PRAISE FOR   
THE Rust programming language, 2nd edition

“The Rust Programming Language has always been, and continues to be, the first place I point anyone looking to learn Rust. It’s referred to simply as “The Book,” and with good reason . . . The 2nd edition is a reflection of the ongoing love for the book in the community, and ensures that it will remain a solid introduction to the best Rust has to offer for years to come.”

—  
Jon Gjengset, author of Rust for Rustaceans

“The Rust Programming Language, 2nd Edition has been improved, refined, and still remains \*the book\* to get started with learning Rust. Whether an experienced or a new Rustacean it’s an invaluable resource on how Rust works.”

—  
Michael Gattozzi, Senior Software Engineer at Fastly

“A great resource for mastering Rust’s fundamentals. . . . If you’re thinking about seriously learning Rust, this book is for you.”

—  
Adam Vartanian, Engineering Manager at Cord

“From a person who read the first edition front to back, the second edition of The Rust Programming Language met and exceeded my expectations! It’s well-written and formatted to help introduce the reader to all the concepts of Rust. . . . [it’s] an excellent addition to a programmer’s personal library.”

—  
Jared Wolff, Owner, Circuit Dojo LLC

The Rust Programming Language

2nd Edition

by Steve Klabnik, Carol Nichols, and Chris Krycho,   
with contributions from   
the Rust Community

THE RUST PROGRAMMING LANGUAGE, 2ND EDITION. Copyright © 2023 by the Rust Foundation and the Rust Project Developers.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed in the United States of America

Second printing

27 26 25 24 23     2 3 4 5 6

ISBN-13: 978-1-7185-0310-6 (print)  
ISBN-13: 978-1-7185-0311-3 (ebook)

Publisher: William Pollock  
Managing Editor: Jill Franklin  
Production Manager: Sabrina Plomitallo-González  
Production Editors: Jennifer Kepler and Katrina Horlbeck Olsen  
Developmental Editor: Liz Chadwick  
Cover Illustration: Karen Rustad Tölva  
Interior Design: Octopod Studios  
Technical Reviewer: JT  
Copyeditor: Audrey Doyle  
Compositor: Jeff Lytle, Happenstance Type-O-Rama  
Proofreader: Liz Wheeler

For information on distribution, bulk sales, corporate sales, or translations, please contact No Starch Press, Inc. directly at info@nostarch.com or:

No Starch Press, Inc.  
245 8th Street, San Francisco, CA 94103  
phone: 1.415.863.9900  
www.nostarch.com

The Library of Congress has catalogued the first edition as follows:

Names: Klabnik, Steve, author. | Nichols, Carol, 1983- eauthor.  
Title: The Rust programming language / by Steve Klabnik and Carol Nichols ; with contributions from   
   the Rust Community.  
Description: San Francisco : No Starch Press, Inc., 2018. | Includes index. Identifiers: LCCN   
   2018014097 (print) | LCCN 2018019844 (ebook) | ISBN 9781593278519 (epub) | ISBN 1593278519 (epub)   
   | ISBN 9781593278281 (paperback) | ISBN 1593278284 (paperback)  
Subjects: LCSH: Rust (Computer programming language) | BISAC: COMPUTERS / Programming / Open Source.  
    | COMPUTERS / Programming Languages / General. | COMPUTERS / Programming / General.  
Classification: LCC QA76.73.R87 (ebook) | LCC QA76.73.R87 K53 2018 (print) | DDC 005.13/3--dc23  
LC record available at https://lccn.loc.gov/2018014097

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the authors nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Authors

Steve Klabnik was the lead for the Rust documentation team and was one of Rust’s core developers. A frequent speaker and a prolific open source contributor, he previously worked on projects such as Ruby and Ruby on Rails.

Carol Nichols is a member of the Rust Crates.io Team and a former member of the Rust Core Team. She’s a co-founder of Integer 32, LLC, the world’s first Rust-focused software consultancy. Nichols has also organized the Rust Belt Rust Conference.

Chris Krycho is a software engineering leader with experience in avionics, developer tools, web frameworks, and more. In addition to his open source software contributions and regular public speaking, he created the *New Rustacean* podcast (2015–2019).

About the Technical Reviewer

JT is a Rust core team member and the co-creator of the Rust error message format, Rust Language Server (RLS), and Nushell. They first started using Rust in 2011, and in 2016 joined Mozilla to work on Rust full-time, helping to shape its direction for widespread use. These days, they are a freelance Rust trainer and advocate for safe systems programming.

Brief Contents

Foreword xix

Preface xxi

Acknowledgments xxiii

Introduction xxv

Chapter 1: Getting Started 1

Chapter 2: Programming a Guessing Game 13

Chapter 3: Common Programming Concepts 31

Chapter 4: Understanding Ownership 59

Chapter 5: Using Structs to Structure Related Data 85

Chapter 6: Enums and Pattern Matching 103

Chapter 7: Managing Growing Projects with Packages, Crates, and Modules 119

Chapter 8: Common Collections 141

Chapter 9: Error Handling 161

Chapter 10: Generic Types, Traits, and Lifetimes 181

Chapter 11: Writing Automated Tests 215

Chapter 12: An I/O Project: Building a Command Line Program 243

Chapter 13: Functional Language Features: Iterators and Closures 273

Chapter 14: More About Cargo and Crates.io 295

Chapter 15: Smart Pointers 315

Chapter 16: Fearless Concurrency 353

Chapter 17: Object-Oriented Programming Features 375

Chapter 18: Patterns and Matching 397

Chapter 19: Advanced Features 419

Chapter 20: Final Project: Building a Multithreaded Web Server 459

Appendix A: Keywords 495

Appendix B: Operators and Symbols 499

Appendix C: Derivable Traits 507

Appendix D: Useful Development Tools 511

Appendix E: Editions 515

Index 517

Contents in Detail

Foreword xix

Preface xxi

Acknowledgments xxiii

Introduction xxv

Who Rust Is For xxvi

Teams of Developers xxvi

Students xxvi

Companies xxvi

Open Source Developers xxvii

People Who Value Speed and Stability xxvii

Who This Book Is For xxvii

How to Use This Book xxvii

Resources and How to Contribute to This Book xxix

1  
Getting Started 1

Installation 1

Installing rustup on Linux or macOS 2

Installing rustup on Windows 3

Troubleshooting 3

Updating and Uninstalling 4

Local Documentation 4

Hello, World! 4

Creating a Project Directory 4

Writing and Running a Rust Program 5

Anatomy of a Rust Program 5

Compiling and Running Are Separate Steps 6

Hello, Cargo! 7

Creating a Project with Cargo 8

Building and Running a Cargo Project 9

Building for Release 11

Cargo as Convention 11

Summary 11

2  
Programming a Guessing Game 13

Setting Up a New Project 14

Processing a Guess 15

Storing Values with Variables 16

Receiving User Input 16

Handling Potential Failure with Result 17

Printing Values with println! Placeholders 18

Testing the First Part 19

Generating a Secret Number 19

Using a Crate to Get More Functionality 19

Generating a Random Number 22

Comparing the Guess to the Secret Number 23

Allowing Multiple Guesses with Looping 26

Quitting After a Correct Guess 28

Handling Invalid Input 28

Summary 30

3  
Common Programming Concepts 31

Variables and Mutability 32

Constants 33

Shadowing 34

Data Types 36

Scalar Types 36

Compound Types 40

Functions 43

Parameters 44

Statements and Expressions 46

Functions with Return Values 47

Comments 49

Control Flow 50

if Expressions 50

Repetition with Loops 54

Summary 58

4  
Understanding Ownership 59

What Is Ownership? 59

Ownership Rules 61

Variable Scope 61

The String Type 62

Memory and Allocation 63

Ownership and Functions 68

Return Values and Scope 69

References and Borrowing 71

Mutable References 73

Dangling References 75

The Rules of References 77

The Slice Type 77

String Slices 79

Other Slices 83

Summary 83

5  
Using Structs to Structure Related Data 85

Defining and Instantiating Structs 86

Using the Field Init Shorthand 87

Creating Instances from Other Instances with Struct Update Syntax 88

Using Tuple Structs Without Named Fields to Create Different Types 89

Unit-Like Structs Without Any Fields 89

An Example Program Using Structs 91

Refactoring with Tuples 92

Refactoring with Structs: Adding More Meaning 93

Adding Useful Functionality with Derived Traits 94

Method Syntax 97

Defining Methods 97

Methods with More Parameters 99

Associated Functions 101

Multiple impl Blocks 102

Summary 102

6  
Enums and Pattern Matching 103

Defining an Enum 103

Enum Values 104

The Option Enum and Its Advantages Over Null Values 108

The match Control Flow Construct 110

Patterns That Bind to Values 112

Matching with Option<T> 113

Matches Are Exhaustive 114

Catch-All Patterns and the \_ Placeholder 115

Concise Control Flow with if let 116

Summary 117

7  
Managing Growing Projects   
with Packages, Crates, and Modules 119

Packages and Crates 120

Defining Modules to Control Scope and Privacy 123

Paths for Referring to an Item in the Module Tree 125

Exposing Paths with the pub Keyword 127

Starting Relative Paths with super 130

Making Structs and Enums Public 130

Bringing Paths into Scope with the use Keyword 132

Creating Idiomatic use Paths 133

Providing New Names with the as Keyword 135

Re-exporting Names with pub use 135

Using External Packages 136

Using Nested Paths to Clean Up Large use Lists 137

The Glob Operator 138

Separating Modules into Different Files 138

Summary 140

8  
Common Collections 141

Storing Lists of Values with Vectors 142

Creating a New Vector 142

Updating a Vector 142

Reading Elements of Vectors 143

Iterating Over the Values in a Vector 145

Using an Enum to Store Multiple Types 145

Dropping a Vector Drops Its Elements 146

Storing UTF-8 Encoded Text with Strings 147

What Is a String? 147

Creating a New String 147

Updating a String 148

Indexing into Strings 151

Slicing Strings 152

Methods for Iterating Over Strings 153

Strings Are Not So Simple 154

Storing Keys with Associated Values in Hash Maps 154

Creating a New Hash Map 154

Accessing Values in a Hash Map 155

Hash Maps and Ownership 156

Updating a Hash Map 156

Hashing Functions 158

Summary 159

9  
Error Handling 161

Unrecoverable Errors with panic! 162

Recoverable Errors with Result 165

Matching on Different Errors 166

Propagating Errors 169

To panic! or Not to panic! 175

Examples, Prototype Code, and Tests 175

Cases in Which You Have More Information Than the Compiler 176

Guidelines for Error Handling 176

Creating Custom Types for Validation 177

Summary 179

10  
Generic Types, Traits, and Lifetimes 181

Removing Duplication by Extracting a Function 182

Generic Data Types 184

In Function Definitions 184

In Struct Definitions 187

In Enum Definitions 188

In Method Definitions 189

Performance of Code Using Generics 191

Traits: Defining Shared Behavior 192

Defining a Trait 192

Implementing a Trait on a Type 193

Default Implementations 195

Traits as Parameters 197

Returning Types That Implement Traits 199

Using Trait Bounds to Conditionally Implement Methods 200

Validating References with Lifetimes 201

Preventing Dangling References with Lifetimes 201

The Borrow Checker 202

Generic Lifetimes in Functions 203

Lifetime Annotation Syntax 205

Lifetime Annotations in Function Signatures 205

Thinking in Terms of Lifetimes 208

Lifetime Annotations in Struct Definitions 209

Lifetime Elision 209

Lifetime Annotations in Method Definitions 212

The Static Lifetime 212

Generic Type Parameters, Trait Bounds, and Lifetimes Together 213

Summary 213

11  
Writing Automated Tests 215

How to Write Tests 216

The Anatomy of a Test Function 216

Checking Results with the assert! Macro 219

Testing Equality with the assert\_eq! and assert\_ne! Macros 222

Adding Custom Failure Messages 224

Checking for Panics with should\_panic 226

Using Result<T, E> in Tests 230

Controlling How Tests Are Run 230

Running Tests in Parallel or Consecutively 231

Showing Function Output 231

Running a Subset of Tests by Name 233

Ignoring Some Tests Unless Specifically Requested 235

Test Organization 236

Unit Tests 236

Integration Tests 237

Summary 242

12  
An I/O Project: Building a Command Line Program 243

Accepting Command Line Arguments 244

Reading the Argument Values 244

Saving the Argument Values in Variables 246

Reading a File 247

Refactoring to Improve Modularity and Error Handling 248

Separation of Concerns for Binary Projects 249

Fixing the Error Handling 253

Extracting Logic from main 256

Splitting Code into a Library Crate 258

Developing the Library’s Functionality with Test-Driven Development 259

Writing a Failing Test 260

Writing Code to Pass the Test 262

Working with Environment Variables 265

Writing a Failing Test for the Case-Insensitive Search Function 265

Implementing the search\_case\_insensitive Function 266

Writing Error Messages to Standard Error Instead of Standard Output 270

Checking Where Errors Are Written 270

Printing Errors to Standard Error 271

Summary 272

13  
Functional Language Features:   
Iterators and Closures 273

Closures: Anonymous Functions That Capture Their Environment 274

Capturing the Environment with Closures 274

Closure Type Inference and Annotation 276

Capturing References or Moving Ownership 278

Moving Captured Values Out of Closures and the Fn Traits 280

Processing a Series of Items with Iterators 284

The Iterator Trait and the next Method 285

Methods That Consume the Iterator 286

Methods That Produce Other Iterators 286

Using Closures That Capture Their Environment 287

Improving Our I/O Project 289

Removing a clone Using an Iterator 289

Making Code Clearer with Iterator Adapters 292

Choosing Between Loops and Iterators 292

Comparing Performance: Loops vs. Iterators 293

Summary 294

14  
More About Cargo and Crates.io 295

Customizing Builds with Release Profiles 296

Publishing a Crate to Crates.io 297

Making Useful Documentation Comments 297

Exporting a Convenient Public API with pub use 300

Setting Up a Crates.io Account 304

Adding Metadata to a New Crate 304

Publishing to Crates.io 305

Publishing a New Version of an Existing Crate 306

Deprecating Versions from Crates.io with cargo yank 306

Cargo Workspaces 307

Creating a Workspace 307

Creating the Second Package in the Workspace 308

Installing Binaries with cargo install 312

Extending Cargo with Custom Commands 313

Summary 313

15  
Smart Pointers 315

Using Box<T> to Point to Data on the Heap 316

Using Box<T> to Store Data on the Heap 317

Enabling Recursive Types with Boxes 317

Treating Smart Pointers Like Regular References with Deref 321

Following the Pointer to the Value 322

Using Box<T> Like a Reference 323

Defining Our Own Smart Pointer 323

Implementing the Deref Trait 324

Implicit Deref Coercions with Functions and Methods 325

How Deref Coercion Interacts with Mutability 326

Running Code on Cleanup with the Drop Trait 327

Rc<T>, the Reference Counted Smart Pointer 330

Using Rc<T> to Share Data 331

Cloning an Rc<T> Increases the Reference Count 333

RefCell<T> and the Interior Mutability Pattern 334

Enforcing Borrowing Rules at Runtime with RefCell<T> 334

Interior Mutability: A Mutable Borrow to an Immutable Value 335

Allowing Multiple Owners of Mutable Data with Rc<T> and RefCell<T> 342

Reference Cycles Can Leak Memory 343

Creating a Reference Cycle 343

Preventing Reference Cycles Using Weak<T> 346

Summary 351

16  
Fearless Concurrency 353

Using Threads to Run Code Simultaneously 354

Creating a New Thread with spawn 355

Waiting for All Threads to Finish Using join Handles 356

Using move Closures with Threads 358

Using Message Passing to Transfer Data Between Threads 361

Channels and Ownership Transference 363

Sending Multiple Values and Seeing the Receiver Waiting 364

Creating Multiple Producers by Cloning the Transmitter 365

Shared-State Concurrency 367

Using Mutexes to Allow Access to Data from One Thread at a Time 367

Similarities Between RefCell<T>/Rc<T> and Mutex<T>/Arc<T> 372

Extensible Concurrency with the Send and Sync Traits 373

Allowing Transference of Ownership Between Threads with Send 373

Allowing Access from Multiple Threads with Sync 373

Implementing Send and Sync Manually Is Unsafe 374

Summary 374

17  
Object-Oriented Programming Features 375

Characteristics of Object-Oriented Languages 376

Objects Contain Data and Behavior 376

Encapsulation That Hides Implementation Details 376

Inheritance as a Type System and as Code Sharing 378

Using Trait Objects That Allow for Values of Different Types 379

Defining a Trait for Common Behavior 380

Implementing the Trait 381

Trait Objects Perform Dynamic Dispatch 384

Implementing an Object-Oriented Design Pattern 384

Defining Post and Creating a New Instance in the Draft State 386

Storing the Text of the Post Content 387

Ensuring the Content of a Draft Post Is Empty 387

Requesting a Review Changes the Post’s State 388

Adding approve to Change the Behavior of content 389

Trade-offs of the State Pattern 392

Summary 396

18  
Patterns and Matching 397

All the Places Patterns Can Be Used 398

match Arms 398

Conditional if let Expressions 399

while let Conditional Loops 400

for Loops 400

let Statements 401

Function Parameters 402

Refutability: Whether a Pattern Might Fail to Match 403

Pattern Syntax 405

Matching Literals 405

Matching Named Variables 405

Multiple Patterns 406

Matching Ranges of Values with ..= 406

Destructuring to Break Apart Values 407

Ignoring Values in a Pattern 411

Extra Conditionals with Match Guards 415

@ Bindings 417

Summary 418

19  
Advanced Features 419

Unsafe Rust 420

Unsafe Superpowers 420

Dereferencing a Raw Pointer 421

Calling an Unsafe Function or Method 423

Accessing or Modifying a Mutable Static Variable 427

Implementing an Unsafe Trait 429

Accessing Fields of a Union 429

When to Use Unsafe Code 429

Advanced Traits 430

Associated Types 430

Default Generic Type Parameters and Operator Overloading 431

Disambiguating Between Methods with the Same Name 433

Using Supertraits 437

Using the Newtype Pattern to Implement External Traits 439

Advanced Types 440

Using the Newtype Pattern for Type Safety and Abstraction 440

Creating Type Synonyms with Type Aliases 440

The Never Type That Never Returns 443

Dynamically Sized Types and the Sized Trait 444

Advanced Functions and Closures 446

Function Pointers 446

Returning Closures 448

Macros 449

The Difference Between Macros and Functions 449

Declarative Macros with macro\_rules! for General Metaprogramming 449

Procedural Macros for Generating Code from Attributes 451

How to Write a Custom derive Macro 452

Attribute-Like Macros 457

Function-Like Macros 458

Summary 458

20  
Final Project: Building a Multithreaded Web Server 459

Building a Single-Threaded Web Server 460

Listening to the TCP Connection 461

Reading the Request 462

A Closer Look at an HTTP Request 464

Writing a Response 465

Returning Real HTML 466

Validating the Request and Selectively Responding 467

A Touch of Refactoring 469

Turning Our Single-Threaded Server into a Multithreaded Server 471

Simulating a Slow Request 471

Improving Throughput with a Thread Pool 472

Graceful Shutdown and Cleanup 487

Implementing the Drop Trait on ThreadPool 487

Signaling to the Threads to Stop Listening for Jobs 490

Summary 493

A  
Keywords 495

Keywords Currently in Use 495

Keywords Reserved for Future Use 497

Raw Identifiers 497

B  
Operators and Symbols 499

Operators 499

Non-operator Symbols 502

C  
Derivable Traits 507

Debug for Programmer Output 508

PartialEq and Eq for Equality Comparisons 508

PartialOrd and Ord for Ordering Comparisons 509

Clone and Copy for Duplicating Values 509

Hash for Mapping a Value to a Value of Fixed Size 510

Default for Default Values 510

D  
Useful Development Tools 511

Automatic Formatting with rustfmt 511

Fix Your Code with rustfix 512

More Lints with Clippy 513

IDE Integration Using rust-analyzer 514

E  
Editions 515

Index 517

Foreword

It wasn’t always so clear, but the Rust programming language is fundamentally about empowerment: no matter what kind of code you are writing now, Rust empowers you to reach further, to program with confidence in a wider variety of domains than you did before.

Take, for example, “systems-level” work that deals with low-level details of memory management, data representation, and concurrency. Traditionally, this realm of programming is seen as arcane, accessible to only a select few who have devoted the necessary years to learning it to avoid its infamous pitfalls. And even those who practice it do so with caution, lest their code be open to exploits, crashes, or corruption.

Rust breaks down these barriers by eliminating the old pitfalls and providing a friendly, polished set of tools to help you along the way. Programmers who need to “dip down” into lower-level control can do so with Rust, without taking on the customary risk of crashes or security holes and without having to learn the fine points of a fickle toolchain. Better yet, the language is designed to guide you naturally toward reliable code that is efficient in terms of speed and memory usage.

Programmers who are already working with low-level code can use Rust to raise their ambitions. For example, introducing parallelism in Rust is a relatively low-risk operation: the compiler will catch the classical mistakes for you. And you can tackle more aggressive optimizations in your code with the confidence that you won’t accidentally introduce crashes or vulnerabilities.

But Rust isn’t limited to low-level systems programming. It’s expressive and ergonomic enough to make CLI apps, web servers, and many other kinds of code quite pleasant to write—you’ll find simple examples later in the book. Working with Rust allows you to build skills that transfer from one domain to another; you can learn Rust by writing a web app, then apply those same skills to target your Raspberry Pi.

This book fully embraces the potential of Rust to empower its users. It’s a friendly and approachable text intended to help you level up not just your knowledge of Rust, but also your reach and confidence as a programmer in general. So dive in, get ready to learn—and welcome to the Rust community!

Nicholas Matsakis and Aaron Turon

Preface

This version of the text assumes you’re using Rust 1.85.0 (released 2025-02-17) or later with edition = "2024" in the Cargo.toml file of all projects to configure them to use Rust 2024 edition idioms. See “Installation” on page 1 for instructions on installing or updating Rust, and see Appendix E for information on editions.

The 2024 edition of the Rust language includes a number of improvements that make Rust more ergonomic and that correct some inconsistencies. On top of a general update to reflect these improvements, this rendition of the book has a number of improvements to address specific feedback:

A new Chapter 17 introduces async programming in Rust, including async and await along with the Future and Stream types. Later chapters have been lightly updated and renumbered to account for this update.

Chapter 20 (Chapter 19 in previous editions) now includes an introduction to Miri, Rust’s dynamic analysis tool for unsafe code. It also includes updates for some of the more significant updates to Rust in the 2024 Edition.

We fixed a number of small errors and imprecise wording throughout the book. Thank you to the readers who reported them!

Note that any code from earlier renditions of this book that compiled will continue to compile with the relevant edition in the project’s Cargo.toml, even as you update the Rust compiler version you’re using. That’s Rust’s backward-compatibility guarantees at work!

Acknowledgments

We would like to thank everyone who has worked on the Rust language for creating an amazing language worth writing a book about. We’re grateful to everyone in the Rust community for being welcoming and creating an environment worth welcoming more folks into.

We’re especially thankful for everyone who read early versions of this book online and provided feedback, bug reports, and pull requests. Special thanks to Eduard-Mihai Burtescu, Alex Crichton, and JT for providing technical review, and to Karen Rustad Tölva for the cover art. Thank you to our team at No Starch, including Bill Pollock, Liz Chadwick, and Janelle Ludowise, for improving this book and bringing it to print.

Carol is grateful for the opportunity to work on this book. She thanks her family for their constant love and support, especially her husband, Jake Goulding, and her daughter, Vivian.

Chris is profoundly grateful to Tim McNamara, Will Crichton, James Munns, Nick Cameron, and Tyler Mandry for dedicated feedback on the new chapter on async programming in Rust. He would also like to thank Integer32 for making this update for the 2024 Edition happen. Finally, and most of all, he is thankful to his wife Jaimie and his daughters Elayne and Katherine for supporting him, including many a “How is the book going, daddy?” over family dinners.

Introduction

Welcome to The Rust Programming Language, an introductory book about Rust. The Rust programming language helps you write faster, more reliable software. High-level ergonomics and low-level control are often at odds in programming language design; Rust challenges that conflict. Through balancing powerful technical capacity and a great developer experience, Rust gives you the option to control low-level details (such as memory usage) without all the hassle traditionally associated with such control.

Who Rust Is For

Rust is ideal for many people for a variety of reasons. Let’s look at a few of the most important groups.

Teams of Developers

Rust is proving to be a productive tool for collaborating among large teams of developers with varying levels of systems programming knowledge. Low-level code is prone to various subtle bugs, which in most other languages can only be caught through extensive testing and careful code review by experienced developers. In Rust, the compiler plays a gatekeeper role by refusing to compile code with these elusive bugs, including concurrency bugs. By working alongside the compiler, the team can spend their time focusing on the program’s logic rather than chasing down bugs.

Rust also brings contemporary developer tools to the systems programming world:

Cargo, the included dependency manager and build tool, makes adding, compiling, and managing dependencies painless and consistent across the Rust ecosystem.

The rustfmt formatting tool ensures a consistent coding style across developers.

The Rust Language Server powers integrated development environment (IDE) integration for code completion and inline error messages.

By using these and other tools in the Rust ecosystem, developers can be productive while writing systems-level code.

Students

Rust is for students and those who are interested in learning about systems concepts. Using Rust, many people have learned about topics like operating systems development. The community is very welcoming and happy to answer students’ questions. Through efforts such as this book, the Rust teams want to make systems concepts more accessible to more people, especially those new to programming.

Companies

Hundreds of companies, large and small, use Rust in production for a variety of tasks, including command line tools, web services, DevOps tooling, embedded devices, audio and video analysis and transcoding, cryptocurrencies, bioinformatics, search engines, Internet of Things applications, machine learning, and even major parts of the Firefox web browser.

Open Source Developers

Rust is for people who want to build the Rust programming language, community, developer tools, and libraries. We’d love to have you contribute to the Rust language.

People Who Value Speed and Stability

Rust is for people who crave speed and stability in a language. By speed, we mean both how quickly Rust code can run and the speed at which Rust lets you write programs. The Rust compiler’s checks ensure stability through feature additions and refactoring. This is in contrast to the brittle legacy code in languages without these checks, which developers are often afraid to modify. By striving for zero-cost abstractions—higher-level features that compile to lower-level code as fast as code written manually—Rust endeavors to make safe code be fast code as well.

The Rust language hopes to support many other users as well; those mentioned here are merely some of the biggest stakeholders. Overall, Rust’s greatest ambition is to eliminate the trade-offs that programmers have accepted for decades by providing safety and productivity, speed and ergonomics. Give Rust a try and see if its choices work for you.

Who This Book Is For

This book assumes that you’ve written code in another programming language, but doesn’t make any assumptions about which one. We’ve tried to make the material broadly accessible to those from a wide variety of programming backgrounds. We don’t spend a lot of time talking about what programming is or how to think about it. If you’re entirely new to programming, you would be better served by reading a book that specifically provides an introduction to programming.

How to Use This Book

In general, this book assumes that you’re reading it in sequence from front to back. Later chapters build on concepts in earlier chapters, and earlier chapters might not delve into details on a particular topic but will revisit the topic in a later chapter.

You’ll find two kinds of chapters in this book: concept chapters and project chapters. In concept chapters, you’ll learn about an aspect of Rust.   
In project chapters, we’ll build small programs together, applying what you’ve learned so far. Chapter 2, Chapter 12, and Chapter 21 are project chapters; the rest are concept chapters.

Chapter 1 explains how to install Rust, how to write a “Hello, world!” program, and how to use Cargo, Rust’s package manager and build tool. Chapter 2 is a hands-on introduction to writing a program in Rust, having you build up a number-guessing game. Here, we cover concepts at a high level, and later chapters will provide additional detail. If you want to get your hands dirty right away, Chapter 2 is the place for that. Chapter 3 covers Rust features that are similar to those of other programming languages, and in Chapter 4 you’ll learn about Rust’s ownership system. If you’re a particularly meticulous learner who prefers to learn every detail before moving on to the next, you might want to skip Chapter 2 and go straight to Chapter 3, returning to Chapter 2 when you’d like to work on a project applying the details you’ve learned.

Chapter 5 discusses structs and methods, and Chapter 6 covers enums, match expressions, and the if let control flow construct. You’ll use structs and enums to make custom types in Rust.

In Chapter 7, you’ll learn about Rust’s module system and about privacy rules for organizing your code and its public application programming interface (API). Chapter 8 discusses some common collection data structures that the standard library provides, such as vectors, strings, and hash maps. Chapter 9 explores Rust’s error-handling philosophy and techniques.

Chapter 10 digs into generics, traits, and lifetimes, which give you the power to define code that applies to multiple types. Chapter 11 is all about testing, which even with Rust’s safety guarantees is necessary to ensure your program’s logic is correct. In Chapter 12, we’ll build our own implementation of a subset of functionality from the grep command line tool that searches for text within files. For this, we’ll use many of the concepts we discussed in the previous chapters.

Chapter 13 explores closures and iterators: features of Rust that come from functional programming languages. In Chapter 14, we’ll examine Cargo in more depth and talk about best practices for sharing your libraries with others. Chapter 15 discusses smart pointers that the standard library provides and the traits that enable their functionality.

In Chapter 16, we’ll walk through different models of concurrent programming and talk about how Rust helps you program in multiple threads fearlessly. In Chapter 17, we build on that by exploring Rust’s async and await syntax, along with tasks, futures, and streams, and the lightweight concurrency model they enable.

Chapter 18 looks at how Rust idioms compare to object-oriented programming principles you might be familiar with. Chapter 19 is a reference on patterns and pattern matching, which are powerful ways of expressing ideas throughout Rust programs. Chapter 20 contains a smorgasbord of advanced topics of interest, including unsafe Rust, macros, and more about lifetimes, traits, types, functions, and closures.

In Chapter 21, we’ll complete a project in which we’ll implement a low-level multithreaded web server!

Finally, some appendixes contain useful information about the language in a more reference-like format. Appendix A covers Rust’s keywords, Appendix B covers Rust’s operators and symbols, Appendix C covers derivable traits provided by the standard library, Appendix D covers some useful development tools, and Appendix E explains Rust editions.

There is no wrong way to read this book: if you want to skip ahead, go for it! You might have to jump back to earlier chapters if you experience any confusion. But do whatever works for you.

An important part of the process of learning Rust is learning how to read the error messages the compiler displays: these will guide you toward working code. As such, we’ll provide many examples that don’t compile along with the error message the compiler will show you in each situation. Know that if you enter and run a random example, it may not compile! Make sure you read the surrounding text to see whether the example you’re trying to run is meant to error. In most situations, we’ll lead you to the correct version of any code that doesn’t compile.

Resources and How to Contribute to This Book

This book is open source. If you find an error, please don’t hesitate to file an issue or send a pull request on GitHub at <https://github.com/rust-lang/book>. Please see CONTRIBUTING.md at <https://github.com/rust-lang/book/blob/main/CONTRIBUTING.md> for more details.

The source code for the examples in this book, errata, and other information are available at <https://nostarch.com/rust-programming-language-2nd-edition>.

PRAISE FOR   
THE ART OF CYBERWARFARE

“Encompasses useful knowledge from the past and modern advanced   
threats seen today. Regardless of your expertise level, this book is an insightful read . . .”

—  
Brittany Day, Director of Communications, Guardian Digital

“For those looking for a guide to help them understand the new world of cyberwar, *The Art of Cyberwarfare* provides readers with a good overview   
of this expanding threat and what they can do to avoid being victims.”

—  
Ben Rothke, Senior Information Security Manager, Tapad

“An informative and explanatory guide for cybersecurity experts and an enlightening read for novices. DiMaggio effectively details both the   
history of cybercrime and how it is seen today.”

—  
Justice Levine, Communications Manager and Cloud Email Security Expert, Guardian Digital

“This book deserves to find a place on the shelf of everyone whose role involves protecting networks.”

—  
Ian Barker, BetaNews

“A cross between an IBM presentation . . . and a Tom Clancy novel!”

—  
The Shepherdess, Amazon Reviewer