

CONTENTS IN DETAIL

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

ACKNOWLEDGMENTS	xxi
------------------------	------------

INTRODUCTION	xxiii
---------------------	--------------

Who Rust Is Forxxiv
Teams of Developersxxiv
Studentsxxiv
Companiesxxiv
Open Source Developersxxiv
People Who Value Speed and Stabilityxxv
Who This Book Is Forxxv
How to Use This Bookxxv
Resources and How to Contribute to This Bookxxvii

1	
GETTING STARTED	1

Installation	1
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	39
Functions.	42
Function Parameters	43
Statements and Expressions in Function Bodies	44
Functions with Return Values.	46
Comments.	48
Control Flow	48
if Expressions	48
Repetition with Loops.	53
Summary	56

4 UNDERSTANDING OWNERSHIP 57

What Is Ownership?	57
Ownership Rules.	59
Variable Scope.	59
The String Type.	60
Memory and Allocation	61
Ownership and Functions.	66
Return Values and Scope	66
References and Borrowing	68
Mutable References.	70
Dangling References	72
The Rules of References	73
The Slice Type	73
String Slices	75
Other Slices	78
Summary	79

5 USING STRUCTS TO STRUCTURE RELATED DATA 81

Defining and Instantiating Structs	81
Using the Field Init Shorthand When Variables and Fields Have the Same Name.	83

Creating Instances from Other Instances with Struct Update Syntax	84
Using Tuple Structs Without Named Fields to Create Different Types	84
Unit-Like Structs Without Any Fields	85
An Example Program Using Structs.	86
Refactoring with Tuples	87
Refactoring with Structs: Adding More Meaning	87
Adding Useful Functionality with Derived Traits	88
Method Syntax	90
Defining Methods	90
Methods with More Parameters.	92
Associated Functions	93
Multiple impl Blocks.	94
Summary	94

6 **ENUMS AND PATTERN MATCHING** **95**

Defining an Enum.	96
Enum Values.	96
The Option Enum and Its Advantages over Null Values	99
The match Control Flow Operator	102
Patterns That Bind to Values	104
Matching with Option<T>	105
Matches Are Exhaustive	106
The _ Placeholder	106
Concise Control Flow with if let	107
Summary	108

7 **USING MODULES TO REUSE AND ORGANIZE CODE** **109**

mod and the Filesystem.	110
Module Definitions	111
Moving Modules to Other Files	112
Rules of Module Filesystems	117
Controlling Visibility with pub	118
Making a Function Public.	119
Privacy Rules	121
Privacy Examples	121
Referring to Names in Different Modules.	123
Bringing Names into Scope with the use Keyword.	123
Bringing All Names into Scope with a Glob	124
Using super to Access a Parent Module	125
Summary	127

8 **COMMON COLLECTIONS** **129**

Storing Lists of Values with Vectors	130
Creating a New Vector	130
Updating a Vector.	131
Dropping a Vector Drops Its Elements	131
Reading Elements of Vectors.	131

Iterating over the Values in a Vector	133
Using an Enum to Store Multiple Types	134
Storing UTF-8 Encoded Text with Strings	135
What Is a String?	135
Creating a New String	135
Updating a String	136
Indexing into Strings	139
Slicing Strings.	140
Methods for Iterating over Strings	141
Strings Are Not So Simple	142
Storing Keys with Associated Values in Hash Maps	142
Creating a New Hash Map	142
Hash Maps and Ownership	143
Accessing Values in a Hash Map	144
Updating a Hash Map.	145
Hashing Functions.	147
Summary	147

9 ERROR HANDLING 149

Unrecoverable Errors with panic!	150
Using a panic! Backtrace	151
Recoverable Errors with Result	153
Matching on Different Errors.	155
Shortcuts for Panic on Error: unwrap and expect.	157
Propagating Errors	158
To panic! or Not to panic!	161
Examples, Prototype Code, and Tests	162
Cases in Which You Have More Information Than the Compiler	162
Guidelines for Error Handling	162
Creating Custom Types for Validation	164
Summary	166

10 GENERIC TYPES, TRAITS, AND LIFETIMES 167

Removing Duplication by Extracting a Function	168
Generic Data Types	170
In Function Definitions	170
In Struct Definitions	173
In Enum Definitions	174
In Method Definitions.	175
Performance of Code Using Generics	177
Traits: Defining Shared Behavior	178
Defining a Trait.	178
Implementing a Trait on a Type.	179
Default Implementations	181
Trait Bounds	182
Fixing the largest Function with Trait Bounds.	183
Using Trait Bounds to Conditionally Implement Methods.	185

Validating References with Lifetimes	187
Preventing Dangling References with Lifetimes.	187
The Borrow Checker	188
Generic Lifetimes in Functions	189
Lifetime Annotation Syntax	190
Lifetime Annotations in Function Signatures.	191
Thinking in Terms of Lifetimes	193
Lifetime Annotations in Struct Definitions.	194
Lifetime Elision	195
Lifetime Annotations in Method Definitions	197
The Static Lifetime	198
Generic Type Parameters, Trait Bounds, and Lifetimes Together	199
Summary	199

11 WRITING AUTOMATED TESTS 201

How to Write Tests.	202
The Anatomy of a Test Function.	202
Checking Results with the <code>assert!</code> Macro.	205
Testing Equality with the <code>assert_eq!</code> and <code>assert_ne!</code> Macros	208
Adding Custom Failure Messages	210
Checking for Panics with <code>should_panic</code>	212
Controlling How Tests Are Run.	215
Running Tests in Parallel or Consecutively.	215
Showing Function Output.	216
Running a Subset of Tests by Name.	218
Ignoring Some Tests Unless Specifically Requested	219
Test Organization	220
Unit Tests	221
Integration Tests	222
Summary	226

12 AN I/O PROJECT: BUILDING A COMMAND LINE PROGRAM 227

Accepting Command Line Arguments	228
Reading the Argument Values.	228
Saving the Argument Values in Variables.	230
Reading a File.	231
Refactoring to Improve Modularity and Error Handling	232
Separation of Concerns for Binary Projects.	233
Fixing the Error Handling	237
Extracting Logic from <code>main</code>	240
Splitting Code into a Library Crate	242
Developing the Library’s Functionality with Test-Driven Development	244
Writing a Failing Test	244
Writing Code to Pass the Test.	247
Working with Environment Variables	249
Writing a Failing Test for the Case-Insensitive <code>search</code> Function	250
Implementing the <code>search_case_insensitive</code> Function	251

Writing Error Messages to Standard Error Instead of Standard Output	254
Checking Where Errors Are Written	254
Printing Errors to Standard Error	255
Summary	256

13

FUNCTIONAL LANGUAGE FEATURES: ITERATORS AND CLOSURES

257

Closures: Anonymous Functions That Can Capture Their Environment	258
Creating an Abstraction of Behavior with Closures	258
Closure Type Inference and Annotation	263
Storing Closures Using Generic Parameters and the Fn Traits	264
Limitations of the Cacher Implementation	267
Capturing the Environment with Closures	268
Processing a Series of Items with Iterators	270
The Iterator Trait and the next Method	271
Methods That Consume the Iterator	272
Methods That Produce Other Iterators	273
Using Closures That Capture Their Environment.	274
Creating Our Own Iterators with the Iterator Trait	275
Improving Our I/O Project	277
Removing a clone Using an Iterator.	278
Making Code Clearer with Iterator Adaptors	280
Comparing Performance: Loops vs. Iterators	281
Summary	283

14

MORE ABOUT CARGO AND CRATES.IO

285

Customizing Builds with Release Profiles	286
Publishing a Crate to Crates.io	287
Making Useful Documentation Comments.	287
Exporting a Convenient Public API with pub use	290
Setting Up a Crates.io Account.	294
Adding Metadata to a New Crate	294
Publishing to Crates.io.	295
Publishing a New Version of an Existing Crate	296
Removing Versions from Crates.io with cargo yank.	296
Cargo Workspaces	297
Creating a Workspace	297
Creating the Second Crate in the Workspace.	298
Installing Binaries from Crates.io with cargo install.	302
Extending Cargo with Custom Commands.	303
Summary	303

15

SMART POINTERS

305

Using Box<T> to Point to Data on the Heap.	306
Using a Box<T> to Store Data on the Heap	307
Enabling Recursive Types with Boxes.	308

Treating Smart Pointers Like Regular References with the Deref Trait	311
Following the Pointer to the Value with the Derefence Operator	312
Using Box<T> Like a Reference	312
Defining Our Own Smart Pointer.	313
Treating a Type Like a Reference by Implementing the Deref Trait	314
Implicit Deref Coercions with Functions and Methods.	315
How Deref Coercion Interacts with Mutability	316
Running Code on Cleanup with the Drop Trait	317
Dropping a Value Early with std::mem::drop	318
Rc<T>, the Reference Counted Smart Pointer	320
Using Rc<T> to Share Data	320
Cloning an Rc<T> Increases the Reference Count	322
RefCell<T> and the Interior Mutability Pattern	323
Enforcing Borrowing Rules at Runtime with RefCell<T>.	324
Interior Mutability: A Mutable Borrow to an Immutable Value	325
Having Multiple Owners of Mutable Data by Combining Rc<T> and RefCell<T>	330
Reference Cycles Can Leak Memory.	332
Creating a Reference Cycle	332
Preventing Reference Cycles: Turning an Rc<T> into a Weak<T>	334
Summary	339

16

FEARLESS CONCURRENCY

341

Using Threads to Run Code Simultaneously	342
Creating a New Thread with spawn	344
Waiting for All Threads to Finish Using join Handles	345
Using move Closures with Threads	347
Using Message Passing to Transfer Data Between Threads	349
Channels and Ownership Transference	352
Sending Multiple Values and Seeing the Receiver Waiting	353
Creating Multiple Producers by Cloning the Transmitter	354
Shared-State Concurrency	355
Using Mutexes to Allow Access to Data from One Thread at a Time	356
Similarities Between RefCell<T>/Rc<T> and Mutex<T>/Arc<T>	362
Extensible Concurrency with the Sync and Send Traits	362
Allowing Transference of Ownership Between Threads with Send	363
Allowing Access from Multiple Threads with Sync	363
Implementing Send and Sync Manually Is Unsafe	363
Summary	364

17

OBJECT-ORIENTED PROGRAMMING FEATURES OF RUST

365

Characteristics of Object-Oriented Languages	365
Objects Contain Data and Behavior	366
Encapsulation That Hides Implementation Details	366
Inheritance as a Type System and as Code Sharing	368
Using Trait Objects That Allow for Values of Different Types	369
Defining a Trait for Common Behavior.	369
Implementing the Trait	371

Trait Objects Perform Dynamic Dispatch	374
Object Safety Is Required for Trait Objects	374
Implementing an Object-Oriented Design Pattern	376
Defining Post and Creating a New Instance in the Draft State	377
Storing the Text of the Post Content	378
Ensuring the Content of a Draft Post Is Empty	378
Requesting a Review of the Post Changes Its State	379
Adding the approve Method that Changes the Behavior of content	380
Trade-offs of the State Pattern	383
Summary	387

18

PATTERNS AND MATCHING 389

All the Places Patterns Can Be Used	390
match Arms	390
Conditional if let Expressions	390
while let Conditional Loops	392
for Loops	392
let Statements	393
Function Parameters	394
Refutability: Whether a Pattern Might Fail to Match	395
Pattern Syntax	396
Matching Literals	396
Matching Named Variables	397
Multiple Patterns	398
Matching Ranges of Values with the ... Syntax	398
Destructuring to Break Apart Values	399
Ignoring Values in a Pattern	403
Creating References in Patterns with ref and ref mut	407
Extra Conditionals with Match Guards	408
@ Bindings	410
Summary	411

19

ADVANCED FEATURES 413

Unsafe Rust	414
Unsafe Superpowers	414
Dereferencing a Raw Pointer	415
Calling an Unsafe Function or Method	417
Accessing or Modifying a Mutable Static Variable	421
Implementing an Unsafe Trait	422
When to Use Unsafe Code	423
Advanced Lifetimes	423
Ensuring One Lifetime Outlives Another with Lifetime Subtyping	423
Lifetime Bounds on References to Generic Types	428
Inference of Trait Object Lifetimes	429
Advanced Traits	430
Specifying Placeholder Types in Trait Definitions with Associated Types	431
Default Generic Type Parameters and Operator Overloading	432

Fully Qualified Syntax for Disambiguation:	
Calling Methods with the Same Name	434
Using Supertraits to Require One Trait’s Functionality Within Another Trait . . .	437
Using the Newtype Pattern to Implement External Traits on External Types . . .	439
Advanced Types	440
Using the Newtype Pattern for Type Safety and Abstraction	440
Creating Type Synonyms with Type Aliases	441
The Never Type That Never Returns.	443
Dynamically Sized Types and the Sized Trait	445
Advanced Functions and Closures	446
Function Pointers	446
Returning Closures	448
Summary	448

20

FINAL PROJECT: BUILDING A MULTITHREADED WEB SERVER 449

Building a Single-Threaded Web Server	450
Listening to the TCP Connection	450
Reading the Request	452
A Closer Look at an HTTP Request.	454
Writing a Response.	455
Returning Real HTML	456
Validating the Request and Selectively Responding	457
A Touch of Refactoring	459
Turning Our Single-Threaded Server into a Multithreaded Server	460
Simulating a Slow Request in the Current Server Implementation	460
Improving Throughput with a Thread Pool	461
Graceful Shutdown and Cleanup	479
Implementing the Drop Trait on ThreadPool	479
Signaling to the Threads to Stop Listening for Jobs.	481
Summary	485

A

KEYWORDS 487

Keywords Currently in Use	487
Keywords Reserved for Future Use	489

B

OPERATORS AND SYMBOLS 491

Operators	491
Non-operator Symbols	493

C

DERIVABLE TRAITS 497

Debug for Programmer Output.	498
PartialEq and Eq for Equality Comparisons	498
PartialOrd and Ord for Ordering Comparisons	499

Clone and Copy for Duplicating Values	499
Hash for Mapping a Value to a Value of Fixed Size	500
Default for Default Values	500

D
MACROS **501**

The Difference Between Macros and Functions.	502
Declarative Macros with <code>macro_rules!</code> for General Metaprogramming	502
Procedural Macros for Custom <code>derive</code>	504
The Future of Macros	510

INDEX **511**