doesn’t implement a `focus()` method. That method *is* implemented on `HtmlElement`, however, so what you must do here is *downcast* `Element` to an `HtmlElement`, and then call the now-available `focus()` method:

```
// Get the Element
let input = document
  .get_element_by_id("username")
  .ok_or("No element with id 'username'");

// Downcast it to HtmlElement
let input = input
  .dyn_into::<HtmlElement>()
  .map_err(|_| "Element with id 'username' is not an HtmlElement");

// Now you can set the focus
input
  .focus()
  .map_err(|_| "Could not set focus");
```



REMEMBER Although in this example I downcast one level (from `Element` to `HtmlElement`), it's fine to downcast multiple levels in one fell swoop. For example, if I wanted to work with methods specific to the `input` element, I could have used `dyn_into::<HtmlInputElement>()`.

Working with DOM

Controlling DOM from Rust almost always follows this pattern:

1. Get a binding to the browser `Window`.
2. From the `Window` binding, get `Document`.
3. From `Document`, look up an element by ID or selector.
4. (Optional) Downcast that element to the specific type you need, such as `HtmlElement`.
5. Call methods as needed on the window, document, or element.

I think an example is in order. Here's the HTML (bk05ch03/update_status/ in this book's example files):

```
<h1>Update Status from Rust</h1>
<div>Current status:</div>
<div id="status">Loading...</div>

<script type="module">
  import init from "../pkg/update_status.js";
  init();
</script>
```

Note, in particular, two things:

- » The HTML includes a `div` element with `id="status"`. This is the HTML element we want to update from Rust.
- » The JavaScript imports and calls only the WASM `init()` function. The code doesn't call a specific Rust function because, as you learn

shortly, I configured `wasm-bindgen` with a default startup function.

Okay, here's the Rust code from `src/lib.rs`:

```
use wasm_bindgen::prelude::*;

// Define the function that follows as the default startup function
#[wasm_bindgen(start)]
pub fn run() -> Result<(), JsValue> {
    // 1. Get the Window
    let window = web_sys::window().ok_or("No global 'window' found"?;

    // 2. Get the Document
    let document = window.document().ok_or("No document on window"?;

    // 3. Get the Element
    let status = document
        .get_element_by_id("status")
        .ok_or("No element with id 'status'");

    // 4. Set the element's text content
    status.set_text_content(Some("Rust says \"Hello, DOM!\" via
WebAssembly!"));

    Ok(())
}
```

Note that the `run()` function is annotated with `#[wasm_bindgen(start)]`. That extra `(start)` designates `run()` as the default startup function, which means it runs automatically when the WASM initialization happens. The rest of the code just runs through the procedure I outlined earlier, using `ok_or()` to handle any errors that might crop up (unlikely, sure, but being a Rust developer means anticipating the unlikely).

After getting a binding to the `document` object, the code uses the `get_element_by_id()` method to get a binding to `status` div. Finally, the `set_text_content()` method sets the text content of the element.

[Figure 3-1](#) shows the page after initialization.

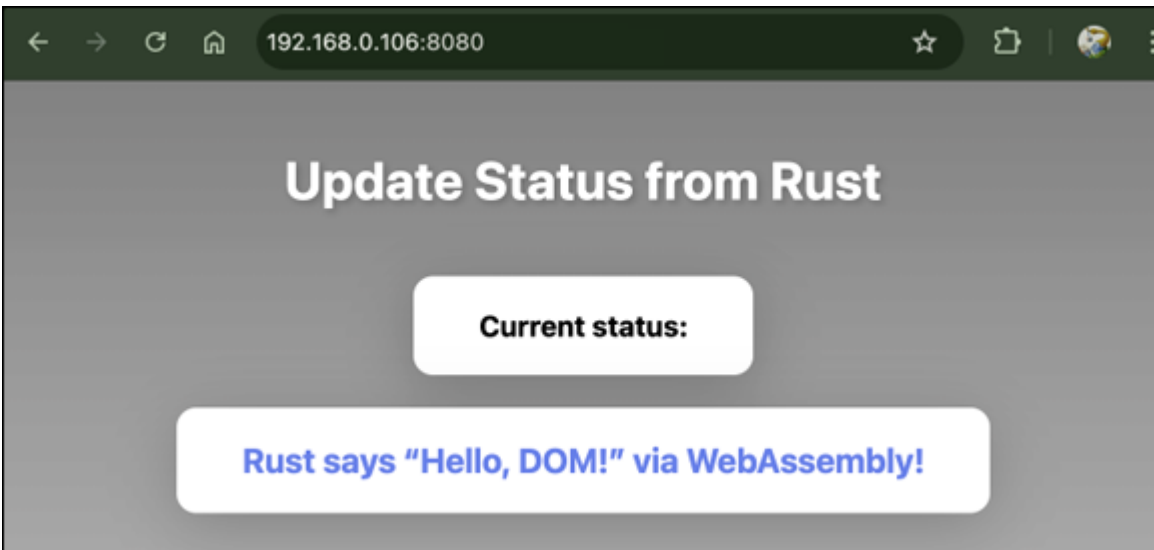


FIGURE 3-1: The status `div` updated from Rust.

Note that any errors in the Rust code would propagate up to the JavaScript caller and be thrown as a JavaScript exception when the WebAssembly module is initialized. To handle the possibility of an error, you can wrap `init()` in a `try/catch`:

```
try {  
    await init();  
} catch (e) {  
    console.error("WASM initialization failed:", e);  
}
```

Referencing DOM elements

In the example in the preceding section, I used the `Document` binding's `get_element_by_id("status")` method to get the `DOM Element` with `id="status"`, but that's not the only way to get `Element`. Two other `Document` methods enable you to use CSS selectors to get an element or a list of elements:

» **`document.querySelector(selector)`**: Returns the first `Element` in `document` that matches `selector`. You could get an `Element` binding for the `div` with `id=status` element as follows:

```
let status = document.querySelector("#status");
```

» **`document.query_selector_all(selector)`**: Returns a `NodeList` binding that contains every `Element` in `document` that matches `selector`. (For this binding to work, you need to add `NodeList` to the list of `web-sys` bindings in your project's `Cargo.toml`.) You could create a `NodeList` binding for all the `div` elements as follows:

```
let node_list = document.query_selector_all("div");
```

`NodeList` is an array-like object that has a `length` and has indexed access via `get()`:

```
// Get the number of nodes
let count = node_list.length();

// Access individual nodes by index
for i in 0..count {
    if let Some(node) = node_list.get(i) {
        // Cast to HTMLElement to use element methods
        if let Ok(element) = node.dyn_into::<HTMLElement>() {
            element.set_inner_text("Modified!");
        }
    }
}
```

Handling Browser Events

DOM manipulation is super-useful, but an interactive application needs to respond to user actions. Clicks, keypresses, and mouse movements are the lifeblood of web applications. Yep, they all happen in the browser, but you can still handle them in your Rust code.

The event listener pattern

In JavaScript, you respond to user actions by attaching event listeners to elements. An element's *event listener* is a function that JavaScript calls whenever a specific type of event occurs on that element. Click a button? The button's `click` listener fires. Press a key while focused on an input? The input's `keydown` listener fires.

In Rust, you respond to user actions also via event listeners, but setting them up requires a bit of ceremony. Fortunately, the pattern you use is essentially the same for all listener types, so once you understand the pattern for one type of listener, you're pretty much set for all the rest.

Handling click events

I begin with the most common event: the humble click. For the `click_handler` example, navigate to your main Rust projects directory and run the following commands:

```
cargo new --lib click_handler
cd click_handler
cargo add wasm-bindgen
cargo add web-sys --features Window,Document,Element,console
```

Here's body of `index.html`:

```
<body>
  <h1>Handling Clicks from Rust</h1>
  <div id="output">Open the console to see the messages.</div>
  <button id="click-me" class="btn">Click Me!</button>

  <script type="module">
    import init from "../pkg/click_handler.js";
    init();
  </script>
</body>
```

Here's `src/lib.rs`:

```
use wasm_bindgen::prelude::*;
use web_sys::{Element, console};

// Run this function upon initialization
#[wasm_bindgen(start)]
pub fn run() -> Result<(), JsValue> {
  // Get access to the DOM
  let window = web_sys::window().ok_or("no global window"?);
  let document = window.document().ok_or("no document"?);

  // Get the button element
  let button = document
    .get_element_by_id("click-me")
    .ok_or("button not found"?);
```

```

    // Set up a listener on the button
    set_up_click_listener(&button)?;

    Ok(())
}

pub fn set_up_click_listener(button: &Element) -> Result<(), JsValue> {
    // 1. Wrap the Rust event handler code in a Closure
    let handler = Closure::wrap(Box::new(move || {
        console::log_1(&"Button was clicked!".into());
    }) as Box<dyn FnMut()>);

    // 2. Attach the event listener
    button.add_event_listener_with_callback(
        "click",
        handler.as_ref().unchecked_ref()?);

    // 3. Tell Rust to forget about this memory
    handler.forget();

    Ok(())
}

```

The `run()` function gets DOM access, gets a binding on the `button` element, and then sends a borrowed reference for that binding to the `set_up_click_listener()` function. This function is where all the action is, so to speak, so a careful run-through of what’s happening is in order. This function

» **Creates a closure wrapper:** It wraps a Rust closure in

`Closure::wrap()` to make it callable from JavaScript. `Box::new` allocates the closure on the heap for a stable memory address. (Refer to Book 4, [Chapter 2](#) to learn more about the `Box` smart pointer.) Two other chores are handled here:

- *Defining the click handler:* The inner closure (`move || { ... }`) logs “Button was clicked!” to the browser console using `console::log_1()`.
- *Setting the type annotation:* `Box<dyn FnMut()>` tells the compiler that this is a boxed trait object that can be called multiple times with no arguments.

» **Attaches the event listener:** Calls

`add_event_listener_with_callback()` (a method implemented on the `EventTarget` struct) on the button element. This method gets two arguments:

- `click`: The event type to listen for
- `handler.as_ref().unchecked_ref()`: Converts the Rust closure to a JavaScript function reference

» **Prevents memory deallocation:** The `handler.forget()` statement intentionally leaks the closure's memory. Without this, Rust would drop the closure when the function ends, causing the event listener to break or crash. The trade-off is that the leaked memory is never reclaimed, which is fine for a small number of long-lived event handlers but could become a problem in a larger application that creates and destroys many handlers over time. (A more robust approach is to store closures in the application state so that Rust manages their lifetimes. I show you how to implement this approach in the “[Managing Application State](#)” section, later in this chapter.)

I know: That's a lot of code for what seems like a relatively simple idea. Happily, this process is identical for every type of event, so there's nothing more to learn. Well, actually, I tell a lie: You need to make a slight modification to `Closure` to enable your code to extract data from the handled event, which I tell you how to do in the next section.

Getting event information

Running code in the inner closure is a perfectly cromulent way to handle an event. However, it's often the case that your handler needs to interrogate the event in some way. For example, when handling a `click` or `pointermove` event, your code might need to know the current pointer position. Similarly, when handling a `keydown` event, your code might want to extract which key was pressed.

Following is an example for the `pointermove` event. To get started, navigate to your main Rust projects folder, and then run the following commands:

```
cargo new --lib pointermove_handler
cd pointermove_handler
cargo add wasm-bindgen
cargo add web-sys --features Window,PointerEvent,console
```

The HTML just initializes WASM (bk05ch03/pointermove_handler/):

```
<script type="module">
  import init from "./pkg/pointermove_handler.js";
  init();
</script>
```

Here's the Rust code from `src/lib.rs`:

```
use wasm_bindgen::prelude::*;
use web_sys::PointerEvent;

#[wasm_bindgen(start)]
pub fn run() -> Result<(), JsValue> {
    let window = web_sys::window().ok_or("no global window")?;

    let handler = Closure::wrap(Box::new(move |event: PointerEvent| {
        let x = event.client_x();
        let y = event.client_y();
        web_sys::console::log_1(&format!("Pointer at coordinates: ({} , {})",
x, y).into());
    }) as Box<dyn FnMut(PointerEvent)>);

    window.add_event_listener_with_callback("pointermove",
handler.as_ref().unchecked_ref())?;

    handler.forget();

    Ok(())
}
```

There are two main differences in this code's `Closure`:

- » The inner closure now has an argument named `event`, which is a `PointerEvent` type.
- » The boxed trait object now has a `PointerEvent` argument.

With these changes, your code can now use `event` to return data about the current event. In this example, the code runs the `client_x()` and

`client_y()` methods, which return the pointer's current (x, y) position in the window, where the top-left corner is (0, 0).



TIP

Sometimes you want to stop the browser from doing its normal thing when an event occurs. For example, you might want to prevent a form from submitting (so you can process the form data yourself) or prevent the spacebar from scrolling the page in a game. To stop the browser from handling an event normally, include the following at the top of your event closure:

```
event.preventDefault();
```

Handling keyboard events

Mercifully, handling keyboard events is nearly identical to handling pointer events. Following is an example that shows you how to get info about which key the user pressed via the `keydown` event. To get going, navigate to your main Rust projects folder, and then run the following commands:

```
cargo new --lib keydown_handler
cd keydown_handler
cargo add wasm-bindgen
cargo add web-sys --features Window,Document,KeyboardEvent,console
```

The HTML just launches the WebAssembly code (bk05ch03/keydown_handler/):

```
<script type="module">
  import init from "../pkg/keydown_handler.js";
  init();
</script>
```

Here's the WebAssembly code from `src/lib.rs`:

```
use wasm_bindgen::prelude::*;
use web_sys::KeyboardEvent;

#[wasm_bindgen(start)]
pub fn run() -> Result<(), JsValue> {
    let window = web_sys::window().ok_or("no global window")?;
```



```

let document = window.document().ok_or("no document")?;

let handler = Closure::wrap(Box::new(move |event: KeyboardEvent| {
    let key = event.key();

    match key.as_str() {
        "ArrowUp" | "w" | "W" => {
            web_sys::console::log_1(&"Moving up!".into());
        }
        "ArrowDown" | "s" | "S" => {
            web_sys::console::log_1(&"Moving down!".into());
        }
        "ArrowLeft" | "a" | "A" => {
            web_sys::console::log_1(&"Moving left!".into());
        }
        "ArrowRight" | "d" | "D" => {
            web_sys::console::log_1(&"Moving right!".into());
        }
        " " => {
            web_sys::console::log_1(&"Spacebar pressed!".into());
        }
        _ => {}
    }
}) as Box<dyn FnMut(KeyboardEvent)>);

document.add_event_listener_with_callback("keydown",
handler.as_ref().unchecked_ref())?;

handler.forget();

Ok(())
}

```

In this example, the closure argument `event` is `KeyboardEvent`, and the handler gets the pressed key by running the `event.key()` method. The closure uses a `match` pattern on `key.as_str()` to check for various keys, printing a message to the console with each match.

Managing Application State

If you're building a game or an app, you need to remember things: the score, the player's name, the contents of a shopping cart, or whatever. These kinds of data represent the application's *state*. In a normal Rust program, you'd just have a variable in `main()`. But in WASM, there's no

long-running `main()`. WASM functions run and return. So where do you keep the data?

You could solve this problem in several ways, but the Rust-iest method is to use shared ownership with interior mutability. If that sounds vaguely familiar, you might be remembering way back to Book 3, [Chapter 6](#), when I describe `Arc` smart pointers and `Arc<Mutex<T>>` for sharing mutable data safely across threads. The single-threaded version of `Arc` is `Rc` (reference counting) and the single-threaded version of `Arc<Mutex<T>>` is `Rc<RefCell<T>>`:

- » **`Rc<T>`**: A smart pointer to the original data that lets multiple owners share the same data. It uses an internal counter that tracks how many `Rcs` exist for a single memory address.
- » **`RefCell<T>`**: Provides interior mutability, meaning the data can be mutated at runtime, even when the `Rc` itself isn't mutable, and Rust enforces borrowing rules dynamically instead of at compile time.

Put together, `Rc<RefCell<T>>` is the standard pattern for a shared, mutable state where compile-time borrowing would otherwise get in the way. Time for an example (`bk05ch03/application_state`).

Navigate to your main Rust projects directory and run the following commands:

```
cargo new --lib application_state
cd application_state
cargo add wasm-bindgen
cargo add web-sys --features Document,Element,Event,HtmlElement,
[CA]HtmlButtonElement,Window
```

Here's the relevant HTML from `index.html`:

```
<body>
  <h1>Application State</h1>

  <div id="display">Count: 0 (step = 1)</div>

  <div class="button-container">
    <button id="increment_btn"> + Increment</button>
    <button id="step_btn"> 1 Increase Step</button>
```

```

</div>

<script type="module">
  import init from "../pkg/application_state.js";
  await init();
</script>
</body>

```

And here's the Rust code from `src/lib.rs`:

```

use std::cell::RefCell;
use std::rc::Rc;
use wasmbindgen::closure::Closure;
use wasmbindgen::prelude::*;
use web_sys::{Event, HtmlButtonElement, HtmlElement};

// Here's the application state
#[derive(Debug)]
struct AppState {
    count: i32,
    step: i32,
}

#[wasm_bindgen(start)]
pub fn start() -> Result<(), JsValue> {
    // Get access to the DOM
    let window = web_sys::window().ok_or("no global window")?;
    let document = window.document().ok_or("no document")?;

    // Get the Increment button and downcast it to HtmlButtonElement
    let increment_button = document
        .get_element_by_id("increment_btn")
        .ok_or("no 'increment_btn' button")?
        .dyn_into::<HtmlButtonElement>()?;

    // Get the Step button and downcast it to HtmlButtonElement
    let step_button = document
        .get_element_by_id("step_btn")
        .ok_or("no 'step_btn' button")?
        .dyn_into::<HtmlButtonElement>()?;

    // Get the Display element and downcast it to HtmlElement
    let display = document
        .get_element_by_id("display")
        .ok_or("no 'display' element")?
        .dyn_into::<HtmlElement>()?;

    // Initialize the application state using Rc<RefCell<AppState>>

```

```

let state = Rc::new(RefCell::new(AppState { count: 0, step: 1 }));

update_display(&display, &state.borrow());

// Click the Increment button: add 'step' to 'count'.
{
    // Get a smart pointer to the state
    let state = Rc::clone(&state);
    let display = display.clone();

    let handler = Closure::wrap(Box::new(move |_event: Event| {
        // Borrow the state mutably
        let mut s = state.borrow_mut();

        // Update the count property
        s.count += s.step;
        update_display(&display, &s);
    }) as Box<dyn FnMut(_)>);

    increment_button
        .add_event_listener_with_callback(
            "click",
            handler.as_ref().unchecked_ref()?);
    handler.forget();
}

// Click the Step button: increase the increment size.
{
    // Get a smart pointer to the state
    let state = Rc::clone(&state);
    let display = display.clone();

    let handler = Closure::wrap(Box::new(move |_event: Event| {
        // Borrow the state mutably
        let mut s = state.borrow_mut();

        // Update the count property
        s.step += 1;
        update_display(&display, &s);
    }) as Box<dyn FnMut(_)>);

    step_button.add_event_listener_with_callback(
        "click",
        handler.as_ref().unchecked_ref()?);
    handler.forget();
}

Ok(())

```

```

}

// Use the application state to update the display
fn update_display(display: &HtmlElement, state: &AppState) {
    display.set_inner_text(&format!("Count: {} (step = {})",
        state.count, state.step));
}

```

The application state here is stored in the `AppState` struct, which has just two properties: `count` and `step`. The `start()` function begins by getting references to various DOM elements, and then it initializes the application state (bound to the `state` variable) as an

`Rc<RefCell<AppState>>>:`

```

let state = Rc::new(RefCell::new(AppState { count: 0, step: 1 }));

```

Then two code blocks follow, the first to handle clicks on the Increment button and the second to handle clicks on the Step button. In each block, the code gets a smart pointer to the application state, and then sets up the event handler `Closure`. Within the inner closure, the application state is borrowed mutably (bound to the `s` variable) and the `count` and `step` properties, respectively, are updated.

[Figure 3-2](#) shows the page.

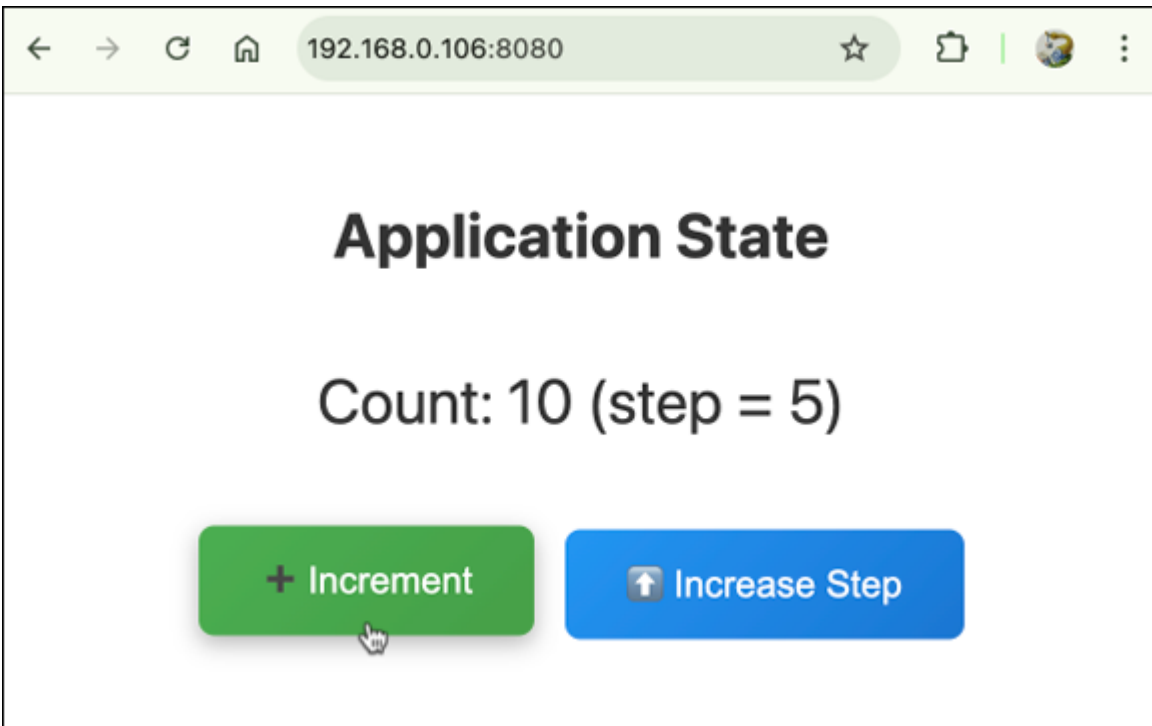


FIGURE 3-2: A web page with its application state managed by Rust.

Project: Populating DOM from Rust

If you have data in your Rust project directory, a handy technique to know is how to use that data to populate the web page. That's what the following project is all about. To begin, make a beeline to your main Rust projects directory and run the following commands:

```
cargo new --lib portfolio
cd portfolio
cargo add wasm-bindgen
cargo add serde --features derive
cargo add serde_json
cargo add web-sys --features Document,Element,Node,Window
```

Here's some JSON data from a file named `portfolio.json` (bk05ch03/portfolio/):

```
{
  "cards": [
    {
      "id": 1,
      "image": "images/portfolio01.png",
```

```

        "title": "I Fell In Love with ChatGPT",
        "description": "I designed this book for Moughton-Hifflin's new line
[CA]of AI-themed romance books."
    },
    {
        "id": 2,
        "image": "images/portfolio02.png",
        "title": "The Shakespeare Time Machine",
        "description": "Maggie O'Farrell meets H.G. Wells in this historical
[CA]fiction book I designed for Juxtaposition Publishing."
    },
    {
        "id": 3,
        "image": "images/portfolio03.png",
        "title": "Our New ChatBot Overlords",
        "description": "A dystopian AI future awaits us all in this science
[CA]fiction title I designed for H. J. Simpson Press."
    },
    {
        "id": 4,
        "image": "images/portfolio04.png",
        "title": "Yo, Yossarian!",
        "description": "A biography of the character John Yossarian from
[CA]Joseph Heller's novel Catch-22; designed for Hyperbole Editions."
    },
    {
        "id": 5,
        "image": "images/portfolio05.png",
        "title": "The Mystery of the Missing Sock",
        "description": "A perennial mystery gets solved in this whodunit, which
[CA]I designed for The Butler Did It Publications."
    },
    {
        "id": 6,
        "image": "images/portfolio06.png",
        "title": "The Old Person's Guide to TikTok",
        "description": "I designed the cover of this TikTok how-to guide for
Old
[CA]People Are People, Too Press."
    }
}
]

```

The goal is to use this data to populate a portfolio web page. Here's the body of the HTML (index.html):

```

<body>
  <header>

```

```

        
        <h1>Cover Me Book Design</h1>
    </header>

    <main>
        <h2 id="main-heading">Loading...</h2>
        <div id="portfolio" class="portfolio">
        </div>
    </main>

    <script type="module">
        import init from './pkg/portfolio.js';
        init();
    </script>

    <footer>
        <p>© Cover Me Book Design</p>
    </footer>
</body>

```

Here's the Rust code (src/lib.rs):

```

use serde::Deserialize;
use wasm_bindgen::prelude::*;

#[derive(Deserialize)]
struct PortfolioData {
    cards: Vec<Card>,
}

#[derive(Deserialize)]
struct Card {
    image: String,
    title: String,
    description: String,
}

// Get the JSON file contents
const PORTFOLIO_JSON: &str = include_str!("portfolio.json");

// Launch at initialization
#[wasm_bindgen(start)]
pub fn render_portfolio() -> Result<(), JsValue> {

    // Get access to the DOM
    let window = web_sys::window()
        .ok_or_else(|| JsValue::from_str("no global window exists"))?;
    let document = window.document()

```



```

        .ok_or_else(|| JsValue::from_str("should have document on window"))?;

// Parse the JSON data
let data: PortfolioData = serde_json::from_str(PORTFOLIO_JSON)
    .map_err(|e| JsValue::from_str(&format!("Failed to parse
portfolio.json: {}", e)))?;

// Get the portfolio container
let portfolio = document
    .get_element_by_id("portfolio")
    .ok_or_else(|| JsValue::from_str("should have portfolio element"))?;

// Create portfolio cards from the data
for card in data.cards {
    // Create card div
    let card_div = document.create_element("div")?;
    card_div.set_class_name("card");

    // Create card image
    let img = document.create_element("img")?;
    img.set_attribute("src", &card.image)?;
    img.set_attribute("alt", &format!("{}", book cover", card.title));

    // Create card heading
    let h2 = document.create_element("h2")?;
    h2.set_text_content(Some(&card.title));

    // Create card paragraph
    let p = document.create_element("p")?;
    p.set_text_content(Some(&card.description));

    // Append the elements to the card
    card_div.append_child(&img)?;
    card_div.append_child(&h2)?;
    card_div.append_child(&p)?;

    // Append the card to the portfolio
    portfolio.append_child(&card_div)?;
}

// Update the heading from "Loading..." to "My Latest Book Designs"
let heading = document
    .get_element_by_id("main-heading")
    .ok_or_else(|| JsValue::from_str("should have main-heading
element"))?;
    heading.set_text_content(Some("My Latest Book Designs"));

```

```
Ok ( () )  
}
```

This code

- » Defines the `PortfolioData` and `Card` structs, which are used for deserializing the JSON portfolio data.
- » Embeds the JSON data at compile time by using the `include_str!` macro to include the contents of `portfolio.json`. You need to do this because WebAssembly has no access to the filesystem, so you have to bake the JSON into the compiled WASM binary.
- » Initializes on WASM load via the `#[wasm_bindgen(start)]` attribute, running the `render_portfolio()` function automatically when the module loads.
- » Accesses DOM by getting the browser's `window` and `document` objects.
- » Parses the JSON data into the Rust structs using `serde_json`.
- » Generates HTML dynamically by creating `div`, `img`, `h2`, and `p` elements for each card.
- » Populates the portfolio by appending all the generated card elements to the portfolio container.
- » Updates the page heading from “Loading...” to “My Latest Book Designs.”
- » Handles errors gracefully using `Result<(), JsValue>` and the `?` operator, propagating errors to JavaScript instead of panicking.

[Figure 3-3](#) shows the resulting page.

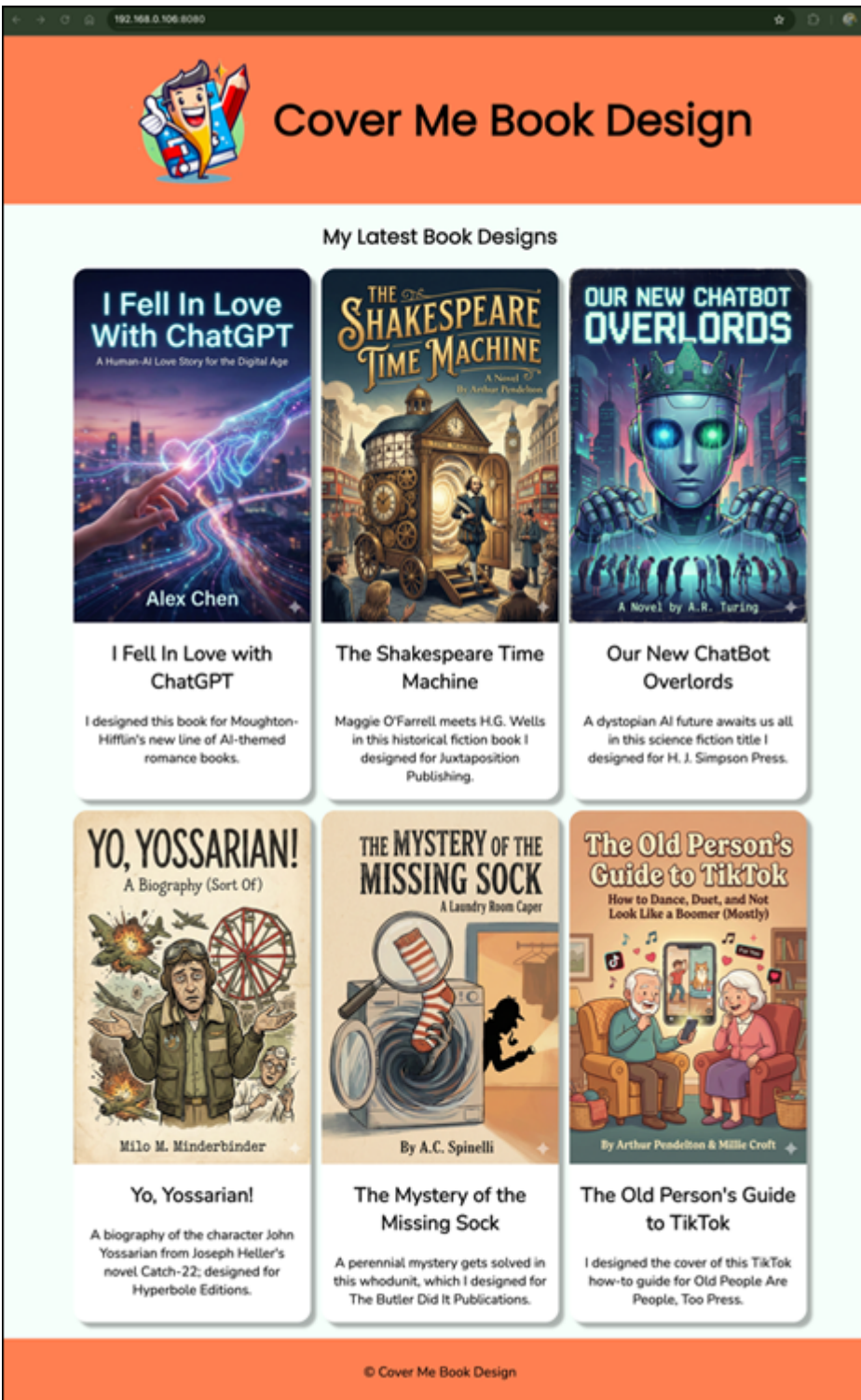


FIGURE 3-3: A web page with DOM (mostly) populated by Rust.

Book 6

Networking with Rust

Contents at a Glance

Chapter 1: Building a Web Server

[Understanding HTTP from the Ground Up](#)

[Getting a Minimal Web Server Off the Ground](#)

[Implementing Routing and Handler Functions](#)

[Organizing Your Web Server Code](#)

[Serving Static Files](#)

[Logging and Debugging](#)

Chapter 2: Building a REST API

[What's an API?](#)

[Okay, So What's a REST API?](#)

[From HTML to JSON: Changing Your Mindset](#)

[Getting Started on the API](#)

[POST, PUT, DELETE: The REST of the HTTP Verbs](#)

[Building a Resource: The Book Library API](#)

[Implementing CRUD Operations](#)

[Validating Data and Handling Errors](#)

[Migrating to a Real Database with SQLx](#)

[Testing Your API](#)

[Deployment Considerations](#)

Chapter 3: Building a Server-Side Web App

[Setting Up Your Web App Foundation](#)

[Templates: HTML with Superpowers](#)

[Serving Static Files \(CSS and Friends\)](#)

[Connecting to the Database](#)

[Authentication: Who Are You?](#)

[Managing Bookmarks](#)

[Extending the App](#)

Chapter 1

Building a Web Server

IN THIS CHAPTER

- » Discovering why Rust is the new heavyweight champion of web servers
 - » Building your very own web server
 - » Serving HTML, CSS, and images like a pro
-

Programming languages are a type of technical infrastructure: specifically, they are tools. Rust is a tool for building other infrastructure: network protocols, web servers, ... etc., etc.

—GRAYDON HOARE

A web server seems like one of those unfathomably big, hideously complex, impossibly abstruse pillars of internet infrastructure. So, you might have read the title of this chapter and scoffed. How is it even possible to build a web server, much less do it in a single chapter?

Prepare to be amazed! In this chapter, you really do learn how to create a web server using Axum, one of Rust's modern web frameworks. You discover a bit about how HTTP works under the hood, how to route requests to different handlers, how to extract data from those requests, and how to send back responses. By the end of the chapter, you'll have built a complete web server that serves a simple website with multiple pages, handles different types of requests, and serves static files such as CSS and JavaScript.

This chapter sets the foundation for the next two chapters. In Book 6, [Chapter 2](#), you build a REST API that handles JSON data. In Book 6, [Chapter 3](#), you create a server-side web app with templates and forms.

But first, you need to understand the fundamentals of how web servers work in Rust.

Understanding HTTP from the Ground Up

Before you write a single line of web server code, it helps to have a bit of background about how web servers and web browsers talk to each other. Don't worry, though: I won't take you deep into the weeds of network protocols and architecture. What lies ahead is a mercifully brief lesson on Hypertext Transfer Protocol — HTTP, for short — the language spoken by web servers. If you don't understand that language, you'll have a hard time building servers that speak it correctly.

What is HTTP, anyway?

Hypertext Transfer Protocol is a gloriously unhelpful name if you're trying to parse what it does. HTTP is just a set of rules for how computers talk to each other over the internet, specifically when one computer (a client) wants to get information from or send information to another computer (a server).

HTTP defines how the client makes a request to the server and how the server responds to that request. This request-response cycle is the essence of HTTP. The client always initiates. The server only responds. The server never randomly sends you data without you asking for it first. (Newer technologies such as WebSockets change this, but that's beyond the scope of this chapter.)

Amazingly, all this back-and-forth is text-based. HTTP messages are human-readable text, not binary gobbledygook. You can read and understand what's being sent back and forth, which makes debugging much easier than if everything were encoded in binary.

HTTP uses methods to describe intent. When you make a request, you specify not just what resource you want, but what you want to do with it. Want to read a blog post? Use the `GET` method. Want to submit a form?

Use the `POST` method. There are other methods, too, which you learn about in Book 6, [Chapter 2](#).

The anatomy of an HTTP request

When your browser wants to retrieve a web page, it sends an HTTP request to the server. That request contains several pieces of information, all formatted in a specific way that servers understand.

Here's an example of what an HTTP request looks like when you visit

<http://example.com/about:>

```
GET /about HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
Accept: text/html,application/xhtml+xml
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
```

The *request line* is the first line: `GET /about HTTP/1.1`. This tells the server three things:

- » The method is `GET`, which tells the server that you want to retrieve something.
- » The path is `/about`, which tells the server what resource you want.
- » The HTTP version is `1.1`, which is the protocol version. (Most of today's web uses HTTP/2 or HTTP/3, but the concepts you're learning here still apply.)

The *request headers* are all the lines after the request line. Each header is a key-value pair that provides metadata about the request:

- » **Host:** `example.com`: Tells the server which website you're trying to reach. (One server could host multiple websites.)
- » **User-Agent:** Identifies your browser and operating system.
- » **Accept:** Tells the server what types of content your browser can handle.
- » **Accept-Language:** Indicates which languages you prefer.

» **Connection: keep-alive:** Tells the server to keep the underlying TCP (Transmission Control Protocol) connection open for additional requests, thereby avoiding the overhead of establishing a new connection each time.

There's also a body, which comes after a blank line, but GET requests typically don't have a body. POST requests do, which you see later when you send JSON (Book 6, [Chapter 2](#)) and submit form data (Book 6, [Chapter 3](#)).

The path is particularly important for web servers. When a request comes in for `/about`, your server needs to figure out what to do with that request. Should it return an HTML page? Should it query a database? Should it return a 404 error because that page doesn't exist? Handling requests is what routing is all about, and you'll be doing a lot of that in this chapter.

The anatomy of an HTTP response

When the server receives your request, it processes it and sends back an HTTP response. In the same way that requests have a specific structure, so do responses. Here's what a typical HTTP response looks like:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 147
Cache-Control: max-age=3600
Date: Sat, 06 Dec 2025 14:30:00 GMT

<!DOCTYPE html>
<html>
<head>
  <title>About Us</title>
</head>
<body>
  <h1>About Our Company</h1>
  <p>We make amazing things...</p>
</body>
</html>
```

The *status line* is the first line: `HTTP/1.1 200 OK`. This tells you

- » The HTTP version the server is using.
- » The *status code*, which is a three-digit number indicating what happened. Status codes are grouped into five categories based on their first digit:
 - **1xx (informational):** The server received your request and is processing it. 1xx status codes are rarely seen these days.
 - **2xx (success):** Everything worked! The most common success code is 200 OK.
 - **3xx (redirection):** The resource you want is somewhere else. For example, 301 Moved Permanently means the resource has moved to the address returned by the server.
 - **4xx (client error):** Something went wrong with the request. The famous 404 Not Found means the path you requested doesn't exist. 403 Forbidden means you don't have permission. 400 Bad Request means your request was malformed.
 - **5xx (server error):** Something went wrong on the server. 500 Internal Server Error means something crashed on the server. 503 Service Unavailable means the server is overloaded or down for maintenance.
- » The *reason phrase*, which is a human-readable description of the status code.

The *response headers* come next, just like in the request. Here are some important response headers you should know:

- » **Content-Type:** Tells the browser what kind of data is in the response body (HTML, JSON, an image, and so on)
- » **Content-Length:** Tells the browser how many bytes to expect
- » **Cache-Control:** Tells the browser how long it can keep this response cached
- » **Date:** Specifies when the response was generated

The body comes after a blank line. This is the content: the HTML page, the JSON data, the image bytes, or whatever the client requested.

Understanding this structure is crucial because when you're writing web server code in Rust, you construct these responses. The Axum crate you'll be using makes it easy, but you need to know what's being sent over the wire.

What you're building in this chapter

Now that you understand HTTP basics, it's time to talk about what you're going to build as you work through this chapter. You'll create a web server that

- » Serves multiple HTML pages (a home page, an about page, and a contact page)
- » Handles different routes (URLs) and maps them to the right handler functions
- » Extracts data from URLs, such as path parameters and query strings
- » Serves static files (CSS and JavaScript) so your pages can look nice and be interactive
- » Logs requests so you can see what's happening

This isn't a toy example. It's a functional web server that serves a real website. You could deploy this server to the internet and people could visit it. (You won't deploy it in this chapter, but you could.)

The server will be organized with separate modules for different concerns. By the time you're done, you'll understand not just how to make Axum work, but why web servers are structured the way they are.

Let's get started.

Getting a Minimal Web Server Off the Ground

Rust has several excellent web frameworks to choose from, including Actix-web and Rocket. They all have their strengths, but for this chapter's project you're going to use a web framework called Axum. It's a modern, actively developed framework built on top of `tokio`, the async runtime I discuss in Book 3, [Chapter 6](#).

Starting the project

Navigate to your main Rust projects folder and then fire off the following commands:

```
cargo new web_server
cd web_server
cargo add axum
cargo add tokio --features full
cargo add tower-http --features fs,trace
cargo add tracing
cargo add tracing-subscriber
cargo add serde --features derive
```

For the dependencies, `axum` is the web framework; `tokio` is the async runtime; `tower-http` is used for serving static files (the `fs` feature) and for request logging (the `trace` feature); and `tracing` and `tracing-subscriber` provide structured logging, which means that when requests come in, you see them in your terminal. I also added the `serde` crate, which you use later to handle query strings.

Building the cutest little web server ever

Let's get right down to it by creating a minimal Axum server. Replace the contents of `src/main.rs` with the following code:

```
use axum::{
    routing::get,
    Router,
};

#[tokio::main]
async fn main() {
    let app = Router::new()
        .route("/", get(homepage));

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
```

```

        .expect("Failed to bind to 127.0.0.1:3000");

println!("Server running on http://127.0.0.1:3000");

axum::serve(listener, app)
    .await
    .expect("Failed to start server");
}

async fn homepage() -> &'static str {
    "Welcome to my website!"
}

```

First, marvel at the fact that you can build a web server in a mere 19 lines of code! Second, check out what this code is doing:

- » The imports bring in the pieces you need from Axum. `routing::get` is a function that creates a route handler for GET requests. `Router` is the type that holds all your routes.
- » The `#[tokio::main]` attribute transforms your main function into an async function that runs on the Tokio runtime. This is necessary because Axum is built on async, and you can't use async code without a runtime.
- » Creating the router happens with `Router::new()`. The router is where you define all your *routes*, the mappings between URL paths and the functions that handle them. In this code, `.route("/", get(homepage))` adds one route: When a GET request comes in for the path `/`, call the `homepage()` function.
- » `tokio::net::TcpListener::bind()` tells your server where to listen for incoming requests. `127.0.0.1` is `localhost` (your own computer), and `3000` is the port number. `await` is necessary because binding might take a moment, and `expect()` crashes if binding fails (which might happen if another program is already using port 3000).
- » Starting the server happens with `axum::serve()`. This function takes the listener and your app and starts serving requests. It runs forever, handling requests as they come in. The `await.expect()` means “wait

for the server to finish” (which it never does, unless something goes wrong) and panic if there’s an error.

- » The `homepage()` function is your first handler. It’s marked `async` because Axum handlers need to be asynchronous. A web server must be able to handle a bunch of requests at a time without blocking, and `async` makes that possible. The function returns `&'static str`, which is a string slice that lives for the entire program. Axum is smart enough to convert this into a proper HTTP response with a 200 status code and the right content type.

Testing your server

Okay, the server certainly qualifies as minimal, but does it work? To find out, scoot over to your terminal and run

```
cargo run
```

After a moment or three (compilation takes a while the first time), you should see

```
Server running on http://127.0.0.1:3000
```

Sweet. Now open a handy web browser and send it to <http://127.0.0.1:3000> (or <http://localhost:3000>). [Figure 1-1](#) shows the result. It’s alive!

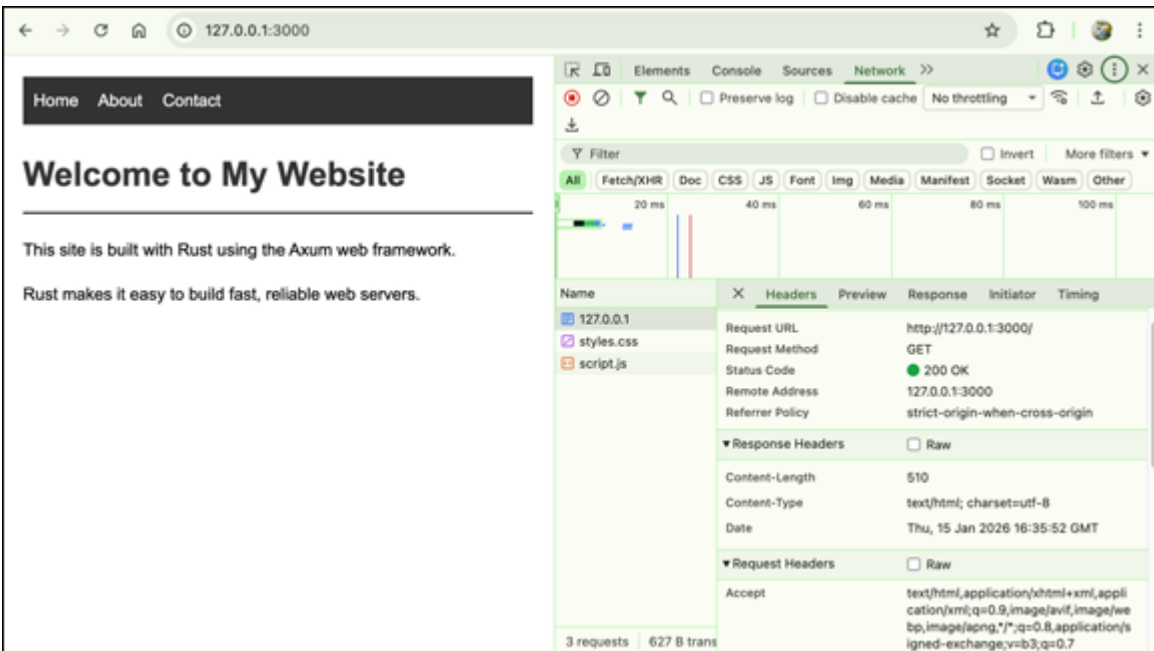


FIGURE 1-1: Your minimal web server is up and serving.

In [Figure 1-1](#), note as well that I opened my browser’s developer tools (press F12 in most browsers), displayed the Network tab, reloaded the page, and then clicked the 127.0.0.1 request to display the request and response headers.

You can also access your new server via the command line using `curl`:

```
curl http://127.0.0.1:3000
```

or

```
curl http://localhost:3000
```

You should see

```
Welcome to my website!
```

What’s happening here? When you visit the URL in your browser or use `curl`, your client (browser or `curl`) sends an HTTP GET request to your server. The request looks something like:

```
GET / HTTP/1.1
Host: localhost:3000
```

Your server receives this request. Axum parses it, looks at the router (app) to find a matching route, finds one, and uses that route to call your

`homepage()` function. That function returns the string `Welcome to my website!` Axum converts that to an HTTP response:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 22

Welcome to my website!
```

Note that the `Content-Type` header says `text/plain`, and the body is just a plain text string with no HTML. The browser or `curl` receives this response and displays the body. Later in this chapter (the next section, in fact), you upgrade your handlers to return proper HTML.

In your browser, try visiting <http://localhost:3000/about>. You'll see an error page that says 404 Not Found. That's because you haven't defined a route for `/about` yet. Axum automatically returns a 404 response when no route matches.

Stop your server by pressing `Ctrl+C` in the terminal where it's running.

You've built your first server. Now let's make it useful.

Implementing Routing and Handler Functions

As downright cool as it is to have your very own, honest-to-goodness server running, let's get real: A web server with just one route isn't much of a web server! Real websites have multiple pages: a homepage, an about page, a contact page, and so on. Each page needs its own route, and each route needs a handler function to generate the response.

How routing works in Axum

Routing is the process of matching an incoming request to the code that should handle it. The router looks at two things: the HTTP method (GET, POST, and so on) and the path (`/`, `/about`, `/contact`, and so on). Based on these two pieces of information, the router calls the appropriate handler function.

In Axum, you build up a router by chaining `route()` calls. Each call adds a new route to the router. Here's what it looks like to add multiple routes:

```
let app = Router::new()
    .route("/", get(homepage))
    .route("/about", get(about_page))
    .route("/contact", get(contact_page));
```

Writing handler functions

Handler functions are where your application logic lives. When a request comes in and a route matches, Axum calls the corresponding handler function. That function processes the request and returns a response.

Handler functions in Axum are flexible. They can return many different types, as long as those types implement a trait called `IntoResponse`. This trait knows how to convert a value into an HTTP response.

Here are some types that implement `IntoResponse`:

- » **String and &str:** Returns plain text with a 200 status.
- » **Html<String>:** Returns HTML with the right content type. (`Html` is an Axum type; you learn about it in the next section.)
- » **Json<T>:** Returns JSON. (You use this in Book 6, [Chapter 2](#).)
- » **Tuples of the form (StatusCode, String):** Returns a custom status code plus a body.
- » **Result<T, E> where both T and E implement IntoResponse:** For error handling.

You added two routes in the preceding section, so now it's time to write handler functions for those routes. Add the following to `main.rs`, just after the `homepage()` function:

```
async fn about_page() -> &'static str {
    "This is the about page. We build things with Rust!"
}

async fn contact_page() -> &'static str {
```

```
"Contact us at hello@example.com"  
}
```

Run this with `cargo run` and visit <http://localhost:3000>, <http://localhost:3000/about> and <http://localhost:3000/contact> in your browser. Each route returns different text.

This works, but plain text is boring. It's time to learn how to return HTML instead.

Returning HTML responses

To return HTML, you need to import Axum's `Html` type and wrap your HTML string in it. First, update your web server's imports:

```
use axum::{  
    response::Html,  
    routing::get,  
    Router,  
};
```

Now update your handlers to return HTML. Here's the updated `homepage()` handler:

```
async fn homepage() -> Html<&'static str> {  
    Html(r#"  
        <!DOCTYPE html>  
        <html>  
        <head>  
            <title>My Rust Website</title>  
        </head>  
        <body>  
            <h1>Welcome!</h1>  
            <p>This website is built with Rust and Axum.</p>  
            <nav>  
                <a href="/">Home</a> |  
                <a href="/about">About</a> |  
                <a href="/contact">Contact</a>  
            </nav>  
        </body>  
        </html>  
    "#)  
}
```

The `Html` type is a wrapper around a string. It tells Axum that what follows is HTML, so it needs to set the `Content-Type` header to

text/html. The `r#"..."#` syntax is a raw string literal, which lets you include quotes and newlines without escaping them.

Run your server and visit the homepage. Now you see a proper HTML page with a heading, some text, and navigation links. Click the links to navigate between pages. Nice!

Reading HTML from files

Hardcoding HTML in your Rust source code is not ideal. It makes your code hard to read and is annoying to edit. In a real project, you keep your HTML in separate files.

Let's create an HTML file for each page. First, create a directory called `pages` in your project root (the same level as `src`):

```
mkdir pages
```

Now create three HTML files in the `pages` directory: `index.html`, `about.html`, and `contact.html`. Populate each file with the HTML code you want to use. For `index.html`, just copy the code from the raw string you used in the preceding section. Here's my code for `about.html` (refer to the GitHub repo for the `contact.html` code):

```
<!DOCTYPE html>
<html>
<head>
  <title>About - My Rust Website</title>
</head>
<body>
  <nav>
    <a href="/">Home</a> |
    <a href="/about">About</a> |
    <a href="/contact">Contact</a>
  </nav>
  <h1>About This Site</h1>
  <p>We're passionate about building web applications with Rust.</p>
  <p>Rust's performance and safety make it perfect for web development.</p>
</body>
</html>
```

Next, you update your handler functions to read these files. To avoid blocking, you use `tokio::fs::read_to_string()`, which is Tokio's

async file reader. First, be sure to import `tokio::fs` for your filesystem work:

```
use tokio::fs;
```

Now you can update your handlers to read the text in each HTML file:

```
async fn homepage() -> Html<String> {
    let html = fs::read_to_string("pages/index.html")
        .await
        .unwrap_or_else(|_| String::from("<h1>Error loading page</h1>"));
    Html(html)
}

async fn about_page() -> Html<String> {
    let html = fs::read_to_string("pages/about.html")
        .await
        .unwrap_or_else(|_| String::from("<h1>Error loading page</h1>"));
    Html(html)
}

async fn contact_page() -> Html<String> {
    let html = fs::read_to_string("pages/contact.html")
        .await
        .unwrap_or_else(|_| String::from("<h1>Error loading page</h1>"));
    Html(html)
}
```

Note that the return type changed from `Html<&'static str>` to `Html<String>`. That's because you're reading the file into `String` at runtime, not using a string literal that exists for the entire program.

Also note the error handling: If reading the file fails (maybe the file doesn't exist or you don't have permission to read it), `unwrap_or_else()` provides a fallback error message instead of panicking. This is better than plain old `unwrap()`, which would crash your entire server if a file was missing.

Restart your server and test it. Now your HTML lives in separate files, making it much easier to maintain. You can edit the HTML files without touching your Rust code.

Everything works, but did you notice something about the route handlers? That's right: There's quite a bit of duplication there because

each handler does basically the same thing. You can fix that by creating a helper function:

```
fn load_page(filename: &str) -> Html<String> {
    let path = format!("pages/{}", filename);
    let html = fs::read_to_string(&path)
        .await
        .unwrap_or_else(|_| format!("<h1>Error loading {}</h1>", filename));
    Html(html)
}

async fn homepage() -> Html<String> {
    load_page("index.html").await
}

async fn about_page() -> Html<String> {
    load_page("about.html").await
}

async fn contact_page() -> Html<String> {
    load_page("contact.html").await
}
```

Ah, that's better. Now each handler is a one-liner that just specifies which file to load.

Reading URL query strings

Consider the following URL:

```
http://localhost/search?q=rust&page=2
```

The *query string* is the part of the URL after the `?` character, and it contains one or more key-value pairs separated by `&`. In the preceding URL, the query string is `q=rust&page=2` and has two parameters: `q` and `page`. This path is a search route, so in this example, the `q` value is the search term, and the `page` value is the page number.

To extract query parameters, use Axum's `Query` extractor. First, import it:

```
use axum::{
    extract::{Query},
    response::Html,
    routing::get,
```

```
Router,
};
use serde::Deserialize;
```

Note, as well, that I'm also importing `Deserialize` from our old friend, the `serde` crate, which was added as a dependency at the start of the project. You might have thought that reading the query string's key-value pairs into a hash map was the way to go here. You could definitely do that, but you can ramp up your app's type safety by deserializing the query string into a struct:

```
#[derive(Deserialize)]
struct SearchParams {
    q: String,
    #[serde(default = "default_page")]
    page: u32,
}

fn default_page() -> u32 {
    1
}

async fn search(Query(params): Query<SearchParams>) -> Html<String> {
    let html = format!(r#"
        <!DOCTYPE html>
        <html>
        <head>
            <title>Search Results</title>
        </head>
        <body>
            <h1>Search Results</h1>
            <p>Searching for: <strong>{}</strong></p>
            <p>Page: {}</p>
            <p>(In a real application, you'd query a database here.)</p>
        </body>
        </html>
    "#, params.q, params.page);

    Html(html)
}
```

You also need to add the following to your router:

```
.route("/search", get(search))
```

The `SearchParams` struct defines what parameters you expect: `q` and `page`, in this case. You can configure parameters to be required or to have default values by using the `#[serde(default)]` attribute:

- » **No attribute:** The parameter is required. In the `SearchParams` struct, the `q` parameter has no attribute, so it's required. If the URL doesn't have a `q` parameter, Axum returns a 400 error automatically.
- » `#[serde(default)]`: The parameter is assigned the default value of its type if the parameter is not part of the query string. This assignment works only if the type implements the `Default` trait. The `page` parameter is `u32`, which does implement `Default`, but the default is `0`, which doesn't work here, so this attribute isn't used in the `SearchParams` struct.
- » `#[serde(default = "function")]`: The parameter is assigned the default value returned by the specified *function* if the parameter is not part of the query string. In the `SearchParams` struct, the `page` parameter has this attribute, and it tells `serde` to call the `default_page()` function, which returns the value `1`.

Now you can access the parameters with `params.q` and `params.page`, and you get type checking. If someone visits `/search?page=notanumber`, Axum returns a 400 (Bad Request) error because `notanumber` can't be parsed as `u32`.

Organizing Your Web Server Code

At this point, your `main.rs` file is getting crowded. You have imports, the main function, several handler functions, helper functions, and a struct. Having everything in `main.rs` is no big deal for a small example, but real web servers have dozens or hundreds of routes. Before things get out of hand, now is a good time to organize your code into modules.

You learned about Rust's module system in Book 2, [Chapter 7](#). Now you'll apply those lessons to organize a web server. The goal is to keep

`main.rs` focused on setup and configuration, while moving the actual handler logic into separate modules.

The most natural split is to move your handler functions into a separate module. Create a new file named `src/handlers.rs`:

```
use axum::{
    extract::{Query},
    response::Html,
};

use std::fs;
use serde::Deserialize;

fn load_page(filename: &str) -> Html<String> {
    let path = format!("pages/{}", filename);
    let html = fs::read_to_string(&path)
        .unwrap_or_else(|_| format!("<h1>Error loading {}</h1>", filename));
    Html(html)
}

pub async fn homepage() -> Html<String> {
    load_page("index.html")
}

pub async fn about_page() -> Html<String> {
    load_page("about.html")
}

pub async fn contact_page() -> Html<String> {
    load_page("contact.html")
}

#[derive(Deserialize)]
pub struct SearchParams {
    q: String,
    #[serde(default = "default_page")]
    page: u32,
}

fn default_page() -> u32 {
    1
}

pub async fn search(Query(params): Query<SearchParams>) -> Html<String> {
    let html = format!(r#"
        <!DOCTYPE html>
```



```

        <html>
        <head>
            <title>Search Results</title>
            <link rel="stylesheet" href="/static/css/styles.css">
        </head>
        <body>
            <nav>
                <a href="/">Home</a>
                <a href="/about">About</a>
                <a href="/contact">Contact</a>
            </nav>
            <h1>Search Results</h1>
            <p>Searching for: <strong>{}</strong></p>
            <p>Page: {}</p>
            <p>(In a real application, you'd query a database here.)</p>
            <script src="/static/js/script.js"></script>
        </body>
        </html>
        "#, params.q, params.page);

    Html(html)
}

```

Note that the `SearchParams` struct and most of the functions are marked `pub`. That's because they need to be accessible from `main.rs`. The `load_page()` and `default_page()` functions are not `pub` because they're only used within this module.

With that done, you're left with a much cleaner `main.rs`:

```

mod handlers;

use axum::{
    routing::get,
    Router,
};

#[tokio::main]
async fn main() {
    let app = Router::new()
        .route("/", get(handlers::homepage))
        .route("/about", get(handlers::about_page))
        .route("/contact", get(handlers::contact_page))
        .route("/search", get(handlers::search));

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
}

```

```
.expect("Failed to bind to 127.0.0.1:3000");

println!("Server running on http://127.0.0.1:3000");

axum::serve(listener, app)
    .await
    .expect("Failed to start server");
}
```

The `mod handlers` line tells Rust to look for a `handlers.rs` file. Then, in your router, you reference the handlers with `handlers::homepage`, `handlers::about_page`, and so on.

Run your server with `cargo run` and verify that everything still works. It should, since you have changed just the organization, not the functionality.

As your server grows, you might want to split things even further. You learn how to structure a modern web app in Book 6, [Chapter 3](#).

Serving Static Files

So far, you've been serving HTML pages, but real websites also need CSS stylesheets, JavaScript files, images, fonts, and other static assets. These files don't change based on who's requesting them or what parameters are in the request. They're just files on disk that need to be sent to the browser.

You could write handlers for each static file, but that would be tedious and slow. Instead, you'll use Axum's static file-serving capabilities, which are provided by the `tower-http` crate.

First, create directories for your static files:

```
mkdir static
mkdir static/css
mkdir static/js
```

Now import the `tower-http` crate:

```
use tower_http::services::ServeDir;
```

Then update your router:

```
let app = Router::new()
    .route("/", get(homepage))
    .route("/about", get(about_page))
    .route("/contact", get(contact_page))
    .route("/search", get(search))
    .nest_service("/static", ServeDir::new("static"));
```

The key line is `nest_service("/static", ServeDir::new("static"))`. This line tells Axum that whenever it comes across a request that starts with `/static`, serve that file from the `static` directory on disk.

Create your static files — for example, `static/css/styles.css` and `static/js/script.js` — and then update your HTML to point to them:

```
<!DOCTYPE html>
<html>
<head>
    <title>Home - My Rust Website</title>
    <link rel="stylesheet" href="/static/css/styles.css">
</head>
<body>
    <nav>
        <a href="/">Home</a>
        <a href="/about">About</a>
        <a href="/contact">Contact</a>
    </nav>
    <h1>Welcome to My Website</h1>
    <p>This site is built with Rust using the Axum web framework.</p>
    <p>Rust makes it easy to build fast, reliable web servers.</p>

    <script src="/static/js/script.js"></script>
</body>
</html>
```

Update `pages/about.html`, `pages/contact.html`, and the search code similarly, adding the same `<link>` and `<script>` tags.

Run your server and visit <http://localhost:3000>. Your page should now have styling from the CSS file. Open your browser's developer tools (press F12), click the Console tab, and you should see the messages from the JavaScript file available on this book's GitHub repo.

Logging and Debugging

When you're developing a web server, you need to see what's happening. Which requests are coming in? What routes are they matching? Are there any errors? Logging helps you answer these questions.

Axum integrates with the `tracing` crate, which provides structured logging. You've already added `tracing` and `tracing-subscriber` to your dependencies, so let's set up logging on your site.

In `main.rs`, add the following imports:

```
use tower_http::trace::TraceLayer;
use tracing_subscriber;
```

Update the `main()` function:

```
#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::DEBUG)
        .init();

    let app = Router::new()
        .route("/", get(handlers::homepage))
        .route("/about", get(handlers::about_page))
        .route("/contact", get(handlers::contact_page))
        .route("/search", get(handlers::search))
        .nest_service("/static", ServeDir::new("static"))
        .layer(TraceLayer::new_for_http());

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
        .expect("Failed to bind to 127.0.0.1:3000");

    println!("Server running on http://127.0.0.1:3000");

    axum::serve(listener, app)
        .await
        .expect("Failed to start server");
}
```

There are two new things to note here:

- » `tracing_subscriber::fmt`: Sets up the logging system. This tells `tracing` to output log messages to your terminal in a readable format.
- » `layer(TraceLayer::new_for_http())`: Adds request logging middleware to your app. Middleware is code that runs for every request before it reaches your handlers. `TraceLayer` logs details about each request and response.

Run your server and make a few requests. In your terminal, you'll see output such as the following:

```
2025-12-07T15:55:26.574559Z DEBUG request{method=GET uri=/about version=[CA]HTTP/1.1}: tower_http::trace::on_request: started processing request
2025-12-07T15:55:26.575637Z DEBUG request{method=GET uri=/about version=[CA]HTTP/1.1}: tower_http::trace::on_response: finished processing request [CA]latency=1 ms status=200
```

Chapter 2

Building a REST API

IN THIS CHAPTER

- » Decoding acronym soup: API, REST, JSON, and CRUD
 - » Understanding what makes an API RESTful
 - » Persisting data with SQLx
 - » Testing and validating your API
-

This is the essence of the systems Rust is best for writing: not flashy, not attention-grabbing, often entirely unnoticed. Just the robust and reliable necessities that enable us to get our work done, to attend to other things, confident that the system will keep humming along unattended.

—GRAYDON HOARE

In Book 6, [Chapter 1](#), I show you how to build a web server that serves nicely formatted web pages with text, headings, and links. That HTML-festooned data is great for humans, but what about programs? What if another application needs to get data from your server, or send data to it, without all the HTML trimmings?

This is where REST APIs come in. Instead of serving HTML pages meant for humans, a REST API serves data in a format that programs can easily consume. In this chapter, you build a complete REST API for managing a book library. You handle JSON data, implement all the major HTTP verbs, manage shared state across multiple requests, validate incoming data, handle errors gracefully, persist data to a database, and test your API to make sure it works. By the end of the chapter, you'll have built a functional API that can power a mobile app, drive a web front end, or integrate with other services.

What's an API?

Before diving into REST specifically, I want to clarify what an API is. API stands for *application programming interface*, which is one of those terms that sounds more complicated than it really is. An API is essentially just a way for programs to talk to each other.

Think of it like this: When you go to a restaurant, you don't walk into the kitchen and start making your own food. Instead, you interact with a waiter by choosing from the menu. You don't need to know the recipes or how the kitchen is organized; you just need to know what's on the menu and how to order. An API works the same way. It defines a set of operations that a server makes available to the outside world. The client doesn't need to know the messy details of how the server works internally. It just needs to know what operations are available and how to call them.

The apps on your phone and computer interact with APIs all the time without realizing it. When a weather app on your phone shows you the forecast, it's using an API to request weather data from a server. When you share a post from one social media app to another, APIs are making that connection happen. When a website shows you a map, it's probably using the Google Maps API to fetch and display that map data. APIs are the connective tissue of modern software. They let different programs and services work together even though they were built by different people, use different technologies, and might even be running on different continents.

Okay, So What's a REST API?

REST stands for *representational state transfer*, which is possibly the most unhelpful acronym expansion in all of web development. Don't worry about memorizing what REST stands for. Instead, just know that REST is a particular architectural style for building these APIs. A REST API is designed for application-to-application communication: Your server provides the data, and the client — whether it's a mobile app,

another server, or a JavaScript front end — decides what to do with it. A REST API uses standard HTTP, the same protocol your web browser uses and that I discuss in Book 6, [Chapter 1](#). REST APIs are so popular because they use the same infrastructure as the web itself.

How REST APIs differ from HTML web servers

The web server I show you how to build in Book 6, [Chapter 1](#) is designed for humans. When someone visits `/about`, they get back a complete HTML page with headings, paragraphs, navigation links, and styling. The browser renders all that HTML into a nice, comprehensible page that humans can read and interact with.

A REST API is designed for programs. When a program (which might be a mobile app, another web service, or a JavaScript frontend) makes a request to your API, the program gets back just the data it asked for, typically formatted as JSON. There's no HTML, no CSS, no JavaScript. Just pure data.

For example, the Book 6, [Chapter 1](#) web server probably returned something like this for the `/about` route:

```
<!DOCTYPE html>
<html>
<head><title>About</title> </head>
<body>
  <h1>About This Site</h1>
  <p>We build things with Rust.</p>
</body>
</html>
```

A REST API serving the same information would return something like this:

```
{
  "title": "About This Site",
  "description": "We build things with Rust."
}
```

The REST API response contains the same information, but programs receiving this data can do whatever they want with it: display it in a mobile app, store it in a database, or combine it with other data.

Why JSON instead of HTML

JSON has become the de facto standard format for REST APIs. It's not the only option: you could use XML, YAML, or even plain text, but JSON has won out for good reasons.

First, JSON is programming-language agnostic. Even though it has *JavaScript* in its name, every major programming language can easily parse and generate JSON. Whether you're writing a mobile app in Swift, a back end service in Python, or a web front end in TypeScript, you can work with JSON effortlessly.

Second, JSON is lightweight. It's just text, and it's more compact than XML. For example, here's a typical XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <title>Rust All-in-One For Dummies</title>
  <author>Paul McFedries</author>
  <isbn>9781394401901</isbn>
  <published_year>2026</published_year>
  <read>true</read>
</book>
```

Here's the equivalent JSON:

```
{
  "title": "Rust All-in-One For Dummies",
  "author": "Paul McFedries",
  "isbn": "9781394401901",
  "published_year": 2026,
  "read": true
}
```

I think you'll agree that the JSON version is more concise and easier to read.

Third, JSON maps naturally to data structures in most programming languages, such as structs in Rust and objects in JavaScript. And JSON uses strings, numbers, and Booleans, so when you receive JSON data, you can immediately work with it using native data types.

Rust has excellent JSON support through the `serde` crate, which you've already used in previous chapters. Serde can serialize (convert Rust

types to JSON) and deserialize (convert JSON to Rust types) with minimal boilerplate.

From HTML to JSON: Changing Your Mindset

Building a REST API requires a shift in thinking from building the HTML web server in Book 6, [Chapter 1](#). Instead of thinking about pages and navigation, you think about resources and operations. Instead of returning formatted HTML, you return structured data.

A good place to begin is to explore how Axum (the web framework that I introduce in the preceding chapter) handles JSON, both for receiving data in requests and sending data in responses. Once you understand the mechanics of JSON handling, you'll be ready to build a complete API.

Axum makes working with JSON blessedly straightforward. You use the `Json<T>` type for both receiving and sending JSON data. When `Json<T>` appears as a function parameter, it deserializes incoming JSON into a Rust type. When `Json<T>` appears as a return type, it serializes a Rust type into JSON.

Getting Started on the API

Since, as they say, a good example is the best teacher, let's just dive right in, shall we? In this chapter, I show you how to build an API for managing a collection of books. To get the API ball rolling, head over to a handy terminal and enter the following commands:

```
cargo new book_api
cd book_api
cargo add axum
cargo add tokio --features full
cargo add serde --features derive
cargo add serde_json
```

For the dependencies, `axum` is the web framework, `tokio` is the async runtime, `serde` handles all serialization and deserialization, and

`serde_json` performs the JSON magic.

In `main.rs`, replace the boilerplate with the following code to create a minimal API that accepts and returns JSON:

```
use axum::{Json, Router, routing::post};
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize)]
struct Greeting {
    name: String,
    message: String,
}

// Send back a greeting, JSON-style
async fn greet(Json(greeting): Json<Greeting>) -> Json<Greeting> {
    let response = Greeting {
        name: greeting.name.clone(),
        message: format!("Hello, {}! {}", greeting.name, greeting.message),
    };
    Json(response)
}

#[tokio::main]
async fn main() {
    // Set up the router
    let app = Router::new().route("/greet", post(greet));

    // Start listening for incoming requests
    let listener = tokio::net::TcpListener::bind("127.0.0.1:3001")
        .await
        .expect("Failed to bind");

    println!("API server running on http://127.0.0.1:3001");

    // Launch the API server
    axum::serve(listener, app)
        .await
        .expect("Failed to start server");
}
```

The `Greeting` struct represents the data structure you're working with. It derives both `Deserialize` and `Serialize` from `serde`, which means `serde` can convert JSON into a `Greeting` (deserialize) and convert a `Greeting` into JSON (serialize).

The `greet()` function takes `Json(greeting): Json<Greeting>` as a parameter. This is Axum's JSON extractor in action. When a request comes in with a JSON body, Axum automatically deserializes that JSON into a `Greeting` struct and passes it to your function. If the JSON is invalid or doesn't match the `Greeting` structure, Axum returns a 400 `Bad Request` error automatically.

The function returns `Json<Greeting>`, which tells Axum to serialize the `Greeting` struct into JSON and send it back as the response body. Axum automatically sets the `Content-Type` header to `application/json`.

To test your API, you need a tool that can send JSON in HTTP requests. In Book 6, [Chapter 1](#), I showed how to use a web browser to test your HTML server, but browsers aren't ideal for testing APIs. Instead, it's better to use `curl`, a command-line tool for making HTTP requests.

Run your server with `cargo run`. You should see the following in the terminal window:

```
API server running on http://127.0.0.1:3001
```

Then, in another terminal window, send a JSON request:

```
curl -X POST http://localhost:3001/greet \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "message": "Nice to meet you!"}'
```

The `-X POST` flag tells `curl` to use the POST method (refer to the next section to learn about POST), the `-H "Content-Type: application/json"` header tells the server you're sending JSON, and the `-d` flag provides the JSON data you're sending.

You should get back

```
{"name": "Alice", "message": "Hello, Alice! Nice to meet you!"}
```

What happens with incomplete or invalid JSON? Try this:

```
curl -X POST http://localhost:3001/greet \
  -H "Content-Type: application/json" \
  -d '{"name": "Bob"}'
```

Here's the result:

```
Failed to deserialize the JSON body into the target type:  
missing field 'message' at line 1 column 15
```

You got an error because the `message` field is missing. Axum automatically validates that the JSON matches your struct. Nice.

POST, PUT, DELETE: The REST of the HTTP Verbs

In Book 6, [Chapter 1](#), you use only `GET`. REST APIs need more: `POST` to create resources, `PUT` to update them, and `DELETE` to remove them. (You might also stumble upon the `PATCH` verb, which is used for partial updates, such as changing just a few fields rather than replacing an entire resource.) These four methods (`GET`, `POST`, `PUT`, `DELETE`) map to CRUD operations: Create, Read, Update, and Delete, respectively:

- » **GET** (reading resources): The client requests a resource by its URL. For example, to retrieve a specific book, you'd send a `GET` request to `/books/5`, or to retrieve all books, you'd `GET /books`. `GET` requests never modify data on the server, they just read it. `GET` is *idempotent*, meaning that making the same `GET` request multiple times always returns the same resource (assuming nothing else has changed it).
- » **POST** (creating resources): The client sends data for a new resource. For example, to create a book, you'd `POST JSON` with the book's title, author, and ISBN to `/books`. `POST` requests are not idempotent, so making the same `POST` twice creates two resources.
- » **PUT** (updating resources): The client sends the complete updated version of a resource, typically with the resource ID in the URL path (like `/books/5`). `PUT` is idempotent: sending the same `PUT` multiple times results in the same final state.
- » **DELETE** (removing resources): The client specifies which resource to delete via the URL (like `/books/5`). `DELETE` is also idempotent:

deleting something that's already deleted results in the same state (the resource doesn't exist).

Building a Resource: The Book Library API

Throughout the rest of this chapter, you build a book library API that stores information about books and supports all CRUD operations. You'll start with in-memory storage, and then migrate to a real database.

Defining the Book struct

Create a new file `src/models.rs`:

```
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Book {
    pub id: u32,
    pub title: String,
    pub author: String,
    pub isbn: String,
    pub published_year: u32,
    pub category: String,
    pub read: bool,
    pub rating: Option<u8>,
}

#[derive(Debug, Deserialize)]
pub struct CreateBook {
    pub title: String,
    pub author: String,
    pub isbn: String,
    pub published_year: u32,
    pub category: String,
}

#[derive(Debug, Deserialize)]
pub struct UpdateBook {
    pub title: Option<String>,
    pub author: Option<String>,
    pub isbn: Option<String>,
    pub published_year: Option<u32>,
    pub category: Option<String>,
    pub read: Option<bool>,
    pub rating: Option<u8>,
}
```

There are three structs here:

- » `Book` represents a complete book with an ID assigned by your API. The `rating` field is `Option<u8>` because not every book has been rated yet.
- » `CreateBook` is used when adding new books. It doesn't have an `id` (the API assigns that), and it doesn't have `read` or `rating` (those get default values).
- » `UpdateBook` is used when updating books. All fields are `Option` because updates might change only some fields. If a field is `None`, it means "don't change this field."

Why three separate structs? It prevents errors. With `CreateBook`, clients can't accidentally send an ID. With `UpdateBook`, clients can update only the fields they want. This makes your API clearer and harder to misuse.

Setting up in-memory storage

Before implementing CRUD operations, you need somewhere to store books. You'll use a `Vec<Book>` wrapped in `Arc<Mutex<...>>` for thread-safe shared access. This is a quick refresher from Book 3, [Chapter 6](#): `Mutex` ensures that only one thread accesses data at a time, and `Arc` allows multiple owners of the same data.

Update `src/main.rs`:

```
mod models;

use axum::{
    extract::State,
    routing::{get, post},
    Router,
    Json,
};
use std::sync::{Arc, Mutex};
use models::Book;

type BookStore = Arc<Mutex<Vec<Book>>>;

#[tokio::main]
```



```

async fn main() {
    let book_store: BookStore = Arc::new(Mutex::new(Vec::new()));

    let app = Router::new()
        .route("/books", get(list_books))
        .route("/books", post(create_book))
        .with_state(book_store);

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
        .expect("Failed to bind");

    println!("Book Library API running on http://127.0.0.1:3000");

    axum::serve(listener, app)
        .await
        .expect("Failed to start server");
}

async fn list_books(State(store): State<BookStore>) -> Json<Vec<Book>> {
    let books = store.lock().unwrap();
    Json(books.clone())
}

async fn create_book(
    State(store): State<BookStore>,
    Json(new_book): Json<models::CreateBook>
) -> Json<Book> {
    let mut books = store.lock().unwrap();
    let id = books.iter().map(|b| b.id).max().unwrap_or(0) + 1;

    let book = Book {
        id,
        title: new_book.title,
        author: new_book.author,
        isbn: new_book.isbn,
        published_year: new_book.published_year,
        category: new_book.category,
        read: false,
        rating: None,
    };

    books.push(book.clone());
    Json(book)
}

```

The `BookStore` type alias makes your code more readable than writing `Arc<Mutex<Vec<Book>>>` everywhere.

In `main()`, you pass the book store to the router with `.with_state(book_store)`, making it available to all handlers. Handlers that need it add `State(store): State<BookStore>` as their first parameter.

Testing your API

Let's test what you've built so far. Run the server:

```
cargo run
```

In another terminal, create a book:

```
curl -X POST http://localhost:3000/books \
-H "Content-Type: application/json" \
-d '{
  "title": "The Rust Programming Language",
  "author": "Steve Klabnik",
  "isbn": "1718503105",
  "published_year": 2023
}'
```

You should get back the created book with ID 1. Create another book:

```
curl -X POST http://localhost:3000/books \
-H "Content-Type: application/json" \
-d '{
  "title": "Programming Rust",
  "author": "Jim Blandy",
  "isbn": "1492052590",
  "published_year": 2021
}'
```

Now list all books:

```
curl http://localhost:3000/books
```

You should see both books in the JSON array response.

Implementing CRUD Operations

Now let's add the remaining CRUD operations: getting a single book, updating, and deleting.

GET a single book with path parameters

Add this import:

```
use axum::extract::Path;
```

Add the route in `main()`:

```
.route("/books/:id", get(get_book))
```

Add the handler:

```
async fn get_book(
    State(store): State<BookStore>,
    Path(id): Path<u32>
) -> Result<Json<Book>, (axum::http::StatusCode, String)> {
    let books = store.lock().unwrap();

    match books.iter().find(|book| book.id == id) {
        Some(book) => Ok(Json(book.clone())),
        None => Err((
            axum::http::StatusCode::NOT_FOUND,
            format!("Book with id {} not found", id)
        )),
    },
}
```

The route `/books/:id` uses `:id` as a path parameter. When a request comes in for `/books/1`, Axum extracts `1` as the value of `id`.

The `Path(id): Path<u32>` parameter extracts and parses that value. If someone requests `/books/notanumber`, Axum automatically returns a 400 Bad Request error.

Test it:

```
curl http://localhost:3000/books/1
curl http://localhost:3000/books/999 # Should return 404
```

PUT to update a book

Add the import and route:

```

use axum::routing::{get, post, put};
use models::UpdateBook;

// In main():
.route("/books/:id", put(update_book))

```

Add the handler:

```

async fn update_book(
    State(store): State<BookStore>,
    Path(id): Path<u32>,
    Json(update): Json<UpdateBook>,
) -> Result<Json<Book>, (axum::http::StatusCode, String)> {
    let mut books = store.lock().unwrap();

    match books.iter_mut().find(|book| book.id == id) {
        Some(book) => {
            if let Some(title) = update.title {
                book.title = title;
            }
            if let Some(author) = update.author {
                book.author = author;
            }
            if let Some(isbn) = update.isbn {
                book.isbn = isbn;
            }
            if let Some(published_year) = update.published_year {
                book.published_year = published_year;
            }
            if let Some(category) = update.category {
                book.category = category;
            }
            if let Some(read) = update.read {
                book.read = read;
            }
            if let Some(rating) = update.rating {
                book.rating = Some(rating);
            }

            Ok(Json(book.clone()))
        }
        None => Err((
            axum::http::StatusCode::NOT_FOUND,
            format!("Book with id {} not found", id)
        )),
    }
}

```

This handler uses `.iter_mut()` to modify books in place. For each field in `UpdateBook`, if the field is `Some`, you update the field. If it's `None`, you leave it unchanged. This allows partial updates:

```
curl -X PUT http://localhost:3000/books/1 \
  -H "Content-Type: application/json" \
  -d '{"read": true, "rating": 5}'
```

This marks the book as read and sets its rating to 5, without changing other fields.

DELETE a book

Add the route:

```
use axum::routing::{get, post, put, delete};

// In main():
.route("/books/{id}", delete(delete_book))
```

Add the handler:

```
async fn delete_book(
    State(store): State<BookStore>,
    Path(id): Path<u32>,
) -> Result<axum::http::StatusCode, (axum::http::StatusCode, String)> {
    let mut books = store.lock().unwrap();

    let initial_len = books.len();
    books.retain(|book| book.id != id);

    if books.len() < initial_len {
        Ok(axum::http::StatusCode::NO_CONTENT)
    } else {
        Err((
            axum::http::StatusCode::NOT_FOUND,
            format!("Book with id {} not found", id)
        ))
    }
}
```

Test the deletion:

```
curl -X DELETE http://localhost:3000/books/1
curl http://localhost:3000/books # Verify the book is gone
```

Validating Data and Handling Errors

Your API should validate incoming data and return helpful error messages. Let's add validation to `create_book()`:

```
use axum::http::StatusCode;

async fn create_book(
    State(store): State<BookStore>,
    Json(new_book): Json<models::CreateBook>,
) -> Result<(StatusCode, Json<Book>), (StatusCode, String)> {
    // Validate title
    if new_book.title.trim().is_empty() {
        return Err((StatusCode::BAD_REQUEST, String::from("Title cannot be empty")));
    }

    // Validate author
    if new_book.author.trim().is_empty() {
        return Err((StatusCode::BAD_REQUEST, String::from("Author cannot be empty")));
    }

    // Validate ISBN (must be 10 or 13 digits)
    let isbn_digits: String = new_book.isbn.chars()
        .filter(|c| c.is_ascii_digit())
        .collect();
    if isbn_digits.len() != 10 && isbn_digits.len() != 13 {
        return Err((
            StatusCode::BAD_REQUEST,
            String::from("ISBN must contain exactly 10 or 13 digits")
        ));
    }

    // Validate publication year
    if new_book.published_year < 1000 || new_book.published_year > 2025 {
        return Err((
            StatusCode::BAD_REQUEST,
            String::from("Publication year must be between 1000 and 2025")
        ));
    }

    // Validate category
    if new_book.category.trim().is_empty() {
```

```

        return Err((
            StatusCode::BAD_REQUEST,
            String::from("Category cannot be empty"),
        ));
    }

    let mut books = store.lock().unwrap();
    let id = books.iter().map(|b| b.id).max().unwrap_or(0) + 1;

    let book = Book {
        id,
        title: new_book.title,
        author: new_book.author,
        isbn: new_book.isbn,
        published_year: new_book.published_year,
        read: false,
        rating: None,
    };

    books.push(book.clone());
    Ok((StatusCode::CREATED, Json(book)))
}

```

Note that validation runs before acquiring the mutex lock. This approach is good practice: validate first, and then acquire locks only when needed.

Test the validation:

```

# Empty title
curl -X POST http://localhost:3000/books \
  -H "Content-Type: application/json" \
  -d '{"title": "", "author": "Test", "isbn": "1234567890", "published_year": 2020}'

# Invalid ISBN
curl -X POST http://localhost:3000/books \
  -H "Content-Type: application/json" \
  -d '{"title": "Test", "author": "Test", "isbn": "12345", "published_year": 2020}'

```

Migrating to a Real Database with SQLx

In-memory storage is fine for development, but production APIs need persistence. If your server restarts, you don't want to lose all your data! Let's add that persistence by changing the in-memory storage model to a SQLite database with SQLx.

Setting up SQLx

Add SQLx to your dependencies in `Cargo.toml`:

```
[dependencies]
axum = "0.7"
tokio = { version = "1", features = ["full"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
sqlx = { version = "0.8", features = ["runtime-tokio", "sqlite"] }
```

Install the SQLx CLI:

```
cargo install sqlx-cli --no-default-features --features sqlite
```

Create your database:

```
sqlx database create --database-url sqlite:books.db
```

Create a migration:

```
sqlx migrate add create_books_table
```

Open the generated file in the `migrations` directory and add:

```
CREATE TABLE IF NOT EXISTS books (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  author TEXT NOT NULL,
  isbn TEXT NOT NULL,
  published_year INTEGER NOT NULL,
  read BOOLEAN NOT NULL DEFAULT 0,
  rating INTEGER
);
```

Run the migration:

```
sqlx migrate run --database-url sqlite:books.db
```

Updating your code for SQLx

The migration to SQLx requires a few modifications to the existing code:

Update `src/models.rs` to derive `FromRow`:

```
use sqlx::FromRow;

#[derive(Debug, Clone, Serialize, Deserialize, FromRow)]
pub struct Book {
    // ... rest stays the same
}
```

Update `src/main.rs` to use `SQLx`:

```
use sqlx::SqlitePool;

#[tokio::main]
async fn main() {
    let pool = SqlitePool::connect("sqlite:books.db")
        .await
        .expect("Failed to connect to database");

    let app = Router::new()
        .route("/books", get(list_books))
        .route("/books/:id", get(get_book))
        .route("/books", post(create_book))
        .route("/books/:id", put(update_book))
        .route("/books/:id", delete(delete_book))
        .with_state(pool);

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
        .expect("Failed to bind");

    println!("Book Library API running on http://127.0.0.1:3000");

    axum::serve(listener, app)
        .await
        .expect("Failed to start server");
}
```

Update `list_books()`:

```
async fn list_books(
    State(pool): State<SqlitePool>
) -> Result<Json<Vec<Book>>, (StatusCode, String)> {
    let books = sqlx::query_as::<_, Book>(
        "SELECT * FROM books"
    )
    .fetch_all(&pool)
    .await
```

```

        .map_err(|e| (StatusCode::INTERNAL_SERVER_ERROR, format!("Database error:
        {}", e)))?;

        Ok(Json(books))
    }

```

Update create_book():

```

async fn create_book(
    State(pool): State<SqlitePool>,
    Json(new_book): Json<models::CreateBook>,
) -> Result<(StatusCode, Json<Book>), (StatusCode, String)> {
    // Keep all the validation from before...
    if new_book.title.trim().is_empty() {
        return Err((StatusCode::BAD_REQUEST, String::from("Title cannot be
        empty")));
    }

    if new_book.author.trim().is_empty() {
        return Err((StatusCode::BAD_REQUEST, String::from("Author cannot be
        empty")));
    }

    let isbn_digits: String = new_book.isbn.chars()
        .filter(|c| c.is_ascii_digit())
        .collect();
    if isbn_digits.len() != 10 && isbn_digits.len() != 13 {
        return Err((
            StatusCode::BAD_REQUEST,
            String::from("ISBN must contain exactly 10 or 13 digits")
        ));
    }

    if new_book.published_year < 1000 || new_book.published_year > 2025 {
        return Err((
            StatusCode::BAD_REQUEST,
            String::from("Publication year must be between 1000 and 2025")
        ));
    }

    let result = sqlx::query(
        "INSERT INTO books (title, author, isbn, published_year, read,
        rating)
        VALUES (?, ?, ?, ?, ?, ?)"
    )
    .bind(&new_book.title)
    .bind(&new_book.author)
    .bind(&new_book.isbn)

```

```

        .bind(new_book.published_year)
        .bind(false)
        .bind(None::<u8>)
        .execute(&pool)
        .await
        .map_err(|e| (StatusCode::INTERNAL_SERVER_ERROR, format!("Database error:
        {}", e)))?;

        let book_id = result.last_insert_rowid() as u32;

        let book = Book {
            id: book_id,
            title: new_book.title,
            author: new_book.author,
            isbn: new_book.isbn,
            published_year: new_book.published_year,
            read: false,
            rating: None,
        };

        Ok((StatusCode::CREATED, Json(book)))
    }
}

```

Update get_book():

```

async fn get_book(
    State(pool): State<SqlitePool>,
    Path(id): Path<u32>
) -> Result<Json<Book>, (StatusCode, String)> {
    let book = sqlx::query_as::<_, Book>("SELECT * FROM books WHERE id = ?")
        .bind(id)
        .fetch_optional(&pool)
        .await
        .map_err(|e| (StatusCode::INTERNAL_SERVER_ERROR, format!("Database
        error: {}", e)))?
        .ok_or_else(|| (StatusCode::NOT_FOUND, format!("Book with id {} not
        found", id)))?;

    Ok(Json(book))
}

```

Update delete_book():

```

async fn delete_book(
    State(pool): State<SqlitePool>,
    Path(id): Path<u32>,
) -> Result<StatusCode, (StatusCode, String)> {
    let result = sqlx::query("DELETE FROM books WHERE id = ?")

```

```

        .bind(id)
        .execute(&pool)
        .await
        .map_err(|e| (StatusCode::INTERNAL_SERVER_ERROR, format!("Database
error: {}", e)))?;

    if result.rows_affected() > 0 {
        Ok(StatusCode::NO_CONTENT)
    } else {
        Err((StatusCode::NOT_FOUND, format!("Book with id {} not found",
id)))
    }
}

```

Now your data persists! Test it by adding books, stopping the server, restarting it, and verifying that the books are still there.

Testing Your API

Automated tests give you confidence that your API works correctly. Add `reqwest` to your dev dependencies in `Cargo.toml`:

```

[dev-dependencies]
reqwest = { version = "0.12", features = ["json"] }
Create tests/api_tests.rs:
use serde_json::json;

#[tokio::test]
async fn test_create_and_get_book() {
    let client = reqwest::Client::new();

    // Create a book
    let response = client
        .post("http://127.0.0.1:3000/books")
        .json(&json!({
            "title": "Test Book",
            "author": "Test Author",
            "isbn": "1234567890",
            "published_year": 2023
        }))
        .send()
        .await
        .expect("Failed to send request");

    assert_eq!(response.status(), 201);
}

```

```
let book: serde_json::Value = response.json().await.unwrap();
let book_id = book["id"].as_u64().unwrap();

// Get the book
let response = client
    .get(format!("http://127.0.0.1:3000/books/{}", book_id))
    .send()
    .await
    .expect("Failed to send request");

assert_eq!(response.status(), 200);
let retrieved: serde_json::Value = response.json().await.unwrap();
assert_eq!(retrieved["title"], "Test Book");
}
```

Make sure your server is running, and then type:

```
cargo test
```

Deployment Considerations

When you're ready to deploy your API, remember to:

- » Build a release binary: `cargo build --release`
- » Use environment variables for configuration (database URLs, ports)
- » Add HTTPS (typically with a reverse proxy like nginx)
- » Consider rate-limiting to prevent abuse
- » Set up proper logging for production

Platforms that make deploying Rust easy include Fly.io, Railway, Shuttle.rs, and the major cloud providers (AWS, Google Cloud, Azure).

You've built a complete REST API! You learned JSON handling, CRUD operations, state management, validation, database integration, and testing. In Book 6, [Chapter 3](#), you'll build a server-side web application with templates and forms.

Chapter 3

Building a Server-Side Web App

IN THIS CHAPTER

- » Creating web templates with Askama
 - » Building forms and handling user input
 - » Implementing authentication
 - » Storing users and bookmarks in a database
-

Describing Rust as a systems programming language ... is like describing Microsoft as Windows or Google as search. Yes, Rust is equipped for systems programming, but its applicability is much wider.

—TIM MCNAMARA

Here's an uncomfortable truth about modern web development: Not everything needs to be a fancy single-page application powered by React or Vue talking to an API. Sometimes you just want a good, old-fashioned web application that generates HTML on the server and sends complete pages to the browser. You know, the way the web used to work before JavaScript frameworks took over the world!

In this chapter, you go full server-side to build LinkLocker, a complete bookmark manager web application. Along the way, you learn how to use templates to generate HTML dynamically, handle form submissions, implement user authentication with sessions and password hashing, and store and retrieve data from a database. Let's get started!

Setting Up Your Web App Foundation

Every journey begins with the proverbial single step, and every web app begins with the not-even-close-to-proverbial proper configuration. In this section, you set up the basic structure for LinkLocker, install the necessary dependencies, design the database schema, and create the initial directory structure. This foundation work might seem tedious, but trust me, taking the time to set things up correctly now will save you hours of headaches later.

Creating the project structure

In Book 6, [Chapter 1](#), I introduce Axum, a web framework built on top of `tokio` that makes building HTTP servers straightforward and type-safe. You'll use Axum again as the foundation for LinkLocker, but this time you'll add several new dependencies for templating, sessions, and password hashing.

Start by creating a new project from your main Rust projects directory:

```
cargo new linklocker
```

Now you need to install the dependencies:

```
cd linklocker
cargo add askama@0.12
cargo add askama_axum
cargo add axum@0.7
cargo add bcrypt
cargo add serde --features derive
cargo add sqlx --features runtime-tokio,sqlite
cargo add tokio --features full
cargo add tower
cargo add tower-http --features fs
cargo add tower-sessions@0.12
```

Most of these dependencies should be familiar from previous chapters; `axum`, `sqlx`, `tokio`, `serde`, and `tower-http` all work the same way as in Book 6, [Chapters 1](#) and [2](#). The new ones are

» **askama and askama_axum:** A templating engine that compiles your HTML templates to Rust code at compile time, giving you type safety and blazing performance. The `askama_axum` crate integrates it with Axum so templates automatically implement `IntoResponse`.

- » **bcrypt:** Password hashing. *Never store passwords in plain text!* `bcrypt` is a one-way hashing algorithm that is intentionally slow (making brute-force attacks harder) and handles salting automatically.
- » **tower-sessions:** Session management middleware. When a user logs in, the server creates a session and gives their browser a cookie with a session ID. On subsequent requests, the browser sends that cookie, letting the server know who's logged in.

Next, create the directory structure for your templates, static files, and database migrations:

```
mkdir -p templates static/css migrations
```

The `templates` directory will hold your HTML templates, which are files with embedded Rust code for displaying dynamic content. The `static/css` directory will hold your CSS files for styling. (You could also add `static/js` for JavaScript and `static/images` for images.) The `migrations` directory will hold SQL files for creating and modifying your database tables. Each migration is a numbered SQL file that modifies the database schema. Everything has its place!

Database design for LinkLocker

LinkLocker needs two database tables: one for users and one for bookmarks. The `users` table stores user credentials (`username` and `password_hash`), and the `bookmarks` table stores the bookmark data with a foreign key linking each bookmark to its owner.

Create a file called `migrations/001_create_tables.sql` and add this SQL code:

```
-- Create the users table
CREATE TABLE IF NOT EXISTS users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  username TEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```



```
-- Create the bookmarks table
CREATE TABLE IF NOT EXISTS bookmarks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    url TEXT NOT NULL,
    title TEXT NOT NULL,
    description TEXT,
    tags TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users (id)
);
```

The schema uses standard SQL constraints: `PRIMARY KEY` `AUTOINCREMENT` gives each row a unique, auto-assigned ID. `NOT NULL` ensures that required fields can't be empty, while `UNIQUE` prevents duplicate usernames. `DEFAULT CURRENT_TIMESTAMP` automatically records creation times. The `FOREIGN KEY` constraint links each bookmark to its owning user, preventing orphaned data. Fields without `NOT NULL` (like `description` and `tags`) are optional.

I'll be storing tags as a `TEXT` field with comma-separated values (like `"rust,web,programming"`) rather than creating a separate `tags` table with a many-to-many relationship through a junction table. For this simple app, comma-separated tags are perfectly fine. Comma-separated tags are easy to work with in SQL using the `LIKE` operator (`WHERE tags LIKE '%rust%'`), and they're easy to split apart in Rust using `split(",")`.

Initializing the database

Now you need to create the database file and run the migration to create the tables. As I mention in Book 6, [Chapter 2](#), SQLx provides a command-line tool called `sqlx-cli` for managing databases. If you don't have it installed yet (or you skipped [Chapter 2](#) — how dare you!), install it now:

```
cargo install sqlx-cli --features sqlite
```

This compilation takes a few minutes because `sqlx-cli` is a substantial program. Go grab a coffee, check your email, pet your dog, or contemplate the eventual heat death of the universe. I'll wait.

Once SQLx is installed, you can use it to create the database and run migrations. First, set an environment variable so SQLx knows where to find the database:

```
export DATABASE_URL="sqlite:linklocker.db"
```

On Windows (PowerShell), use:

```
$env:DATABASE_URL="sqlite:linklocker.db"
```

Now create the database and run the migration:

```
sqlx database create  
sqlx migrate run
```

The first command creates a SQLite database file called `linklocker.db` in your project directory. The second command reads all the `.sql` files in your `migrations` directory (currently just `001_create_tables.sql`) and executes them in order. You should see output confirming that the migration ran successfully:

```
Applied 1/migrate create tables (263µs)
```

With the database set up, you're ready to start building the application. The foundation is in place and now comes the fun part: writing code!

Templates: HTML with Superpowers

Writing HTML by hand gets old fast. Sure, for a simple static page it's fine to write `<h1>Hello, World!</h1>` and call it a day, but when you need to display dynamic data — such as a list of bookmarks that changes based on who's logged in — you need something more powerful.

That something is a *templating engine*. Think of it as HTML with variables, loops, and conditional logic built in. You write garden-variety HTML, but with special syntax for the dynamic parts. The templating engine replaces the special syntax with data when someone requests the page.

Meet Askama: Rust-flavored templates

Askama is a templating engine that enables you to write HTML with embedded Rust-like syntax for loops, conditionals, and variable interpolation. You create an HTML file with special syntax for the dynamic parts, and then in your Rust code you define a struct that holds the data you want to display. Askama takes that struct and the template, generates Rust code that renders the HTML, and compiles it all together. At runtime, when someone requests the page, that compiled code runs blazingly fast to produce the final HTML.

Why compile templates instead of parsing them at runtime like many other frameworks? Two reasons: speed (no parsing overhead on each request) and safety (template errors are caught at compile time, not when a user hits a broken page in production).

Template syntax uses the following patterns:

- » **`{{ variable }}`**: Outputs the value of a variable. Askama automatically HTML-escapes the value to prevent XSS attacks unless you explicitly use the `safe` filter.
- » **`{% for item in items %}`**: Starts a loop. Everything between this and **`{% endfor %}`** repeats for each item in the collection. You can iterate over any type that implements `Iterator`.
- » **`{% if condition %}`**: Conditional rendering. The content appears only if the condition is `true`. You can use **`{% else %}`** and **`{% else if other_condition %}`** just like in Rust.
- » **`{% extends "base.html" %}`**: Inherits from another template. With this approach, you avoid repeating the same header and footer on every page. You define the common structure once in a base template, and then each page template extends it and fills in the unique parts.
- » **`{% block name %}`**: Defines a section that child templates can override. The base template provides defaults, and child templates fill in or replace those sections with page-specific content.

Creating your first template

Let's start with a base layout template that defines the overall structure of your pages. Every page on LinkLocker will share this basic structure: header with navigation, main content area, and footer. Just swap out the main content depending on which page you're viewing.

This approach is called the *template inheritance pattern*, and it's one of the most useful features of modern templating engines. Instead of copying and pasting the same header and footer code into every page template (which is a maintenance nightmare; change one thing and you have to update it everywhere), you define the common structure once in a base template and then extend it.

Create a file called `templates/base.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>{% block title %}LinkLocker{% endblock %}</title>
  <link rel="stylesheet" href="/static/css/styles.css">
</head>
<body>
  <header>
    <h1><a href="/">LinkLocker</a></h1>
    <nav>
      <a href="/">Home</a>
      {% if let Some(user) = username %}
        <a href="/bookmarks">My Bookmarks</a>
        <a href="/bookmarks/new">Add Bookmark</a>
        <span class="user-info">Welcome, {{ user }}!</span>
        <form method="post" action="/logout" class="logout-form">
          <button type="submit" class="link-button">Logout</button>
        </form>
      {% else %}
        <a href="/login">Login</a>
        <a href="/register">Register</a>
      {% endif %}
    </nav>
  </header>
  <main>
    {% block content %}{% endblock %}
  </main>
```

```

    <footer>
        <p>© LinkLocker - Your Personal Bookmark Manager</p>
    </footer>
</body>
</html>

```

The template uses `{% block title %}` to let child templates override the page title and uses `{% block content %}` for page-specific content. The `{% if let Some(user) = username %}` conditional shows different navigation links depending on login status. When `{{ user }}` displays the username, Askama automatically HTML-escapes it to prevent XSS attacks.

Now create a simple homepage template to see how inheritance works. Create `templates/home.html`:

```

{% extends "base.html" %}

{% block title %}Home - LinkLocker{% endblock %}

{% block content %}
<div class="hero">
    <h2>Never Lose a Link Again</h2>
    <p>LinkLocker helps you organize all your favorite URLs in one place.</p>
    <p>Save bookmarks with titles, descriptions, and tags. Search and filter
        to find exactly what you're looking for.</p> <!--<ce:anchor id="pp:623
np:624" role="page-break"/>-->
    {% if let Some(_user) = username %}
        <a href="/bookmarks" class="button primary">View My Bookmarks</a>
    {% else %}
        <a href="/register" class="button primary">Get Started</a>
        <a href="/login" class="button">Log In</a>
    {% endif %}
</div>
{% endblock %}

```

This template extends the base layout and fills in two blocks:

- » **title block:** Sets the page title to “Home - LinkLocker” instead of just “LinkLocker”.
- » **content block:** Provides the unique content for the home page, specifically a hero section with a welcome message and call-to-action buttons.

Note how clean this is. You don't need to repeat the HTML boilerplate, the header, the navigation, or the footer. You just say `{% extends "base.html" %}` and then provide the content that's unique to this page. Beautiful!

Rendering templates from routes

To use these templates in your Rust code, you define a struct that implements the `Template` trait from Askama. This struct holds the data that the template needs to render. Askama uses the `#[derive(Template)]` macro to generate the rendering code automatically.

Replace the `src/main.rs` boilerplate with this code:

```
use askama_axum::Template;
use axum::{Router, routing::get};
use tower_http::services::ServeDir;

#[derive(Template)]
#[template(path = "home.html")]
struct HomeTemplate {
    username: Option<String>,
}

async fn home() -> HomeTemplate {
    HomeTemplate { username: None }
}

#[tokio::main]
async fn main() {
    let app = Router::new()
        .route("/", get(home))
        .nest_service("/static", ServeDir::new("static"));

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
        .await
        .unwrap();

    println!("LinkLocker running at http://127.0.0.1:3000");
    axum::serve(listener, app).await.unwrap();
}
```

The `#[derive(Template)]` macro tells Askama to generate rendering code, and `#[template(path = "home.html")]` specifies which template file to use (from the `templates` directory). The struct's fields correspond to variables in the template: `username: Option<String>` maps to `{{ username }}` in the template. The `askama_axum` integration makes templates implement `IntoResponse`, so just return the template struct and Axum automatically converts it to an HTML response with the correct headers.

Serving Static Files (CSS and Friends)

Before going further with application logic, you need to add CSS to make LinkLocker look presentable. Nobody wants to use ugly software. While function trumps form, there's no reason you can't have both, and a little CSS goes a long way toward making your app feel professional and polished instead of like a 1995 GeoCities page.

Create `static/css/style.css` and populate it with your preferred CSS, or use the CSS that comes with this project's repo, which uses clean, modern styles.

You already added the following `ServeDir` line in `main.rs`:

```
.nest_service("/static", ServeDir::new("static"))
```

Let me explain what it does. The `nest_service` method mounts the static file server at the `"/static"` path. This means requests to `"/static/css/style.css"` will serve the file from `static/css/style.css` in your project directory. It's like saying "Hey Axum, whenever someone requests a URL starting with `/static`, just serve them the corresponding file from the static folder. Don't run any handler code; don't do any routing logic; just send the file."

Now you can run the server:

```
cargo run
```

Visit <http://127.0.0.1:3000> in your browser, and you should see your beautifully styled home page!

Connecting to the Database

Before you can start building features that save and retrieve bookmarks, you need to connect to the database and set up the application state. You'll create an `AppState` struct that holds the database connection pool and share it across all your route handlers using Axum's state management.

Setting up the database pool

Add these imports to the top of `src/main.rs`:

```
use sqlx::{FromRow, SqlitePool};
use std::sync::Arc;
```

The `SqlitePool` is a connection pool that manages multiple database connections. Instead of opening a new database connection for every request (which is slow), you create a pool of connections when the server starts, and requests borrow connections from the pool when they need database access. The `Arc` (atomic reference counter) lets you share the connections across multiple threads safely because each handler gets a clone of `Arc`, which is cheap (it just increments a counter), and they all share the same underlying pool.

Define the application state and database models:

```
#[derive(Clone)]
struct AppState {
    pool: SqlitePool,
}

#[derive(FromRow)]
struct User {
    id: i64,
    username: String,
    password_hash: String,
}

#[derive(FromRow)]
```



```
struct Bookmark {
    id: i64,
    user_id: i64,
    url: String,
    title: String,
    description: Option<String>,
    tags: Option<String>,
}
```

The `#[derive(FromRow)]` attribute tells SQLx how to convert database rows into these structs. When you run a query that returns rows, SQLx maps the columns to the struct fields by name. If the struct field names don't match the column names, you can use `#[sqlx(rename = "column_name")]` to specify the mapping.

The `Clone` derive on `AppState` is required because Axum needs to clone the state for each handler. Since `AppState` just contains a `SqlitePool`, which is designed to be cloned (cloning the pool is cheap because it's just cloning an `Arc` internally), this approach works perfectly.



REMEMBER Hey, wait a minute! If `SqlitePool` is already internally `Arc`-wrapped, why wrap `AppState` in another `Arc`? You're right: Wrapping `SqlitePool` in another `Arc` is redundant. However, keeping it is a reasonable pattern for two reasons:

- » **Future-proofing:** If you add non-`Arc` fields to `AppState` later, the outer `Arc` becomes necessary.
- » **Consistency:** Wrapping the pool in `Arc` is a common Axum pattern and makes the code clearer about intent.

Update `main()` to connect to the database and set up the state:

```
#[tokio::main]
async fn main() {
    // Connect to the database
    let pool = SqlitePool::connect("sqlite:linklocker.db")
        .await
        .expect("Failed to connect to database");
```

```
// Create the application state
let state = Arc::new(AppState { pool });

// Build the router with state
let app = Router::new()
    .route("/", get(home))
    .nest_service("/static", ServeDir::new("static"))
    .with_state(state);

// The rest of the code is the same
```

The `SqlitePool::connect()` call opens a connection to the database file. `await` means this is an async operation, so it doesn't block the thread while waiting for the database connection to establish. `expect()` will panic with the given message if the connection fails (for example, if the database file doesn't exist or is corrupted).

`Arc::new(AppState { pool })` wraps the state in an atomic reference counter, making it safe to share across threads. The `with_state(state)` method tells Axum to make this state available to all route handlers.

Now your handlers can access the database! When you need database access in a handler, you use the `State` extractor like this:

```
async fn handler(State(state): State<Arc<AppState>>) {
    // Use state.pool to run database queries
}
```

The `State` extractor automatically extracts the shared state from the request context and gives it to your handler. This is one of Axum's nicest features because it handles all the plumbing for you.

Authentication: Who Are You?

Now comes one of the most important parts of any web application: authentication. You need to let users create accounts, log in, stay logged in as they navigate between pages, and log out when they're done. You also need to keep their passwords secure, which means hashing them properly with `bcrypt`.



WARNING The authentication code in this section is designed to teach you the basic mechanics of registration, login, and sessions. It is *not* production-ready! A real-world web application needs additional security measures, including Cross-Site Request Forgery (CSRF) protection, rate limiting, password complexity requirements, input sanitization, protection against SQL injection and XSS attacks, enforced HTTPS, and session ID regeneration after login. For production applications, I strongly recommend using a well-tested authentication library or service (such as `axum-login`) rather than implementing authentication yourself.

Setting up session management

Before you can implement registration and login, you need to set up session management. HTTP is *stateless*, meaning that each request is independent, so the server has no built-in way to remember who you are between requests. *Sessions* solve this by storing a small identifier (a session ID) in a browser cookie. When the browser sends that cookie with each request, the server can look up the associated data (such as a username) and know who's making the request. This is how “staying logged in” works.

To add session support, first add this import:

```
use tower_sessions::{ MemoryStore, Session, SessionManagerLayer };
```

Update `main()` to add the session layer:

```
#[tokio::main]
async fn main() {
    let pool = SqlitePool::connect("sqlite:linklocker.db")
        .await
        .expect("Failed to connect to database");

    let state = Arc::new(AppState { pool });

    // Create session store and layer
    let session_store = MemoryStore::default();
    let session_layer = SessionManagerLayer::new(session_store);
```

```

let app = Router::new()
  .route("/", get(home))
  .nest_service("/static", ServeDir::new("static"))
  .layer(session_layer) // Add the session layer
  .with_state(state);

// The rest of the code is the same

```

The `MemoryStore` stores sessions in memory. This approach is fine for development, but in production you'd want to use a persistent store like Redis or a database so sessions survive server restarts. More importantly, `MemoryStore` doesn't work if you run multiple server instances (each would have its own separate session store), and it lacks built-in session expiration, which is a security concern. The `SessionManagerLayer` is middleware that runs on every request to load the session from the cookie (if present) and make it available to handlers.

The `layer()` method adds middleware to the router. Middleware wraps your handlers and can inspect or modify requests and responses. The session middleware looks for a session cookie in the request, loads the corresponding session data, and makes it available via the `Session` extractor.

User registration

Now let's implement a basic user registration flow to see how it works. Users will submit a username and password, you'll hash the password with `bcrypt` (a well-known password hashing algorithm designed to be slow enough to resist brute-force attacks), and store both in the database. If the username already exists, you'll show an error. If everything's good, you'll create a session (logging them in automatically) and redirect to their bookmarks page.

First, create the registration template. Create `templates/register.html`:

```

{% extends "base.html" %}

{% block title %}Register - LinkLocker{% endblock %}

{% block content %}

```

```

<div class="auth-container">
  <h2>Create Your Account</h2>
  <p>Start organizing your bookmarks today!</p>

  {% if let Some(err) = error %}
  <div class="error">{{ err }}</div>
  {% endif %}

  <form method="post">
    <label for="username">Username</label>
    <input type="text" id="username" name="username" required
      value="{% if let Some(u) = username_prefill %}{{ u }}{% endif
%}">

    <label for="password">Password</label>
    <input type="password" id="password" name="password" required>

    <button type="submit" class="button primary">Create Account</button>
  </form>

  <p class="auth-footer">
    Already have an account? <a href="/login">Login here</a>
  </p>
</div>
{% endblock %}

```

This template extends the base and provides a registration form. Note the `{% if let Some(err) = error %}` block, which displays error messages (such as “Username already exists”) if present. The code prefills the username field if the user had a validation error, so they don’t have to retype it.

Add the template struct and form struct to `main.rs`:

```

use axum::{
    response::Redirect,
    extract::{Form, State},
};
use serde::Deserialize;

#[derive(Template)]
#[template(path = "register.html")]
struct RegisterTemplate {
    username: Option<String>,
    error: Option<String>,
    username_prefill: Option<String>,
}

```

```

}

#[derive(Deserialize)]
struct RegisterForm {
    username: String,
    password: String,
}

```

The `RegisterTemplate` holds the data needed to render the registration page: the current username (if logged in, though that shouldn't happen), an optional error message, and an optional prefilled username for after validation errors.

The `RegisterForm` is what Axum will deserialize from the HTTP form submission. When the user submits the form, Axum parses the `application/x-www-form-urlencoded` data and creates this struct. The `#[derive(Deserialize)]` macro tells `serde` how to do this.

Now implement the handler functions:

```

async fn show_register() -> RegisterTemplate {
    RegisterTemplate {
        username: None,
        error: None,
        username_prefill: None,
    }
}

async fn handle_register(
    State(state): State<Arc<AppState>>,
    session: Session,
    Form(form): Form<RegisterForm>,
) -> Result<Redirect, RegisterTemplate> {
    // Validate that username and password aren't empty
    if form.username.trim().is_empty() {
        return Err(RegisterTemplate {
            username: None,
            error: Some("Username cannot be empty".to_string()),
            username_prefill: Some(form.username),
        });
    }

    if form.password.len() < 8 {
        return Err(RegisterTemplate {
            username: None,
            error: Some("Password must be at least 8

```

```

characters".to_string()),
    username_prefill: Some(form.username),
  });
}

// Hash the password
let hash = match bcrypt::hash(&form.password, bcrypt::DEFAULT_COST) {
  Ok(h) => h,
  Err(_) => {
    return Err(RegisterTemplate {
      username: None,
      error: Some("Error hashing password".to_string()),
      username_prefill: Some(form.username),
    });
  }
};

// Try to insert the user
let result = sqlx::query(
  "INSERT INTO users (username, password_hash) VALUES (?, ?)"
)
  .bind(&form.username)
  .bind(&hash)
  .execute(&state.pool)
  .await;

match result {
  Ok(_) => {
    // Successfully created user, log them in
    if let Err(_) = session.insert("username",
form.username.clone()).await {
      return Err(RegisterTemplate {
        username: None,
        error: Some("Failed to create session. Please try
again.".to_string()),
        username_prefill: Some(form.username),
      });
    }

    // Redirect to bookmarks page
    Ok(Redirect::to("/bookmarks"))
  }
  Err(_) => {
    // Username already exists
    Err(RegisterTemplate {
      username: None,
      error: Some("Username already exists".to_string()),
      username_prefill: Some(form.username),
    })
  }
}

```

```

    })
  }
}
}

```

The handler returns `Result<Redirect, RegisterTemplate>`. On success, it redirects to bookmarks; on error, it redisplay the form with an error message. Axum automatically deserializes the form data into a struct. The handler validates the input, hashes the password with `bcrypt::hash()`, inserts the user into the database, creates a session with `session.insert("username", ...)`, and redirects. If the database insert fails (likely a duplicate username), it returns an error template.

Add the routes to `main()`:

```

use axum::routing::post;

let app = Router::new()
    .route("/", get(home))
    .route("/register", get(show_register).post(handle_register))
    .nest_service("/static", ServeDir::new("static"))
    .layer(session_layer)
    .with_state(state);

```

The `.route("/register", get(show_register).post(handle_register))` line sets up two handlers for the same path: GET requests show the form, and POST requests process the submission. This pattern is standard for forms.

Now users can register! Try it out by visiting <http://127.0.0.1:3000/register>, creating an account, and seeing the redirect. (It will 404 on `"/bookmarks"` for now since you haven't implemented that yet, but that's fine. The important thing is that the user was created and you got redirected.)

User login

Login works similarly to registration, but instead of creating a user, you look up an existing user and verify their password. Create `templates/login.html`:

```

{% extends "base.html" %}

```



```

{% block title %}Login - LinkLocker{% endblock %}

{% block content %}
<div class="auth-container">
  <h2>Welcome Back!</h2>
  <p>Log in to access your bookmarks.</p>
  {% if let Some(err) = error %}
  <div class="error">{{ err }}</div>
  {% endif %}
  <form method="post">
    <label for="username">Username</label>
    <input type="text" id="username" name="username" required
      value="{% if let Some(u) = username_prefill %}{{ u }}{% endif
%}">

    <label for="password">Password</label>
    <input type="password" id="password" name="password" required>

    <button type="submit" class="button primary">Login</button>
  </form>

  <p class="auth-footer">
    Don't have an account? <a href="/register">Register here</a>
  </p>
</div>
{% endblock %}

```

Add the template and form structs:

```

#[derive(Template)]
#[template(path = "login.html")]
struct LoginTemplate {
  username: Option<String>,
  error: Option<String>,
  username_prefill: Option<String>,
}

// Reuse RegisterForm since it has the same fields
type LoginForm = RegisterForm;

```

Implement the handlers:

```

async fn show_login() -> LoginTemplate {
  LoginTemplate {
    username: None,
    error: None,
    username_prefill: None,
  }
}

```

```

}

async fn handle_login(
    State(state): State<Arc<AppState>>,
    session: Session,
    Form(form): Form<LoginForm>,
) -> Result<Redirect, LoginTemplate> {
    // Look up the user
    let user_result = sqlx::query_as::<_, User>(
        "SELECT id, username, password_hash FROM users WHERE username = ?"
    )
    .bind(&form.username)
    .fetch_optional(&state.pool)
    .await;

    let user = match user_result {
        Ok(Some(u)) => u,
        Ok(None) | Err(_) => {
            // User doesn't exist (or database error)
            return Err(LoginTemplate {
                username: None,
                error: Some("Invalid username or password".to_string()),
                username_prefill: Some(form.username),
            });
        }
    };

    // Verify the password
    let valid = bcrypt::verify(&form.password, &user.password_hash)
        .unwrap_or(false);

    if !valid {
        return Err(LoginTemplate {
            username: None,
            error: Some("Invalid username or password".to_string()),
            username_prefill: Some(form.username),
        });
    }

    // Password is correct, create session
    session.insert("username", user.username)
        .await
        .ok();

    Ok(Redirect::to("/bookmarks"))
}

```

The `handle_login` function queries the database for the user with `fetch_optional()`, and then verifies the password using `bcrypt::verify()`. If valid, it creates a session and redirects. Note that the error message is deliberately vague — `Invalid username or password` — rather than revealing whether the username exists, preventing attackers from enumerating valid usernames.

Add the login routes:

```
let app = Router::new()
  .route("/", get(home))
  .route("/register", get(show_register).post(handle_register))
  .route("/login", get(show_login).post(handle_login))
  .nest_service("/static", ServeDir::new("static"))
  .layer(session_layer)
  .with_state(state);
```

Now users can log in! The session cookie will persist across requests, so users will stay logged in as they navigate your site.

User logout

Logout is simple: Just delete the session and redirect to the home page.

Add this handler:

```
async fn logout(session: Session) -> Redirect {
  session.delete().await.ok();
  Redirect::to("/")
}
```

Add the route:

```
.route("/logout", post(logout))
```

Now clicking the Logout button (which appears only when the user is logged in) will clear the session and redirect to the home page.

Updating the home page with login status

You need to update the `home` handler to check if someone is logged in and then pass that info to the template:

```
async fn home(session: Session) -> HomeTemplate {
  let username = session.get::<String>("username")
    .await
```

```
        .ok()
        .flatten();

    HomeTemplate { username }
}
```

The `session.get::<String>("username")` tries to retrieve the username from the session. It returns `Result<Option>: Err` if there was a session error, `Ok(None)` if the key doesn't exist, or `Ok(Some(username))` if it exists. The `.ok().flatten()` converts `Result` to `Option`, discarding errors and treating them as not logged in.

Now restart the server, log in, and you'll see the navigation change. The Login and Register links disappear, and My Bookmarks, Add Bookmark, and Logout appear instead. You'll also see Welcome, *username*! Nice!

Managing Bookmarks

Now that users can create accounts and log in, it's time to build the core functionality: managing bookmarks. You'll implement the full CRUD (create, read, update, delete) lifecycle for bookmarks.

Displaying bookmarks

Let's start by showing users their bookmarks. Create `templates/bookmarks.html`:

```
{% extends "base.html" %}

{% block title %}My Bookmarks - LinkLocker{% endblock %}

{% block content %}
<div class="page-header">
    <h2>My Bookmarks</h2>
    <a href="/bookmarks/new" class="button primary">Add Bookmark</a>
</div>

{% if bookmarks.is_empty() %}
<div class="empty-state">
    <h3>No bookmarks yet!</h3>
    <p>Start building your collection by adding your first bookmark.</p>
    <a href="/bookmarks/new" class="button primary">Add Your First
```

```

Bookmark</a>
</div>
{% else %}
<div class="bookmarks-list">
  {% for bookmark in bookmarks %}
    <div class="bookmark">
      <h3><a href="{{ bookmark.url }}" target="_blank">{{ bookmark.title }}
</a></h3>
      <p class="bookmark-url">{{ bookmark.url }}</p>

      {% if let Some(desc) = bookmark.description %}
      <p class="bookmark-description">{{ desc }}</p>
      {% endif %}

      {% if let Some(tags) = bookmark.tags %}
      <div class="bookmark-tags">
        {% for tag in tags.split(",") %}
          <span class="tag">{{ tag.trim() }}</span>
        {% endfor %}
      </div>
      {% endif %}

      <div class="bookmark-actions">
        <a href="/bookmarks/{{ bookmark.id }}/edit"
class="button">Edit</a>
        <form method="post" action="/bookmarks/{{ bookmark.id }}/delete"
          class="inline-form">
          <button type="submit" class="button danger"
            onclick="return confirm('Are you sure you want to
delete this bookmark?')">
            Delete
          </button>
        </form>
      </div>
    </div>
  {% endfor %}
</div>
{% endif %}
{% endblock %}

```

This template shows the list of bookmarks, or an “empty state” message if there are none. Each bookmark displays the title (as a link), the URL, an optional description, tags (split by comma), and Edit/Delete buttons.

Note `bookmark.tags.split(",")` in the template. Askama lets you call methods on values, so you can split the comma-separated tags and loop

over them right in the template. Neat!

Add the template struct:

```
#[derive(Template)]
#[template(path = "bookmarks.html")]
struct BookmarksTemplate {
    username: Option<String>,
    bookmarks: Vec<Bookmark>,
}
```

Refer to `bk06ch03/linklocker/src/main.rs` to check out the code for the `show_bookmarks()` handler. This handler

- » Gets the username from the session. If not logged in, the handler redirects to login. The `?` operator propagates the redirect.
- » Looks up the user's ID from the database. Again, if this fails (shouldn't happen unless the database is corrupted), the handler redirects to login.
- » Queries all bookmarks for this user, ordered by creation time (newest first). The `unwrap_or_default()` returns an empty vector if the query fails.
- » Returns the template with the username and bookmarks.

Add the route:

```
.route("/bookmarks", get(show_bookmarks))
```

Now you can view bookmarks! Of course, you don't have any yet, so you'll see the empty state. Let's fix that by implementing the feature to add bookmarks.

Adding bookmarks

Create `templates/new_bookmark.html`:

```
{% extends "base.html" %}

{% block title %}Add Bookmark - LinkLocker{% endblock %}

{% block content %}
<h2>Add New Bookmark</h2>
{% endblock %}
```

```

{% if let Some(err) = error %}
<div class="error">{{ err }}</div>
{% endif %}

<form method="post">
  <label for="url">URL *</label>
  <input type="url" id="url" name="url" required
    placeholder="https://example.com"
    value="{% if let Some(u) = url %}{{ u }}{% endif %}">

  <label for="title">Title *</label>
  <input type="text" id="title" name="title" required
    placeholder="Example Website"
    value="{% if let Some(t) = title %}{{ t }}{% endif %}">

  <label for="description">Description</label>
  <textarea id="description" name="description"
    placeholder="Optional description...">{% if let Some(d) =
description %}{{ d }}{% endif %}</textarea>

  <label for="tags">Tags</label>
  <input type="text" id="tags" name="tags"
    placeholder="rust, web, tutorial (comma-separated)"
    value="{% if let Some(t) = tags %}{{ t }}{% endif %}">

  <button type="submit" class="button primary">Save Bookmark</button>
  <a href="/bookmarks" class="button">Cancel</a>
</form>
{% endblock %}

```

Add the template and form structs:

```

#[derive(Template)]
#[template(path = "new_bookmark.html")]
struct NewBookmarkTemplate {
  username: Option<String>,
  error: Option<String>,
  url: Option<String>,
  title: Option<String>,
  description: Option<String>,
  tags: Option<String>,
}

#[derive(Deserialize)]
struct BookmarkForm {
  url: String,
  title: String,

```

```
description: Option<String>,  
tags: Option<String>,  
}
```

Note that `description` and `tags` are `Option` because they're optional fields. If the form doesn't include them (or they're empty), they'll be `None`.

Refer to `bk06ch03/linklocker/src/main.rs` to read the code for the `show_new_bookmark()` and `handle_new_bookmark()` handlers. The `handle_new_bookmark` function validates the input, gets the user ID, and inserts the bookmark. On success, it redirects to the bookmarks list. On any error, it returns the form with an error message and the values prefilled so the user doesn't have to retype everything.

Add the routes:

```
.route("/bookmarks/new", get(show_new_bookmark).post(handle_new_bookmark))
```

Now you can add bookmarks! Try adding a few different bookmarks with various titles, URLs, descriptions, and tags. Visit `"/bookmarks"` to see them displayed in the list.

Editing bookmarks

Editing is similar to adding, but you pre-fill the form with existing data and update instead of insert. Create `templates/edit_bookmark.html`:

```
{% extends "base.html" %}  
  
{% block title %}Edit Bookmark - LinkLocker{% endblock %}  
  
{% block content %}  
<h2>Edit Bookmark</h2>  
  
{% if let Some(err) = error %}  
<div class="error">{{ err }}</div>  
{% endif %}  
  
<form method="post">  
  <label for="url">URL *</label>  
  <input type="url" id="url" name="url" required  
    value="{{ url }}">  
  
  <label for="title">Title *</label>
```



```

<input type="text" id="title" name="title" required
      value="{{ title }}">

<label for="description">Description</label>
<textarea id="description" name="description">{% if let Some(d) =
description %}{{ d }}{% endif %}</textarea>

<label for="tags">Tags</label>
<input type="text" id="tags" name="tags"
      value="{% if let Some(t) = tags %}{{ t }}{% endif %}">

<button type="submit" class="button primary">Save Changes</button>
<a href="/bookmarks" class="button">Cancel</a>
</form>
{% endblock %}

```

Add the template struct:

```

#[derive(Template)]
#[template(path = "edit_bookmark.html")]
struct EditBookmarkTemplate {
    username: Option<String>,
    bookmark_id: i64,
    url: String,
    title: String,
    description: Option<String>,
    tags: Option<String>,
    error: Option<String>,
}

```

Refer to `bk06ch03/linklocker/src/main.rs` to peruse the code for the lengthy `show_edit_bookmark()` and `handle_edit_bookmark()` handlers.

When examining the code, note that the key difference between editing and adding is that you include `AND user_id = ?` in both the `SELECT` and `UPDATE` queries. This ensures users can edit only their own bookmarks, not other people's bookmarks! This difference is crucial for security because, without it, anyone could edit anyone's bookmarks by guessing bookmark IDs.

Add the routes:

```

.route("/bookmarks/:id/edit", get(show_edit_bookmark)
.post(handle_edit_bookmark))

```

`:id` is a path parameter that Axum extracts using the `Path` extractor.

Deleting bookmarks

Deletion is simple: Just run a DELETE query and redirect. Refer to `bk06ch03/linklocker/src/main.rs` to read the code for the `handle_delete_bookmark()` handler.

Add the route:

```
.route("/bookmarks/:id/delete", post(handle_delete_bookmark))
```

Note that this is POST-only, not GET. You should never perform destructive actions (like deleting data) with GET requests because browsers and web crawlers can follow GET links automatically. POST requires an explicit form submission, which is safer.

The Delete button in the bookmarks template is a small form that POSTs to this route. The `onclick="return confirm(...)"` JavaScript shows a confirmation dialog before submitting.

Now you have full CRUD! Users can create, view, edit, and delete bookmarks. Try it all out and make sure everything works.

Extending the App

The LinkLocker app is a nice little tool that's simple but useful. Could it be better? Well, sure. You could add searching and filtering. You could set up better handling of errors such as 404. And you could deploy LinkLocker to the web so that you (and anyone else) can use it from anywhere. Alas, I have run out of room to write any more. You're not stuck, however. In `bk06ch03/linklocker`, you'll find a file named `linklocker_extra.md`, which gives you full instructions for extending LinkLocker with searching, filtering, and error-handling, as well as for deploying LinkLocker to the web. Have fun!

Index

Symbols

- < > angle brackets, [328](#)
- (=) assignment operator, [41](#), [102](#), [141](#), [184](#)
- (\) backslash, path separator, [58](#), [228](#), [229](#)
- (\$) command-line prompt, Linux, [19](#)
- (%) command-line prompt, macOS Terminal, [19](#)
- (>) command-line prompt, Windows Terminal, [19](#)
- *complement dereferences, [209](#)
- (/) division sign operator, [62](#)
- (") double quotation marks, [55](#), [228](#)
- (==) equality operators, [308](#), [335](#)
- (=) equals sign operator, [62](#), [65](#), [74](#), [184](#)
- (?) error propagation operator, [193](#)
- (\ ") escape sequence, [229](#)
- (\ n) escape sequence, [229](#)
- \ t escape sequence, [241](#), [245](#)
- (>-greater than symbol) thin arrow return type, [114](#)
- (#) hashtag, [219](#)
- (-) minus sign, [55](#)
- (*/) multiple-line comments, [46](#)
- (*) multiplication operator, [62](#)
- (!=) nonequality operators, [308](#)

`::` operator, [58](#)
 πr^2 formula, [68](#)
`(|)` pipe symbol, [97](#)
`(+)` plus sign, [55](#)
`{:p}` pointer formatter, [150](#)
`(%)` remainder operator, [64](#)–[65](#), [89](#)
`(;)` semicolon, [41](#)
`(//)` single-line comments, [46](#)
`(')` single quotation marks, [55](#), [228](#)
`# [...]` syntax, [309](#)
`(_)` underscore separator, [52](#)
`(~)` user account directory, [28](#)
`(_)` wildcard character, [94](#), [98](#), [102](#), [189](#)

A

`a.cmp(b)` method, [258](#), [259](#)
adapting iterators
 definition, [269](#)
 `enumerate()` method, [270](#)–[271](#)
 `filter()` method, [269](#)
 `map()` method, [270](#)
`add_numbers()` function, [115](#)
`add_one()`s function, [123](#)
Adobe Photoshop, [506](#)
`analyze_password()` function, [523](#)–[524](#)

AND (&&) operators, [81](#)–[82](#), [148](#)

anonymous loop, [88](#)

ANSI escape sequences, [493](#)

AppConfig struct, [357](#)

application programming interface (API), definition of, [594](#)

APP_NAME variable, [119](#)

Arc (atomic reference counter), [411](#), [559](#), [602](#), [626](#), [627](#)

 sharing ownership, [407](#)–[409](#)

archive subcommand, [500](#)–[501](#)

area variable, [75](#), [99](#)

arguments, [255](#)

 function of, [115](#), [117](#), [142](#), [158](#), [205](#), [210](#), [228](#), [239](#), [348](#), [376](#), [534](#)

 definition of, [110](#)

 multiple arguments, [112](#)–[113](#)

 as mutable, [113](#)–[114](#)

 read-only text arguments, [113](#)

 single arguments, [111](#)–[112](#)

 text argument, [156](#)

arithmetic assignment operators, [65](#)–[66](#)

arithmetic operators, [62](#)–[64](#)

 division sign (/) operator, [62](#)

 multiplication (*) operator, [62](#)

 remainder operator (%), [64](#)–[65](#)

array_contains() function, [334](#)–[336](#)

arrays, [529](#)

- bounds checking in, [203](#)

- definition of, [201](#)

- element in, [203](#), [204](#)

- functions of, [204](#)–206

- index, [202](#)

- iteration, [203](#)–204

- JavaScript, [532](#)–535

- numeric, [530](#)

- repeat expression syntax, [202](#)

- types of, [204](#)–206

- vector creation, [212](#)–213

Askama, [621](#)–625, [639](#)

as keyword, [64](#)

assertion macros, [427](#)

assignment operator (=), [41](#), [102](#), [141](#), [184](#)

associated functions, [168](#)–169, [171](#)

- constructor, [169](#)

asynchronous programming, [414](#)

async keyword, [414](#)–416, [513](#), [578](#)

atomic reference counter (Arc), [411](#), [559](#), [602](#), [626](#), [627](#)

- sharing ownership, [407](#)–409

- authentication, for web application, [628](#)
 - session management, [629](#)–630
 - updating the home page with login status, [637](#)
 - user login, [634](#)–636
 - user logout, [636](#)–637
 - user registration, [630](#)–634
- authentication flow modelling project
 - error propagation operator (?), [193](#)
 - example of, [195](#)
 - `main()` function, [194](#)
 - `ResultString, io:Error` return type, [193](#)
 - `unwrap_or_default()` method, [194](#)
 - user input, [193](#)
- `author: author` function, [164](#)
- AutoCAD, [507](#)
- `await` keyword, [414](#)–416, [541](#), [628](#)
- Axum, [571](#), [575](#)–579, [618](#), [625](#)–628, [631](#), [633](#), [642](#)
 - routing works in, [580](#)

B

- backslash (`\`) path separator, [58](#), [228](#), [229](#)

Bank Account Manager project

adding implementation block, [171](#)–173

adding user input functions, [173](#)

`check_balance()` method, [172](#)

creating instance of struct, [173](#)–174

`deposit()` method, [172](#)

`display_account_info()` method, [173](#), [174](#)

error handling, [191](#)

`get_amount()` function, [173](#)

`get_user_input()` function, [173](#)

main menu loop implementation, [175](#)

setting up the struct, [171](#)

`unwrap_or_default()` method, [173](#)

`withdraw()` method, [172](#), [191](#)

`BankAccount` struct, [173](#), [174](#)

behavioral trait, [305](#)

binary (base 2), [52](#)

binary crate, [280](#)

binary size, [507](#)

binding generator, [515](#)

blank placeholder, [47](#)–48

blocks, expressions, [74](#)–75, [255](#)

block scope, [117](#)

definition of, [99](#)

example, [99](#)–100

block syntax, [78](#)

book library API, [600](#)–604

 database migration with SQLx, [609](#)–614

 deployment considerations, [615](#)

 error handling, [607](#)–609

 testing, [614](#)–615

 validation, [607](#)–609

bookmarks, in database

 adding, [639](#)–641

 deletion, [643](#)

 displaying, [637](#)–639

 editing, [641](#)–642

BookStore type, [603](#)

Book struct, [161](#), [162](#), [169](#), [355](#), [601](#)

bool data type, [54](#)–55, [329](#)

borrow checker, [14](#), [146](#)–149, [152](#), [153](#), [156](#), [158](#), [264](#), [345](#), [350](#)

borrowed string. *See* [\(&str\)_string slice type](#)

borrowing, [140](#), [144](#)

 and function calls, [145](#)

 golden rule of, [146](#)–149

 immutable borrowing, [144](#)–149, [154](#)

 mutable borrowing, [146](#)–149, [154](#)

 non-owning variable, [144](#)

 references, [149](#)–153

 sharing XOR mutating, [148](#)

 string parameters, [145](#)

bounds checking in arrays, [203](#)

`break` keyword, [85](#)

`break` mode, [436](#)

`break` statement, [84](#), [89](#), [90](#), [189](#)

B-Tree map, [221](#)

buffer overflows, [135](#)

`BufReader`, [473](#)

`BufRead` trait, [458](#)

C

C++, [13](#), [14](#)

`calculate_tip()` function, [116](#), [121](#)

Cargo

- capabilities of, [25](#)

- `Cargo.toml` file (*see* [Cargo.toml file](#))

- commands, [26](#)

- `rand` version 0.9.2, [296](#)

- `src` subdirectory, [29](#)–[30](#)

`cargo build` command, [26](#), [34](#), [280](#), [282](#), [361](#), [449](#)

`cargo check` command, [26](#), [34](#)

`cargo clean` command, [26](#)

`cargo: command not found` error, [23](#)

`cargo doc` commands, [26](#)

`Cargo.lock` file, [31](#)

`cargo new` command, [282](#), [431](#)

`cargo new project` commands, [26](#)

`cargo run` command, [26](#), [34](#)

`cargo test` command, [26](#), [424](#), [426](#), [429](#), [433](#), [445](#)

`Cargo.toml` file, [280](#), [296](#), [297](#), [302](#), [415](#), [461](#), [494](#), [495](#), [521](#), [530](#), [539](#), [548](#), [552](#), [609](#), [614](#)

dependencies, [31](#), [297](#)

edition, [31](#)

name, [31](#)

version, [31](#)

`cargo --version` commands, [23](#)

case-conversion methods

example, [243](#)–[244](#)

output, [244](#)

`string.make_ascii_lowercase()` method, [244](#)

`string.make_ascii_uppercase()` method, [244](#)

`string.to_lowercase()` method, [243](#)

`string.to_uppercase()` method, [243](#)

`cd path` (change directory), [20](#)

`cd project` command, [35](#)

cedilla variable, [54](#)

change directory (`cd path`), [20](#)

channel, message passing, [403](#)–[404](#)

multiple producers, single consumer, [406](#)–[407](#)

sender and receiver, [404](#)–[406](#)

`char` data type, [54](#), [237](#)

`char.is_ascii_alphanumeric()` method, [233](#)

`char.len_utf8()` method, [237](#)

`chars()` method, [185](#), [232](#)–[233](#)

`check_balance()` method, [172](#)

choice variable, [102](#), [103](#), [189](#)

clamp, [337](#)

cleaning service of code, [439](#)

Clippy, [441](#)–443

`rustfmt`, [439](#)–441

Clippy, [441](#)–443

`clone()` method, [141](#), [142](#), [308](#), [312](#)

`Clone` trait, [308](#)–310, [312](#)

cloning, [141](#)–142, [154](#), [235](#), [309](#)

closures

capturing local variables, [256](#)–257

definition of, [255](#)

general syntax for, [255](#)

`sort_by_key()` method, [257](#)–258

`sort_by()` method, [258](#)–259

to write, [259](#)–260

code

- cleaning service, [439](#)

 - Clippy, [441](#)–443

 - `rustfmt`, [439](#)–441

- debugging, [433](#)

 - VS Code user, [435](#)–437

 - `dbg!` macro, [434](#)–435

 - `println!` macro, [438](#)–439

- documentation, [443](#)

 - adding comments, [443](#)–445

 - building your program, [446](#)

 - running documentation examples as tests, [445](#)–446

- testing, [422](#)–423

 - assertion macros, [427](#)

 - code panics, [428](#)

 - integration tests, [432](#)–433

 - library, [431](#)

 - module, [429](#)–431

 - returns `Result`, [427](#)–428

 - writing, [423](#)–426

- CodeLLDB, [435](#)

code organization

access control

encapsulation, [292](#)–293

private by default, [290](#)–292

crate

binary crate, [280](#)

definition of, [280](#), [296](#)

documentation, [297](#)

library crate, [280](#)

as project dependency, [296](#)

rand rate, [296](#)

dependency, [296](#)–297

module

crate root module, [280](#)

definition of, [280](#)

and files, [281](#)

inline module, [283](#)–284

shortening the module paths, [294](#)–295

tree of, [280](#), [281](#)

in multiple files

creating own module files, [288](#)–290

mapping rules, [287](#)–288

using `main.rs` and `lib.rs`, [287](#)

in one file

creating modules in `main.rs`, [283](#)–284

`main.rs` vs `lib.rs` decision, [282](#)–283

nesting modules, [284](#)–287

package, [280](#)

real-world organization patterns, [298](#)–302

`use` keyword, code importing, [293](#)–294

coding process

abstraction, [14](#)

repeat, [12](#)

revise, [12](#)

run, [12](#)

write, [12](#)

Collatz Conjecture, [416](#)–420

collection, definition of, [87](#)

`collect()` method, [245](#), [268](#), [273](#), [275](#)

color scheme, [65](#)

command line, [21](#)

basic command-line survival skills, [20](#)

definition of, [18](#)

keyboard shortcuts, [19](#)–20

launching terminal, [18](#)–19

shell, [18](#)

terminal, [18](#)

command-line interface (CLI) tools, [449](#)–451, [465](#), [487](#)–489, [491](#), [493](#), [494](#), [496](#), [498](#)

- arguments, [451](#)–452

- clarity, [450](#)

- composability, [450](#)

- courtesy, [450](#)

- file operations, [459](#)–460

- parsing arguments, [452](#)–457

- standard input (`stdin`), [457](#)–458

- with subcommands

 - adding, [488](#)–490

 - working with nested, [491](#)–493

- `word-replacer` tool, [461](#)–464

comma-separated values (CSV)

- reading files, [477](#)–479

- writing files, [477](#)–479

comment, definition of, [46](#)

comparison expressions, [54](#)–55, [78](#), [86](#)

- comparison operators, [68](#)

- definition of, [68](#)

- example, [69](#)

comparison operators, [68](#)

compiler, definition of, [11](#)

computer, description, [8](#), [254](#)

concatenation operator, [234](#)

concurrency safety of Rust, [13](#)

concurrency vs parallelism, [396](#)–397, [403](#)

- ConfigError **enum**, [378](#)
- config.rs **file**, [299](#)
- constants, [56–57](#), [119](#), [272](#)
- constructor, [169](#), [171](#), [341](#)
- consuming iterators
 - collect() **method**, [268](#)
 - example, [265–266](#)
 - find() **method**, [266–268](#)
 - iterator.count() **method**, [265](#)
 - iterator.last() **method**, [265](#)
 - iterator.max() **method**, [265](#)
 - iterator.min() **method**, [265](#)
 - iterator.product() **method**, [265](#)
 - iterator.sum() **method**, [265](#)
 - output, [266](#)
- contains() **method**, [242](#), [243](#), [273](#), [324](#)
- continue **statement**, [89](#)
- control block, [151](#)
- conversion operator, [64](#)
- copied() **method**, [359](#)
- Copy **trait**, [137–138](#), [140](#), [166](#), [212](#), [219](#), [337](#)
- cost_per_splice() **function**, [187](#)
- counts.get_mut(w) **method**, [274](#)
- C programming language, [10](#), [13](#), [14](#)
- CPU-bound operation, [397](#)

crate

- binary crate, [280](#)

- definition of, [280](#), [296](#)

- documentation, [297](#)

- library crate, [280](#)

- multiple crates, [301](#)–302

- as project dependency, [296](#)

- root module, [280](#)

CreateBook structs, [601](#)

Cross-Site Request Forgery (CSRF), [628](#)

CRUD operations, [600](#)

- book update, [605](#)–606

- deletion, [606](#)–607

- single book with path parameters, [604](#)–605

CSS files, [571](#), [575](#), [589](#), [590](#), [625](#)–626

curl command, [22](#)

D

dangling pointer, [135](#), [158](#), [344](#)

database

bookmarks

adding, [639](#)–641

deletion, [643](#)

displaying, [637](#)–639

editing, [641](#)–642

connection pool, [626](#)–628

design for LinkLocker, [619](#)–620

initialization, [620](#)–621

`database.rs` file, [299](#)

data cleanup procedure

garbage collection, [134](#), [135](#)

manual memory management, [134](#), [135](#)

data storage

arrays, [201](#)–206

hash maps, [217](#)–222

slices, [206](#)–211

tuple, [198](#)–201

vectors, [211](#)–217

`days_in_month()` function, [122](#)

`days_in_month` variable, [96](#)

`dbg!` macro, [434](#)–435, [438](#)

debugging of code, [433](#), [591](#)–592

VS Code user, [435](#)–437

`dbg!` macro, [434](#)–435

`println!` macro, [438](#)–439

- Debug trait, [206](#), [310](#), [316](#)
- declarative macro, [384](#), [387](#)
- deep copy, [141](#)
- deep-copying heap data, [141](#)–142
- Default trait, [318](#)–319
- DELETE (removing resources), [600](#)
- demonstrate_parse_error() function, [373](#)–374
- dependency
 - adding a crate, [296](#)
 - definition of, [296](#)
 - external dependency, [296](#)
 - importing to Cargo.toml, [297](#)
- deposit() method, [172](#)
- deref coercion, [239](#)–240, [548](#)
- dereferencing, [150](#)
 - with mutable slice, [209](#)–210
- derive attribute, [309](#)
- deserializing data, [480](#)–482
- destructuring, tuple, [200](#)
- directory (dir) on Windows, [20](#)
- dir (directory) on Windows, [20](#)
- display_account_info() method, [173](#), [174](#)
- display_app_title() function, [285](#)
- Display trait, [306](#)–307, [310](#), [316](#)–318, [324](#), [341](#), [377](#), [379](#)
- display_welcome() function, [286](#)
- division sign (/) operator, [62](#)

doctest, [446](#)

Document Object Model (DOM), [506](#), [514](#), [520](#)

Don't repeat yourself (DRY). *See also* [generics](#)

definition, [325](#)

violations, [327](#)

dot notation, [162](#)–163

double-free, [135](#)

double quotation marks ("), [55](#), [228](#)

E

elision rules, for lifetimes, [354](#)–356

empty tuple, [199](#)

empty vector, [211](#), [264](#), [363](#), [364](#), [411](#), [639](#)

enum (enumerations), [159](#), [161](#). *See also* [generics](#)

- adding data, [180](#)–181

- adding trait to, [309](#)

- creating an instance of, [178](#)–179

- definition of, [177](#), [180](#)

- error handling

 - assigning default value, [189](#)–190

 - example, [188](#)

 - if let pattern-matching, [192](#)

 - return functions, [190](#)–192

 - (_) wildcard character, [189](#)

- general syntax of, [178](#)

- if let pattern-matching, [179](#)–180, [184](#)

- LoginError enum, [281](#)

- Option enum (*see* [Option enum](#))

- Ordering enum, [258](#)–259

- PascalCase, [178](#)

- Result, [93](#), [188](#)–190

- TrafficLight enum, [178](#), [179](#)

- variants, [178](#)

enumerate() method, [270](#)–271

equality (==) operators, [308](#), [335](#)

equals sign operator (=), [62](#), [65](#), [74](#), [184](#)

error, location of, [53](#)

error handling

pattern matching with `match` keyword, [365](#)–366

problems with, [372](#)–373

`Result` type, [364](#)–365

error propagation operator (`?`), [193](#)

error-related helper methods

`unwrap()` and `expect()`, [366](#)–367

`unwrap_or_default()` method, [370](#)–371

`unwrap_or_else()` method, [369](#)–370

`unwrap_or()` method, [367](#)–369

error types, [373](#)–374

`Err(String)`, [377](#)–380

`Err` variant, [189](#)–191, [365](#), [366](#), [371](#)

escape sequence (`\"`), [229](#)

escape sequence (`\n`), [229](#)

escape sequence (`\t`), [229](#)

Escape the Dungeon project (text-based adventure game)

setting up the game, [102](#)–103

user input, [101](#)–102

pattern-matching, [104](#)–106

variable shadowing, [103](#)

excerpt variable, [156](#)

`expect()` method, [366](#)–369

explicit lifetimes, [346](#)

expression-based return, [115](#)

expressions, [61](#). *See also specific expressions*

blocks, [74](#)–[75](#), [255](#)

comparison, [54](#)–[55](#), [78](#), [86](#)

definition of, [62](#)

logical, [78](#), [79](#), [86](#)

numeric, [62](#)–[68](#)

operands, [62](#)

operator, [62](#)

remainder, [65](#)

structure, [62](#)

external data, [480](#)

F

`factorial()` function, [123](#)

fat pointer, [151](#)–[152](#)

`favorite_number` variable, [41](#), [42](#), [44](#), [45](#), [51](#)

fearless concurrency, [14](#), [396](#)

`fetch_and_render()` function, [541](#)

field init shorthand, [164](#)–[165](#)

Figma, [507](#)

file-handling techniques, [482](#)–[485](#)

file operations, for command-line interface (CLI) tools, [459](#)–[460](#)

files vs modules, [281](#)

file system

fiddling, [470](#)–471

manipulating paths, [466](#)–470

building, [467](#)–468

components, [468](#)–470

new, [467](#)

`filter()` method, [269](#)

`filter(|w| !STOPWORDS.contains(&w.as_str()))` method, [273](#)

`filter(|w| !w.is_empty())` method, [273](#)

`find()` method, [182](#), [183](#), [185](#), [186](#), [243](#), [266](#)–268

fixed-sized storage. *See* [arrays](#)

`Float32Array`, [530](#), [531](#)

floating-point constants, [66](#)–67

floating-point expressions, [64](#)

floating-point methods, [67](#)–68

floating-point number

`bool`, [138](#), [161](#)

`char`, [138](#), [161](#)

definition of, [53](#)

`f32`, [53](#), [205](#), [329](#)

`f64`, [53](#), [173](#), [329](#)

flow control

- `if...else` statement, [79](#)–80

- `if` statement, [78](#)–79

- making multiple decisions

 - `if` expression, [82](#)–83

 - multiple `if/else` blocks, [80](#)–81

 - using AND (`&&`) and OR (`||`) operators, [81](#)–82

- using loops, [83](#)–91

- `fmt` module, [307](#), [316](#)

- `fn main() { }` program, [197](#)

- `for` loops, [87](#)–89, [249](#), [251](#), [262](#)–264, [381](#), [393](#), [405](#), [419](#), [439](#)

- `format_currency()` function, [116](#), [121](#)

- `format!` macro, [235](#), [323](#), [324](#)

foundation

- database design for LinkLocker, [619](#)–620

- database initialization, [620](#)–621

- project structure, [618](#)–619

- `from()` method, [212](#), [218](#), [219](#), [541](#)

function. *See also* [generics](#)

- advantages, [120](#)

- arguments, [115](#), [117](#), [142](#), [158](#), [205](#), [210](#), [228](#), [239](#), [348](#), [376](#), [534](#)

 - definition of, [110](#)

 - multiple arguments, [112](#)–113

 - as mutable, [113](#)–114

 - read-only text arguments, [113](#)

 - single argument, [110](#)–111

- basic structure of, [109](#)

- definition of, [108](#)

- helper function, [121](#)–122

- importance of, [108](#)–109

- with no arguments, [110](#)

- recursive function, [122](#)–124

- return values, [114](#)–116

- separating responsibilities, [120](#)–121

functional programming

closures

capturing local variables, [256](#)–257

definition of, [255](#)

general syntax for, [255](#)

`sort_by_key()` method, [257](#)–258

`sort_by()` method, [258](#)–259

to write, [259](#)–260

definition of, [254](#)

iterator

adapting iterators, [269](#)–271

consuming iterators, [264](#)–268

creation, [261](#)–264

definition of, [260](#)

uses of, [260](#)

`function_name` function, [110](#)

function scope

advantage, [118](#)

code, [117](#)

definition of, [117](#)

`local_variable` variable, [118](#)

output, [118](#)

function signature, [112](#), [115](#), [240](#), [327](#), [329](#), [335](#), [338](#), [349](#)

G

Games, [507](#)

- garbage collection, [13](#), [134](#)
- garbage collector, [100](#), [135](#), [507](#)
- `generate_wave()` function, [531](#)–532
- generic lifetime parameter declaration, [348](#), [349](#), [353](#)
- generics, [340](#)–342
 - `<T>` (type parameter), [327](#)–335
 - `<U>` (type parameter), [333](#)
 - `<V>` (type parameter), [333](#)
 - description of, [327](#)
 - making enum generic, [331](#)–332
 - making function generic, [328](#)–329
 - making struct generic, [329](#)–330
 - trait bounds, [335](#)–338
- `GET` (reading resources), [600](#)
- `get_amount()` function, [173](#)
- `get_character_count()` function, [156](#)
- `get_duration()` method, [321](#), [323](#)
- `get_first_element()` function, [363](#)
- `get_length()` function, [142](#), [143](#), [156](#)
- `get_length(&msg)` function, [145](#)
- `get_longest_word()` function, [156](#)
- `get_menu_choice()` function, [125](#), [126](#), [129](#)
- `get_mut()` method, [274](#)
- `get_rating()` method, [323](#)
- `get_review_count()` method, [323](#)
- `get_searchable_text()` method, [321](#), [324](#)

`get_temp()` function, [126](#), [129](#)
`get_user_input()` function, [173](#), [175](#)
`get_value()` method, [359](#)
`get_word_count()` function, [156](#)
`.gitignore` file, [32](#)
global scope, [119](#), [518](#)
golden rule of borrowing, [146](#)–149
Google Earth, [507](#)
`greet()` function, [598](#)

H

handler functions, [580](#)–581, [583](#), [586](#), [587](#), [631](#)
`handlers.rs` file, [299](#)
hashing algorithm, [619](#), [630](#)
`HashMap` data type, [217](#)
`hash_map.get()` method, [220](#)
`hash_map.insert(key, value)` method, [220](#)
`HashMap<K, V>`, [218](#)
`hash_map.remove(key)` method, [221](#)

hash maps

- accessing values, [219](#)–220

- creation

 - from array of tuples, [218](#)–219

 - new hash map, [218](#)

- definition of, [217](#)

- `HashMap<K, V>`, [218](#)

- iteration, [221](#)–222

- key-value pairs, [217](#)

 - adding and removing, [220](#)–221

- mutable hash map, [220](#), [221](#)

hashtag (#), [219](#)

heap, [138](#)–141

- deep-copying heap data, [141](#)–142

heap-allocated variable, [141](#)

height variable, [99](#)

helper function, [126](#)

- `days_in_month()` function, [122](#)

- definition of, [121](#)

- examples, [121](#)–122

- `is_leap_year()` function, [122](#)

- `is_valid_date()` function, [122](#)

- output, [122](#)

hexadecimal (base 16), [52](#)

Hoare, Graydon (programmer), [12](#)

HTML, [446](#), [512](#)–514, [518](#), [523](#), [573](#), [575](#), [579](#), [589](#), [590](#), [617](#)

reading HTML from files, [582](#)–584

return HTML responses, [581](#)–582

with superpowers, [621](#)–625

human language, [9](#)

Hypertext Transfer Protocol (HTTP), [575](#)–576, [618](#), [628](#), [629](#), [631](#)

request, [572](#)–573

response, [574](#)–575

I

i32 integers, [50](#), [125](#), [126](#), [204](#), [209](#)–211, [218](#), [227](#), [328](#), [329](#)

idiomatic Rust, [441](#)

if...else statement, [79](#)–80

if expression, [78](#), [79](#), [82](#)–83, [86](#)

if let pattern-matching, [192](#)

if statement, [78](#)–82, [97](#), [104](#), [254](#)

immutable borrowing, [144](#)–149, [154](#), [209](#)

immutable variable, [44](#), [56](#)

imperative programming, [254](#)

implementation (impl) block, [167](#), [169](#), [171](#)–173, [314](#)

implicit lifetimes, [345](#)

impl keyword, [167](#)

inbugging, [433](#)

index, arrays, [202](#)

indexing operator method, [213](#)–214

infinite recursion, [123](#)

`infinity_symbol` variable, [54](#)

`initialize()` function, [119](#)

inline bounds, [338](#)

inline module, [283](#)–286

inner documentation, [443](#)

installation, Rust

- components installed, [22](#)

- on Linux, [22](#)

- on macOS, [22](#)

- verification, [22](#)–23

- on Windows, [21](#)

`instance` syntax, [165](#), [166](#)

`Int32Array`, [530](#)

integers, [43](#), [63](#), [137](#), [161](#)

- with floating-point type, [64](#)

- `i32`, [50](#), [125](#), [126](#), [204](#), [210](#), [211](#), [218](#)

- platform-dependent, [51](#)

- types of, [49](#)–53

- `u8`, [51](#), [189](#), [327](#), [329](#)

integration tests, [432](#)–433

internal data, [480](#)

interpreter, definition of, [10](#)

`into_iter()` iterator, [263](#)–264, [267](#)

`IntoResponse`, [580](#), [581](#), [618](#), [625](#)

I/O-bound operation, [397](#)

`io` module, [101](#), [285](#), [289](#)

`is_empty()` method, [233](#)

`isize` type, [51](#)

`is_leap_year()` function, [122](#)

`is_prime` vector, [224](#)

`is_valid_date()` function, [122](#)

`item_average` variable, [62](#)

`item_count` variable, [62](#)

item-level documentation, [444](#)

`item_total` variable, [62](#)

iterator, [157](#)

adapting iterators

definition, [269](#)

`enumerate()` method, [270](#)–271

`filter()` method, [269](#)

`map()` method, [270](#)

consuming iterators

`collect()` method, [268](#)

example, [265](#)–266

`find()` method, [266](#)–268

`iterator.count()` method, [265](#)

`iterator.last()` method, [265](#)

`iterator.max()` method, [265](#)

`iterator.min()` method, [265](#)

`iterator.product()` method, [265](#)

`iterator.sum()` method, [265](#)

output, [266](#)

creation

borrowing immutably with `iter()`, [261](#)–262

borrowing mutably with `iter_mut()`, [262](#)–263

owning with `into_iter()`, [263](#)–264

definition of, [260](#)

uses of, [260](#)

`iterator.any(closure)` method, [268](#)

`iterator.count()` method, [265](#)

`iterator.for_each(closure)` method, [271](#)

`iterator.last()` method, [265](#)
`iterator.max()` method, [265](#)
`iterator.min()` method, [265](#)
`iterator.product()` method, [265](#)
`iterator.sum()` method, [265](#)
`iter()` method, [261](#), [264](#)
`iter_mut()` method, [262](#)–263

J

JavaScript, [33](#), [225](#), [506](#), [571](#), [575](#), [589](#), [591](#)
 array of strings to Rust, [534](#)–535
 array to Rust, [532](#)–533
 functions, [515](#)–519
 namespaces, [515](#)–517
 Rust user profile struct to, [525](#), [535](#)
 Rust vector to, [530](#)
 string, [527](#)–529
 vector of strings to, [533](#)–534
JavaScript Object Notation (JSON), [477](#), [480](#)–483, [597](#)
 Greeting struct, [598](#)
 vs HTML, [595](#)–596

K

key, definition of, [257](#)
keyboard shortcuts, command line, [19](#)–20

L

labeling loops, [90](#)–91

layout template, [622](#)–624

`len()` method, [142](#), [232](#)

`let` statement, [51](#), [345](#)

lexical scope. *See* [block scope](#)

library crate, [280](#)

`lib.rs` decision, [282](#)–283, [287](#), [298](#)

lifetimes, [343](#)–346

- annotation, [346](#)–347

- definition, [343](#)

- elision rules, [354](#)–356

- facts about, [344](#)

- function, [347](#)–348

 - multiple lifetime annotations, [350](#)–351

 - return reference, [350](#)

 - signature, [349](#)

 - unsafe designs, [351](#)

- `static` keyword, [356](#)–357

- structs, [352](#)–353

- zero-copy parser approach, [357](#)–360

linear memory, [527](#)

LinkLocker

- database design for, [619](#)–620

- extending instructions, [643](#)

`linklocker.db` file, [621](#)

Linux

Rust installation, [22](#)

Terminal app launching, [19](#)

Linux kernel, [14](#)

`load_game()` function, [482](#)

`load_server_config()` function, [380](#)

local scope. *See* [function scope](#)

`local_variable` variable, [118](#)

`lock()` method, [458](#)

logging, in web server, [591](#)–592

logical expressions, [78](#), [79](#), [86](#)

definition of, [69](#)

example of, [70](#)

logical operators, [69](#)

logical operators, [69](#)

`LoginError` enum, [281](#)

`login_screen` module, [281](#)

`longest_word()` function, [275](#)–277

`longest_word` variable, [157](#), [158](#)

- loops, [203](#), [216](#)
 - anonymous loop, [88](#)
 - basic loop, [84](#)–85
 - bypassing loop statements, [89](#)
 - conditional repetition, [86](#)–87
 - continue statement, [89](#)
 - definition of, [84](#)
 - labeling loops, [90](#)–91
 - for loops, [87](#)–89
 - main menu loop, [175](#)
 - nesting loops, [89](#)–90
 - using loop as expression, [85](#)
 - while loops, [86](#)–87, [224](#)
- lowercase() method, [103](#)
- ls (list), Mac and Linux, [20](#)

M

- machine language, [10](#)
- macOS
 - Rust installation, [22](#)
 - Terminal app launching, [18](#)–19
- macro_rules!, [384](#)–387, [389](#), [390](#), [392](#)

macros, [381](#)

- code, [383](#)–384

- `format!` macro, [235](#), [323](#), [324](#)

- `println!` (see [println! macro](#))

- syntax, [384](#)–387

- timer macro, [391](#)–393

- `vec!` macro, [213](#), [382](#)–383

- writing, [387](#)–391

 - converting of key-value pairs to vector, [390](#)–391

 - sums any number of inputs, [389](#)–390

`main()` function, [40](#), [74](#), [77](#), [108](#), [111](#), [118](#)–120, [127](#), [145](#), [191](#), [194](#), [199](#), [252](#), [272](#), [276](#)–277, [282](#), [283](#), [292](#), [463](#)–464, [484](#)–485, [498](#)–499

`main.rs` file, [29](#), [30](#), [40](#), [111](#), [155](#), [160](#), [171](#), [223](#), [279](#), [281](#)–283, [287](#)

`make_ascii_lowercase()` method, [263](#)

`make_cookies()` function, [292](#)

`make_directory` (`mkdir name`), [20](#)

`make_string_tuple()` function, [326](#), [328](#), [329](#)

manual memory management, [134](#)

`map()` method, [270](#)

mapping rules, module files

- submodules, [288](#)

- top-level modules, [287](#)

`map(|w| w.to_lowercase())` method, [273](#)

Markdown, [444](#)–446, [538](#)–543

`matches_query()` method, [321](#), [324](#)

`match` expression, [129](#)

- arm of, [94](#)

- example, [95](#)–96

- general syntax, [94](#)

- matching one value or another, [97](#)

- matching range of values, [96](#)

- matching with a condition, [97](#)–98

- pattern alternative syntax, [97](#)

`match` keyword, [179](#), [365](#)–366

`match` structure, [100](#), [104](#), [129](#), [179](#), [253](#)

`max_by_key()` method, [276](#)

`MAX_PLAYERS` variable, [119](#)

Mcnamara, Tim, [617](#)

Measurement enum, [332](#)

media library building project

- Display trait, [324](#)

- Movie struct, [321](#)–322

- Playable trait, [321](#)

- Rateable trait, [320](#), [322](#)–323

- Searchable trait, [321](#), [323](#)–324

memory, facts of, [33](#)

memory leak, [134](#)

- memory management, [100](#), [507](#)
 - failure of
 - buffer overflows, [135](#)
 - dangling pointer, [135](#)
 - double-free, [135](#)
 - memory leak, [134](#)
- memory safety, [13](#), [135](#), [141](#), [143](#), [148–149](#), [507](#)
- memory storage, variables, [42–43](#)
- message passing, [403–404](#)
 - multiple producers, single consumer, [406–407](#)
 - sender and receiver, [404–406](#)
- metaprogramming, [382](#)
- method, definition of, [166](#)
- method chaining, [266](#)
- mise en place* principle, [40](#)
- missing command-line argument, [451](#)
- `mkdir name` (make directory), [20](#)
- `mod block`, [283](#), [284](#)
- module-level documentation, [443](#)

modules, [58](#), [280](#)

`config.rs` file, [299](#)

 crate root module, [280](#)

`database.rs` file, [299](#)

 definition of, [280](#)

 and files, [281](#)

`fmt` module, [316](#)

`handlers.rs` file, [299](#)

 inline module, [283](#)–284

`io` module, [101](#), [285](#), [289](#)

`login_screen` module, [281](#)

 in `main.rs`, [283](#)–284

 mapping rules, [287](#)–288

 nesting modules, [284](#)–287

 shortening the module paths, [294](#)–295

`std::io` module, [125](#), [129](#), [171](#)

 tree of, [280](#), [281](#)

`user_interface` module, [285](#), [286](#), [288](#)–290, [294](#)

`utils.rs` file, [299](#)

monomorphization, [329](#)

`msg` variable, [142](#), [143](#)

multiblock statement, [80](#)

- multicommand file organizer, [496](#)–501
 - archive subcommand, [500](#)–501
 - command-line interface structure, [498](#)
 - main() function, [498](#)–499
 - scan subcommand, [499](#)
 - stats subcommand, [501](#)
- multiple arguments, function
 - examples, [112](#), [113](#)
 - output, [113](#)
 - &str, [113](#)
- multiple crates, [301](#)–302
- multiple if/else blocks, [80](#)–81
- multiple lifetime annotations, [350](#)–351
- multiple-line comments (* /), [46](#)
- multiplication (*) operator, [62](#)
- multi-threaded code, [395](#)
- mutable borrowing, [146](#)–149, [154](#)
- Mutex, [410](#)–411
- mut keyword, [45](#), [114](#), [146](#), [163](#)–164, [290](#), [291](#)

N

- named placeholders, [47](#)
- nesting loops, [89](#)–90
- nesting modules, [284](#)–287
- nesting parentheses, [73](#)
- newline character, [47](#)

`next()` method, [454](#)–455

Node.js installation, [509](#)

nonequality (`!=`) operators, [308](#)

non-lexical lifetime, [149](#)

note-taking tool, [489](#)–490

numbered placeholders, [48](#)

numeric expressions

- arithmetic operators in, [62](#)–64

- definition of, [62](#)

- floating-point constants in, [66](#)–67

- floating-point methods, [67](#)–68

- remainder operator (`%`) in, [64](#)–65

- using arithmetic assignment operators, [65](#)–66

numeric type conversion, [64](#)

O

octal (base 8), [52](#)

`ok_or_else()` method, [376](#)–377

`ok_or()` method, [375](#)–376

`open()` method, [476](#)

operand, definition of, [62](#)

operator, definition of, [62](#)

`Option` enum, [93](#), [181](#)–183, [331](#), [374](#)–375

`ok_or_else()` method, [376](#)–377

`ok_or()` method, [375](#)–376

 None arm, [184](#)

 assigning a default value to, [185](#)–186

 return functions, [186](#)–187

 Some arm, [184](#)

`Optionf64` enum, [187](#)

`Option<usize>` method, [182](#)

`Ordering` enum, [258](#)–259

 Equal, [258](#)

 Greater, [258](#)

 Less, [258](#)

order of precedence

 control of, [72](#)–73

 for operators, [70](#), [71](#)

OR (`||`) operators, [81](#)–82, [148](#)

owned string. *See* [String type](#)

ownership

borrowing values, [143](#)–149

cloning, [141](#)–142

copying owned values, [137](#)–138

Copy trait, [137](#)–138, [140](#), [166](#), [212](#), [219](#)

example of, [135](#)

passing values into functions, [142](#)–143

of password, [148](#)

String type, [138](#)–140, [142](#)–145, [150](#), [157](#), [158](#)

three rules of

each piece of data can have only one owner, [136](#), [137](#)

every value has an owner, [136](#)

when the owner goes out of scope, the value is dropped, [136](#)

transferring ownership to new variable, [141](#)

P

package, definition of, [280](#)

package manager

definition of, [24](#)

Maven (Java), [24](#)

npm (Node.js), [24](#)

pip (Python), [24](#)

pages directory, [582](#)

parallelism vs concurrency, [396](#)–397, [403](#)

parse_age() function, [372](#)

ParseIntError, [367](#), [373](#)–374, [379](#)

- `parse()` method, [125](#), [189](#)
- parsing arguments, [452](#)–457
- `PartialEq` trait, [307](#)–308, [310](#), [311](#), [335](#), [336](#)
- `PartialOrd` trait, [337](#)
- PascalCase*, [161](#), [178](#)
- password analyzer, [520](#)–524
- password sharing, [148](#)
- pattern alternative syntax, [97](#)
- pattern-matching
 - `if let` pattern-matching, [192](#)
 - `match` expression, [94](#)–98
 - `with match` keyword, [365](#)–366
 - `Option` enum, [179](#)–180, [184](#)
 - and scope, [98](#)–100
 - user input, [104](#)–106
- `pi` (`PI`) constant, [58](#), [59](#), [68](#)
- pipe symbol (`|`), [97](#)
- platform-dependent, integer, [51](#)
- `Playable` trait, [321](#), [322](#)
- `play()` method, [321](#), [323](#)
- pointer, [135](#), [139](#), [144](#), [149](#), [151](#)–152, [170](#), [230](#), [239](#), [408](#), [409](#), [478](#)–479
- pointer formatter (`{:p}`), [150](#)
- positional parameters, [112](#)
- POST (creating resources), [600](#)
- PowerShell, [18](#)
- prelude, standard library, [58](#)

`prev_hyphen` Boolean, [248](#), [249](#)
`price_per_unit()` function, [364](#)–365
prime-explorer project, [222](#)–224
`primes.iter()` method, [262](#)
`print_converted_temp()` function, [127](#)
`println!` macro, [29](#), [30](#), [46](#)–48, [58](#), [74](#), [79](#), [99](#), [101](#), [137](#), [143](#)–145, [152](#),
[155](#), [205](#), [209](#), [219](#), [256](#), [264](#), [292](#), [382](#)–383, [431](#), [434](#), [438](#)–439
print working directory (`pwd`), [20](#)
`process_file()` function, [463](#)
programming language. *See also* [Rust](#)
 C programming language, [10](#), [13](#), [14](#)
 definition of, [9](#)
 role of, [9](#)–11
progress bars implementation, [487](#), [494](#)–496
`prompt()` function, [194](#)
`ProvidesEnergy` trait, [339](#), [340](#)
`pub` keyword, [284](#), [291](#), [292](#)
`pull-down-cmark`, [539](#), [540](#)
`push_hyphen()` function, [248](#)
`push()` method, [138](#)
`push_str()` method, [138](#)
`PUT` (updating resources), [600](#)
`pwd` (print working directory), [20](#)
Python, [33](#), [225](#)

Q

`question.find("nothinf")`, [183](#)

question variable, [183](#)

R

range syntax, [96](#), [237](#)–238

Rateable trait, [320](#), [322](#)–323

`rate()` method, [320](#), [323](#)

raw strings, [229](#), [582](#)

Raymond, Eric (programmer), [325](#)

Rayon, parallelizing iterations with, [413](#)

reading files, [471](#)–472

- appending text, [475](#)–476

- comma-separated values, [477](#)–479

- increment, [472](#)–473

- one fell swoop, [472](#)

`read_line()` function, [101](#)–103

read-only data segment, [230](#)

read-only text arguments, [113](#)

`read_shopping_list()` function, [472](#)

`read_stock_count()` function, [371](#)

`read_to_string()` function, [472](#)–474, [481](#), [482](#), [484](#)

real-world organization patterns

- organizing larger applications, [298](#)–300

- re-exports, [300](#)–301

- splitting project into multiple crates, [301](#)–302

recoverable error vs unrecoverable error, [362](#)–364

recursion, [122](#)–124

recursive function

`add_one()` function, [123](#)

 examples of, [122](#)–123

`factorial()` function, [123](#)

re-exports, [300](#)–301

references, borrowing, [144](#), [149](#), [156](#)

 dereferencing, [150](#)

 fat pointer, [151](#)–152

 lifetimes, [153](#)

 memory address, [150](#)

 pointer formatter (`{:p}`), [150](#)

 and `println!` macro, [152](#)

`RegisterTemplate`, [631](#)–633

release version, [34](#), [35](#)

remainder expression, [65](#)

remainder operator (%), [64](#)–65, [89](#)

rendering templates, [538](#)–543

 from routes, [624](#)–625

repeat expression syntax, [202](#)

`replace()` function, [157](#), [158](#)

`replace_in_file()` function, [474](#)–475

`replacen()` method, [236](#)

representational state transfer (REST), [594](#)–595

reserved words/keywords, [9](#), [41](#), [110](#)

REST API, [571](#), [593](#)

vs HTML web server, [595](#)

Result enum, [93](#), [188](#)–190, [331](#), [364](#)–365, [427](#)–428

`result.is_err()` method, [189](#)

`result.is_none()` method, [183](#)

`result.is_ok()` method, [189](#)

`result.is_some()` method, [183](#)

ResultString, `io::Error` return type, [193](#)

return keyword, [115](#)

return values

`add_numbers()` function, [115](#)

example, [115](#)

expression-based return, [115](#)

function signature, [115](#)

practical use case for, [116](#)

thin arrow (`>`-greater than symbol) return type, [114](#)

unit type return, [116](#)

revised signature, [349](#)

Rng trait, [297](#)

`.rs` file extension, [26](#), [432](#)

rules and words, human language, [9](#)

rules and words, programming language, [9](#)

`running_total` variable, [65](#)

Rust, [361](#), [465](#), [505](#). *See also specific entries*

- advantages of, [13](#), [14](#)

- vs C, [13](#), [14](#)

- vs C++, [13](#), [14](#)

- code cleaning service, [439](#)

 - Clippy, [441](#)–443

 - `rustfmt`, [439](#)–441

- code debugging, [433](#)

 - VS Code user, [435](#)–437

 - `dbg!` macro, [434](#)–435

 - `println!` macro, [438](#)–439

- code documentation, [443](#)

 - adding comments, [443](#)–445

 - building your program, [446](#)

 - running documentation examples as tests, [445](#)–446

- code testing, [422](#)–423

 - assertion macros, [427](#)

 - code panics, [428](#)

 - integration tests, [432](#)–433

 - library, [431](#)

 - module, [429](#)–431

 - returns `Result`, [427](#)–428

 - writing, [423](#)–426

- command-line tools, [487](#)

 - multicommand file organizer, [496](#)–501

 - progress bars implementation, [487](#), [494](#)–496

 - with subcommands, [488](#)–493

compiling, [32](#)–36

error handling

 pattern matching with `match` keyword, [365](#)–366

 problems with, [372](#)–373

`Result` type, [364](#)–365

error-related helper methods

`unwrap()` and `expect()`, [366](#)–367

`unwrap_or_default()` method, [370](#)–371

`unwrap_or_else()` method, [369](#)–370

`unwrap_or()` method, [367](#)–369

error types, [373](#)–374

`Err(String)`, [377](#)–380

file-handling techniques, [482](#)–485

file system

 fiddling, [470](#)–471

 manipulating paths, [466](#)–470

functions in web browser, [510](#)

 building your project, [512](#)

 creating, [512](#)–513

 projects directory, [511](#)

 spinning up a local server for, [513](#)–514

 WebAssembly development, [514](#)–515

installation

- components installed, [22](#)

- on Linux, [22](#)

- on macOS, [22](#)

- verification, [22](#)–23

- on Windows, [21](#)

JavaScript functions, [515](#)–519

Option, [374](#)–375

- `ok_or()` method, [375](#)–376

- `ok_or_else()` method, [376](#)–377

projects directory, [618](#)

reading files, [471](#)–472

- appending text, [475](#)–476

- comma-separated values, [477](#)–479

- increment, [472](#)–473

- one fell swoop, [472](#)

recoverable error vs unrecoverable error, [362](#)–364

rust-powered password analyzer, [520](#)–524

Rust-to-WASM toolchain

- Node.js, [509](#)

- test project, [510](#)

- `wasm32-unknown-unknown`, [508](#)

- `wasm-pack`, [509](#)

serializing and deserializing data, [480](#)–482

update, [23](#)–24

`wasm-pack` commands, [519](#)–520

WebAssembly, [506](#)–508

web servers, [571](#)

- Axum, [571](#), [575](#)–579

- code, [586](#)–589

- HTTP, [572](#)–576

- logging and debugging, [591](#)–592

- reading HTML from files, [582](#)–584

- return HTML responses, [581](#)–582

- routing works in Axum, [580](#)

- starting the project, [576](#)

- static files, [589](#)–591

- testing, [578](#)–580

- URL query strings, [585](#)–586

- writing handler functions, [580](#)–581

writing files, [471](#)–472

- appending text, [475](#)–476

- comma-separated values, [477](#)–479

- safety, [474](#)–475

rust-analyzer extension installation, [27](#)–28

`rustc: command not found` error, [23](#)

`rustc --version` commands, [23](#)

Rust-flavored templates, [621](#)–622

`rustfmt`, [439](#)–441

Rust multitasking, [395](#)–396

- Arc, sharing ownership, [407](#)–409

- `async/await` keywords, [414](#)–416

- Collatz Conjecture, [416](#)–420

- message passing, [403](#)–404

 - multiple producers, single consumer, [406](#)–407

 - sender and receiver, [404](#)–406

- Mutex, modifying shared data safely with, [410](#)–411

- parallelism vs concurrency, [396](#)–397

- Rayon, parallelizing iterations with, [413](#)

- scoped threads, borrowing data with, [411](#)–413

- spawning threads

 - creating and joining, [397](#)–400

 - moving data, [400](#)–402

Rust-powered password analyzer, [520](#)–524

Rust Rover, [26](#)

Rust statements, [110](#)

Rust-to-WASM toolchain

- Node.js, [509](#)

- test project, [510](#)

- `wasm32-unknown-unknown`, [508](#)

- `wasm-pack`, [509](#)

Rust WASM community, [508](#)
browser functions, [518](#)–519
project building, [512](#)
projects directory, [511](#)
`pub fn hello()`, [511](#)
`wasm-bindgen` library, [511](#)

S

`save_game()` function, [482](#)
`say_hello()` function, [283](#)
scalar value, [49](#)
`scan` subcommand, [499](#)
scope
block scope, [99](#), [100](#), [117](#)
definition of, [98](#)
function scope, [117](#)–118
global scope, [119](#)
memory management, [100](#)
scoped threads, [411](#)–413
`Searchable` trait, [321](#), [323](#)–324
`secret_recipe()` function, [292](#)
`self` keyword, [167](#), [168](#)
semicolon (`;`), [41](#)
Serde, [530](#)
`serde` crate, [480](#)–483, [585](#), [596](#)
`serde_json`, [597](#)

serializing data, [480](#)–482

server-side web app, [617](#)

authentication, [628](#)

session management, [629](#)–630

updating the home page with login status, [637](#)

user login, [634](#)–636

user logout, [636](#)–637

user registration, [630](#)–634

bookmarks

adding, [639](#)–641

deletion, [643](#)

displaying, [637](#)–639

editing, [641](#)–642

database connection pool, [626](#)–628

foundation

database design for LinkLocker, [619](#)–620

database initialization, [620](#)–621

project structure, [618](#)–619

instructions for extending LinkLocker app, [643](#)

static file server, [625](#)–626

templates

layout template, [622](#)–624

rendering templates from routes, [624](#)–625

Rust-flavored templates, [621](#)–622

session management, [619](#), [629](#)–630

`SessionManagerLayer` middleware, [629](#)

sharing. *See* [immutable borrowing](#)

shell, command line, [18](#)

`show_menu()` function, [124](#), [129](#)

Sieve of Eratosthenes (mathematical algorithm), [222](#)

signed integer data type, [51](#)

single arguments, function

- example of, [111](#)

- function set up, [112](#)

- output, [112](#)

single-line comments (`//`), [46](#)

single-line syntax, [78](#)

single quotation marks (`'`), [55](#), [228](#)

sleep apnea, [361](#)

`slice.replace(from, to)` method, [157](#)

slices

- creation of, [207](#)–208

- definition of, [206](#)

- to make functions, [210](#)

- mutable slice

 - data modification, [208](#)–209

 - dereferencing, [209](#)–210

slug, definition of, [246](#)

slugifying a string project

- appending hyphen to slug, [247](#)–248

- avoiding consecutive hyphens addition, [248](#)

- checking for ASCII alphanumeric character, [249](#)

- checking for case of characters, [250](#)

- checking for common punctuation characters, [251](#)

- checking for common separators, [250](#)

- checking for whitespace character, [249](#)

- example, [246](#)

- `main()` function, [252](#)

- `prev_hyphen` Boolean, [248](#), [249](#)

- process steps, [247](#)

- `push_hyphen()` function, [248](#)

- `slugify()` function, [247](#), [248](#), [251](#), [252](#)

- substitutions of accented characters, [248](#)

- `transliterate()` function, [248](#), [251](#)

- `trimmed.chars()` method, [249](#)

smart pointer, [383](#), [408](#), [478](#)–479, [559](#), [562](#)

`sort_by_key()` method, [257](#)–258

`sort_by()` method, [258](#)–259, [275](#)

spawning threads, [403](#)

- creating and joining, [397](#)–400

- moving data, [400](#)–402

`speak()` method, [313](#)–315

`Speak` trait, [313](#)

`split(|c: char| !c.is_alphanumeric() && c != "'')` method, [273](#)

splitting, string

on substring, [246](#)

on whitespace, [244](#)–245

`split_whitespace()` method, [157](#), [244](#)–245

`SqlitePool`, [626](#)–628

`SQLx`, [609](#)–614, [627](#)

`sqlx-cli` command-line tool, [620](#)

standard input (`stdin`), [193](#), [451](#), [457](#)–458

standard library

definition of, [57](#)

modules, [58](#)

prelude, [58](#)

standard output (`stdout`), [46](#)–49, [101](#), [306](#), [382](#), [451](#)

`statements-if-false` statement, [79](#), [81](#)

`statements-if-true` statement, [79](#), [81](#)

static file server, [625](#)–626

`static` keyword, [356](#)–357

`stats` subcommand, [501](#)

`std::f32::consts::PI` constant, [58](#)

`stdin()` function, [101](#)

`std::io` module, [125](#), [129](#), [171](#)

`STOPWORDS` constant, [272](#)–273

`string.clear()` method, [236](#)

`string.contains(substring)` method, [242](#)

`string.ends_with(substring)` method, [242](#)

`string.find(substring)` method, [242](#)

string literals, [55](#)–[56](#), [138](#), [228](#)–[229](#), [272](#)

lifetime, [229](#)

&'static str, [229](#)

storage, [230](#)

&str type, [229](#)

string.make_ascii_lowercase() method, [244](#)

string.make_ascii_uppercase() method, [244](#)

string.pop() method, [236](#)

string.push(char) method, [233](#)

string.push_str(slice) method, [234](#)

(&str) string slice type, [145](#), [151](#), [156](#)–[158](#), [170](#), [227](#), [229](#)

adding to end of string, [234](#)

concatenation operation, [235](#)

deref coercion, [239](#)–[240](#)

get() method, [238](#)

string slicing with range syntax, [237](#)–[238](#)

usage rules, [227](#)–[228](#)

string.split(substring) method, [246](#)

string.split_whitespace() method, [244](#)–[245](#)

string.starts_with(substring) method, [242](#)

strings_you_bow variable, [245](#)

string.to_lowercase() method, [243](#)

string.to_uppercase() method, [243](#)

string.trim_end() method, [241](#)

string.trim_start() method, [241](#)

`String` type, [138](#)–[140](#), [142](#)–[145](#), [150](#), [157](#)–[159](#), [161](#), [170](#), [182](#), [183](#), [185](#), [226](#), [328](#)–[330](#)

- adding character to end of string, [233](#)

- adding `&str` slice to end of string, [234](#)

- case-conversion methods, [243](#)–[244](#)

- `chars()`, [232](#)–[233](#)

- clearing a string, [236](#)

- concatenation operation, [234](#)–[235](#)

- conversion to `&str`, [231](#)

- creation, [230](#)–[231](#)

- `from(str)` method, [231](#)

- `is_empty()` method, [233](#)

- `len()` method, [232](#)

- `new()` method, [230](#)

- replacing text in, [236](#)

- searching and testing, [242](#)–[243](#)

- splitting

 - on substring, [246](#)

 - on whitespace, [244](#)–[245](#)

- `str.to_string()` method, [231](#)

- trimming whitespace from, [240](#)–[241](#)

- usage rules, [227](#)–[228](#)

- `with_capacity(bytes)` method, [231](#)

structs, [306](#), [352](#)–353. *See also* [generics](#)

- adding trait to, [309](#)

- associated functions, [168](#)–169

- `Book` struct, [161](#), [162](#), [169](#)

- creating an instance of, [161](#)–162

- data fields, [161](#)

- definition of, [160](#)

- dot notation, [162](#)–163

- general syntax, [160](#)–161

- method, [166](#)–167

- mutable, [163](#)–164

- `mut` keyword, [163](#)–164

- PascalCase*, [161](#)

- shortcuts

 - field init shorthand, [164](#)–165

 - struct update syntax, [165](#)–166

- teaching new behaviors, [166](#)–168

- Use `&mut self`, [167](#)

- Use `&self`, [167](#)

struct update syntax, [165](#)–166

subcommands, command-line tools with, [487](#)

- adding, [488](#)–490

- `archive` subcommand, [500](#)–501

- `scan` subcommand, [499](#)

- `stats` subcommand, [501](#)

- working with nested, [491](#)–493

substring, [182](#)

sum() function, [211](#)

T

temperature converter project

get_menu_choice() function, [125](#), [126](#), [129](#)

get_temp() function, [126](#), [129](#)

helper functions, [126](#)

main() function, [127](#)–129

match structure, [129](#)

parse() method, [125](#)

print_converted_temp() function, [127](#)

show_menu() function, [124](#), [129](#)

standard formulas, [127](#)

std::io module, [129](#)

trim() method, [125](#)

unwrap_or() method, [125](#), [126](#)

template inheritance pattern, [622](#)

templates

layout template, [622](#)–624

rendering templates from routes, [624](#)–625

Rust-flavored templates, [621](#)–622

terminal, command line, [18](#)

Terminal app launching

- on Linux, [19](#)

- on macOS, [18](#)–19

- on Windows, [18](#)

testing of code, [422](#)–423

- assertion macros, [427](#)

- code panics, [428](#)

- integration tests, [432](#)–433

- library, [431](#)

- module, [429](#)–431

- returns `Result`, [427](#)–428

- writing, [423](#)–426

`tests` directory, [432](#)

text analyzer project, [155](#)–158

text argument, [156](#)

text types

- owned string, [226](#)

- string slice (see [\(&str\)_string slice type](#))

`text_without_dashes` local variable, [157](#), [158](#)

thin arrow (`>`–greater than symbol) return type, [114](#)

thread, [404](#)–414, [419](#)

- creating and joining, [397](#)–400

- moving data, [400](#)–402

- spawning, [403](#)

`thread` module, [397](#), [411](#)

`tidy`, [496](#)–498

timer macro, [391](#)–393

title: title function, [164](#)

to_digit() method, [185](#)

tokenization, [272](#)–274

tokenize() function, [273](#), [277](#)

tokens, definition of, [272](#)

“Tom’s Obvious, Minimal Language” (TOML), [31](#), [477](#), [480](#), [483](#)–485

top_words() function, [275](#), [277](#)

to_vec() method, [212](#), [213](#)

T programming Truth, [422](#)

TrafficLight enum, [178](#), [179](#)

trait bounds

- adding to generic parameter, [336](#)

- clamp, [337](#)

- definition of, [335](#)

- inline bounds, [338](#)

- multiple trait bounds, [336](#)–337

traits

behavioral trait, [305](#)

Copy trait, [137](#)–[138](#), [140](#), [166](#), [212](#), [219](#)

custom trait, [313](#)

Debug trait, [206](#), [310](#), [316](#)

Default trait, [318](#)–[319](#)

definition of, [305](#)

extensibility, [314](#)

implementing Clone trait, [308](#)–[310](#), [312](#)

implementing Display trait, [306](#)–[307](#), [310](#), [316](#)–[318](#)

implementing PartialEq trait, [307](#)–[308](#), [310](#), [311](#)

polymorphism, [313](#)

Rng trait, [297](#)

shared contracts, [314](#)

speak() method, [313](#)–[315](#)

Speak trait, [313](#)

transliterate() function, [251](#)

trimmed.chars() method, [249](#)

trim() method, [103](#), [125](#)

tuple

definition of, [198](#)

destructuring, [200](#)

empty tuple, [199](#)

limitations, [200](#)–201

tuple.i syntax, [198](#)

unit tuple, [199](#)

uses, [199](#)

tuple.i syntax, [198](#)

type inference, [56](#)

type-safe language, [57](#)

U

u8 integers, [51](#), [189](#), [327](#), [329](#)

u32 type, [57](#), [166](#), [423](#), [432](#), [586](#)

UInt8Array, [530](#)

underscore (`_`) separator, [52](#)

Unicode, [54](#)

Unicode Transformation Format-8-bit (UTF-8), [226](#), [227](#), [230](#), [232](#), [237](#), [238](#)

unit tuple, [199](#)

unit type return, [116](#)

unrecoverable error vs recoverable error, [362](#)–364

unsigned integer data type, [51](#)

`unwrap()` method, [366](#)–367

`unwrap_or_default()` method, [370](#)–371

`unwrap_or_else()` method, [369](#)–370

`unwrap_or()` method/`unwrap_or_default()` method, [125](#), [126](#), [173](#), [190](#), [194](#), [367](#)–369

`unwrap_or()` Option method, [186](#)

`UpdateBook`, [601](#)

UpperCamelCase. See [PascalCase](#)

URL query strings, [585](#)–586

`use` keyword, [59](#), [290](#), [293](#)–294, [297](#)

user account directory (~), [28](#)

`user_interface` module, [285](#), [286](#), [288](#)–290, [294](#)

user login, [634](#)–636

user logout, [636](#)–637

user registration, [630](#)–634

`usize` type, [51](#)

`utils.rs` file, [299](#)

V

variable-length encoding, [226](#)

variables, [61](#), [135](#), [141](#), [144](#). *See also specific variables*

- choice variable, [189](#)

- declarations, [41](#), [279](#)

- definition of, [40](#)

- immutable variable, [44](#), [56](#)

- memory locations, [42](#)–43

- mutable variable, [45](#)

- naming rules, [41](#), [42](#)

- question variable, [183](#)

- scope of, [98](#)

variable shadowing, [103](#)

`vec!` macro, [213](#), [382](#)–383

`vector` (`Vec<T>`), [170](#), [529](#)

- adding and removing elements, [214](#)–216

- advantages of, [214](#)

- creation

 - from array, [212](#)–213

 - new vector, [211](#)

 - using `vec!` macro, [213](#)

- definition of, [211](#)

- elements, accessing of, [213](#)–214

- empty vector, [211](#)

- indexing operator, [213](#)–214

- iteration, [216](#)–217

- mutable vector, [214](#)–215

- numeric, [530](#)

`vector.get()` method, [214](#)

version

1.89.0, [23](#)–24

1.90, [24](#)

`v.insert(value, index)` method, [215](#)

`v.is_empty()` method, [215](#)

Visual Studio, [21](#)

Visual Studio Code

compiling and running from, [35](#)

installation, [27](#)–28

`v.push(value)` method, [215](#)

`v.remove(index)` method, [215](#)

VS Code user, [424](#)–426, [433](#), [435](#)–437

W

`walkdir`, [497](#)–501

WASM. *See* [WebAssembly](#)

`wasm32-unknown-unknown`, [508](#)

`wasm-bindgen-futures`, [539](#), [541](#)

`wasm-bindgen` library, [507](#), [526](#)–527

`wasm-pack`, [507](#), [509](#)

commands, [519](#)–520

installation, [509](#)

WebAssembly (WASM), [506](#)–508

- browser integration, [520](#)

- development, [513](#)–515

- module size, [507](#)

- wasm32 target, [510](#)

- wasm-bindgen, [510](#)

- web page creation, [512](#)–513

web servers, [571](#)

- Axum, [571](#), [575](#)–579

- code, [586](#)–589

- HTTP, [572](#)–576

- logging and debugging, [591](#)–592

- reading HTML from files, [582](#)–584

- return HTML responses, [581](#)–582

- routing works in Axum, [580](#)

- starting the project, [576](#)

- static files, [589](#)–591

- testing, [578](#)–580

- URL query strings, [585](#)–586

- writing handler functions, [580](#)–581

web_sys, [539](#)

while loops, [86](#)–87, [224](#)

whitespace

definition of, [240](#)

example, [241](#)

output, [241](#)

splitting, string on, [244](#)–245

`string.trim_end()` method, [241](#)

`string.trim()` method, [241](#)

`string.trim_start()` method, [241](#)

width variable, [99](#)

wildcard character (`_`), [94](#), [98](#), [102](#), [189](#)

Windows

Rust installation, [21](#)

Terminal app launching, [18](#)

`withdraw()` method, [172](#), [191](#)

`word_counts()` function, [274](#), [275](#)

word-replacer tool, [461](#)–464

words and rules, human language, [9](#)

words and rules, programming language, [9](#)

`words.iter_mut()` method, [263](#)

Word Wrangler project

counting word frequencies, [274](#)

`longest_word()` function, [275](#)–276

`main()` function, [276](#)–277

tokenization, [272](#)–274

`top_words()` function, [275](#)

- writing files, [471](#)–472
 - appending text, [475](#)–476
 - comma-separated values, [477](#)–479
 - safety, [474](#)–475

X

- XOR operator, [148](#)

Y

- YAML, [480](#), [595](#)

Z

- zagging, [422](#)
- zero-copy parser approach, [357](#)–360
- `zero_out()` function, [209](#)
- zigging, [422](#)

About the Author

Paul McFedries is the president of Logophilia Limited, a technical writing company, and has worked with computers large and small since 1975. While now primarily a writer, Paul has worked as a programmer, consultant, database developer, and website developer. He has written more than 110 books that have sold over four million copies worldwide. Paul invites everyone to drop by his personal website at <https://paulmcfedries.com>, or to follow him on X (@paulmcf), Facebook (www.facebook.com/PaulMcFedries/), or LinkedIn (<https://www.linkedin.com/in/paulmcfedries/>).

Dedication

To Karen and Chase, of course.

Author's Acknowledgments

If we're ever at the same cocktail party and you overhear me saying something like "I wrote a book," I hereby give you permission to wag your finger at me and say "Tsk, tsk." Why the scolding? Because although I did write this book's text and take its screenshots, those represent only a part of what constitutes a book. The rest of it is brought to you by the dedication and professionalism of Wiley's editing, graphics, and production teams, who toiled long and hard to turn my text and images into an actual book.

I offer my heartfelt thanks to everyone at Wiley who made this book possible, but I'd like to extend some special thank-you's to the folks I worked with directly: executive editor Steve Hayes and project editor and copy editor Susan Pink.

Publisher's Acknowledgments

Executive Editor: Steve Hayes

Senior Editorial Assistant: Hanna Sytsma

Project and Copy Editor: Susan Pink

Managing Editor: Murari Mukundan

Production Editor: Magesh Elangovan

Cover Image: © ABCDstock/stock.adobe.com

Take Dummies with you everywhere you go!



Go to our [Website](#)



Like us on [Facebook](#)



Follow us on [Twitter](#)



Watch us on [YouTube](#)



Join us on [LinkedIn](#)



Pin us on [Pinterest](#)



Subscribe to our [newsletter](#)



Create your own [Dummies book cover](#)

**for
dummies®**
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.