

CONTENTS IN DETAIL

FOREWORD by Nicholas Matsakis and Aaron Turon **xix**

PREFACE **xxi**

ACKNOWLEDGMENTS **xxiii**

INTRODUCTION **xxv**

| | |
|--|-------|
| Who Rust Is For | xxvi |
| Teams of Developers | xxvi |
| Students | xxvi |
| Companies | xxvi |
| Open Source Developers | xxvi |
| 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 |
| Command Line Notation | 2 |
| Installing rustup on Linux or macOS | 2 |
| Installing rustup on Windows | 3 |
| Updating and Uninstalling | 3 |
| Troubleshooting | 3 |
| 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 | 10 |
| Cargo as Convention | 11 |
| Summary | 11 |

2
PROGRAMMING A GUESSING GAME **13**

| | |
|---|----|
| Setting Up a New Project | 14 |
| Processing a Guess | 14 |
| Storing Values with Variables | 15 |
| Handling Potential Failure with the Result Type | 17 |

| | |
|--|----|
| Printing Values with println! Placeholders | 18 |
| Testing the First Part | 18 |
| Generating a Secret Number | 19 |
| Using a Crate to Get More Functionality | 19 |
| Generating a Random Number | 21 |
| Comparing the Guess to the Secret Number | 23 |
| Allowing Multiple Guesses with Looping | 26 |
| Quitting After a Correct Guess | 27 |
| Handling Invalid Input | 28 |
| Summary | 30 |

3

COMMON PROGRAMMING CONCEPTS 31

| | |
|---|----|
| Variables and Mutability | 32 |
| Differences Between Variables and Constants | 34 |
| Shadowing | 34 |
| Data Types | 36 |
| Scalar Types | 36 |
| Compound Types | 40 |
| Functions | 43 |
| Function Parameters | 44 |
| Statements and Expressions in Function Bodies | 45 |
| Functions with Return Values | 47 |
| Comments | 49 |
| Control Flow | 49 |
| if Expressions | 49 |
| Repetition with Loops | 54 |
| Summary | 57 |

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 | 68 |
| References and Borrowing | 70 |
| Mutable References | 72 |
| Dangling References | 74 |
| The Rules of References | 75 |
| The Slice Type | 75 |
| String Slices | 77 |
| Other Slices | 81 |
| Summary | 81 |

| | | |
|--|--|------------|
| 5 | | |
| USING STRUCTS TO STRUCTURE RELATED DATA | | 83 |
| Defining and Instantiating Structs | | 83 |
| Using the Field Init Shorthand When Variables and Fields | | |
| Have the Same Name. | | 85 |
| Creating Instances from Other Instances with Struct Update Syntax | | 86 |
| Using Tuple Structs Without Named Fields to Create Different Types | | 86 |
| Unit-Like Structs Without Any Fields | | 87 |
| An Example Program Using Structs. | | 88 |
| Refactoring with Tuples | | 89 |
| Refactoring with Structs: Adding More Meaning | | 89 |
| Adding Useful Functionality with Derived Traits. | | 90 |
| Method Syntax | | 92 |
| Defining Methods | | 92 |
| Methods with More Parameters. | | 94 |
| Associated Functions | | 95 |
| Multiple impl Blocks. | | 96 |
| Summary | | 96 |
| | | |
| 6 | | |
| ENUMS AND PATTERN MATCHING | | 97 |
| Defining an Enum. | | 98 |
| Enum Values. | | 98 |
| The Option Enum and Its Advantages over Null Values | | 101 |
| The match Control Flow Operator | | 104 |
| Patterns That Bind to Values | | 106 |
| Matching with Option<T> | | 107 |
| Matches Are Exhaustive. | | 108 |
| The _ Placeholder | | 108 |
| Concise Control Flow with if let | | 109 |
| Summary | | 110 |
| | | |
| 7 | | |
| MANAGING GROWING PROJECTS WITH PACKAGES, CRATES, AND MODULES | | 111 |
| Packages and Crates | | 112 |
| Defining Modules to Control Scope and Privacy. | | 113 |
| Paths for Referring to an Item in the Module Tree | | 115 |
| Exposing Paths with the pub Keyword | | 117 |
| Starting Relative Paths with super | | 119 |
| Making Structs and Enums Public | | 120 |
| Bringing Paths into Scope with the use Keyword | | 121 |
| Creating Idiomatic use Paths. | | 123 |
| Providing New Names with the as Keyword. | | 124 |
| Re-exporting Names with pub use | | 124 |
| Using External Packages | | 125 |
| Using Nested Paths to Clean Up Large use Lists | | 126 |
| The Glob Operator | | 127 |
| Separating Modules into Different Files | | 127 |
| Summary | | 128 |

| | | |
|--|---|------------|
| 8 | COMMON COLLECTIONS | 131 |
| Storing Lists of Values with Vectors | | 132 |
| Creating a New Vector | | 132 |
| Updating a Vector. | | 132 |
| Dropping a Vector Drops Its Elements | | 133 |
| Reading Elements of Vectors. | | 133 |
| Iterating over the Values in a Vector | | 135 |
| Using an Enum to Store Multiple Types | | 136 |
| Storing UTF-8 Encoded Text with Strings | | 137 |
| What Is a String? | | 137 |
| Creating a New String | | 137 |
| Updating a String | | 138 |
| Indexing into Strings | | 141 |
| Slicing Strings. | | 142 |
| Methods for Iterating over Strings | | 143 |
| Strings Are Not So Simple | | 144 |
| Storing Keys with Associated Values in Hash Maps | | 144 |
| Creating a New Hash Map | | 144 |
| Hash Maps and Ownership | | 145 |
| Accessing Values in a Hash Map | | 146 |
| Updating a Hash Map. | | 147 |
| Hashing Functions. | | 149 |
| Summary | | 149 |
| | | |
| 9 | ERROR HANDLING | 151 |
| Unrecoverable Errors with panic! | | 152 |
| Using a panic! Backtrace | | 153 |
| Recoverable Errors with Result | | 155 |
| Matching on Different Errors. | | 158 |
| Shortcuts for Panic on Error: unwrap and expect. | | 159 |
| Propagating Errors | | 160 |
| To panic! or Not to panic! | | 164 |
| Examples, Prototype Code, and Tests | | 165 |
| Cases in Which You Have More Information Than the Compiler | | 165 |
| Guidelines for Error Handling | | 166 |
| Creating Custom Types for Validation | | 167 |
| Summary | | 169 |
| | | |
| 10 | GENERIC TYPES, TRAITS, AND LIFETIMES | 171 |
| Removing Duplication by Extracting a Function | | 172 |
| Generic Data Types | | 174 |
| In Function Definitions | | 174 |
| In Struct Definitions | | 177 |
| In Enum Definitions | | 178 |
| In Method Definitions. | | 179 |
| Performance of Code Using Generics | | 181 |

| | |
|---|-----|
| Traits: Defining Shared Behavior | 182 |
| Defining a Trait. | 182 |
| Implementing a Trait on a Type. | 183 |
| Default Implementations. | 185 |
| Traits as Parameters | 186 |
| Returning Types that Implement Traits. | 188 |
| Fixing the largest Function with Trait Bounds. | 189 |
| Using Trait Bounds to Conditionally Implement Methods. | 191 |
| Validating References with Lifetimes | 192 |
| Preventing Dangling References with Lifetimes. | 193 |
| The Borrow Checker | 194 |
| Generic Lifetimes in Functions. | 195 |
| Lifetime Annotation Syntax. | 196 |
| Lifetime Annotations in Function Signatures. | 197 |
| Thinking in Terms of Lifetimes | 199 |
| Lifetime Annotations in Struct Definitions. | 200 |
| Lifetime Elision | 201 |
| Lifetime Annotations in Method Definitions | 203 |
| The Static Lifetime | 204 |
| Generic Type Parameters, Trait Bounds, and Lifetimes Together | 205 |
| Summary | 205 |

11

WRITING AUTOMATED TESTS 207

| | |
|--|-----|
| How to Write Tests. | 208 |
| The Anatomy of a Test Function. | 208 |
| Checking Results with the <code>assert!</code> Macro. | 211 |
| Testing Equality with the <code>assert_eq!</code> and <code>assert_ne!</code> Macros | 214 |
| Adding Custom Failure Messages | 216 |
| Checking for Panics with <code>should_panic</code> | 218 |
| Using <code>Result<T, E></code> in Tests | 221 |
| Controlling How Tests Are Run. | 221 |
| Running Tests in Parallel or Consecutively. | 222 |
| Showing Function Output. | 222 |
| Running a Subset of Tests by Name. | 224 |
| Ignoring Some Tests Unless Specifically Requested | 226 |
| Test Organization | 227 |
| Unit Tests | 227 |
| Integration Tests | 228 |
| Summary | 232 |

12

AN I/O PROJECT: BUILDING A COMMAND LINE PROGRAM 233

| | |
|--|-----|
| Accepting Command Line Arguments | 234 |
| Reading the Argument Values. | 234 |
| Saving the Argument Values in Variables. | 236 |
| Reading a File. | 237 |
| Refactoring to Improve Modularity and Error Handling | 238 |
| Separation of Concerns for Binary Projects. | 239 |
| Fixing the Error Handling. | 243 |

| | |
|---|-----|
| Extracting Logic from main | 246 |
| Splitting Code into a Library Crate | 248 |
| Developing the Library's Functionality with Test-Driven Development | 250 |
| Writing a Failing Test | 250 |
| Writing Code to Pass the Test | 253 |
| Working with Environment Variables | 255 |
| Writing a Failing Test for the Case-Insensitive search Function | 255 |
| Implementing the search_case_insensitive Function | 257 |
| Writing Error Messages to Standard Error Instead of Standard Output | 260 |
| Checking Where Errors Are Written | 260 |
| Printing Errors to Standard Error | 261 |
| Summary | 262 |

13

FUNCTIONAL LANGUAGE FEATURES: ITERATORS AND CLOSURES **263**

| | |
|--|-----|
| Closures: Anonymous Functions That Can Capture Their Environment | 264 |
| Creating an Abstraction of Behavior with Closures | 264 |
| Closure Type Inference and Annotation | 269 |
| Storing Closures Using Generic Parameters and the Fn Traits | 270 |
| Limitations of the Cacher Implementation | 273 |
| Capturing the Environment with Closures | 274 |
| Processing a Series of Items with Iterators | 276 |
| The Iterator Trait and the next Method | 277 |
| Methods That Consume the Iterator | 278 |
| Methods That Produce Other Iterators | 279 |
| Using Closures That Capture Their Environment | 280 |
| Creating Our Own Iterators with the Iterator Trait | 281 |
| Improving Our I/O Project | 283 |
| Removing a clone Using an Iterator | 284 |
| Making Code Clearer with Iterator Adaptors | 286 |
| Comparing Performance: Loops vs. Iterators | 287 |
| Summary | 289 |

14

MORE ABOUT CARGO AND CRATES.IO **291**

| | |
|--|-----|
| Customizing Builds with Release Profiles | 292 |
| Publishing a Crate to Crates.io | 293 |
| Making Useful Documentation Comments | 293 |
| Exporting a Convenient Public API with pub use | 296 |
| Setting Up a Crates.io Account | 300 |
| Adding Metadata to a New Crate | 300 |
| Publishing to Crates.io | 301 |
| Publishing a New Version of an Existing Crate | 302 |
| Removing Versions from Crates.io with cargo yank | 302 |
| Cargo Workspaces | 303 |
| Creating a Workspace | 303 |
| Creating the Second Crate in the Workspace | 304 |

| | |
|---|-----|
| Installing Binaries from Crates.io with cargo install | 308 |
| Extending Cargo with Custom Commands | 309 |
| Summary | 309 |

15 SMART POINTERS 311

| | |
|---|-----|
| Using Box<T> to Point to Data on the Heap | 312 |
| Using a Box<T> to Store Data on the Heap | 313 |
| Enabling Recursive Types with Boxes | 314 |
| Treating Smart Pointers Like Regular References with the Deref Trait | 317 |
| Following the Pointer to the Value with the Derefence Operator | 318 |
| Using Box<T> Like a Reference | 318 |
| Defining Our Own Smart Pointer | 319 |
| Treating a Type Like a Reference by Implementing the Deref Trait | 320 |
| Implicit Deref Coercions with Functions and Methods | 321 |
| How Deref Coercion Interacts with Mutability | 322 |
| Running Code on Cleanup with the Drop Trait | 323 |
| Dropping a Value Early with std::mem::drop | 325 |
| Rc<T>, the Reference Counted Smart Pointer | 326 |
| Using Rc<T> to Share Data | 327 |
| Cloning an Rc<T> Increases the Reference Count | 329 |
| RefCell<T> and the Interior Mutability Pattern | 330 |
| Enforcing Borrowing Rules at Runtime with RefCell<T> | 330 |
| Interior Mutability: A Mutable Borrow to an Immutable Value | 331 |
| Having Multiple Owners of Mutable Data by Combining Rc<T> and RefCell<T> | 337 |
| Reference Cycles Can Leak Memory | 339 |
| Creating a Reference Cycle | 339 |
| Preventing Reference Cycles: Turning an Rc<T> into a Weak<T> | 341 |
| Summary | 346 |

16 FEARLESS CONCURRENCY 347

| | |
|---|-----|
| Using Threads to Run Code Simultaneously | 348 |
| Creating a New Thread with spawn | 350 |
| Waiting for All Threads to Finish Using join Handles | 351 |
| Using move Closures with Threads | 353 |
| Using Message Passing to Transfer Data Between Threads | 355 |
| Channels and Ownership Transference | 358 |
| Sending Multiple Values and Seeing the Receiver Waiting | 359 |
| Creating Multiple Producers by Cloning the Transmitter | 360 |
| Shared-State Concurrency | 361 |
| Using Mutexes to Allow Access to Data from One Thread at a Time | 362 |
| Similarities Between RefCell<T>/Rc<T> and Mutex<T>/Arc<T> | 368 |
| Extensible Concurrency with the Sync and Send Traits | 368 |
| Allowing Transference of Ownership Between Threads with Send | 369 |
| Allowing Access from Multiple Threads with Sync | 369 |
| Implementing Send and Sync Manually Is Unsafe | 369 |
| Summary | 370 |

17

OBJECT-ORIENTED PROGRAMMING FEATURES OF RUST **371**

| | |
|--|-----|
| Characteristics of Object-Oriented Languages | 371 |
| Objects Contain Data and Behavior | 372 |
| Encapsulation That Hides Implementation Details | 372 |
| Inheritance as a Type System and as Code Sharing | 374 |
| Using Trait Objects That Allow for Values of Different Types | 375 |
| Defining a Trait for Common Behavior | 375 |
| Implementing the Trait | 377 |
| Trait Objects Perform Dynamic Dispatch | 380 |
| Object Safety Is Required for Trait Objects | 380 |
| Implementing an Object-Oriented Design Pattern | 382 |
| Defining Post and Creating a New Instance in the Draft State | 383 |
| Storing the Text of the Post Content | 384 |
| Ensuring the Content of a Draft Post Is Empty | 384 |
| Requesting a Review of the Post Changes Its State | 385 |
| Adding the approve Method that Changes the Behavior of content | 386 |
| Trade-offs of the State Pattern | 389 |
| Summary | 393 |

18

PATTERNS AND MATCHING **395**

| | |
|---|-----|
| All the Places Patterns Can Be Used | 396 |
| match Arms | 396 |
| Conditional if let Expressions | 396 |
| while let Conditional Loops | 398 |
| for Loops | 398 |
| let Statements | 399 |
| Function Parameters | 400 |
| Refutability: Whether a Pattern Might Fail to Match | 401 |
| Pattern Syntax | 402 |
| Matching Literals | 402 |
| Matching Named Variables | 403 |
| Multiple Patterns | 404 |
| Matching Ranges of Values with the ... Syntax | 404 |
| Destructuring to Break Apart Values | 405 |
| Ignoring Values in a Pattern | 409 |
| Extra Conditionals with Match Guards | 413 |
| @ Bindings | 415 |
| Summary | 416 |

19

ADVANCED FEATURES **417**

| | |
|--|-----|
| Unsafe Rust | 418 |
| Unsafe Superpowers | 418 |
| Dereferencing a Raw Pointer | 419 |
| Calling an Unsafe Function or Method | 421 |
| Accessing or Modifying a Mutable Static Variable | 425 |
| Implementing an Unsafe Trait | 426 |
| When to Use Unsafe Code | 427 |

| | |
|---|-----|
| Advanced Traits | 427 |
| Specifying Placeholder Types in Trait Definitions with Associated Types | 427 |
| Default Generic Type Parameters and Operator Overloading | 429 |
| Fully Qualified Syntax for Disambiguation: | |
| Calling Methods with the Same Name | 431 |
| Using Supertraits to Require One Trait’s Functionality Within Another Trait | 434 |
| Using the Newtype Pattern to Implement External Traits on External Types | 436 |
| Advanced Types | 437 |
| Using the Newtype Pattern for Type Safety and Abstraction | 437 |
| Creating Type Synonyms with Type Aliases | 438 |
| The Never Type That Never Returns | 440 |
| Dynamically Sized Types and the Sized Trait | 441 |
| Advanced Functions and Closures | 443 |
| Function Pointers | 443 |
| Returning Closures | 445 |
| Macros | 446 |
| The Difference Between Macros and Functions | 446 |
| Declarative Macros with <code>macro_rules!</code> for General Metaprogramming | 446 |
| Procedural Macros for Generating Code from Attributes | 449 |
| How to Write a Custom <code>derive</code> Macro | 449 |
| Attribute-like macros | 454 |
| Function-like macros | 455 |
| Summary | 455 |

20

FINAL PROJECT: BUILDING A MULTITHREADED WEB SERVER 457

| | |
|--|-----|
| Building a Single-Threaded Web Server | 458 |
| Listening to the TCP Connection | 458 |
| Reading the Request | 460 |
| A Closer Look at an HTTP Request | 462 |
| Writing a Response | 463 |
| Returning Real HTML | 464 |
| Validating the Request and Selectively Responding | 465 |
| A Touch of Refactoring | 466 |
| Turning Our Single-Threaded Server into a Multithreaded Server | 468 |
| Simulating a Slow Request in the Current Server Implementation | 468 |
| Improving Throughput with a Thread Pool | 469 |
| Graceful Shutdown and Cleanup | 487 |
| Implementing the Drop Trait on ThreadPool | 487 |
| Signaling to the Threads to Stop Listening for Jobs | 489 |
| 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 | | 501 |
| | | |
| 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 the Rust Language Server | | 514 |
| | | |
| E | | |
| EDITIONS | | 515 |
| | | |
| INDEX | | 517 |