Foreword

It wasn't always so clear, but the Rust programm *empowerment*: no matter what kind of code you to reach farther, to program with confidence in did before.

Take, for example, "systems-level" work that dea management, data representation, and concurr programming is seen as arcane, accessible only necessary years learning to avoid its infamous p do so with caution, lest their code be open to ex

Rust breaks down these barriers by eliminating friendly, polished set of tools to help you along t "dip down" into lower-level control can do so wit customary risk of crashes or security holes, and points of a fickle toolchain. Better yet, the langu towards reliable code that is efficient in terms o

Programmers who are already working with low ambitions. For example, introducing parallelism operation: the compiler will catch the classical m more aggressive optimizations in your code with accidentally introduce crashes or vulnerabilities.

But Rust isn't limited to low-level systems progra ergonomic enough to make CLI apps, web serve quite pleasant to write — you'll find simple exam Working with Rust allows you to build skills that you can learn Rust by writing a web app, then ap Raspberry Pi.

This book fully embraces the potential of Rust to approachable text intended to help you level up also your reach and confidence as a programme learn—and welcome to the Rust community!

— Nicholas Matsakis and Aaron Turon

Introduction

> Note: This edition of the book is the same as
> available in print and ebook format from No

Welcome to *The Rust Programming Language*, an
Rust programming language helps you write fas
ergonomics and low-level control are often at od
Rust challenges that conflict. Through balancing
great developer experience, Rust gives you the c
(such as memory usage) without all the hassle tr
control.

Who Rust Is For

Rust is ideal for many people for a variety of rea
important groups.

Teams of Developers

Rust is proving to be a productive tool for collab
developers with varying levels of systems progra
prone to a variety of subtle bugs, which in most
through extensive testing and careful code revie
the compiler plays a gatekeeper role by refusing
bugs, including concurrency bugs. By working al
spend their time focusing on the program's logic

Rust also brings contemporary developer tools t

- Cargo, the included dependency manager
  compiling, and managing dependencies pa
  ecosystem.
- Rustfmt ensures a consistent coding style a
- The Rust Language Server powers Integrat
  integration for code completion and inline

By using these and other tools in the Rust ecosy
while writing systems-level code.

Students

Rust is for students and those who are intereste
Using Rust, many people have learned about to
development. The community is very welcoming
questions. Through efforts such as this book, th
concepts more accessible to more people, espe

Companies

Hundreds of companies, large and small, use Ru
Those tasks include command line tools, web se
devices, audio and video analysis and transcodi
search engines, Internet of Things applications,
parts of the Firefox web browser.

Open Source Developers

Rust is for people who want to build the Rust pr
developer tools, and libraries. We'd love to have

People Who Value Speed and Stability

Rust is for people who crave speed and stability
speed of the programs that you can create with
you write them. The Rust compiler's checks ensu
and refactoring. This is in contrast to the brittle
these checks, which developers are often afraid
abstractions, higher-level features that compile
written manually, Rust endeavors to make safe

The Rust language hopes to support many other
are merely some of the biggest stakeholders. O
eliminate the trade-offs that programmers have
safety *and* productivity, speed *and* ergonomics.
work for you.

Who This Book Is For

This book assumes that you've written code in a
doesn't make any assumptions about which one
broadly accessible to those from a wide variety
don't spend a lot of time talking about what pro
If you're entirely new to programming, you woul
that specifically provides an introduction to prog

How to Use This Book

In general, this book assumes that you're readin
Later chapters build on concepts in earlier chapt
delve into details on a topic; we typically revisit t

You'll find two kinds of chapters in this book: co
In concept chapters, you'll learn about an aspect
build small programs together, applying what yo
and 20 are project chapters; the rest are concep

Chapter 1 explains how to install Rust, how to w
to use Cargo, Rust's package manager and build
introduction to the Rust language. Here we cove
chapters will provide additional detail. If you wa
Chapter 2 is the place for that. At first, you migh
covers Rust features similar to those of other pr
straight to Chapter 4 to learn about Rust's owne
particularly meticulous learner who prefers to le
the next, you might want to skip Chapter 2 and
Chapter 2 when you'd like to work on a project a

Chapter 5 discusses structs and methods, and C
expressions, and the `if let` control flow const
make custom types in Rust.

In Chapter 7, you'll learn about Rust's module sy
organizing your code and its public Application F
8 discusses some common collection data struct
provides, such as vectors, strings, and hash map
handling philosophy and techniques.

Chapter 10 digs into generics, traits, and lifetime
code that applies to multiple types. Chapter 11 i
Rust's safety guarantees is necessary to ensure
Chapter 12, we'll build our own implementation

`grep` command line tool that searches for text w
the concepts we discussed in the previous chapt

Chapter 13 explores closures and iterators: featu
functional programming languages. In Chapter 1
and talk about best practices for sharing your lib
discusses smart pointers that the standard libra
their functionality.

In Chapter 16, we'll walk through different mode
talk about how Rust helps you to program in mu
looks at how Rust idioms compare to object-orie
might be familiar with.

Chapter 18 is a reference on patterns and patter
of expressing ideas throughout Rust programs.
of advanced topics of interest, including unsafe
types, functions, and closures.

In Chapter 20, we'll complete a project in which w
multithreaded web server!

Finally, some appendixes contain useful informa
reference-like format. Appendix A covers Rust's
operators and symbols, Appendix C covers deriv
library, and Appendix D covers macros.

There is no wrong way to read this book: if you w
might have to jump back to earlier chapters if yo
whatever works for you.

An important part of the process of learning Rus
messages the compiler displays: these will guide
we'll provide many examples of code that doesn
message the compiler will show you in each situ
a random example, it may not compile! Make su
see whether the example you're trying to run is
we'll lead you to the correct version of any code

Source Code

The source files from which this book is generat

Getting Started

Let's start your Rust journey! There's a lot to lear
somewhere. In this chapter, we'll discuss:

- Installing Rust on Linux, macOS, and Windo
- Writing a program that prints `Hello, worl`
- Using `cargo`, Rust's package manager and

Installation

The first step is to install Rust. We'll download Ru
tool for managing Rust versions and associated
connection for the download.

> Note: If you prefer not to use `rustup` for som
> installation page for other options.

The following steps install the latest stable versi
stability guarantees ensure that all the examples
continue to compile with newer Rust versions. T
between versions, because Rust often improves
other words, any newer, stable version of Rust y
work as expected with the content of this book.

> ### Command Line Notation
>
> In this chapter and throughout the book, we'l
> the terminal. Lines that you should enter in a
> need to type in the `$` character; it indicates t
> that don't start with `$` typically show the outp
> Additionally, PowerShell-specific examples wi

Installing `rustup` on Linux or macOS

If you're using Linux or macOS, open a terminal

```
$ curl https://sh.rustup.rs -sSf | sh
```

The command downloads a script and starts the
which installs the latest stable version of Rust. Y
password. If the install is successful, the followir

```
Rust is installed now. Great!
```

If you prefer, feel free to download the script an

The installation script automatically adds Rust to
login. If you want to start using Rust right away i
run the following command in your shell to add

```
$ source $HOME/.cargo/env
```

Alternatively, you can add the following line to y

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

Additionally, you'll need a linker of some kind. It'
when you try to compile a Rust program and get
not execute, that means a linker isn't installed or
install one manually. C compilers usually come w
platform's documentation for how to install a C
packages depend on C code and will need a C co
installing one now.

Installing `rustup` on Windows

On Windows, go to https://www.rust-lang.org/in
for installing Rust. At some point in the installati
explaining that you'll also need the C++ build too
easiest way to acquire the build tools is to instal
The tools are in the Other Tools and Framework

The rest of this book uses commands that work
there are specific differences, we'll explain which

Updating and Uninstalling

After you've installed Rust via `rustup`, updating
your shell, run the following update script:

```
$ rustup update
```

To uninstall Rust and `rustup`, run the following

```
$ rustup self uninstall
```

Troubleshooting

To check whether you have Rust installed correc

```
$ rustc --version
```

You should see the version number, commit has
stable version that has been released in the follo

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

If you see this information, you have installed Ru
information and you're on Windows, check that
variable. If that's all correct and Rust still isn't wo
you can get help. The easiest is the #rust IRC cha
can access through Mibbit. At that address you c
nickname we call ourselves) who can help you o
Users forum and Stack Overflow.

Local Documentation

The installer also includes a copy of the docume
offline. Run `rustup doc` to open the local docum

Any time a type or function is provided by the st
what it does or how to use it, use the applicatior
documentation to find out!

Hello, World!

Now that you've installed Rust, let's write your fi
learning a new language to write a little program
to the screen, so we'll do the same here!

> Note: This book assumes basic familiarity witl
> no specific demands about your editing or to
> you prefer to use an integrated development
> command line, feel free to use your favorite I
> degree of Rust support; check the IDE's docur
> Rust team has been focusing on enabling gre
> been made rapidly on that front!

Creating a Project Directory

You'll start by making a directory to store your R
where your code lives, but for the exercises and
making a *projects* directory in your home directo
there.

Open a terminal and enter the following comma
a directory for the Hello, world! project within th

For Linux and macOS, enter this:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

For Windows CMD, enter this:

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

For Windows PowerShell, enter this:

```
> mkdir $env:USERPROFILE\projects
> cd $env:USERPROFILE\projects
> mkdir hello_world
> cd hello_world
```

Writing and Running a Rust Program

Next, make a new source file and call it *main.rs*.
extension. If you're using more than one word ir
separate them. For example, use *hello_world.rs* r

Now open the *main.rs* file you just created and e

Filename: main.rs

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

Listing 1-1: A program that prints `Hello, world`

Save the file and go back to your terminal windo
following commands to compile and run the file

```
$ rustc main.rs
$ ./main
Hello, world!
```

On Windows, enter the command `.\main.exe` i

```
> rustc main.rs
> .\main.exe
Hello, world!
```

Regardless of your operating system, the string
terminal. If you don't see this output, refer back
Installation section for ways to get help.

If `Hello, world!` did print, congratulations! You
That makes you a Rust programmer—welcome!

Anatomy of a Rust Program

Let's review in detail what just happened in your
first piece of the puzzle:

```
fn main() {

}
```

These lines define a function in Rust. The `main`
code that runs in every executable Rust progran
named `main` that has no parameters and returr
they would go inside the parentheses, `()` .

Also, note that the function body is wrapped in
these around all function bodies. It's good style
the same line as the function declaration, addin

At the time of this writing, an automatic formatt
development. If you want to stick to a standard
will format your code in a particular style. The R
this tool with the standard Rust distribution, like
read this book, it might already be installed on y
documentation for more details.

Inside the `main` function is the following code:

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

This line does all the work in this little program:
four important details to notice here. First, Rust
a tab.

Second, `println!` calls a Rust macro. If it called
entered as `println` (without the `!`). We'll discu
Appendix D. For now, you just need to know tha
a macro instead of a normal function.

Third, you see the `"Hello, world!"` string. We
`println!` , and the string is printed to the scree

Fourth, we end the line with a semicolon (`;`), wh
over and the next one is ready to begin. Most lir

Compiling and Running Are Separate S

You've just run a newly created program, so let's

Before running a Rust program, you must comp
entering the `rustc` command and passing it the

```
$ rustc main.rs
```

If you have a C or C++ background, you'll notice
After compiling successfully, Rust outputs a bina

On Linux and macOS you can see the executable
your shell as follows:

```
$ ls
main  main.rs
```

With PowerShell on Windows, you can use `ls` a

```
> ls


    Directory: Path:\to\the\project


Mode                 LastWriteTime
----                 -------------
-a----          6/1/2018   7:31 AM
-a----          6/1/2018   7:31 AM         1
-a----          6/1/2018   7:31 AM
```

With CMD on Windows, you would enter the foll

```
> dir /B %= the /B option says to only sho
main.exe
main.pdb
main.rs
```

This shows the source code file with the *.rs* exte
Windows, but *main* on all other platforms), and,
debugging information with the *.pdb* extension.
main.exe file, like this:

```
$ ./main # or .\main.exe on Windows
```

If *main.rs* was your Hello, world! program, this li
your terminal.

If you're more familiar with a dynamic language,
you might not be used to compiling and running
an *ahead-of-time compiled* language, meaning yo
executable to someone else, and they can run it
you give someone a *.rb*, *.py*, or *.js* file, they need
implementation installed (respectively). But in th
command to compile and run your program. Ev

design.

Just compiling with `rustc` is fine for simple prog
you'll want to manage all the options and make
introduce you to the Cargo tool, which will help

Hello, Cargo!

Cargo is Rust's build system and package manag
manage their Rust projects because Cargo hand
building your code, downloading the libraries yo
those libraries. (We call libraries your code need

The simplest Rust programs, like the one we've
dependencies. So if we had built the Hello, worl
use the part of Cargo that handles building your
Rust programs, you'll add dependencies, and if
dependencies will be much easier to do.

Because the vast majority of Rust projects use C
that you're using Cargo too. Cargo comes install
installers discussed in the "Installation" section.
other means, check whether Cargo is installed b
terminal:

```
$ cargo --version
```

If you see a version number, you have it! If you s
`command not found`, look at the documentation
determine how to install Cargo separately.

Creating a Project with Cargo

Let's create a new project using Cargo and look
Hello, world! project. Navigate back to your *proj*
decided to store your code). Then, on any opera

```
$ cargo new hello_cargo
$ cd hello_cargo
```

The first command creates a new directory calle project *hello_cargo*, and Cargo creates its files in

Go into the *hello_cargo* directory and list the files
two files and one directory for us: a *Cargo.toml* f
file inside. It has also initialized a new Git reposi

> Note: Git is a common version control system
> use a different version control system or no v
> `--vcs` flag. Run `cargo new --help` to see th

Open *Cargo.toml* in your text editor of choice. It
Listing 1-2.

Filename: Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

Listing 1-2: Contents of *Cargo.toml* generated by

This file is in the *TOML* (*Tom's Obvious, Minimal L*
configuration format.

The first line, `[package]`, is a section heading th
statements are configuring a package. As we ad
add other sections.

The next three lines set the configuration inform
program: the name, the version, and who wrote
information from your environment, so if that in
information now and then save the file.

The last line, `[dependencies]`, is the start of a s
project's dependencies. In Rust, packages of cod
need any other crates for this project, but we wi
we'll use this dependencies section then.

Now open *src/main.rs* and take a look:

Filename: src/main.rs

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

Cargo has generated a Hello, world! program fo
Listing 1-1! So far, the differences between our p
generates are that Cargo placed the code in the
Cargo.toml configuration file in the top directory

Cargo expects your source files to live inside the
directory is just for README files, license inform
anything else not related to your code. Using Ca
There's a place for everything, and everything is

If you started a project that doesn't use Cargo, a
you can convert it to a project that does use Car
directory and create an appropriate *Cargo.toml* f

Building and Running a Cargo Project

Now let's look at what's different when we build
with Cargo! From your *hello_cargo* directory, buil
following command:

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///p
    Finished dev [unoptimized + debuginfo]
```

This command creates an executable file in *targ*
\hello_cargo.exe on Windows) rather than in your
executable with this command:

```
$ ./target/debug/hello_cargo # or .\target
Windows
Hello, world!
```

If all goes well, `Hello, world!` should print to tl
for the first time also causes Cargo to create a n
file keeps track of the exact versions of depende
doesn't have dependencies, so the file is a bit sp
this file manually; Cargo manages its contents fo

We just built a project with `cargo build` and ra
`./target/debug/hello_cargo`, but we can also

and then run the resulting executable all in one

```
$ cargo run
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/hello_cargo`
Hello, world!
```

Notice that this time we didn't see output indica `hello_cargo` . Cargo figured out that the files ha If you had modified your source code, Cargo wo running it, and you would have seen this output

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///p
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/hello_cargo`
Hello, world!
```

Cargo also provides a command called `cargo c` your code to make sure it compiles but doesn't p

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///pr
    Finished dev [unoptimized + debuginfo]
```

Why would you not want an executable? Often, `cargo build` , because it skips the step of produ continually checking your work while writing the up the process! As such, many Rustaceans run ` their program to make sure it compiles. Then th ready to use the executable.

Let's recap what we've learned so far about Carg

- We can build a project using `cargo build`
- We can build and run a project in one step
- Instead of saving the result of the build in t stores it in the *target/debug* directory.

An additional advantage of using Cargo is that th which operating system you're working on. So, a specific instructions for Linux and macOS versus

Building for Release

When your project is finally ready for release, yc compile it with optimizations. This command wil instead of *target/debug*. The optimizations make turning them on lengthens the time it takes for y there are two different profiles: one for develop quickly and often, and another for building the f won't be rebuilt repeatedly and that will run as f benchmarking your code's running time, be sure benchmark with the executable in *target/release*.

Cargo as Convention

With simple projects, Cargo doesn't provide a lot will prove its worth as your programs become m composed of multiple crates, it's much easier to

Even though the `hello_cargo` project is simple, you'll use in the rest of your Rust career. In fact, can use the following commands to check out th project's directory, and build:

```
$ git clone someurl.com/someproject
$ cd someproject
$ cargo build
```

For more information about Cargo, check out its

Summary

You're already off to a great start on your Rust jc how to:

- Install the latest stable version of Rust usin
- Update to a newer Rust version
- Open locally installed documentation
- Write and run a Hello, world! program usin
- Create and run a new project using the cor

This is a great time to build a more substantial p writing Rust code. So, in Chapter 2, we'll build a g rather start by learning how common programn

Chapter 3 and then return to Chapter 2.

Programming a Guessi

Let's jump into Rust by working through a hands
introduces you to a few common Rust concepts
real program. You'll learn about `let`, `match`, m
external crates, and more! The following chapte
detail. In this chapter, you'll practice the fundam

We'll implement a classic beginner programmin
how it works: the program will generate a rando
then prompt the player to enter a guess. After a
indicate whether the guess is too low or too high
print a congratulatory message and exit.

Setting Up a New Project

To set up a new project, go to the *projects* direct
make a new project using Cargo, like so:

```
$ cargo new guessing_game
$ cd guessing_game
```

The first command, `cargo new`, takes the name
the first argument. The second command chang

Look at the generated *Cargo.toml* file:

Filename: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

If the author information that Cargo obtained fr
fix that in the file and save it again.

As you saw in Chapter 1, `cargo new` generates a
Check out the *src/main.rs* file:

Filename: src/main.rs

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

Now let's compile this "Hello, world!" program ar
`cargo run` command:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:/,
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/guessing_game`
Hello, world!
```

The `run` command comes in handy when you n
we'll do in this game, quickly testing each iteratio

Reopen the *src/main.rs* file. You'll be writing all tl

Processing a Guess

The first part of the guessing game program will
and check that the input is in the expected form
input a guess. Enter the code in Listing 2-1 into s

Filename: src/main.rs

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Listing 2-1: Code that gets a guess from the user

This code contains a lot of information, so let's g

input and then print the result as output, we nee
library into scope. The `io` library comes from th
`std`):

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

By default, Rust brings only a few types into the
prelude. If a type you want to use isn't in the prel
scope explicitly with a `use` statement. Using the
number of useful features, including the ability t

As you saw in Chapter 1, the `main` function is th

```
fn main() {
```

The `fn` syntax declares a new function, the pare
parameters, and the curly bracket, `{`, starts the

As you also learned in Chapter 1, `println!` is a
screen:

```
println!("Guess the number!");

println!("Please input your guess.");
```

This code is printing a prompt stating what the g
user.

Storing Values with Variables

Next, we'll create a place to store the user input,

```
let mut guess = String::new();
```

Now the program is getting interesting! There's a
that this is a `let` statement, which is used to cre
example:

```
let foo = bar;
```

This line creates a new variable named `foo` and
variable. In Rust, variables are immutable by def
in detail in the "Variables and Mutability" section

shows how to use `mut` before the variable nam

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

> Note: The `//` syntax starts a comment that c
> Rust ignores everything in comments, which
> Chapter 3.

Now you know that `let mut guess` will introdu
On the other side of the equal sign (`=`) is the val
the result of calling `String::new` , a function tha
`String` is a string type provided by the standar
encoded bit of text.

The `::` syntax in the `::new` line indicates that
`String` type. An associated function is impleme
rather than on a particular instance of a `String`
method.

This `new` function creates a new, empty string. Y
types, because it's a common name for a functic
kind.

To summarize, the `let mut guess = String::n`
variable that is currently bound to a new, empty

Recall that we included the input/output functio
`use std::io;` on the first line of the program. N
`stdin` , on `io` :

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

If we hadn't listed the `use std::io` line at the b
have written this function call as `std::io::stdi`
instance of `std::io::Stdin` , which is a type tha
input for your terminal.

The next part of the code, `.read_line(&mut gue`
the standard input handle to get input from the
argument to `read_line` : `&mut guess` .

The job of `read_line` is to take whatever the us place that into a string, so it takes that string as needs to be mutable so the method can change user input.

The `&` indicates that this argument is a *referenc* parts of your code access one piece of data with memory multiple times. References are a compl advantages is how safe and easy it is to use refe of those details to finish this program. For now, variables, references are immutable by default. rather than `&guess` to make it mutable. (Chapte thoroughly.)

Handling Potential Failure with the `Re`

We're not quite done with this line of code. Altho single line of text, it's only the first part of the si part is this method:

```
.expect("Failed to read line");
```

When you call a method with the `.foo()` synta and other whitespace to help break up long line

```
io::stdin().read_line(&mut guess).expect('
```

However, one long line is difficult to read, so it's method calls. Now let's discuss what this line do

As mentioned earlier, `read_line` puts what the passing it, but it also returns a value—in this cas of types named `Result` in its standard library: a versions for submodules, such as `io::Result`.

The `Result` types are *enumerations*, often refer type that can have a fixed set of values, and tho *variants*. Chapter 6 will cover enums in more de

For `Result`, the variants are `Ok` or `Err`. The `Ok` successful, and inside `Ok` is the successfully ge the operation failed, and `Err` contains informat failed.

The purpose of these `Result` types is to encode
of the `Result` type, like any type, have methods
`io::Result` has an `expect` method that you ca
is an `Err` value, `expect` will cause the program
that you passed as an argument to `expect` . If th
, it would likely be the result of an error coming
system. If this instance of `io::Result` is an `Ok`
value that `Ok` is holding and return just that val
case, that value is the number of bytes in what t

If you don't call `expect` , the program will compi

```
$ cargo build
   Compiling guessing_game v0.1.0 (file://
warning: unused `std::result::Result` whic
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess)
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by
```

Rust warns that you haven't used the `Result` va
indicating that the program hasn't handled a po

The right way to suppress the warning is to actu
you just want to crash this program when a prol
You'll learn about recovering from errors in Cha

Printing Values with `println!` Placeho

Aside from the closing curly brackets, there's on
added so far, which is the following:

```
println!("You guessed: {}", guess);
```

This line prints the string we saved the user's inp
a placeholder: think of `{}` as little crab pincers t
print more than one value using curly brackets:
the first value listed after the format string, the s
and so on. Printing multiple values in one call to

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

This code would print `x = 5 and y = 10`.

Testing the First Part

Let's test the first part of the guessing game. Ru

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:/
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

At this point, the first part of the game is done: v and then printing it.

Generating a Secret Number

Next, we need to generate a secret number that number should be different every time so the ga Let's use a random number between 1 and 100 doesn't yet include random number functionalit Rust team does provide a `rand` crate.

Using a Crate to Get More Functionalit

Remember that a crate is a package of Rust cod *binary crate*, which is an executable. The `rand` c code intended to be used in other programs.

Cargo's use of external crates is where it really s uses `rand`, we need to modify the *Cargo.toml* fil dependency. Open that file now and add the fol [dependencies] section header that Cargo crea

Filename: Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

In the *Cargo.toml* file, everything that follows a h continues until another section starts. The [dep Cargo which external crates your project depen crates you require. In this case, we'll specify the specifier 0.3.14 . Cargo understands Semantic which is a standard for writing version numbers shorthand for ^0.3.14 , which means "any versi with version 0.3.14."

Now, without changing any of the code, let's bui

```
$ cargo build
    Updating registry `https://github.com/
 Downloading rand v0.3.14
 Downloading libc v0.2.14
   Compiling libc v0.2.14
   Compiling rand v0.3.14
   Compiling guessing_game v0.1.0 (file://
    Finished dev [unoptimized + debuginfo]
```

Listing 2-2: The output from running cargo bui dependency

You may see different version numbers (but the thanks to SemVer!), and the lines may be in a dif

Now that we have an external dependency, Carg everything from the *registry*, which is a copy of d people in the Rust ecosystem post their open sc

After updating the registry, Cargo checks the [d any crates you don't have yet. In this case, altho dependency, Cargo also grabbed a copy of libc work. After downloading the crates, Rust compil project with the dependencies available.

If you immediately run cargo build again with get any output aside from the Finished line. Ca

and compiled the dependencies, and you haven
your *Cargo.toml* file. Cargo also knows that you h
code, so it doesn't recompile that either. With no

If you open up the *src/main.rs* file, make a trivial
again, you'll only see two lines of output:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:/
    Finished dev [unoptimized + debuginfo]
```

These lines show Cargo only updates the build w
src/main.rs file. Your dependencies haven't chan
what it has already downloaded and compiled fo
the code.

Ensuring Reproducible Builds with the *Cargo.*

Cargo has a mechanism that ensures you can re
or anyone else builds your code: Cargo will use o
you specified until you indicate otherwise. For ex
version `v0.3.15` of the `rand` crate comes out a
also contains a regression that will break your co

The answer to this problem is the *Cargo.lock* file,
ran `cargo build` and is now in your *guessing_ga*
project for the first time, Cargo figures out all th
the criteria and then writes them to the *Cargo.lo*
the future, Cargo will see that the *Cargo.lock* file
there rather than doing all the work of figuring o
a reproducible build automatically. In other wor
until you explicitly upgrade, thanks to the *Cargo.*

Updating a Crate to Get a New Version

When you *do* want to update a crate, Cargo prov
which will ignore the *Cargo.lock* file and figure ou
specifications in *Cargo.toml*. If that works, Cargo
Cargo.lock file.

But by default, Cargo will only look for versions l
`0.4.0`. If the `rand` crate has released two new
would see the following if you ran `cargo update`

```
$ cargo update
    Updating registry `https://github.com/
    Updating rand v0.3.14 -> v0.3.15
```

At this point, you would also notice a change in y
version of the `rand` crate you are now using is

If you wanted to use `rand` version `0.4.0` or any
have to update the *Cargo.toml* file to look like thi

```
[dependencies]

rand = "0.4.0"
```

The next time you run `cargo build`, Cargo will
and reevaluate your `rand` requirements accord
specified.

There's a lot more to say about Cargo and its ec
Chapter 14, but for now, that's all you need to kn
reuse libraries, so Rustaceans are able to write s
from a number of packages.

Generating a Random Number

Now that you've added the `rand` crate to *Cargo.*
step is to update *src/main.rs*, as shown in Listing

Filename: src/main.rs

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng()

    println!("The secret number is: {}", s

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Listing 2-3: Adding code to generate a random n

First, we add a line that lets Rust know we'll be u dependency. This also does the equivalent of ca anything in the `rand` crate by placing `rand::` b

Next, we add another `use` line: `use rand::Rng` random number generators implement, and thi: those methods. Chapter 10 will cover traits in de

Also, we're adding two more lines in the middle. give us the particular random number generato local to the current thread of execution and see call the `gen_range` method on the random num defined by the `Rng` trait that we brought into sc statement. The `gen_range` method takes two nu a random number between them. It's inclusive c the upper bound, so we need to specify `1` and and 100.

> Note: You won't just know which traits to use to call from a crate. Instructions for using a cr documentation. Another neat feature of Carg `cargo doc --open` command, which will buil

your dependencies locally and open it in your
other functionality in the `rand` crate, for exar
click `rand` in the sidebar on the left.

The second line that we added to the code print
while we're developing the program to be able t
final version. It's not much of a game if the progr
starts!

Try running the program a few times:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file://
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

You should get different random numbers, and
1 and 100. Great job!

Comparing the Guess to the Se

Now that we have user input and a random num
is shown in Listing 2-4. Note that this code won't

Filename: src/main.rs

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    // ---snip---

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too sm
        Ordering::Greater => println!("Too
        Ordering::Equal => println!("You w
    }
}
```

Listing 2-4: Handling the possible return values o

The first new bit here is another `use` statement `std::cmp::Ordering` into scope from the stand another enum, but the variants for `Ordering` ar are the three outcomes that are possible when y

Then we add five new lines at the bottom that u

The `cmp` method compares two values and can compared. It takes a reference to whatever you comparing the `guess` to the `secret_number`. Th `Ordering` enum we brought into scope with the expression to decide what to do next based on w returned from the call to `cmp` with the values in

A `match` expression is made up of *arms*. An arm that should be run if the value given to the begin that arm's pattern. Rust takes the value given to pattern in turn. The `match` construct and patter let you express a variety of situations your code you handle them all. These features will be cove 18, respectively.

Let's walk through an example of what would ha used here. Say that the user has guessed 50 and number this time is 38. When the code compare return `Ordering::Greater`, because 50 is great gets the `Ordering::Greater` value and starts ch

the first arm's pattern, `Ordering::Less`, and se
does not match `Ordering::Less`, so it ignores t
next arm. The next arm's pattern, `Ordering::Gr`
`Ordering::Greater`! The associated code in tha
`Too big!` to the screen. The `match` expression
at the last arm in this scenario.

However, the code in Listing 2-4 won't compile y

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:/
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^ ex
`std::string::String`, found integral var
   |
   = note: expected type `&std::string::St
   = note:    found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

The core of the error states that there are *mism*
type system. However, it also has type inference
`let mut guess = String::new();`, Rust was ab
`String` and didn't make us write the type. The
a number type. A few number types can have a
bit number; `u32`, an unsigned 32-bit number; `i`
others. Rust defaults to an `i32`, which is the typ
type information elsewhere that would cause Ru
The reason for the error is that Rust cannot com

Ultimately, we want to convert the `String` the p
number type so we can compare it numerically t
adding the following two lines to the `main` funct

Filename: src/main.rs

```
// --snip--

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too sm
        Ordering::Greater => println!("Too
        Ordering::Equal => println!("You w
    }
}
```

The two new lines are:

```
let guess: u32 = guess.trim().parse()
    .expect("Please type a number!");
```

We create a variable named `guess` . But wait, dc variable named `guess` ? It does, but Rust allows `guess` with a new one. This feature is often use convert a value from one type to another type. S variable name rather than forcing us to create tv and `guess` for example. (Chapter 3 covers shad

We bind `guess` to the expression `guess.trim()` expression refers to the original `guess` that was `trim` method on a `String` instance will elimina and end. Although `u32` can contain only numer enter to satisfy `read_line` . When the user pres added to the string. For example, if the user typ like this: `5\n` . The `\n` represents "newline," the method eliminates `\n` , resulting in just `5` .

The `parse` method on strings parses a string in method can parse a variety of number types, we type we want by using `let guess: u32` . The col annotate the variable's type. Rust has a few built is an unsigned, 32-bit integer. It's a good default You'll learn about other number types in Chapte

in this example program and the comparison wi
will infer that `secret_number` should be a `u32` a
between two values of the same type!

The call to `parse` could easily cause an error. If,
`A👍%`, there would be no way to convert that to
`parse` method returns a `Result` type, much as
(discussed earlier in "Handling Potential Failure w
`Result` the same way by using the `expect` met
`Result` variant because it couldn't create a num
will crash the game and print the message we gi
convert the string to a number, it will return the
will return the number that we want from the `O

Let's run the program now!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:/
    Finished dev [unoptimized + debuginfo]
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! Even though spaces were added before th
that the user guessed 76. Run the program a few
behavior with different kinds of input: guess the
that is too high, and guess a number that is too

We have most of the game working now, but the
change that by adding a loop!

Allowing Multiple Guesses wit

The `loop` keyword creates an infinite loop. We'l
chances at guessing the number:

Filename: src/main.rs

```
// --snip--

    println!("The secret number is: {}", s

    loop {
        println!("Please input your guess.

        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("To
            Ordering::Greater => println!(
            Ordering::Equal => println!("Y
        }
    }
}
```

As you can see, we've moved everything into a lo
onward. Be sure to indent the lines inside the lo
the program again. Notice that there is a new pr
exactly what we told it to do: ask for another gu
user can quit!

The user could always halt the program by using
there's another way to escape this insatiable mo
discussion in "Comparing the Guess to the Secre
number answer, the program will crash. The use
to quit, as shown here:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:/
    Finished dev [unoptimized + debuginfo]
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a r
InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a ba
error: Process didn't exit successfully: `
101)
```

Typing `quit` actually quits the game, but so will
However, this is suboptimal to say the least. We
when the correct number is guessed.

Quitting After a Correct Guess

Let’s program the game to quit when the user w

Filename: src/main.rs

```
// --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("To
            Ordering::Greater => println!(
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Adding the `break` line after `You win!` makes th
user guesses the secret number correctly. Exitin
program, because the loop is the last part of `ma

Handling Invalid Input

To further refine the game's behavior, rather tha
user inputs a non-number, let's make the game
continue guessing. We can do that by altering th
a `String` to a `u32`:

```
let guess: u32 = match guess.trim().parse(
    Ok(num) => num,
    Err(_) => continue,
};
```

Switching from an `expect` call to a `match` expre
from crashing on an error to handling the error.
`Result` type and `Result` is an enum that has th
`match` expression here, as we did with the `Orde

If `parse` is able to successfully turn the string in
value that contains the resulting number. That `(
pattern, and the `match` expression will just retu
produced and put inside the `Ok` value. That nur
it in the new `guess` variable we're creating.

If `parse` is *not* able to turn the string into a num
contains more information about the error. The
`Ok(num)` pattern in the first `match` arm, but it d
second arm. The underscore, `_`, is a catchall val
want to match all `Err` values, no matter what ir
the program will execute the second arm's code
next iteration of the `loop` and ask for another g
ignores all errors that `parse` might encounter!

Now everything in the program should work as (

```
$ cargo run
   Compiling guessing_game v0.1.0 (file://
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny final tweak, we will finis
program is still printing the secret number. That
the game. Let's delete the `println!` that outpu
shows the final code:

Filename: src/main.rs

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng()

    loop {
        println!("Please input your guess.

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line")

        let guess: u32 = match guess.trim(
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess)

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("To
            Ordering::Greater => println!(
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Listing 2-5: Complete guessing game code

Summary

At this point, you've successfully built the guessi

This project was a hands-on way to introduce yo `match`, methods, associated functions, the use o next few chapters, you'll learn about these conc concepts that most programming languages hav

functions, and shows how to use them in Rust. C
feature that makes Rust different from other lar
and method syntax, and Chapter 6 explains how

Common Programmin

This chapter covers concepts that appear in alm
how they work in Rust. Many programming lang
core. None of the concepts presented in this cha
discuss them in the context of Rust and explain
concepts.

Specifically, you'll learn about variables, basic ty
flow. These foundations will be in every Rust pro
give you a strong core to start from.

Keywords

The Rust language has a set of *keywords* that
language only, much as in other languages. K
these words as names of variables or functio
special meanings, and you'll be using them to
programs; a few have no current functionalit
been reserved for functionality that might be
can find a list of the keywords in Appendix A.

Variables and Mutability

As mentioned in Chapter 2, by default variables
nudges Rust gives you to write your code in a wa
and easy concurrency that Rust offers. However
your variables mutable. Let's explore how and w
immutability and why sometimes you might wan

When a variable is immutable, once a value is bo
value. To illustrate this, let's generate a new proj
directory by using `cargo new variables` .

Then, in your new *variables* directory, open *src/m*
following code that won't compile just yet:

Filename: src/main.rs

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Save and run the program using `cargo run`. Yo
shown in this output:

```
error[E0384]: cannot assign twice to immut
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
3 |     println!("The value of x is: {}",
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immut
```

This example shows how the compiler helps you
though compiler errors can be frustrating, they
doing what you want it to do yet; they do *not* me
programmer! Experienced Rustaceans still get c

The error indicates that the cause of the error is
`cannot assign twice to immutable variable`
second value to the immutable `x` variable.

It's important that we get compile-time errors w
that we previously designated as immutable bec
bugs. If one part of our code operates on the as
change and another part of our code changes th
part of the code won't do what it was designed t
can be difficult to track down after the fact, espe
changes the value only *sometimes*.

In Rust, the compiler guarantees that when you
really won't change. That means that when you'r
have to keep track of how and where a value mi
to reason through.

But mutability can be very useful. Variables are i

in Chapter 2, you can make them mutable by ad
name. In addition to allowing this value to chang
readers of the code by indicating that other part
variable value.

For example, let's change *src/main.rs* to the follo

Filename: src/main.rs

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

When we run the program now, we get this:

```
$ cargo run
   Compiling variables v0.1.0 (file:///pro
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

We're allowed to change the value that `x` binds
some cases, you'll want to make a variable muta
convenient to write than if it had only immutabl

There are multiple trade-offs to consider in addi
example, in cases where you're using large data
place may be faster than copying and returning
smaller data structures, creating new instances
programming style may be easier to think throu
worthwhile penalty for gaining that clarity.

Differences Between Variables and Co

Being unable to change the value of a variable m
programming concept that most other language
variables, constants are values that are bound t
change, but there are a few differences betwee

First, you aren't allowed to use `mut` with consta
by default—they're always immutable.

You declare constants using the `const` keyworc
type of the value *must* be annotated. We're abou
in the next section, "Data Types," so don't worry
that you must always annotate the type.

Constants can be declared in any scope, includir
them useful for values that many parts of code r

The last difference is that constants may be set (
result of a function call or any other value that c

Here's an example of a constant declaration wh
`MAX_POINTS` and its value is set to 100,000. (Rus
use all uppercase with underscores between wo
in numeric literals to improve readability):

```
const MAX_POINTS: u32 = 100_000;
```

Constants are valid for the entire time a prograr
declared in, making them a useful choice for val
multiple parts of the program might need to kno
number of points any player of a game is allowe

Naming hardcoded values used throughout you
conveying the meaning of that value to future m
have only one place in your code you would nee
needed to be updated in the future.

Shadowing

As you saw in the "Comparing the Guess to the S
you can declare a new variable with the same na
new variable shadows the previous variable. Rus
shadowed by the second, which means that the s
appears when the variable is used. We can shad
variable's name and repeating the use of the `le`

Filename: src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("The value of x is: {}", x);
}
```

This program first binds `x` to a value of `5`. Then
taking the original value and adding `1` so the va
statement also shadows `x`, multiplying the prev
value of `12`. When we run this program, it will o

```
$ cargo run
   Compiling variables v0.1.0 (file:///pro
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/variables`
The value of x is: 12
```

Shadowing is different than marking a variable a
time error if we accidentally try to reassign to th
keyword. By using `let`, we can perform a few tr
the variable be immutable after those transform

The other difference between `mut` and shadowi
creating a new variable when we use the `let` ke
of the value but reuse the same name. For exam
show how many spaces they want between som
but we really want to store that input as a numb

```
let spaces = "   ";
let spaces = spaces.len();
```

This construct is allowed because the first `space
second `spaces` variable, which is a brand-new v
same name as the first one, is a number type. Sh
to come up with different names, such as `space
can reuse the simpler `spaces` name. However, i
here, we'll get a compile-time error:

```
let mut spaces = "   ";
spaces = spaces.len();
```

The error says we're not allowed to mutate a vai

```
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected &st
  |
  = note: expected type `&str`
             found type `usize`
```

Now that we've explored how variables work, let
have.

Data Types

Every value in Rust is of a certain *data type*, whic
specified so it knows how to work with that data
scalar and compound.

Keep in mind that Rust is a *statically typed* langu
the types of all variables at compile time. The co
want to use based on the value and how we use
possible, such as when we converted a `String`
"Comparing the Guess to the Secret Number" se
type annotation, like this:

```
let guess: u32 = "42".parse().expect("Not
```

If we don't add the type annotation here, Rust w
means the compiler needs more information fro
use:

```
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("N
  |         ^^^^^
  |         |
  |         cannot infer type for `_`
  |         consider giving `guess` a type
```

You'll see different type annotations for other da

Scalar Types

A *scalar* type represents a single value. Rust has
floating-point numbers, Booleans, and characte
other programming languages. Let's jump into h

Integer Types

An *integer* is a number without a fractional comp
Chapter 2, the `u32` type. This type declaration ir
with should be an unsigned integer (signed integ
that takes up 32 bits of space. Table 3-1 shows t
variant in the Signed and Unsigned columns (for
declare the type of an integer value.

Table 3-1: Integer Types in Rust

Length	Signed
8-bit	`i8`
16-bit	`i16`
32-bit	`i32`
64-bit	`i64`
128-bit	`i128`
arch	`isize`

Each variant can be either signed or unsigned ar
unsigned refer to whether it's possible for the nu
other words, whether the number needs to hav
will only ever be positive and can therefore be r
It's like writing numbers on paper: when the sign
plus sign or a minus sign; however, when it's saf
it's shown with no sign. Signed numbers are sto
representation (if you're unsure what this is, you
explanation is outside the scope of this book).

Each signed variant can store numbers from -(2
the number of bits that variant uses. So an `i8` c
which equals -128 to 127. Unsigned variants can
`u8` can store numbers from 0 to 2^8 - 1, which ec

Additionally, the `isize` and `usize` types depen

program is running on: 64 bits if you're on a 64-l
on a 32-bit architecture.

You can write integer literals in any of the forms
number literals except the byte literal allow a ty
visual separator, such as `1_000`.

Table 3-2: Integer Literals in Rust

Number literals	
Decimal	`9`
Hex	`0`
Octal	`0`
Binary	`0`
Byte (`u8` only)	`b`

So how do you know which type of integer to us
generally good choices, and integer types defaul
fastest, even on 64-bit systems. The primary situ
`usize` is when indexing some sort of collection.

Integer Overflow

Let's say that you have a `u8`, which can hold val
happens if you try to change it to `256`? This is c
some interesting rules around this behavior. Wh
checks for this kind of issue and will cause your
Rust uses when a program exits with an error. W

In release builds, Rust does not check for overfl
called "two's complement wrapping." In short, `2`
etc. Relying on overflow is considered an error, e
want this behavior explicitly, the standard librar
it explicitly.

Floating-Point Types

Rust also has two primitive types for *floating-poi*
decimal points. Rust's floating-point types are `f`
bits in size, respectively. The default type is `f64`
the same speed as `f32` but is capable of more p

Here's an example that shows floating-point nur

Filename: src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Floating-point numbers are represented accordi
type is a single-precision float, and `f64` has dou

Numeric Operations

Rust supports the basic mathematical operation
types: addition, subtraction, multiplication, divis
code shows how you'd use each one in a `let` st

Filename: src/main.rs

```
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;

    // remainder
    let remainder = 43 % 5;
}
```

Each expression in these statements uses a mat
single value, which is then bound to a variable. A
operators that Rust provides.

The Boolean Type

As in most other programming languages, a Boo
values: `true` and `false`. The Boolean type in R
example:

Filename: src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit
}
```

The main way to consume Boolean values is thr
expression. We'll cover how `if` expressions wo
section.

Booleans are one byte in size.

The Character Type

So far we've worked only with numbers, but Rus
type is the language's most primitive alphabetic
one way to use it. (Note that the `char` literal is s
opposed to string literals, which use double quo

Filename: src/main.rs

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let heart_eyed_cat = '😻';
}
```

Rust's `char` type represents a Unicode Scalar Va
lot more than just ASCII. Accented letters; Chine
emoji; and zero-width spaces are all valid `char`
range from `U+0000` to `U+D7FF` and `U+E000` to `U
"character" isn't really a concept in Unicode, so y
"character" is may not match up with what a `cha
detail in "Strings" in Chapter 8.

Compound Types

Compound types can group multiple values into o
compound types: tuples and arrays.

The Tuple Type

A tuple is a general way of grouping together so
variety of types into one compound type. Tuples
they cannot grow or shrink in size.

We create a tuple by writing a comma-separatec
Each position in the tuple has a type, and the typ
don't have to be the same. We've added optiona

Filename: src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1
}
```

The variable `tup` binds to the entire tuple, beca
compound element. To get the individual values
matching to destructure a tuple value, like this:

Filename: src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

This program first creates a tuple and binds it to
pattern with `let` to take `tup` and turn it into th
This is called *destructuring*, because it breaks the
the program prints the value of `y`, which is `6.4`

In addition to destructuring through pattern ma
directly by using a period (`.`) followed by the in
For example:

Filename: src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

This program creates a tuple, `x`, and then make
using their index. As with most programming lan

The Array Type

Another way to have a collection of multiple valu
every element of an array must have the same t
arrays in some other languages because arrays

In Rust, the values going into an array are writte
square brackets:

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

Arrays are useful when you want your data alloc
heap (we will discuss the stack and the heap mo
ensure you always have a fixed number of elem
vector type, though. A vector is a similar collecti
library that *is* allowed to grow or shrink in size. I
array or a vector, you should probably use a vec
more detail.

An example of when you might want to use an a
program that needs to know the names of the m
that such a program will need to add or remove
because you know it will always contain 12 item

```
let months = ["January", "February", "Marc
"July",
              "August", "September", "Octo
```

Arrays have an interesting type; it looks like this:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

First, there's square brackets; they look like the s
there's two pieces of information, separated by
each element of the array. Since all elements ha
it once. After the semicolon, there's a number th
Since an array has a fixed size, this number is alw
elements are modified, it cannot grow or shrink.

Accessing Array Elements

An array is a single chunk of memory allocated o
of an array using indexing, like this:

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

In this example, the variable named `first` will g
value at index `[0]` in the array. The variable nam
from index `[1]` in the array.

Invalid Array Element Access

What happens if you try to access an element of
array? Say you change the example to the follow
with an error when it runs:

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

    println!("The value of element is: {}"
}
```

Running this code using `cargo run` produces th

```
$ cargo run
   Compiling arrays v0.1.0 (file:///projec
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/arrays`
thread '<main>' panicked at 'index out of
index is
 10', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for a ba
```

The compilation didn't produce any errors, but t
error and didn't exit successfully. When you atte
indexing, Rust will check that the index you've sp
the index is greater than the length, Rust will pa

This is the first example of Rust's safety principle
languages, this kind of check is not done, and wh
invalid memory can be accessed. Rust protects y
immediately exiting instead of allowing the mem
discusses more of Rust's error handling.

Functions

Functions are pervasive in Rust code. You've alre
functions in the language: the `main` function, wh
programs. You've also seen the `fn` keyword, wh
functions.

Rust code uses *snake case* as the conventional st
In snake case, all letters are lowercase and unde
program that contains an example function defi

Filename: src/main.rs

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

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Function definitions in Rust start with `fn` and ha
function name. The curly brackets tell the compi
and ends.

We can call any function we've defined by enteri
parentheses. Because `another_function` is def
from inside the `main` function. Note that we def
`main` function in the source code; we could hav
doesn't care where you define your functions, o

Let's start a new binary project named *functions*
the `another_function` example in *src/main.rs* a
output:

```
$ cargo run
   Compiling functions v0.1.0 (file:///pro
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/functions`
Hello, world!
Another function.
```

The lines execute in the order in which they app
"Hello, world!" message prints, and then `anothe
is printed.

Function Parameters

Functions can also be defined to have *parameter*
part of a function's signature. When a function h
with concrete values for those parameters. Tech
arguments, but in casual conversation, people te
argument interchangeably for either the variable
concrete values passed in when you call a functi

The following rewritten version of `another_func
like in Rust:

Filename: src/main.rs

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {}", x);
}
```

Try running this program; you should get the fol

```
$ cargo run
   Compiling functions v0.1.0 (file:///prc
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/functions`
The value of x is: 5
```

The declaration of `another_function` has one p
specified as `i32` . When `5` is passed to `another`
`5` where the pair of curly brackets were in the f

In function signatures, you *must* declare the type
deliberate decision in Rust's design: requiring ty
means the compiler almost never needs you to
figure out what you mean.

When you want a function to have multiple para
declarations with commas, like this:

Filename: src/main.rs

```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

This example creates a function with two param
The function then prints the values in both of its
parameters don't all need to be the same type, t
example.

Let's try running this code. Replace the program
src/main.rs file with the preceding example and

```
$ cargo run
   Compiling functions v0.1.0 (file:///prc
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/functions`
The value of x is: 5
The value of y is: 6
```

Because we called the function with `5` as the va
value for `y`, the two strings are printed with the

Function Bodies

Function bodies are made up of a series of state
expression. So far, we've only covered functions
you have seen an expression as part of stateme
based language, this is an important distinction
have the same distinctions, so let's look at what
how their differences affect the bodies of functi

Statements and Expressions

We've actually already used statements and exp
that perform some action and do not return a v
resulting value. Let's look at some examples.

Creating a variable and assigning a value to it wi
Listing 3-1, `let y = 6;` is a statement:

Filename: src/main.rs

```
fn main() {
    let y = 6;
}
```

Listing 3-1: A `main` function declaration containi

Function definitions are also statements; the en
in itself.

Statements do not return values. Therefore, you
another variable, as the following code tries to d

Filename: src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

When you run this program, the error you'll get

```
$ cargo run
   Compiling functions v0.1.0 (file:///pro
error: expected expression, found statemen
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: variable declaration using `let`
```

The `let y = 6` statement does not return a val bind to. This is different from what happens in c where the assignment returns the value of the a can write `x = y = 6` and have both `x` and `y` h in Rust.

Expressions evaluate to something and make up you'll write in Rust. Consider a simple math oper expression that evaluates to the value `11`. Expr Listing 3-1, the `6` in the statement `let y = 6;` value `6`. Calling a function is an expression. Call block that we use to create new scopes, `{}`, is a

Filename: src/main.rs

```
fn main() {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {}", y);
}
```

This expression:

```
{
    let x = 3;
    x + 1
}
```

is a block that, in this case, evaluates to `4`. That
`let` statement. Note the `x + 1` line without a s
most of the lines you've seen so far. Expressions
you add a semicolon to the end of an expressior
will then not return a value. Keep this in mind as
and expressions next.

Functions with Return Values

Functions can return values to the code that call
but we do declare their type after an arrow (`->`)
function is synonymous with the value of the fin
of a function. You can return early from a functi
specifying a value, but most functions return the
example of a function that returns a value:

Filename: src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {}", x);
}
```

There are no function calls, macros, or even `let`
just the number `5` by itself. That's a perfectly va
function's return type is specified, too, as `-> i3`
should look like this:

```
$ cargo run
   Compiling functions v0.1.0 (file:///prc
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/functions`
The value of x is: 5
```

The `5` in `five` is the function's return value, wh

Let's examine this in more detail. There are two
`let x = five();` shows that we're using the re
variable. Because the function `five` returns a 5
following:

```
let x = 5;
```

Second, the `five` function has no parameters a
value, but the body of the function is a lonely `5`
expression whose value we want to return.

Let's look at another example:

Filename: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Running this code will print `The value of x is`
the end of the line containing `x + 1`, changing i
we'll get an error.

Filename: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

Compiling this code produces an error, as follow

```
error[E0308]: mismatched types
 --> src/main.rs:7:28
  |
7 |   fn plus_one(x: i32) -> i32 {
  |  ____________________________^
8 | |     x + 1;
  | |          - help: consider removing t
9 | | }
  | |_^ expected i32, found ()
  |
  = note: expected type `i32`
             found type `()`
```

The main error message, "mismatched types," re
The definition of the function `plus_one` says tha
statements don't evaluate to a value, which is ex
Therefore, nothing is returned, which contradict
an error. In this output, Rust provides a message
suggests removing the semicolon, which would

Comments

All programmers strive to make their code easy
explanation is warranted. In these cases, progra
their source code that the compiler will ignore b
may find useful.

Here's a simple comment:

```
// Hello, world.
```

In Rust, comments must start with two slashes a
For comments that extend beyond a single line,
line, like this:

```
// So we're doing something complicated he
// multiple lines of comments to do it! Wh
// explain what's going on.
```

Comments can also be placed at the end of lines

Filename: src/main.rs

```
fn main() {
    let lucky_number = 7; // I'm feeling l
}
```

But you'll more often see them used in this form
line above the code it's annotating:

Filename: src/main.rs

```
fn main() {
    // I'm feeling lucky today.
    let lucky_number = 7;
}
```

Rust also has another kind of comment, docume
discuss in Chapter 14.

Control Flow

Deciding whether or not to run some code depe
deciding to run some code repeatedly while a co
blocks in most programming languages. The mo
control the flow of execution of Rust code are `i`

`if` Expressions

An `if` expression allows you to branch your cod
provide a condition and then state, "If this condi
the condition is not met, do not run this block of

Create a new project called *branches* in your *pro*
expression. In the *src/main.rs* file, input the follo

Filename: src/main.rs

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

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

All `if` expressions start with the keyword `if`, w
this case, the condition checks whether or not th
than 5. The block of code we want to execute if t
immediately after the condition inside curly brac
the conditions in `if` expressions are sometimes
`match` expressions that we discussed in the “Co
Number” section of Chapter 2.

Optionally, we can also include an `else` express
give the program an alternative block of code to
evaluate to false. If you don’t provide an `else` e
the program will just skip the `if` block and mov

Try running this code; you should see the follow

```
$ cargo run
   Compiling branches v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/branches`
condition was true
```

Let’s try changing the value of `number` to a value
see what happens:

```
let number = 7;
```

Run the program again, and look at the output:

```
$ cargo run
   Compiling branches v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/branches`
condition was false
```

It’s also worth noting that the condition in this co
isn’t a `bool`, we’ll get an error. For example:

Filename: src/main.rs

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

    if number {
        println!("number was three");
    }
}
```

The `if` condition evaluates to a value of `3` this

```
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected bool, found int
  |
  = note: expected type `bool`
             found type `{integer}`
```

The error indicates that Rust expected a `bool` b
such as Ruby and JavaScript, Rust will not autom
types to a Boolean. You must be explicit and alw
condition. If we want the `if` code block to run o
for example, we can change the `if` expression

Filename: src/main.rs

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

    if number != 0 {
        println!("number was something oth
    }
}
```

Running this code will print `number was someth`

Handling Multiple Conditions with `else if`

You can have multiple conditions by combining
expression. For example:

Filename: src/main.rs

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4
    } else if number % 3 == 0 {
        println!("number is divisible by 3
    } else if number % 2 == 0 {
        println!("number is divisible by 2
    } else {
        println!("number is not divisible
    }
}
```

This program has four possible paths it can take following output:

```
$ cargo run
   Compiling branches v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/branches`
number is divisible by 3
```

When this program executes, it checks each `if` first body for which the condition holds true. No we don't see the output `number is divisible b` `number is not divisible by 4, 3, or 2` text Rust only executes the block for the first true co doesn't even check the rest.

Using too many `else if` expressions can clutte one, you might want to refactor your code. Chap branching construct called `match` for these case

Using `if` in a `let` Statement

Because `if` is an expression, we can use it on t Listing 3-2:

Filename: src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    println!("The value of number is: {}",
}
```

Listing 3-2: Assigning the result of an `if` expres

The `number` variable will be bound to a value ba expression. Run this code to see what happens:

```
$ cargo run
   Compiling branches v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/branches`
The value of number is: 5
```

Remember that blocks of code evaluate to the la by themselves are also expressions. In this case, expression depends on which block of code exe have the potential to be results from each arm c Listing 3-2, the results of both the `if` arm and t the types are mismatched, as in the following ex

Filename: src/main.rs

```
fn main() {
    let condition = true;

    let number = if condition {
        5
    } else {
        "six"
    };

    println!("The value of number is: {}",
}
```

When we try to compile this code, we'll get an er value types that are incompatible, and Rust indi problem in the program:

```
error[E0308]: if and else have incompatibl
 --> src/main.rs:4:18
  |
4 |       let number = if condition {
  |  __________________^
5 | |         5
6 | |     } else {
7 | |         "six"
8 | |     };
  | |_____^ expected integral variable, fo
  |
  = note: expected type `{integer}`
             found type `&str`
```

The expression in the `if` block evaluates to an i `else` block evaluates to a string. This won't wor single type. Rust needs to know at compile time definitively, so it can verify at compile time that i `number`. Rust wouldn't be able to do that if the t at runtime; the compiler would be more comple about the code if it had to keep track of multiple

Repetition with Loops

It's often useful to execute a block of code more provides several *loops*. A loop runs through the and then starts immediately back at the beginni make a new project called *loops*.

Rust has three kinds of loops: `loop`, `while`, and

Repeating Code with `loop`

The `loop` keyword tells Rust to execute a block or until you explicitly tell it to stop.

As an example, change the *src/main.rs* file in you

Filename: src/main.rs

```
fn main() {
    loop {
        println!("again!");
    }
}
```

When we run this program, we'll see `again!` pri
we stop the program manually. Most terminals s
halt a program that is stuck in a continual loop. (

```
$ cargo run
   Compiling loops v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

The symbol `^C` represents where you pressed c
word `again!` printed after the `^C`, depending o
when it received the halt signal.

Fortunately, Rust provides another, more reliabl
place the `break` keyword within the loop to tell
the loop. Recall that we did this in the guessing g
Guess" section of Chapter 2 to exit the program
guessing the correct number.

Returning from loops

One of the uses of a `loop` is to retry an operatio
if a thread completed its job. However, you migh
operation to the rest of your code. If you add it t
stop the loop, it will be returned by the broken l

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    assert_eq!(result, 20);
}
```

Conditional Loops with `while`

It's often useful for a program to evaluate a con
condition is true, the loop runs. When the condi
calls `break`, stopping the loop. This loop type c
combination of `loop`, `if`, `else`, and `break`; yo
you'd like.

However, this pattern is so common that Rust h
called a `while` loop. Listing 3-3 uses `while`: the
down each time, and then, after the loop, it prin

Filename: src/main.rs

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

    while number != 0 {
        println!("{}!", number);

        number = number - 1;
    }

    println!("LIFTOFF!!!");
}
```

Listing 3-3: Using a `while` loop to run code whil

This construct eliminates a lot of nesting that wo
`if`, `else`, and `break`, and it's clearer. While a c
otherwise, it exits the loop.

Looping Through a Collection with `for`

You could use the `while` construct to loop over
an array. For example, let's look at Listing 3-4:

Filename: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[inc

        index = index + 1;
    }
}
```

Listing 3-4: Looping through each element of a c

Here, the code counts up through the elements then loops until it reaches the final index in the a longer true). Running this code will print every e

```
$ cargo run
   Compiling loops v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

All five array values appear in the terminal, as ex reach a value of `5` at some point, the loop stops sixth value from the array.

But this approach is error prone; we could cause length is incorrect. It's also slow, because the co the conditional check on every element on every

As a more concise alternative, you can use a `fo` each item in a collection. A `for` loop looks like t

Filename: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", eleme
    }
}
```

Listing 3-5: Looping through each element of a c

When we run this code, we'll see the same outpu we've now increased the safety of the code and might result from going beyond the end of the a missing some items.

For example, in the code in Listing 3-4, if you rer forgot to update the condition to `while index` `for` loop, you wouldn't need to remember to ch the number of values in the array.

The safety and conciseness of `for` loops make t
construct in Rust. Even in situations in which you
number of times, as in the countdown example
most Rustaceans would use a `for` loop. The wa
which is a type provided by the standard library
sequence starting from one number and ending

Here's what the countdown would look like usin
we've not yet talked about, `rev`, to reverse the r

Filename: src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{}!", number);
    }
    println!("LIFTOFF!!!");
}
```

This code is a bit nicer, isn't it?

Summary

You made it! That was a sizable chapter: you lea
compound data types, functions, comments, `if`
to practice with the concepts discussed in this ch
the following:

- Convert temperatures between Fahrenheit
- Generate the nth Fibonacci number.
- Print the lyrics to the Christmas carol "The
  advantage of the repetition in the song.

When you're ready to move on, we'll talk about a
commonly exist in other programming language

Understanding Owners

Ownership is Rust's most unique feature, and it
guarantees without needing a garbage collector
understand how ownership works in Rust. In thi

as well as several related features: borrowing, sl
memory.

What Is Ownership?

Rust’s central feature is *ownership*. Although the
it has deep implications for the rest of the langu

All programs have to manage the way they use a
Some languages have garbage collection that co
memory as the program runs; in other language
allocate and free the memory. Rust uses a third
through a system of ownership with a set of rule
time. None of the ownership features slow dow

Because ownership is a new concept for many p
to get used to. The good news is that the more e
and the rules of the ownership system, the more
code that is safe and efficient. Keep at it!

When you understand ownership, you’ll have a s
the features that make Rust unique. In this chap
through some examples that focus on a very co

> ### The Stack and the Heap
>
> In many programming languages, you don’t h
> the heap very often. But in a systems prograr
> a value is on the stack or the heap has more c
> behaves and why you have to make certain d
> described in relation to the stack and the hea
> brief explanation in preparation.
>
> Both the stack and the heap are parts of men
> to use at runtime, but they are structured in c
> values in the order it gets them and removes
> This is referred to as *last in, first out*. Think of a
> more plates, you put them on top of the pile,
> take one off the top. Adding or removing plat
> wouldn’t work as well! Adding data is called *p*

data is called *popping off the stack*.

The stack is fast because of the way it accesse
for a place to put new data or a place to get d
always the top. Another property that makes
stack must take up a known, fixed size.

Data with a size unknown at compile time or
stored on the heap instead. The heap is less
the heap, you ask for some amount of space.
empty spot somewhere in the heap that is big
and returns a *pointer*, which is the address of
allocating on the heap, sometimes abbreviated
onto the stack is not considered allocating. Be
size, you can store the pointer on the stack, b
you have to follow the pointer.

Think of being seated at a restaurant. When y
people in your group, and the staff finds an e
leads you there. If someone in your group co
been seated to find you.

Accessing data in the heap is slower than acc
you have to follow a pointer to get there. Con
they jump around less in memory. Continuing
restaurant taking orders from many tables. It
orders at one table before moving on to the n
table A, then an order from table B, then one
again would be a much slower process. By th
its job better if it works on data that's close to
rather than farther away (as it can be on the h
space on the heap can also take time.

When your code calls a function, the values p
potentially, pointers to data on the heap) and
pushed onto the stack. When the function is o
the stack.

Keeping track of what parts of code are using
the amount of duplicate data on the heap, an
heap so you don't run out of space are all pro
Once you understand ownership, you won't n
the heap very often, but knowing that manag
exists can help explain why it works the way i

Ownership Rules

First, let's take a look at the ownership rules. Ke
through the examples that illustrate them:

---

1. Each value in Rust has a variable that's c
2. There can only be one owner at a time.
3. When the owner goes out of scope, the

Variable Scope

We've walked through an example of a Rust pro
we're past basic syntax, we won't include all the
you're following along, you'll have to put the foll
function manually. As a result, our examples wil
focus on the actual details rather than boilerpla

As a first example of ownership, we'll look at the
the range within a program for which an item is
looks like this:

```
let s = "hello";
```

The variable `s` refers to a string literal, where th
into the text of our program. The variable is vali
until the end of the current *scope*. Listing 4-1 ha
variable `s` is valid:

```
{                      // s is not valid h
    let s = "hello";   // s is valid from

    // do stuff with s
}                      // this scope is no
```

Listing 4-1: A variable and the scope in which it i

In other words, there are two important points i

- When `s` comes *into scope*, it is valid.
- It remains valid until it goes *out of scope*.

At this point, the relationship between scopes a
to that in other programming languages. Now w
by introducing the `String` type.

The `String` Type

To illustrate the rules of ownership, we need a d
the ones we covered in the "Data Types" section
previously are all stored on the stack and poppe
over, but we want to look at data that is stored c
knows when to clean up that data.

We'll use `String` as the example here and conc
relate to ownership. These aspects also apply to
by the standard library and that you create. We'l
Chapter 8.

We've already seen string literals, where a string
String literals are convenient, but they aren't sui
may want to use text. One reason is that they're
string value can be known when we write our co
take user input and store it? For these situations
`String`. This type is allocated on the heap and a
text that is unknown to us at compile time. You
literal using the `from` function, like so:

```
let s = String::from("hello");
```

The double colon (`::`) is an operator that allows
`from` function under the `String` type rather th
`string_from`. We'll discuss this syntax more in t
Chapter 5 and when we talk about namespacing
in Chapter 7.

This kind of string *can* be mutated:

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appe

println!("{}", s); // This will print `he
```

So, what's the difference here? Why can `String` difference is how these two types deal with men

Memory and Allocation

In the case of a string literal, we know the conte hardcoded directly into the final executable. Thi efficient. But these properties only come from tl Unfortunately, we can't put a blob of memory in whose size is unknown at compile time and whc the program.

With the `String` type, in order to support a mu to allocate an amount of memory on the heap, u contents. This means:

- The memory must be requested from the
- We need a way of returning this memory t done with our `String`.

That first part is done by us: when we call `Strin` the memory it needs. This is pretty much univer

However, the second part is different. In langua GC keeps track and cleans up memory that isn't need to think about it. Without a GC, it's our res no longer being used and call code to explicitly r Doing this correctly has historically been a diffic forget, we'll waste memory. If we do it too early, it twice, that's a bug too. We need to pair exactly `free`.

Rust takes a different path: the memory is autor that owns it goes out of scope. Here's a version using a `String` instead of a string literal:

```
{
    let s = String::from("hello"); // s is

    // do stuff with s
}                                  // this
                                   // long
```

There is a natural point at which we can return t

operating system: when `s` goes out of scope. W
calls a special function for us. This function is ca
of `String` can put the code to return the memc
the closing `}` .

> Note: In C++, this pattern of deallocating reso
> lifetime is sometimes called *Resource Acquisiti*
> function in Rust will be familiar to you if you'v

This pattern has a profound impact on the way I
simple right now, but the behavior of code can b
situations when we want to have multiple variab
the heap. Let's explore some of those situations

Ways Variables and Data Interact: Move

Multiple variables can interact with the same da
at an example using an integer in Listing 4-2:

```
let x = 5;
let y = x;
```

Listing 4-2: Assigning the integer value of variabl

We can probably guess what this is doing: "bind
of the value in `x` and bind it to `y` ." We now hav
equal `5` . This is indeed what is happening, beca
known, fixed size, and these two `5` values are p

Now let's look at the `String` version:

```
let s1 = String::from("hello");
let s2 = s1;
```

This looks very similar to the previous code, so w
would be the same: that is, the second line woul
bind it to `s2` . But this isn't quite what happens.

Take a look at Figure 4-1 to see what is happenir
`String` is made up of three parts, shown on the
holds the contents of the string, a length, and a

on the stack. On the right is the memory on the

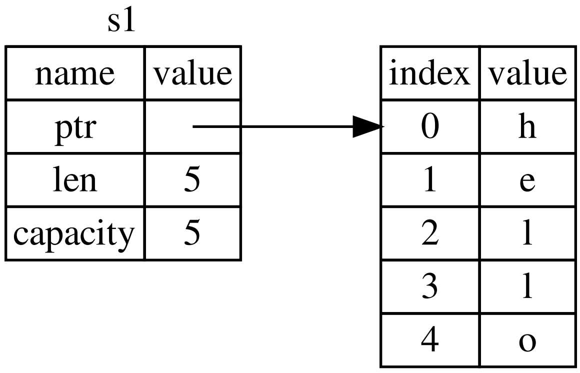

Figure 4-1: Representation in memory of a `Stri`
to `s1`

The length is how much memory, in bytes, the c
using. The capacity is the total amount of memo
received from the operating system. The differe
matters, but not in this context, so for now, it's f

When we assign `s1` to `s2`, the `String` data is c
the length, and the capacity that are on the stac
heap that the pointer refers to. In other words, t
looks like Figure 4-2.

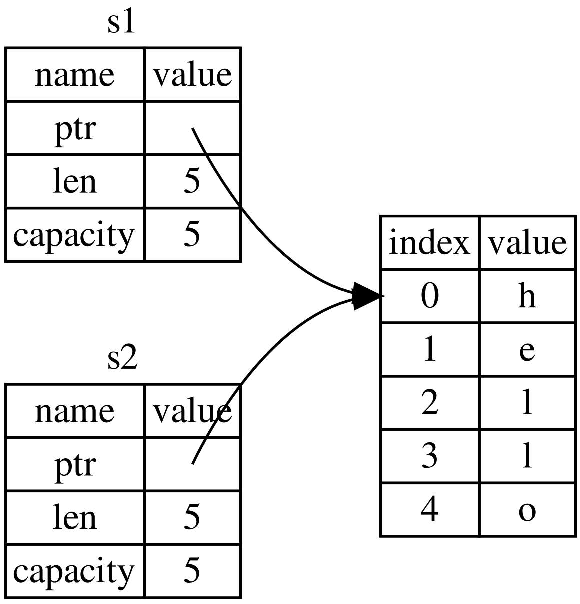

Figure 4-2: Representation in memory of the var
pointer, length, and capacity of `s1`

The representation does *not* look like Figure 4-3,

like if Rust instead copied the heap data as well.
`s2 = s1` could be very expensive in terms of ru
heap were large.

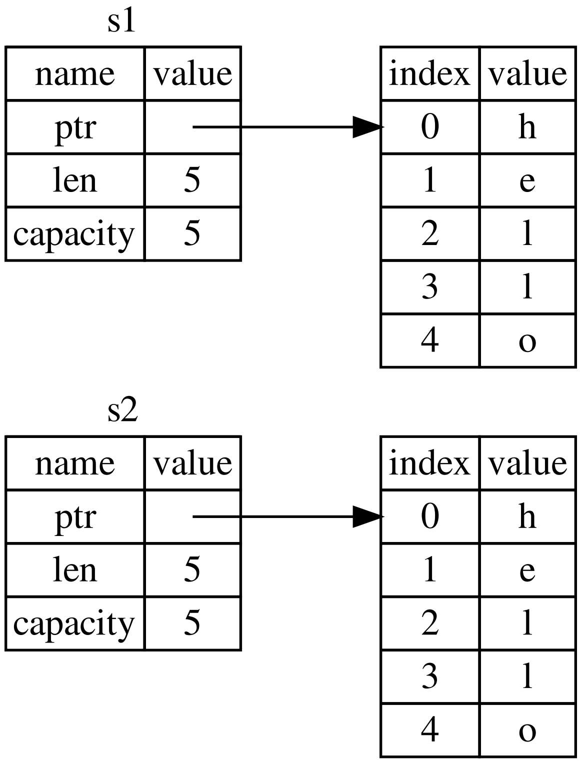

Figure 4-3: Another possibility for what `s2 = s1`
data as well

Earlier, we said that when a variable goes out of
`drop` function and cleans up the heap memory
both data pointers pointing to the same locatior
go out of scope, they will both try to free the sar
free error and is one of the memory safety bugs
memory twice can lead to memory corruption, v
vulnerabilities.

To ensure memory safety, there's one more det
Rust. Instead of trying to copy the allocated mer
be valid and, therefore, Rust doesn't need to fre
scope. Check out what happens when you try to
work:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

You'll get an error like this because Rust prevent
reference:

```
error[E0382]: use of moved value: `s1`
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value us
  |
  = note: move occurs because `s1` has typ
does
  not implement the `Copy` trait
```

If you've heard the terms *shallow copy* and *deep*
languages, the concept of copying the pointer, le
the data probably sounds like making a shallow
invalidates the first variable, instead of being cal
move. Here we would read this by saying that `s1`
actually happens is shown in Figure 4-4.

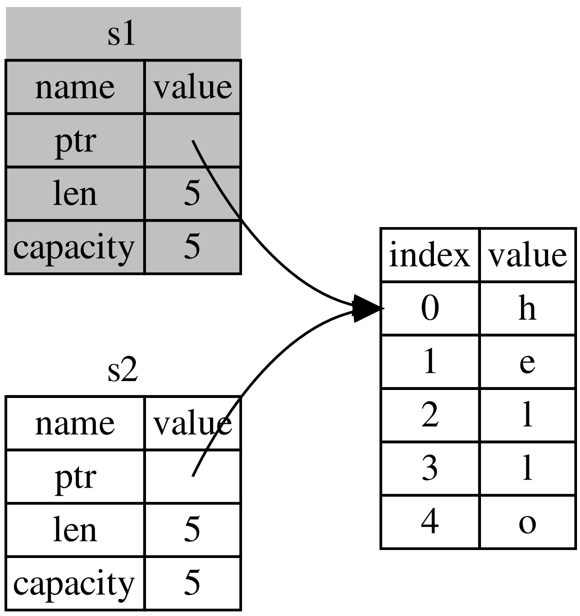

Figure 4-4: Representation in memory after `s1`

That solves our problem! With only `s2` valid, wh
free the memory, and we're done.

In addition, there's a design choice that's implied
create "deep" copies of your data. Therefore, any
to be inexpensive in terms of runtime performa

Ways Variables and Data Interact: Clone

If we *do* want to deeply copy the heap data of th
can use a common method called `clone`. We'll
but because methods are a common feature in
probably seen them before.

Here's an example of the `clone` method in actic

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

This works just fine and explicitly produces the
the heap data *does* get copied.

When you see a call to `clone`, you know that so
and that code may be expensive. It's a visual ind
going on.

Stack-Only Data: Copy

There's another wrinkle we haven't talked about
which was shown earlier in Listing 4-2, works an

```
let x = 5;
let y = x;

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

But this code seems to contradict what we just l
but `x` is still valid and wasn't moved into `y`.

The reason is that types such as integers that ha
stored entirely on the stack, so copies of the act
means there's no reason we would want to prev
create the variable `y`. In other words, there's nc
shallow copying here, so calling `clone` wouldn't
shallow copying and we can leave it out.

Rust has a special annotation called the `Copy` tr
integers that are stored on the stack (we'll talk m

type has the `Copy` trait, an older variable is still let us annotate a type with the `Copy` trait if the t implemented the `Drop` trait. If the type needs s value goes out of scope and we add the `Copy` ar compile time error. To learn about how to add t "Derivable Traits" in Appendix C.

So what types are `Copy` ? You can check the doc sure, but as a general rule, any group of simple s nothing that requires allocation or is some form of the types that are `Copy` :

- All the integer types, such as `u32` .
- The Boolean type, `bool` , with values `true`
- All the floating point types, such as `f64` .
- The character type, `char` .
- Tuples, but only if they contain types that a `(i32, i32)` is `Copy` , but `(i32, String)` i

Ownership and Functions

The semantics for passing a value to a function a value to a variable. Passing a variable to a functi assignment does. Listing 4-3 has an example wit variables go into and out of scope:

Filename: src/main.rs

```
fn main() {
    let s = String::from("hello");  // s c

    takes_ownership(s);             // s's
function...
                                    // ...

    let x = 5;                      // x c

    makes_copy(x);                  // x w
                                    // but
still
                                    // use

} // Here, x goes out of scope, then s. Bu
nothing
  // special happens.

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
} // Here, some_string goes out of scope a
  // memory is freed.

fn makes_copy(some_integer: i32) { // some
    println!("{}", some_integer);
} // Here, some_integer goes out of scope.
```

Listing 4-3: Functions with ownership and scope

If we tried to use `s` after the call to `takes_owne` time error. These static checks protect us from r that uses `s` and `x` to see where you can use th prevent you from doing so.

Return Values and Scope

Returning values can also transfer ownership. Li annotations to those in Listing 4-3:

Filename: src/main.rs

```
fn main() {
    let s1 = gives_ownership();         //
return
                                        //

    let s2 = String::from("hello");     //

    let s3 = takes_and_gives_back(s2);  //
                                        //
also
                                        //
} // Here, s3 goes out of scope and is dr
was
  // moved, so nothing happens. s1 goes ou

fn gives_ownership() -> String {
its

function


    let some_string = String::from("hello'
scope

    some_string
and


}

// takes_and_gives_back will take a String
fn takes_and_gives_back(a_string: String)
into


    a_string  // a_string is returned and
function
}
```

Listing 4-4: Transferring ownership of return val

The ownership of a variable follows the same pa another variable moves it. When a variable that scope, the value will be cleaned up by `drop` unl owned by another variable.

Taking ownership and then returning ownershi What if we want to let a function use a value but annoying that anything we pass in also needs to again, in addition to any data resulting from the want to return as well.

It’s possible to return multiple values using a tup

Filename: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.",
}

fn calculate_length(s: String) -> (String,
    let length = s.len(); // len() returns

    (s, length)
}
```

Listing 4-5: Returning ownership of parameters

But this is too much ceremony and a lot of work
common. Luckily for us, Rust has a feature for th

References and Borrowing

The issue with the tuple code in Listing 4-5 is tha
the calling function so we can still use the `Strin`
because the `String` was moved into `calculate`

Here is how you would define and use a `calcul`
reference to an object as a parameter instead of

Filename: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.",
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

First, notice that all the tuple code in the variabl
value is gone. Second, note that we pass `&s1` in
definition, we take `&String` rather than `String`

These ampersands are *references*, and they allov
taking ownership of it. Figure 4-5 shows a diagra

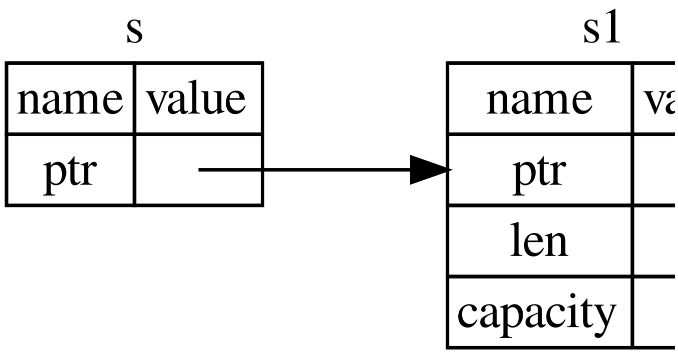

Figure 4-5: A diagram of `&String s` pointing at

> Note: The opposite of referencing by using `&`
> accomplished with the dereference operator,
> dereference operator in Chapter 8 and discus
> Chapter 15.

Let's take a closer look at the function call here:

```
let s1 = String::from("hello");

let len = calculate_length(&s1);
```

The `&s1` syntax lets us create a reference that *r*
own it. Because it does not own it, the value it p
reference goes out of scope.

Likewise, the signature of the function uses `&` to
parameter `s` is a reference. Let's add some exp

```
fn calculate_length(s: &String) -> usize {
    s.len()
} // Here, s goes out of scope. But becaus
what
  // it refers to, nothing happens.
```

The scope in which the variable `s` is valid is the
scope, but we don't drop what the reference poi
because we don't have ownership. When functio
instead of the actual values, we won't need to re
ownership, because we never had ownership.

We call having references as function paramete
owns something, you can borrow it from them.
back.

So what happens if we try to modify something
Listing 4-6. Spoiler alert: it doesn't work!

Filename: src/main.rs

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Listing 4-6: Attempting to modify a borrowed va

Here's the error:

```
error[E0596]: cannot borrow immutable borr
mutable
 --> error.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- use `&m
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ cannot borrow as mutab
```

Just as variables are immutable by default, so ar
modify something we have a reference to.

Mutable References

We can fix the error in the code from Listing 4-6

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

First, we had to change `s` to be `mut`. Then we h
`&mut s` and accept a mutable reference with `so`

But mutable references have one big restriction
reference to a particular piece of data in a partic

Filename: src/main.rs

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;
```

Here's the error:

```
error[E0499]: cannot borrow `s` as mutable
 --> borrow_twice.rs:5:19
  |
4 |     let r1 = &mut s;
  |                   - first mutable borr
5 |     let r2 = &mut s;
  |                   ^ second mutable bor
6 | }
  | - first borrow ends here
```

This restriction allows for mutation but in a very
that new Rustaceans struggle with, because mos
you'd like.

The benefit of having this restriction is that Rust
time. A *data race* is similar to a race condition ar
behaviors occur:

- Two or more pointers access the same dat
- At least one of the pointers is being used t
- There's no mechanism being used to synch

Data races cause undefined behavior and can b
you're trying to track them down at runtime; Ru
happening because it won't even compile code v

As always, we can use curly brackets to create a
mutable references, just not *simultaneous* ones:

```
let mut s = String::from("hello");

{
    let r1 = &mut s;

} // r1 goes out of scope here, so we can
problems.

let r2 = &mut s;
```

A similar rule exists for combining mutable and
results in an error:

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM
```

Here's the error:

```
error[E0502]: cannot borrow `s` as mutable
immutable
 --> borrow_thrice.rs:6:19
  |
4 |     let r1 = &s; // no problem
  |               - immutable borrow occur
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |                   ^ mutable borrow occ
7 | }
  | - immutable borrow ends here
```

Whew! We *also* cannot have a mutable referenc
Users of an immutable reference don't expect th
from under them! However, multiple immutable
who is just reading the data has the ability to aff

Even though these errors may be frustrating at t
compiler pointing out a potential bug early (at co
and showing you exactly where the problem is. T
why your data isn't what you thought it was.

Dangling References

In languages with pointers, it's easy to erroneou
that references a location in memory that may h
freeing some memory while preserving a pointe
the compiler guarantees that references will nev
a reference to some data, the compiler will ensu
scope before the reference to the data does.

Let's try to create a dangling reference, which Ru
error:

Filename: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Here’s the error:

```
error[E0106]: missing lifetime specifier
 --> dangle.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected lifetime par
  |
  = help: this function's return type cont
is
  no value for it to be borrowed from
  = help: consider giving it a 'static li
```

This error message refers to a feature we haven
lifetimes in detail in Chapter 10. But, if you disre
message does contain the key to why this code i

```
this function's return type contains a bor
value
for it to be borrowed from.
```

Let’s take a closer look at exactly what’s happeni

```
fn dangle() -> &String { // dangle return

    let s = String::from("hello"); // s i

    &s // we return a reference to the St
} // Here, s goes out of scope, and is dr
  // Danger!
```

Because `s` is created inside `dangle`, when the
deallocated. But we tried to return a reference t
be pointing to an invalid `String`. That’s no goo

The solution here is to return the `String` direct

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

This works without any problems. Ownership is
deallocated.

The Rules of References

Let's recap what we've discussed about referenc

- At any given time, you can have *either* (but
  or any number of immutable references.
- References must always be valid.

Next, we'll look at a different kind of reference: s

The Slice Type

Another data type that does not have ownershi
contiguous sequence of elements in a collection

Here's a small programming problem: write a fu
the first word it finds in that string. If the functio
the whole string must be one word, so the entir

Let's think about the signature of this function:

```
fn first_word(s: &String) -> ?
```

This function, `first_word`, has a `&String` as a
so this is fine. But what should we return? We d
part of a string. However, we could return the in
that, as shown in Listing 4-7:

Filename: src/main.rs

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumera
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

Listing 4-7: The `first_word` function that returr
parameter

Because we need to go through the `String` ele
value is a space, we'll convert our `String` to an
method:

```
let bytes = s.as_bytes();
```

Next, we create an iterator over the array of byt

```
for (i, &item) in bytes.iter().enumerate()
```

We'll discuss iterators in more detail in Chapter
method that returns each element in a collectio
result of `iter` and returns each element as par
of the tuple returned from `enumerate` is the ind
reference to the element. This is a bit more con
ourselves.

Because the `enumerate` method returns a tuple
that tuple, just like everywhere else in Rust. So i
that has `i` for the index in the tuple and `&item`
Because we get a reference to the element from
the pattern.

Inside the `for` loop, we search for the byte that
byte literal syntax. If we find a space, we return t
length of the string by using `s.len()` :

```
        if item == b' ' {
            return i;
        }
    }
    s.len()
```

We now have a way to find out the index of the
there's a problem. We're returning a `usize` on i
number in the context of the `&String`. In other
from the `String`, there's no guarantee that it w
the program in Listing 4-8 that uses the `first_w`

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello world'

    let word = first_word(&s); // word wi

    s.clear(); // This empties the String,

    // word still has the value 5 here, bu
    // we could meaningfully use the value
invalid!
}
```

Listing 4-8: Storing the result from calling the `fi`
the `String` contents

This program compiles without any errors and v
calling `s.clear()`. Because `word` isn't connecte
contains the value `5`. We could use that value `5`
the first word out, but this would be a bug becau
since we saved `5` in `word`.

Having to worry about the index in `word` getting
tedious and error prone! Managing these indice
`second_word` function. Its signature would have

```
fn second_word(s: &String) -> (usize, usiz
```

Now we're tracking a starting *and* an ending ind
that were calculated from data in a particular sta
We now have three unrelated variables floating

Luckily, Rust has a solution to this problem: strir

String Slices

A *string slice* is a reference to part of a `String`, a

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

This is similar to taking a reference to the whole
bit. Rather than a reference to the entire `String`
`String`. The `start..end` syntax is a range that
but not including, `end`. If we wanted to include

```
let s = String::from("hello world");

let hello = &s[0..=4];
let world = &s[6..=10];
```

The `=` means that we're including the last numb
difference between `..` and `..=`.

We can create slices using a range within bracke
`[starting_index..ending_index]`, where `star`
slice and `ending_index` is one more than the la
slice data structure stores the starting position a
corresponds to `ending_index` minus `starting_`
`let world = &s[6..11];`, `world` would be a sli
byte of `s` and a length value of 5.

Figure 4-6 shows this in a diagram.

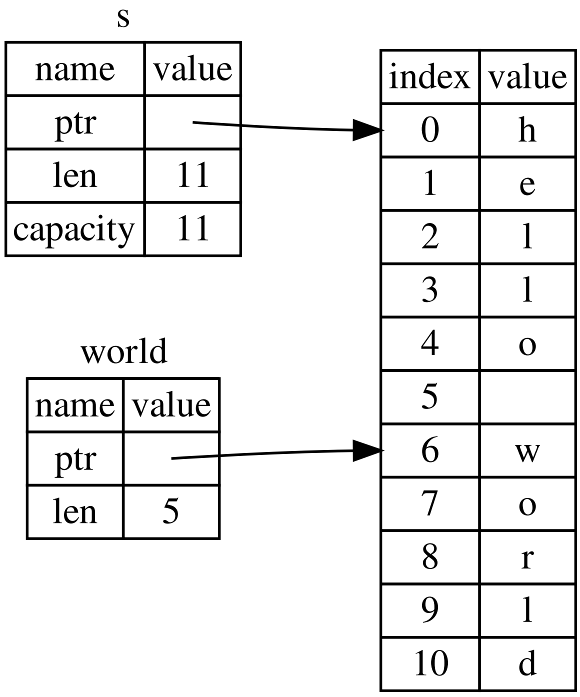

Figure 4-6: String slice referring to part of a `Str`

With Rust's `..` range syntax, if you want to start
the value before the two periods. In other words

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

By the same token, if your slice includes the last
trailing number. That means these are equal:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

You can also drop both values to take a slice of t

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

Note: String slice range indices must occur at
If you attempt to create a string slice in the m
program will exit with an error. For the purpo
are assuming ASCII only in this section; a mor
handling is in the “Strings” section of Chapter

With all this information in mind, let’s rewrite `fi`
that signifies “string slice” is written as `&str`:

Filename: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumera
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

We get the index for the end of the word in the s
looking for the first occurrence of a space. Wher
slice using the start of the string and the index o
indices.

Now when we call `first_word`, we get back a si
underlying data. The value is made up of a refer
and the number of elements in the slice.

Returning a slice would also work for a `second_`

```
fn second_word(s: &String) -> &str {
```

We now have a straightforward API that’s much

compiler will ensure the references into the `Str`
in the program in Listing 4-8, when we got the ir
then cleared the string so our index was invalid?
didn't show any immediate errors. The problem:
trying to use the first word index with an emptie
impossible and let us know we have a problem w
slice version of `first_word` will throw a compile

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello world'

    let word = first_word(&s);

    s.clear(); // Error!
}
```

Here's the compiler error:

```
error[E0502]: cannot borrow `s` as mutable
immutable
 --> src/main.rs:6:5
  |
4 |     let word = first_word(&s);
  |                           - immutable
5 |
6 |     s.clear(); // Error!
  |     ^ mutable borrow occurs here
7 | }
  | - immutable borrow ends here
```

Recall from the borrowing rules that if we have a
we cannot also take a mutable reference. Becau
`String`, it tries to take a mutable reference, wh
API easier to use, but it has also eliminated an e

String Literals Are Slices

Recall that we talked about string literals being s
know about slices, we can properly understand

```
let s = "Hello, world!";
```

The type of `s` here is `&str`: it's a slice pointing t

This is also why string literals are immutable; &s

String Slices as Parameters

Knowing that you can take slices of literals and improvement on `first_word`, and that's its sign

```
fn first_word(s: &String) -> &str {
```

A more experienced Rustacean would write the allows us to use the same function on both `Str

```
fn first_word(s: &str) -> &str {
```

If we have a string slice, we can pass that directl slice of the entire `String`. Defining a function to reference to a `String` makes our API more gen functionality:

Filename: src/main.rs

```
fn main() {
    let my_string = String::from("hello wo

    // first_word works on slices of `Str
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word works on slices of strin
    let word = first_word(&my_string_liter

    // Because string literals *are* strin
    // this works too, without the slice s
    let word = first_word(my_string_litera
}
```

Other Slices

String slices, as you might imagine, are specific t slice type, too. Consider this array:

```
let a = [1, 2, 3, 4, 5];
```

Just as we might want to refer to a part of a strir
an array. We'd do so like this:

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];
```

This slice has the type `&[i32]` . It works the sam
reference to the first element and a length. You'
other collections. We'll discuss these collections
in Chapter 8.

Summary

The concepts of ownership, borrowing, and slice
programs at compile time. The Rust language gi
usage in the same way as other systems progra
owner of data automatically clean up that data v
means you don't have to write and debug extra

Ownership affects how lots of other parts of Rus
concepts further throughout the rest of the boo
look at grouping pieces of data together in a `st`

Using Structs to Struct

A *struct*, or *structure*, is a custom data type that l
multiple related values that make up a meaning
object-oriented language, a *struct* is like an obje
we'll compare and contrast tuples with structs, c
discuss how to define methods and associated f
associated with a struct's data. Structs and enun
building blocks for creating new types in your pr
advantage of Rust's compile time type checking.

Defining and Instantiating Str

Structs are similar to tuples, which were discuss

pieces of a struct can be different types. Unlike w
data so it's clear what the values mean. As a resu
flexible than tuples: you don't have to rely on th
access the values of an instance.

To define a struct, we enter the keyword `struct`
struct's name should describe the significance o
together. Then, inside curly brackets, we define
data, which we call *fields*. For example, Listing 5-
information about a user account:

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

Listing 5-1: A `User` struct definition

To use a struct after we've defined it, we create a
concrete values for each of the fields. We create
the struct and then add curly brackets containin
are the names of the fields and the values are th
fields. We don't have to specify the fields in the s
them in the struct. In other words, the struct def
the type, and instances fill in that template with
type. For example, we can declare a particular u

```
let user1 = User {
    email: String::from("someone@example.c
    username: String::from("someusername12
    active: true,
    sign_in_count: 1,
};
```

Listing 5-2: Creating an instance of the `User` str

To get a specific value from a struct, we can use
user's email address, we could use `user1.email`
value. If the instance is mutable, we can change
assigning into a particular field. Listing 5-3 show
`email` field of a mutable `User` instance:

```
let mut user1 = User {
    email: String::from("someone@example.c
    username: String::from("someusername12
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@e
```

Listing 5-3: Changing the value in the `email` fiel

Note that the entire instance must be mutable; l certain fields as mutable.

As with any expression, we can construct a new expression in the function body to implicitly retu shows a `build_user` function that returns a `Us` username. The `active` field gets the value of `t` value of `1`.

```
fn build_user(email: String, username: Str
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

Listing 5-4: A `build_user` function that takes an `User` instance

It makes sense to name the function parameters fields, but having to repeat the `email` and `user` bit tedious. If the struct had more fields, repeati annoying. Luckily, there's a convenient shorthan

Using the Field Init Shorthand when Va Same Name

Because the parameter names and the struct fie Listing 5-4, we can use the *field init shorthand* sy behaves exactly the same but doesn't have the r shown in Listing 5-5.

```
fn build_user(email: String, username: Str
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Listing 5-5: A `build_user` function that uses fiel
and `username` parameters have the same name

Here, we're creating a new instance of the `User`
`email`. We want to set the `email` field's value t
the `build_user` function. Because the `email` fi
the same name, we only need to write `email` ra

Creating Instances From Other Instanc

It's often useful to create a new instance of a str
values but changes some. You'll do this using *str*

First, Listing 5-6 shows how we create a new `Us`
update syntax. We set new values for `email` an
same values from `user1` that we created in List

```
let user2 = User {
    email: String::from("another@example.c
    username: String::from("anotherusernam
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

Listing 5-6: Creating a new `User` instance using

Using struct update syntax, we can achieve the s
in Listing 5-7. The syntax `..` specifies that the r
should have the same value as the fields in the g

```
let user2 = User {
    email: String::from("another@example.c
    username: String::from("anotherusernam
    ..user1
};
```

Listing 5-7: Using struct update syntax to set nev `User` instance but use the rest of the values fro `user1` variable

The code in Listing 5-7 also creates an instance i `email` and `username` but has the same values f fields from `user1` .

Tuple Structs without Named Fields to

You can also define structs that look similar to tu have the added meaning the struct name provic with their fields; rather, they just have the types when you want to give the whole tuple a name a than other tuples, and naming each field as in a redundant.

To define a tuple struct start with the `struct` ke by the types in the tuple. For example, here are structs named `Color` and `Point` :

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Note that the `black` and `origin` values are diff instances of different tuple structs. Each struct y though the fields within the struct have the sam takes a parameter of type `Color` cannot take a both types are made up of three `i32` values. Ot behave like tuples: you can destructure them int a `.` followed by the index to access an individua

Unit-Like Structs Without Any Fields

You can also define structs that don't have any f
because they behave similarly to `()` , the unit ty
situations in which you need to implement a tra
data that you want to store in the type itself. We

> ### Ownership of Struct Data
>
> In the `User` struct definition in Listing 5-1, we
> rather than the `&str` string slice type. This is
> want instances of this struct to own all of its c
> for as long as the entire struct is valid.
>
> It's possible for structs to store references to
> to do so requires the use of *lifetimes*, a Rust fe
> 10. Lifetimes ensure that the data referenced
> the struct is. Let's say you try to store a refere
> lifetimes, like this, which won't work:
>
> Filename: src/main.rs
>
> ```
> struct User {
>     username: &str,
>     email: &str,
>     sign_in_count: u64,
>     active: bool,
> }
>
> fn main() {
>     let user1 = User {
>         email: "someone@example.com",
>         username: "someusername123",
>         active: true,
>         sign_in_count: 1,
>     };
> }
> ```
>
> The compiler will complain that it needs lifeti

```
error[E0106]: missing lifetime specifie
 -->
  |
2 |     username: &str,
  |               ^ expected lifetime p

error[E0106]: missing lifetime specifie
 -->
  |
3 |     email: &str,
  |            ^ expected lifetime para
```

In Chapter 10, we'll discuss how to fix these e structs, but for now, we'll fix errors like these instead of references like `&str`.

An Example Program Using Str

To understand when we might want to use struc calculates the area of a rectangle. We'll start with the program until we're using structs instead.

Let's make a new binary project with Cargo calle and height of a rectangle specified in pixels and Listing 5-8 shows a short program with one way *src/main.rs*:

Filename: src/main.rs

```rust
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} s
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Listing 5-8: Calculating the area of a rectangle sp variables

Now, run this program using `cargo run`:

```
The area of the rectangle is 1500 square p
```

Even though Listing 5-8 works and figures out th
`area` function with each dimension, we can do l
related to each other because together they des

The issue with this code is evident in the signatu

```
fn area(width: u32, height: u32) -> u32 {
```

The `area` function is supposed to calculate the
function we wrote has two parameters. The para
expressed anywhere in our program. It would b
manageable to group width and height together
might do that in "The Tuple Type" section of Cha

Refactoring with Tuples

Listing 5-9 shows another version of our progra

Filename: src/main.rs

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} s
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Listing 5-9: Specifying the width and height of th

In one way, this program is better. Tuples let us
passing just one argument. But in another way,
name their elements, so our calculation has bec
have to index into the parts of the tuple.

It doesn't matter if we mix up width and height f
to draw the rectangle on the screen, it would ma

that `width` is the tuple index `0` and `height` is t
worked on this code, they would have to figure t
would be easy to forget or mix up these values a
conveyed the meaning of our data in our code.

Refactoring with Structs: Adding More

We use structs to add meaning by labeling the d
using into a data type with a name for the whole
shown in Listing 5-10:

Filename: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, hei

    println!(
        "The area of the rectangle is {} s
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Listing 5-10: Defining a `Rectangle` struct

Here we've defined a struct and named it `Recta`
defined the fields as `width` and `height`, both o
we created a particular instance of `Rectangle` t
50.

Our `area` function is now defined with one para
`rectangle`, whose type is an immutable borrow
mentioned in Chapter 4, we want to borrow the
it. This way, `main` retains its ownership and can
reason we use the `&` in the function signature a

The `area` function accesses the `width` and `hei`
Our function signature for `area` now says exact

of `Rectangle` , using its `width` and `height` fielc
height are related to each other, and it gives des
than using the tuple index values of `0` and `1` . T

Adding Useful Functionality with Deriv

It'd be nice to be able to print an instance of `Rec`
program and see the values for all its fields. Listi
macro as we have used in previous chapters. Th

Filename: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, hei

    println!("rect1 is {}", rect1);
}
```

Listing 5-11: Attempting to print a `Rectangle` in

When we run this code, we get an error with this

```
error[E0277]: the trait bound `Rectangle:
satisfied
```

The `println!` macro can do many kinds of forn
tell `println!` to use formatting known as `Disp`
user consumption. The primitive types we've see
default, because there's only one way you'd wan
type to a user. But with structs, the way `printlr`
clear because there are more display possibilitie
you want to print the curly brackets? Should all t
ambiguity, Rust doesn't try to guess what we wa
implementation of `Display` .

If we continue reading the errors, we'll find this l

```
`Rectangle` cannot be formatted with the d
`:?` instead if you are using a format str
```

Let's try it! The `println!` macro call will now loc `println!("rect1 is {:?}", rect1);`. Putting t brackets tells `println!` we want to use an outp trait that enables us to print our struct in a way t see its value while we're debugging our code.

Run the code with this change. Drat! We still get

```
error[E0277]: the trait bound `Rectangle:
satisfied
```

But again, the compiler gives us a helpful note:

```
`Rectangle` cannot be formatted using `:?`
crate, add `#[derive(Debug)]` or manually
```

Rust *does* include functionality to print out debu explicitly opt in to make that functionality availal the annotation `#[derive(Debug)]` just before th Listing 5-12:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, hei

    println!("rect1 is {:?}", rect1);
}
```

Listing 5-12: Adding the annotation to derive the `Rectangle` instance using debug formatting

Now when we run the program, we won't get an output:

```
rect1 is Rectangle { width: 30, height: 50
```

Nice! It's not the prettiest output, but it shows th instance, which would definitely help during deb it's useful to have output that's a bit easier to rea

instead of `{:?}` in the `println!` string. When w
example, the output will look like this:

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

Rust has provided a number of traits for us to us
can add useful behavior to our custom types. Th
listed in Appendix C, “Derivable Traits.” We’ll cov
custom behavior as well as how to create your c

Our `area` function is very specific: it only compu
be helpful to tie this behavior more closely to ou
work with any other type. Let’s look at how we c
turning the `area` function into an `area` *method*

Method Syntax

Methods are similar to functions: they’re declare
name, they can have parameters and a return va
is run when they’re called from somewhere else
from functions in that they’re defined within the
trait object, which we cover in Chapters 6 and 17
parameter is always `self`, which represents the
being called on.

Defining Methods

Let’s change the `area` function that has a `Recta`
instead make an `area` method defined on the
5-13:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, he

    println!(
        "The area of the rectangle is {} s
        rect1.area()
    );
}
```

Listing 5-13: Defining an `area` method on the `R

To define the function within the context of `Rec (implementation) block. Then we move the `are brackets and change the first (and in this case, o signature and everywhere within the body. In `m function and passed `rect1` as an argument, we the `area` method on our `Rectangle` instance. T instance: we add a dot followed by the method arguments.

In the signature for `area`, we use `&self` instead Rust knows the type of `self` is `Rectangle` due `impl Rectangle` context. Note that we still need we did in `&Rectangle`. Methods can take owner immutably as we've done here, or borrow `self` parameter.

We've chosen `&self` here for the same reason version: we don't want to take ownership, and w struct, not write to it. If we wanted to change the method on as part of what the method does, we parameter. Having a method that takes ownersh as the first parameter is rare; this technique is u transforms `self` into something else and you w the original instance after the transformation.

The main benefit of using methods instead of fu
syntax and not having to repeat the type of `sel`
organization. We've put all the things we can do
`impl` block rather than making future users of c
`Rectangle` in various places in the library we pr

Where's the `->` Operator?

In C and C++, two different operators are use
you're calling a method on the object directly
method on a pointer to the object and need t
other words, if `object` is a pointer, `object->`
`(*object).something()` .

Rust doesn't have an equivalent to the `->` op
called *automatic referencing and dereferencing*
places in Rust that has this behavior.

Here's how it works: when you call a method
automatically adds in `&`, `&mut`, or `*` so `obje`
method. In other words, the following are the

```
p1.distance(&p2);
(&p1).distance(&p2);
```

The first one looks much cleaner. This autom
because methods have a clear receiver—the t
and name of a method, Rust can figure out de
reading (`&self`), mutating (`&mut self`), or co
makes borrowing implicit for method receive
ownership ergonomic in practice.

Methods with More Parameters

Let's practice using methods by implementing a
struct. This time, we want an instance of `Rectan`
`Rectangle` and return `true` if the second `Rect`
otherwise it should return `false` . That is, we wa
shown in Listing 5-14, once we've defined the `ca`

Filename: src/main.rs

```
fn main() {
    let rect1 = Rectangle { width: 30, hei
    let rect2 = Rectangle { width: 10, hei
    let rect3 = Rectangle { width: 60, hei

    println!("Can rect1 hold rect2? {}", r
    println!("Can rect1 hold rect3? {}", r
}
```

Listing 5-14: Using the as-yet-unwritten `can_hol`

And the expected output would look like the foll `rect2` are smaller than the dimensions of `rect`

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

We know we want to define a method, so it will l The method name will be `can_hold`, and it will t `Rectangle` as a parameter. We can tell what the looking at the code that calls the method: `rect1` `&rect2`, which is an immutable borrow to `rect` makes sense because we only need to read `rec` mean we'd need a mutable borrow), and we war so we can use it again after calling the `can_hold` `can_hold` will be a Boolean, and the implement and height of `self` are both greater than the w `Rectangle`, respectively. Let's add the new `can_` Listing 5-13, shown in Listing 5-15:

Filename: src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle)
        self.width > other.width && self.h
    }
}
```

Listing 5-15: Implementing the `can_hold` methc `Rectangle` instance as a parameter

When we run this code with the `main` function i
output. Methods can take multiple parameters t
`self` parameter, and those parameters work ju

Associated Functions

Another useful feature of `impl` blocks is that we
`impl` blocks that *don't* take `self` as a paramete
functions because they're associated with the str
methods, because they don't have an instance o
already used the `String::from` associated func

Associated functions are often used for construc
the struct. For example, we could provide an ass
dimension parameter and use that as both widt
create a square `Rectangle` rather than having t

Filename: src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: s
    }
}
```

To call this associated function, we use the `::` s
`let sq = Rectangle::square(3);` is an exampl
struct: the `::` syntax is used for both associated
by modules. We'll discuss modules in Chapter 7.

Multiple `impl` Blocks

Each struct is allowed to have multiple `impl` blc
equivalent to the code shown in Listing 5-16, wh
block:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle)
        self.width > other.width && self.h
    }
}
```

Listing 5-16: Rewriting Listing 5-15 using multipl

There's no reason to separate these methods in is valid syntax. We'll see a case in which multiple where we discuss generic types and traits.

Summary

Structs let you create custom types that are mea structs, you can keep associated pieces of data each piece to make your code clear. Methods let instances of your structs have, and associated fu functionality that is particular to your struct with

But structs aren't the only way you can create cu feature to add another tool to your toolbox.

Enums and Pattern Ma

In this chapter we'll look at *enumerations*, also re to define a type by enumerating its possible valu enum to show how an enum can encode meani a particularly useful enum, called `Option`, which something or nothing. Then we'll look at how pa expression makes it easy to run different code f we'll cover how the `if let` construct is another available to you to handle enums in your code.

Enums are a feature in many languages, but the Rust's enums are most similar to *algebraic data t*

F#, OCaml, and Haskell.

Defining an Enum

Let's look at a situation we might want to expres
useful and more appropriate than structs in this
addresses. Currently, two major standards are u
and version six. These are the only possibilities f
will come across: we can *enumerate* all possible
gets its name.

Any IP address can be either a version four or a
the same time. That property of IP addresses ma
appropriate, because enum values can only be c
and version six addresses are still fundamentally
treated as the same type when the code is hand
of IP address.

We can express this concept in code by defining
listing the possible kinds an IP address can be,
variants of the enum:

```
enum IpAddrKind {
    V4,
    V6,
}
```

`IpAddrKind` is now a custom data type that we

Enum Values

We can create instances of each of the two varia

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

Note that the variants of the enum are namespa
double colon to separate the two. The reason th
`IpAddrKind::V4` and `IpAddrKind::V6` are of th
then, for instance, define a function that takes a

```
fn route(ip_type: IpAddrKind) { }
```

And we can call this function with either variant:

```
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

Using enums has even more advantages. Thinki
the moment we don't have a way to store the ac
what *kind* it is. Given that you just learned about
this problem as shown in Listing 6-1:

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

Listing 6-1: Storing the data and `IpAddrKind` va

Here, we've defined a struct `IpAddr` that has tw
`IpAddrKind` (the enum we defined previously) a
We have two instances of this struct. The first, `h`
as its `kind` with associated address data of `127`
`loopback`, has the other variant of `IpAddrKind`
address `::1` associated with it. We've used a str
values together, so now the variant is associatec

We can represent the same concept in a more c
than an enum inside a struct, by putting data dir
new definition of the `IpAddr` enum says that bc

associated `String` values:

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.

let loopback = IpAddr::V6(String::from("::
```

We attach data to each variant of the enum dire
struct.

There's another advantage to using an enum rat
have different types and amounts of associated
will always have four numeric components that
we wanted to store `V4` addresses as four `u8` va
as one `String` value, we wouldn't be able to wit
with ease:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::
```

We've shown several different ways to define da
and version six IP addresses. However, as it turr
and encode which kind they are is so common t
definition we can use! Let's look at how the stan
the exact enum and variants that we've defined
data inside the variants in the form of two differ
differently for each variant:

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

This code illustrates that you can put any kind o numeric types, or structs, for example. You can standard library types are often not much more come up with.

Note that even though the standard library cont still create and use our own definition without c the standard library's definition into our scope. into scope in Chapter 7.

Let's look at another example of an enum in List types embedded in its variants:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Listing 6-2: A `Message` enum whose variants ea of values

This enum has four variants with different types

- `Quit` has no data associated with it at all.
- `Move` includes an anonymous struct inside
- `Write` includes a single `String`.
- `ChangeColor` includes three `i32` values.

Defining an enum with variants like the ones in different kinds of struct definitions, except the e and all the variants are grouped together under structs could hold the same data that the preced

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple stru
struct ChangeColorMessage(i32, i32, i32);
```

But if we used the different structs, which each h
easily define a function to take any of these kind
`Message` enum defined in Listing 6-2, which is a

There is one more similarity between enums and
methods on structs using `impl`, we're also able
a method named `call` that we could define on

```
impl Message {
    fn call(&self) {
        // method body would be defined he
    }
}

let m = Message::Write(String::from("hello
m.call();
```

The body of the method would use `self` to get
on. In this example, we've created a variable `m` t
`Message::Write(String::from("hello"))`, and
of the `call` method when `m.call()` runs.

Let's look at another enum in the standard librar
`Option`.

The `Option` Enum and Its Advantages

In the previous section, we looked at how the `I`
system to encode more information than just th
explores a case study of `Option`, which is anoth
library. The `Option` type is used in many places
scenario in which a value could be something or
concept in terms of the type system means the

handled all the cases you should be handling; th
are extremely common in other programming la

Programming language design is often thought
include, but the features you exclude are import
feature that many other languages have. *Null* is
there. In languages with null, variables can alwa
null.

In his 2009 presentation "Null References: The B
inventor of null, has this to say:

> I call it my billion-dollar mistake. At that time,
> comprehensive type system for references in
> goal was to ensure that all use of references s
> checking performed automatically by the com
> temptation to put in a null reference, simply b
> implement. This has led to innumerable error
> crashes, which have probably caused a billion
> last forty years.

The problem with null values is that if you try to
you'll get an error of some kind. Because this nu
extremely easy to make this kind of error.

However, the concept that null is trying to expre
that is currently invalid or absent for some reaso

The problem isn't really with the concept but wit
such, Rust does not have nulls, but it does have
of a value being present or absent. This enum is
standard library as follows:

```
enum Option<T> {
    Some(T),
    None,
}
```

The `Option<T>` enum is so useful that it's even i
need to bring it into scope explicitly. In addition,
and `None` directly without the `Option::` prefix.
regular enum, and `Some(T)` and `None` are still v

The `<T>` syntax is a feature of Rust we haven't t
parameter, and we'll cover generics in more det
need to know is that `<T>` means the `Some` vari
piece of data of any type. Here are some examp
number types and string types:

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

If we use `None` rather than `Some`, we need to te
have, because the compiler can't infer the type t
looking only at a `None` value.

When we have a `Some` value, we know that a val
within the `Some`. When we have a `None` value, i
thing as null: we don't have a valid value. So why
having null?

In short, because `Option<T>` and `T` (where `T` c
the compiler won't let us use an `Option<T>` valu
For example, this code won't compile because it
`Option<i8>`:

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

If we run this code, we get an error message like

```
error[E0277]: the trait bound `i8: std::op
is
not satisfied
 -->
  |
5 |     let sum = x + y;
  |                 ^ no implementation fo
  |
```

Intense! In effect, this error message means tha
an `i8` and an `Option<i8>`, because they're diff
a type like `i8` in Rust, the compiler will ensure t
can proceed confidently without having to check

when we have an `Option<i8>` (or whatever type
have to worry about possibly not having a value
handle that case before using the value.

In other words, you have to convert an `Option<`
operations with it. Generally, this helps catch on
null: assuming that something isn't null when it

Not having to worry about incorrectly assuming
confident in your code. In order to have a value
explicitly opt in by making the type of that value
value, you are required to explicitly handle the c
Everywhere that a value has a type that isn't an
that the value isn't null. This was a deliberate de
pervasiveness and increase the safety of Rust co

So, how do you get the `T` value out of a `Some` v
`Option<T>` so you can use that value? The `Opti`
methods that are useful in a variety of situations
documentation. Becoming familiar with the met
useful in your journey with Rust.

In general, in order to use an `Option<T>` value,
handle each variant. You want some code that w
value, and this code is allowed to use the inner
you have a `None` value, and that code doesn't ha
expression is a control flow construct that does
run different code depending on which variant c
use the data inside the matching value.

The `match` Control Flow Opera

Rust has an extremely powerful control flow ope
compare a value against a series of patterns and
pattern matches. Patterns can be made up of lit
and many other things; Chapter 18 covers all the
they do. The power of `match` comes from the ex
fact that the compiler confirms that all possible

Think of a `match` expression as being like a coin
track with variously sized holes along it, and eac
encounters that it fits into. In the same way, valu

`match`, and at the first pattern the value "fits," th
block to be used during execution.

Because we just mentioned coins, let's use them
write a function that can take an unknown Unite
the counting machine, determine which coin it is
shown here in Listing 6-3:

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Listing 6-3: An enum and a `match` expression th
patterns

Let's break down the `match` in the `value_in_ce`
keyword followed by an expression, which in thi
very similar to an expression used with `if`, but
expression needs to return a Boolean value, but
`coin` in this example is the `Coin` enum that we

Next are the `match` arms. An arm has two parts
arm here has a pattern that is the value `Coin::I`
separates the pattern and the code to run. The c
Each arm is separated from the next with a com

When the `match` expression executes, it compa
pattern of each arm, in order. If a pattern match
that pattern is executed. If that pattern doesn't r
to the next arm, much as in a coin-sorting mach
need: in Listing 6-3, our `match` has four arms.

The code associated with each arm is an expres
expression in the matching arm is the value that

expression.

Curly brackets typically aren't used if the match
where each arm just returns a value. If you want
match arm, you can use curly brackets. For exar
"Lucky penny!" every time the method was callec
return the last value of the block, `1`:

```
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Patterns that Bind to Values

Another useful feature of match arms is that the
that match the pattern. This is how we can extra

As an example, let's change one of our enum va
1999 through 2008, the United States minted qu
of the 50 states on one side. No other coins got
this extra value. We can add this information to
variant to include a `UsState` value stored inside
6-4:

```
#[derive(Debug)] // So we can inspect the
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Listing 6-4: A `Coin` enum in which the `Quarter`

Let's imagine that a friend of ours is trying to col sort our loose change by coin type, we'll also cal with each quarter so if it's one our friend doesn' collection.

In the match expression for this code, we add a that matches values of the variant `Coin::Quart` the `state` variable will bind to the value of that `state` in the code for that arm, like so:

```
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {
            25
        },
    }
}
```

If we were to call `value_in_cents(Coin::Quarte` be `Coin::Quarter(UsState::Alaska)` . When we match arms, none of them match until we reach point, the binding for `state` will be the value `U` that binding in the `println!` expression, thus g `Coin` enum variant for `Quarter` .

Matching with `Option<T>`

In the previous section, we wanted to get the inn when using `Option<T>` ; we can also handle `Opt` the `Coin` enum! Instead of comparing coins, we , but the way that the `match` expression works r

Let's say we want to write a function that takes a inside, adds 1 to that value. If there isn't a value `None` value and not attempt to perform any ope

This function is very easy to write, thanks to `mat`

```
fn plus_one(x: Option<i32>) -> Option<i32>
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Listing 6-5: A function that uses a `match` expres

Let's examine the first execution of `plus_one` in `plus_one(five)` , the variable `x` in the body of `Some(5)` . We then compare that against each m

```
None => None,
```

The `Some(5)` value doesn't match the pattern `N`

```
Some(i) => Some(i + 1),
```

Does `Some(5)` match `Some(i)` ? Why yes it does binds to the value contained in `Some` , so `i` take arm is then executed, so we add 1 to the value c with our total `6` inside.

Now let's consider the second call of `plus_one` i enter the `match` and compare to the first arm.

```
None => None,
```

It matches! There's no value to add to, so the pr value on the right side of `=>` . Because the first a compared.

Combining `match` and enums is useful in many in Rust code: `match` against an enum, bind a va execute code based on it. It's a bit tricky at first, you had it in all languages. It's consistently a use

Matches Are Exhaustive

There's one other aspect of `match` we need to d
`plus_one` function that has a bug and won't cor

```
fn plus_one(x: Option<i32>) -> Option<i32>
    match x {
        Some(i) => Some(i + 1),
    }
}
```

We didn't handle the `None` case, so this code wi
knows how to catch. If we try to compile this coc

```
error[E0004]: non-exhaustive patterns: `Nc
 -->
  |
6 |         match x {
  |               ^ pattern `None` not cov
```

Rust knows that we didn't cover every possible c
we forgot! Matches in Rust are *exhaustive*: we mu
order for the code to be valid. Especially in the c
prevents us from forgetting to explicitly handle t
assuming that we have a value when we might h
dollar mistake discussed earlier.

The `_` Placeholder

Rust also has a pattern we can use when we dor
example, a `u8` can have valid values of 0 throug
values 1, 3, 5, and 7, we don't want to have to lis
255. Fortunately, we don't have to: we can use th

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

The `_` pattern will match any value. By putting i
match all the possible cases that aren't specified
value, so nothing will happen in the `_` case. As a

nothing for all the possible values that we don't

However, the `match` expression can be a bit wo
care about *one* of the cases. For this situation, R

Concise Control Flow with `if`

The `if let` syntax lets you combine `if` and `l
values that match one pattern while ignoring th
6-6 that matches on an `Option<u8>` value but o
is 3:

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

Listing 6-6: A `match` that only cares about execu

We want to do something with the `Some(3)` ma
`Some<u8>` value or the `None` value. To satisfy th
`_ => ()` after processing just one variant, whicl

Instead, we could write this in a shorter way usi
behaves the same as the `match` in Listing 6-6:

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

The syntax `if let` takes a pattern and an expre
the same way as a `match`, where the expressior
pattern is its first arm.

Using `if let` means you have less typing, less i
However, you lose the exhaustive checking that
`match` and `if let` depends on what you're doi
whether gaining conciseness is an appropriate t
checking.

In other words, you can think of `if let` as synt when the value matches one pattern and then ig

We can include an `else` with an `if let`. The bl
is the same as the block of code that would go w
expression that is equivalent to the `if let` and
definition in Listing 6-4, where the `Quarter` vari
wanted to count all non-quarter coins we see wh
quarters, we could do that with a `match` expres

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("Stat
    _ => count += 1,
}
```

Or we could use an `if let` and `else` expressio

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", s
} else {
    count += 1;
}
```

If you have a situation in which your program ha
using a `match`, remember that `if let` is in you

Summary

We've now covered how to use enums to create
of enumerated values. We've shown how the sta
you use the type system to prevent errors. When
you can use `match` or `if let` to extract and us
many cases you need to handle.

Your Rust programs can now express concepts i
enums. Creating custom types to use in your AP
will make certain your functions get only values

In order to provide a well-organized API to your
and only exposes exactly what your users will ne

Using Modules to Reus
Code

When you start writing programs in Rust, your c
function. As your code grows, you'll eventually m
for reuse and better organization. By splitting yc
make each chunk easier to understand on its ov
many functions? Rust has a module system that
organized fashion.

In the same way that you extract lines of code ir
functions (and other code, like structs and enum
a namespace that contains definitions of functic
whether those definitions are visible outside the
Here's an overview of how modules work:

- The `mod` keyword declares a new module.
  either immediately following this declaratic
  file.
- By default, functions, types, constants, and
  keyword makes an item public and therefc
- The `use` keyword brings modules, or the c
  so it's easier to refer to them.

We'll look at each of these parts to see how they

`mod` and the Filesystem

We'll start our module example by making a nev
creating a binary crate, we'll make a library crate
into their projects as a dependency. For exampl
Chapter 2 is a library crate that we used as a de
project.

We'll create a skeleton of a library that provides
functionality; we'll concentrate on the organizati
we won't worry about what code goes in the fun
`communicator`. To create a library, pass the `--l`

```
$ cargo new communicator --lib
$ cd communicator
```

Notice that Cargo generated *src/lib.rs* instead of
the following:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Cargo creates an example test to help us get ou
and `mod tests` syntax in the "Using `super` to A
in this chapter, but for now, leave this code at th

Because we don't have a *src/main.rs* file, there's
`cargo run` command. Therefore, we'll use the
our library crate's code.

We'll look at different options for organizing you
a variety of situations, depending on the intent

Module Definitions

For our `communicator` networking library, we'll
that contains the definition of a function called
Rust starts with the `mod` keyword. Add this code
above the test code:

Filename: src/lib.rs

```
mod network {
    fn connect() {
    }
}
```

After the `mod` keyword, we put the name of the
of code in curly brackets. Everything inside this

`network`. In this case, we have a single function,
function from code outside the `network` modul
module and use the namespace syntax `::` like

We can also have multiple modules, side by side
example, to also have a `client` module that ha
add it as shown in Listing 7-1:

Filename: src/lib.rs

```
mod network {
    fn connect() {
    }
}

mod client {
    fn connect() {
    }
}
```

Listing 7-1: The `network` module and the `clien`
src/lib.rs

Now we have a `network::connect` function and
can have completely different functionality, and
with each other because they're in different mod

In this case, because we're building a library, the
building our library is *src/lib.rs*. However, in resp
nothing special about *src/lib.rs*. We could also cr
binary crate in the same way as we're creating n
crate. In fact, we can put modules inside of mod
modules grow to keep related functionality orga
functionality apart. The way you choose to orgar
think about the relationship between the parts c
code and its `connect` function might make mor
were inside the `network` namespace instead, as

Filename: src/lib.rs

```rust
mod network {
    fn connect() {
    }

    mod client {
        fn connect() {
        }
    }
}
```

Listing 7-2: Moving the `client` module inside th

In your *src/lib.rs* file, replace the existing `mod ne` with the ones in Listing 7-2, which have the `cli` `network`. The functions `network::connect` and named `connect`, but they don't conflict with eac namespaces.

In this way, modules form a hierarchy. The cont level, and the submodules are at lower levels. H example in Listing 7-1 looks like when thought o

```
communicator
 ├── network
 └── client
```

And here's the hierarchy corresponding to the e

```
communicator
 └── network
     └── client
```

The hierarchy shows that in Listing 7-2, `client` rather than a sibling. More complicated projects need to be organized logically in order for you t means in your project is up to you and depends think about your project's domain. Use the techn side modules and nested modules in whatever s

Moving Modules to Other Files

Modules form a hierarchical structure, much like you're used to: filesystems! We can use Rust's m to split up Rust projects so not everything lives i example, let's start with the code in Listing 7-3:

Filename: src/lib.rs

```
mod client {
    fn connect() {
    }
}

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

Listing 7-3: Three modules, `client`, `network`, a
src/lib.rs

The file *src/lib.rs* has this module hierarchy:

```
communicator
 ├── client
 └── network
     └── server
```

If these modules had many functions, and those
would be difficult to scroll through this file to fin
Because the functions are nested inside one or
inside the functions will start getting lengthy as
separate the `client`, `network`, and `server` mo
into their own files.

First, let's replace the `client` module code with
module so that *src/lib.rs* looks like code shown in

Filename: src/lib.rs

```
mod client;

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

Listing 7-4: Extracting the contents of the `clien`
in *src/lib.rs*

We're still *declaring* the `client` module here, bu
semicolon, we're telling Rust to look in another l
the scope of the `client` module. In other word

```
mod client {
    // contents of client.rs
}
```

Now we need to create the external file with tha
in your *src/* directory and open it. Then enter the
function in the `client` module that we remove

Filename: src/client.rs

```
fn connect() {
}
```

Note that we don't need a `mod` declaration in th
the `client` module with `mod` in *src/lib.rs*. This fi
`client` module. If we put a `mod client` here, w
own submodule named `client`!

Rust only knows to look in *src/lib.rs* by default. If
project, we need to tell Rust in *src/lib.rs* to look i
needs to be defined in *src/lib.rs* and can't be defi

Now the project should compile successfully, alt
Remember to use `cargo build` instead of `carg`
crate rather than a binary crate:

```
$ cargo build
   Compiling communicator v0.1.0 (file:///
warning: function is never used: `connect`
 --> src/client.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
 --> src/lib.rs:4:5
  |
4 | /     fn connect() {
5 | |     }
  | |_____^

warning: function is never used: `connect`
 --> src/lib.rs:8:9
  |
8 | /         fn connect() {
9 | |         }
  | |_________^
```

These warnings tell us that we have functions th these warnings for now; we'll address them later Visibility with `pub`" section. The good news is th built successfully!

Next, let's extract the `network` module into its o *src/lib.rs*, delete the body of the `network` modul declaration, like so:

Filename: src/lib.rs

```
mod client;

mod network;
```

Then create a new *src/network.rs* file and enter t

Filename: src/network.rs

```
fn connect() {
}

mod server {
    fn connect() {
    }
}
```

Notice that we still have a `mod` declaration withi
still want `server` to be a submodule of `networ`

Run `cargo build` again. Success! We have one
Because it's a submodule—that is, a module wit
extracting a module into a file named after that
so you can see the error. First, change *src/netwo*
the `server` module's contents:

Filename: src/network.rs

```
fn connect() {
}

mod server;
```

Then create a *src/server.rs* file and enter the con
extracted:

Filename: src/server.rs

```
fn connect() {
}
```

When we try to `cargo build`, we'll get the error

```
$ cargo build
   Compiling communicator v0.1.0 (file:///
error: cannot declare a new module at this
 --> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
  |
note: maybe move this module `src/network.
`src/network/mod.rs`
 --> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
note: ... or maybe `use` the module `serve
redeclaring it
 --> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
```

Listing 7-5: Error when trying to extract the `serv`

The error says we `cannot declare a new modul` to the `mod server;` line in *src/network.rs*. So *src/* somehow: keep reading to understand why.

The note in the middle of Listing 7-5 is actually v something we haven't yet talked about doing:

```
note: maybe move this module `network` to
`network/mod.rs`
```

Instead of continuing to follow the same file-nar can do what the note suggests:

1. Make a new *directory* named *network*, the p
2. Move the *src/network.rs* file into the new *ne* *src/network/mod.rs*.
3. Move the submodule file *src/server.rs* into t

Here are commands to carry out these steps:

```
$ mkdir src/network
$ mv src/network.rs src/network/mod.rs
$ mv src/server.rs src/network
```

Now when we try to run `cargo build`, compilat

though). Our module layout still looks exactly th
code in *src/lib.rs* in Listing 7-3:

```
communicator
 ├── client
 └── network
     └── server
```

The corresponding file layout now looks like this

```
└── src
    ├── client.rs
    ├── lib.rs
    └── network
        ├── mod.rs
        └── server.rs
```

So when we wanted to extract the `network::se`
change the *src/network.rs* file to the *src/network/*
`network::server` in the *network* directory in *src*
just extract the `network::server` module into *s*
wouldn't be able to recognize that `server` was
`network` if the *server.rs* file was in the *src* directo
let's consider a different example with the follow
definitions are in *src/lib.rs*:

```
communicator
 ├── client
 └── network
     └── client
```

In this example, we have three modules again:
`network::client`. Following the same steps we
into files, we would create *src/client.rs* for the `cl`
module, we would create *src/network.rs*. But we
`network::client` module into a *src/client.rs* file
top-level `client` module! If we could put the co
`network::client` modules in the *src/client.rs* file
whether the code was for `client` or for `networ`

Therefore, in order to extract a file for the `netw`
`network` module, we needed to create a directo
a *src/network.rs* file. The code that is in the `netw`
src/network/mod.rs file, and the submodule `netw`
src/network/client.rs file. Now the top-level *src/cli*

belongs to the `client` module.

Rules of Module Filesystems

Let's summarize the rules of modules with regar

- If a module named `foo` has no submodule `foo` in a file named *foo.rs*.
- If a module named `foo` does have submo for `foo` in a file named *foo/mod.rs*.

These rules apply recursively, so if a module nar `bar` and `bar` does not have submodules, you s *src* directory:

```
└── foo
    ├── bar.rs (contains the declarations
    └── mod.rs (contains the declarations
```

The modules should be declared in their parent

Next, we'll talk about the `pub` keyword and get

Controlling Visibility with `pub`

We resolved the error messages shown in Listin `network::server` code into the *src/network/moc* respectively. At that point, `cargo build` was ab warning messages about the `client::connect` , `network::server::connect` functions not being

So why are we receiving these warnings? After a functions that are intended to be used by our *us* own project, so it shouldn't matter that these `co` point of creating them is that they will be used b

To understand why this program invokes these `communicator` library from another project, call create a binary crate in the same directory as ou file containing this code:

Filename: src/main.rs

```
extern crate communicator;

fn main() {
    communicator::client::connect();
}
```

We use the `extern crate` command to bring th
scope. Our package now contains *two* crates. Ca
of a binary crate, which is separate from the exi
src/lib.rs. This pattern is quite common for execu
a library crate, and the binary crate uses that lib
can also use the library crate, and it's a nice sep

From the point of view of a crate outside the `co`
modules we've been creating are within a modu
crate, `communicator`. We call the top-level modu

Also note that even if we're using an external cr
the `extern crate` should go in our root module
in our submodules, we can refer to items from e
level modules.

Right now, our binary crate just calls our library'
module. However, invoking `cargo build` will no

```
error[E0603]: module `client` is private
 --> src/main.rs:4:5
  |
4 |     communicator::client::connect();
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Ah ha! This error tells us that the `client` modul
warnings. It's also the first time we've run into th
context of Rust. The default state of all code in R
to use the code. If you don't use a private functi
program is the only code allowed to use that fur
function has gone unused.

After you specify that a function such as `client`
call to that function from your binary crate be al
function is unused will go away. Marking a funct
function will be used by code outside of your pr
external usage that's now possible as the functi
function is marked public, Rust will not require t
will stop warning that the function is unused.

Making a Function Public

To tell Rust to make a function public, we add th declaration. We'll focus on fixing the warning tha gone unused for now, as well as the `module `cl` binary crate. Modify *src/lib.rs* to make the `clien`

Filename: src/lib.rs

```
pub mod client;

mod network;
```

The `pub` keyword is placed right before `mod`. Le

```
error[E0603]: function `connect` is privat
 --> src/main.rs:4:5
  |
4 |     communicator::client::connect();
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Hooray! We have a different error! Yes, different celebration. The new error shows `function `cc` *src/client.rs* to make `client::connect` public too

Filename: src/client.rs

```
pub fn connect() {
}
```

Now run `cargo build` again:

```
warning: function is never used: `connect`
 --> src/network/mod.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
 --> src/network/server.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
```

The code compiled, and the warning that `clien

Unused code warnings don't always indicate tha made public: if you *didn't* want these functions t code warnings could be alerting you to code you delete. They could also be alerting you to a bug all places within your library where this function

But in this case, we *do* want the other two functi API, so let's mark them as `pub` as well to get rid *src/network/mod.rs* to look like the following:

Filename: src/network/mod.rs

```
pub fn connect() {
}

mod server;
```

Then compile the code:

```
warning: function is never used: `connect`
 --> src/network/mod.rs:1:1
  |
1 | / pub fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
 --> src/network/server.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
```

Hmmm, we're still getting an unused function w `network::connect` is set to `pub`. The reason is t module, but the `network` module that the funct working from the interior of the library out this t we worked from the outside in. We need to char too, like so:

Filename: src/lib.rs

```
pub mod client;

pub mod network;
```

Now when we compile, that warning is gone:

```
warning: function is never used: `connect`
 --> src/network/server.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_^
  |
  = note: #[warn(dead_code)] on by default
```

Only one warning is left—try to fix this one on yo

Privacy Rules

Overall, these are the rules for item visibility:

- If an item is public, it can be accessed throu
- If an item is private, it can be accessed only any of the parent's child modules.

Privacy Examples

Let's look at a few more privacy examples to get project and enter the code in Listing 7-6 into you

Filename: src/lib.rs

```
mod outermost {
    pub fn middle_function() {}

    fn middle_secret_function() {}

    mod inside {
        pub fn inner_function() {}

        fn secret_function() {}
    }
}

fn try_me() {
    outermost::middle_function();
    outermost::middle_secret_function();
    outermost::inside::inner_function();
    outermost::inside::secret_function();
}
```

Listing 7-6: Examples of private and public funct

Before you try to compile this code, make a gue: function will have errors. Then, try compiling the right—and read on for the discussion of the errc

Looking at the Errors

The `try_me` function is in the root module of ou `outermost` is private, but the second privacy rul allowed to access the `outermost` module becau module, as is `try_me` .

The call to `outermost::middle_function` will wc public and `try_me` is accessing `middle_functio` `outermost` . We determined in the previous para

The call to `outermost::middle_secret_functio` Because `middle_secret_function` is private, th module is neither the current module of `middle` nor is it a child module of the current module of

The module named `inside` is private and has n only by its current module `outermost` . That mea allowed to call `outermost::inside::inner_func` `outermost::inside::secret_function` .

Fixing the Errors

Here are some suggestions for changing the coc
Make a guess as to whether it will fix the errors
compile the code to see whether or not you're ri
understand why. Feel free to design more exper

- What if the `inside` module were public?
- What if `outermost` were public and `insid`
- What if, in the body of `inner_function`, yc
  `::outermost::middle_secret_function()`
  mean that we want to refer to the modules

Next, let's talk about bringing items into scope w

Referring to Names in Differe

We've covered how to call functions defined witl
as part of the call, as in the call to the `nested_mc`
7-7:

Filename: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

fn main() {
    a::series::of::nested_modules();
}
```

Listing 7-7: Calling a function by fully specifying i

As you can see, referring to the fully qualified na
Rust has a keyword to make these calls more co

Bringing Names into Scope with the `u`

Rust's `use` keyword shortens lengthy function c

function you want to call into scope. Here's an e
a::series::of module into a binary crate's roc

Filename: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of;

fn main() {
    of::nested_modules();
}
```

The line `use a::series::of;` means that rathe
path wherever we want to refer to the `of` modu

The `use` keyword brings only what we've specifi
children of modules into scope. That's why we st
when we want to call the `nested_modules` funct

We could have chosen to bring the function into
function in the `use` as follows:

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of::nested_modules;

fn main() {
    nested_modules();
}
```

Doing so allows us to exclude all the modules ar

Because enums also form a sort of namespace l
variants into scope with `use` as well. For any kir
multiple items from one namespace into scope,
and commas in the last position, like so:

```rust
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::{Red, Yellow};

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = TrafficLight::Green;
}
```

We're still specifying the `TrafficLight` namesp
didn't include `Green` in the `use` statement.

Nested groups in `use` declarations

If you have a complex module tree with many di
import a few items from each one, it might be u
same declaration to keep your code clean and a
name.

The `use` declaration supports nesting to help yo
imports and glob ones. For example this snippet
`baz` and `Bar`:

```rust
use foo::{
    bar::{self, Foo},
    baz::{*, quux::Bar},
};
```

Bringing All Names into Scope with a G

To bring all the items in a namespace into scope
which is called the *glob operator*. This example b
scope without having to list each specifically:

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::*;

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = Green;
}
```

The `*` will bring into scope all the visible items i
should use globs sparingly: they are convenient,
items than you expected and cause naming con

Using `super` to Access a Parent Modul

As you saw at the beginning of this chapter, whe
makes a `tests` module for you. Let's go into mo
`communicator` project, open *src/lib.rs*:

Filename: src/lib.rs

```
pub mod client;

pub mod network;

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Chapter 11 explains more about testing, but par
now: we have a module named `tests` that lives
contains one function named `it_works`. Even th
the `tests` module is just another module! So ou

```
communicator
 ├── client
 ├── network
 |   └── client
 └── tests
```

Tests are for exercising the code within our libra `client::connect` function from this `it_works` checking any functionality right now. This won't

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        client::connect();
    }
}
```

Run the tests by invoking the `cargo test` comn

```
$ cargo test
   Compiling communicator v0.1.0 (file:///
error[E0433]: failed to resolve. Use of un
 --> src/lib.rs:9:9
  |
9 |         client::connect();
  |         ^^^^^^ Use of undeclared type
```

The compilation failed, but why? We don't need the function, as we did in *src/main.rs*, because w `communicator` library crate here. The reason is t current module, which here is `tests`. The only e where paths are relative to the crate root by def `client` module in its scope!

So how do we get back up one module in the mo `client::connect` function in the `tests` modul either use leading colons to let Rust know that w the whole path, like this:

```
::client::connect();
```

Or, we can use `super` to move up one module i module, like this:

```
super::client::connect();
```

These two options don't look that different in th
module hierarchy, starting from the root every t
those cases, using `super` to get from the currer
shortcut. Plus, if you've specified the path from t
and then rearrange your modules by moving a s
needing to update the path in several places, wh

It would also be annoying to have to type `super`
seen the tool for that solution: `use`! The `super:`
give to `use` so it is relative to the parent module

For these reasons, in the `tests` module especia
the best solution. So now our test looks like this

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::client;

    #[test]
    fn it_works() {
        client::connect();
    }
}
```

When we run `cargo test` again, the test will pa
output will be the following:

```
$ cargo test
   Compiling communicator v0.1.0 (file:///
     Running target/debug/communicator-92

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

Summary

Now you know some new techniques for organi
to group related functionality together, keep file

present a tidy public API to your library users.

Next, we'll look at some collection data structure
use in your nice, neat code.

Common Collections

Rust's standard library includes a number of ver
collections. Most other data types represent one
contain multiple values. Unlike the built-in array
collections point to is stored on the heap, which
need to be known at compile time and can grow
kind of collection has different capabilities and c
one for your current situation is a skill you'll dev
discuss three collections that are used very ofte

- A *vector* allows you to store a variable num
- A *string* is a collection of characters. We've previously, but in this chapter we'll talk abo
- A *hash map* allows you to associate a value implementation of the more general data s

To learn about the other kinds of collections pro
documentation.

We'll discuss how to create and update vectors,
what makes each special.

Storing Lists of Values with Ve

The first collection type we'll look at is `Vec<T>`, a
you to store more than one value in a single dat
next to each other in memory. Vectors can only
are useful when you have a list of items, such as
of items in a shopping cart.

Creating a New Vector

To create a new, empty vector, we can call the `V
8-1:

```
let v: Vec<i32> = Vec::new();
```

Listing 8-1: Creating a new, empty vector to hold

Note that we added a type annotation here. Bec into this vector, Rust doesn't know what kind of an important point. Vectors are implemented us generics with your own types in Chapter 10. For provided by the standard library can hold any ty specific type, the type is specified within angle b that the `Vec<T>` in `v` will hold elements of the

In more realistic code, Rust can often infer the ty you insert values, so you rarely need to do this t create a `Vec<T>` that has initial values, and Rust convenience. The macro will create a new vecto Listing 8-2 creates a new `Vec<i32>` that holds th

```
let v = vec![1, 2, 3];
```

Listing 8-2: Creating a new vector containing valu

Because we've given initial `i32` values, Rust can and the type annotation isn't necessary. Next, w

Updating a Vector

To create a vector and then add elements to it, v shown in Listing 8-3:

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

Listing 8-3: Using the `push` method to add value

As with any variable, if we want to be able to cha mutable using the `mut` keyword, as discussed ir

inside are all of type `i32`, and Rust infers this fr
`Vec<i32>` annotation.

Dropping a Vector Drops Its Elements

Like any other `struct`, a vector is freed when it
Listing 8-4:

```
{
    let v = vec![1, 2, 3, 4];

    // do stuff with v

} // <- v goes out of scope and is freed h
```

Listing 8-4: Showing where the vector and its ele

When the vector gets dropped, all of its content
integers it holds will be cleaned up. This may see
can get a bit more complicated when you start t
elements of the vector. Let's tackle that next!

Reading Elements of Vectors

Now that you know how to create, update, and d
their contents is a good next step. There are two
vector. In the examples, we've annotated the typ
from these functions for extra clarity.

Listing 8-5 shows the method of accessing a valu

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
```

Listing 8-5: Using indexing syntax to access an it

Listing 8-6 shows the method of accessing a valu

```
let v = vec![1, 2, 3, 4, 5];
let v_index = 2;

match v.get(v_index) {
    Some(_) => { println!("Reachable eleme
    None => { println!("Unreachable elemer
}
```

Listing 8-6: Using the `get` method to access an

Note two details here. First, we use the index va vectors are indexed by number, starting at zero. element are by using `&` and `[]`, which gives us method with the index passed as an argument,

Rust has two ways to reference an element so y behaves when you try to use an index value that for. As an example, let's see what a program will elements and then tries to access an element at

```
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

Listing 8-7: Attempting to access the element at elements

When we run this code, the first `[]` method will it references a nonexistent element. This metho program to crash if there's an attempt to access vector.

When the `get` method is passed an index that i without panicking. You would use this method if range of the vector happens occasionally under then have logic to handle having either `Some(&e` Chapter 6. For example, the index could be com If they accidentally enter a number that's too lar value, you could tell the user how many items a another chance to enter a valid value. That woul the program due to a typo!

When the program has a valid reference, the bo

and borrowing rules (covered in Chapter 4) to en
references to the contents of the vector remain
can't have mutable and immutable references in
Listing 8-8, where we hold an immutable referen
try to add an element to the end, which won't wo

```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);
```

Listing 8-8: Attempting to add an element to a ve
item

Compiling this code will result in this error:

```
error[E0502]: cannot borrow `v` as mutable
immutable
 -->
  |
4 |     let first = &v[0];
  |                  - immutable borrow oc
5 |
6 |     v.push(6);
  |     ^ mutable borrow occurs here
7 |
8 | }
  | - immutable borrow ends here
```

The code in Listing 8-8 might look like it should w
first element care about what changes at the en
the way vectors work: adding a new element on
allocating new memory and copying the old elem
enough room to put all the elements next to eac
In that case, the reference to the first element w
memory. The borrowing rules prevent programs

> Note: For more on the implementation details
> Rustonomicon" at https://doc.rust-lang.org/st

Iterating over the Values in a Vector

If we want to access each element in a vector in

elements rather than use indexes to access one
use a `for` loop to get immutable references to
values and print them:

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

Listing 8-9: Printing each element in a vector by
`for` loop

We can also iterate over mutable references to
order to make changes to all the elements. The
to each element:

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Listing 8-10: Iterating over mutable references t

To change the value that the mutable reference
dereference operator (`*`) to get to the value in
. We'll talk more about `*` in Chapter 15.

Using an Enum to Store Multiple Types

At the beginning of this chapter, we said that ve
the same type. This can be inconvenient; there
store a list of items of different types. Fortunate
defined under the same enum type, so when we
type in a vector, we can define and use an enum

For example, say we want to get values from a r
the columns in the row contain integers, some f
strings. We can define an enum whose variants
then all the enum variants will be considered the
we can create a vector that holds that enum and
We've demonstrated this in Listing 8-11:

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("bl
    SpreadsheetCell::Float(10.12),
];
```

Listing 8-11: Defining an `enum` to store values of

Rust needs to know what types will be in the vec exactly how much memory on the heap will be r secondary advantage is that we can be explicit a vector. If Rust allowed a vector to hold any type, more of the types would cause errors with the o of the vector. Using an enum plus a `match` expr compile time that every possible case is handlec

When you're writing a program, if you don't kno program will get at runtime to store in a vector, Instead, you can use a trait object, which we'll cc

Now that we've discussed some of the most con review the API documentation for all the many u the standard library. For example, in addition to returns the last element. Let's move on to the ne

Storing UTF-8 Encoded Text wi

We talked about strings in Chapter 4, but we'll lc Rustaceans commonly get stuck on strings due t Rust's propensity for exposing possible errors, s structure than many programmers give them cr combine in a way that can seem difficult when y programming languages.

It's useful to discuss strings in the context of coll implemented as a collection of bytes, plus some functionality when those bytes are interpreted a

the operations on `String` that every collection t
and reading. We'll also discuss the ways in which
collections, namely how indexing into a `String`
between how people and computers interpret `s

What Is a String?

We'll first define what we mean by the term *strin*
core language, which is the string slice `str` that
`&str`. In Chapter 4, we talked about *string slices*,
encoded string data stored elsewhere. String lite
binary output of the program and are therefore

The `String` type, which is provided by Rust's sta
the core language, is a growable, mutable, owne
Rustaceans refer to "strings" in Rust, they usually
slice `&str` types, not just one of those types. Alt
`String`, both types are used heavily in Rust's sta
string slices are UTF-8 encoded.

Rust's standard library also includes a number o
`OsString`, `OsStr`, `CString`, and `CStr`. Library
for storing string data. See how those names all
owned and borrowed variants, just like the `Stri
previously. These string types can store text in d
in memory in a different way, for example. We w
in this chapter; see their API documentation for
when each is appropriate.

Creating a New String

Many of the same operations available with `Vec
well, starting with the `new` function to create a s

```
let mut s = String::new();
```

Listing 8-11: Creating a new, empty `String`

This line creates a new empty string called `s`, w
Often, we'll have some initial data that we want t

use the `to_string` method, which is available o
`Display` trait, as string literals do. Listing 8-12 s

```
let data = "initial contents";

let s = data.to_string();

// the method also works on a literal dire
let s = "initial contents".to_string();
```

Listing 8-12: Using the `to_string` method to cre

This code creates a string containing `initial c`

We can also use the function `String::from` to c
The code in Listing 8-13 is equivalent to the code
`to_string`:

```
let s = String::from("initial contents");
```

Listing 8-13: Using the `String::from` function t

Because strings are used for so many things, we
for strings, providing us with a lot of options. So
they all have their place! In this case, `String::f`
thing, so which you choose is a matter of style.

Remember that strings are UTF-8 encoded, so w
data in them, as shown in Listing 8-14:

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שָׁלוֹם");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

Listing 8-14: Storing greetings in different langua

All of these are valid `String` values.

Updating a String

A `String` can grow in size and its contents can c
`Vec<T>` , if you push more data into it. In additio
operator or the `format!` macro to concatenate

Appending to a String with `push_str` and `pus`

We can grow a `String` by using the `push_str` r
shown in Listing 8-15:

```
let mut s = String::from("foo");
s.push_str("bar");
```

Listing 8-15: Appending a string slice to a `Strin`

After these two lines, `s` will contain `foobar` . Th
slice because we don't necessarily want to take c
example, the code in Listing 8-16 shows that it w
able to use `s2` after appending its contents to `s`

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {}", s2);
```

Listing 8-16: Using a string slice after appending

If the `push_str` method took ownership of `s2` ,
on the last line. However, this code works as we'

The `push` method takes a single character as a p
Listing 8-17 shows code that adds the letter l to

```
let mut s = String::from("lo");
s.push('l');
```

Listing 8-17: Adding one character to a `String` v

As a result of this code, `s` will contain `lol` .

Concatenation with the `+` Operator or the `f`

Often, you'll want to combine two existing string
as shown in Listing 8-18:

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // Note s1 has been mo
used
```

Listing 8-18: Using the `+` operator to combine t
value

The string `s3` will contain `Hello, world!` as a
no longer valid after the addition and the reaso
with the signature of the method that gets calle
operator uses the `add` method, whose signatur

```
fn add(self, s: &str) -> String {
```

This isn't the exact signature that's in the standa
`add` is defined using generics. Here, we're looki
concrete types substituted for the generic ones,
this method with `String` values. We'll discuss g
gives us the clues we need to understand the tri

First, `s2` has an `&`, meaning that we're adding
first string because of the `s` parameter in the `a`
to a `String`; we can't add two `String` values to
`&String`, not `&str`, as specified in the second
Listing 8-18 compile?

The reason we're able to use `&s2` in the call to
the `&String` argument into a `&str`. When we c
coercion, which here turns `&s2` into `&s2[..]`. W
depth in Chapter 15. Because `add` does not tak
will still be a valid `String` after this operation.

Second, we can see in the signature that `add` ta
`self` does *not* have an `&`. This means `s1` in Lis
call and no longer be valid after that. So althoug
will copy both strings and create a new one, this
of `s1`, appends a copy of the contents of `s2`, a
result. In other words, it looks like it's making a
implementation is more efficient than copying.

If we need to concatenate multiple strings, the b
unwieldy:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

At this point, `s` will be `tic-tac-toe`. With all of
to see what's going on. For more complicated st
`format!` macro:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}-{}-{}", s1, s2, s3);
```

This code also sets `s` to `tic-tac-toe`. The `for`
`println!`, but instead of printing the output to
the contents. The version of the code using `for`
doesn't take ownership of any of its parameters

Indexing into Strings

In many other programming languages, accessi
referencing them by index is a valid and commo
access parts of a `String` using indexing syntax
the invalid code in Listing 8-19:

```
let s1 = String::from("hello");
let h = s1[0];
```

Listing 8-19: Attempting to use indexing syntax w

This code will result in the following error:

```
error[E0277]: the trait bound `std::string
std::ops::Index<{integer}>` is not satisf
 -->
  |
3 |     let h = s1[0];
  |             ^^^^^ the type `std::strin
`{integer}`
  |
  = help: the trait `std::ops::Index<{inte
`std::string::String`
```

The error and the note tell the story: Rust string
To answer that question, we need to discuss hov

Internal Representation

A `String` is a wrapper over a `Vec<u8>`. Let's loc
UTF-8 example strings from Listing 8-14. First, th

```
let len = String::from("Hola").len();
```

In this case, `len` will be 4, which means the vect
long. Each of these letters takes 1 byte when enc
following line? (Note that this line begins with th
Arabic number 3.)

```
let len = String::from("Здравствуйте").len
```

Asked how long the string is, you might say 12. H
the number of bytes it takes to encode "Здравст
Unicode scalar value in that string takes 2 bytes
the string's bytes will not always correlate to a va
demonstrate, consider this invalid Rust code:

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

What should the value of `answer` be? Should it b
in UTF-8, the first byte of `З` is `208` and the seco
be `208`, but `208` is not a valid character on its o
a user would want if they asked for the first lette
only data that Rust has at byte index 0. Users ge
returned, even if the string contains only Latin le

that returned the byte value, it would return `10
unexpected value and causing bugs that might r
doesn't compile this code at all and prevents mi
development process.

Bytes and Scalar Values and Grapheme Clust

Another point about UTF-8 is that there are actu
strings from Rust's perspective: as bytes, scalar
closest thing to what we would call *letters*).

If we look at the Hindi word "नमस्ते" written in the
vector of `u8` values that looks like this:

```
[224, 164, 168, 224, 164, 174, 224, 164, 1
164,
224, 165, 135]
```

That's 18 bytes and is how computers ultimately
Unicode scalar values, which are what Rust's `ch

```
['न', 'म', 'स', '्', 'त', 'े']
```

There are six `char` values here, but the fourth a
diacritics that don't make sense on their own. Fi
clusters, we'd get what a person would call the f
word:

```
["न", "म", "स्", "ते"]
```

Rust provides different ways of interpreting the
so that each program can choose the interpreta
language the data is in.

A final reason Rust doesn't allow us to index into
indexing operations are expected to always take
possible to guarantee that performance with a
walk through the contents from the beginning to
valid characters there were.

Slicing Strings

Indexing into a string is often a bad idea becaus

the string-indexing operation should be: a byte v
cluster, or a string slice. Therefore, Rust asks you
to use indices to create string slices. To be more
that you want a string slice, rather than indexing
can use `[]` with a range to create a string slice c

```
let hello = "Здравствуйте";

let s = &hello[0..4];
```

Here, `s` will be a `&str` that contains the first 4 b
mentioned that each of these characters was 2 b

What would happen if we used `&hello[0..1]`? T
runtime in the same way as if an invalid index w

```
thread 'main' panicked at 'byte index 1 is
inside 'З' (bytes 0..2) of `Здравствуйте`'
```

You should use ranges to create string slices wit
your program.

Methods for Iterating Over Strings

Fortunately, you can access elements in a string

If you need to perform operations on individual
to do so is to use the `chars` method. Calling `ch
returns six values of type `char`, and you can ite
each element:

```
for c in "नमस्ते".chars() {
    println!("{}", c);
}
```

This code will print the following:

```
न
म
स
्
त
े
```

The `bytes` method returns each raw byte, whicl
domain:

```
for b in "नमस्ते".bytes() {
    println!("{}", b);
}
```

This code will print the 18 bytes that make up th

```
224
164
// --snip--
165
135
```

But be sure to remember that valid Unicode sca
than 1 byte.

Getting grapheme clusters from strings is compl
provided by the standard library. Crates are ava
functionality you need.

Strings Are Not So Simple

To summarize, strings are complicated. Differen
different choices about how to present this com
chosen to make the correct handling of `String`
programs, which means programmers have to p
data upfront. This trade-off exposes more of the
apparent in other programming languages, but
errors involving non-ASCII characters later in you

Let's switch to something a bit less complex: has

Storing Keys with Associated V

The last of our common collections is the *hash m* mapping of keys of type `K` to values of type `V`. which determines how it places these keys and v programming languages support this kind of da different name, such as hash, map, object, hash just to name a few.

Hash maps are useful when you want to look up can with vectors, but by using a key that can be you could keep track of each team's score in a h name and the values are each team's score. Give score.

We'll go over the basic API of hash maps in this s hiding in the functions defined on `HashMap<K, V` check the standard library documentation for m

Creating a New Hash Map

You can create an empty hash map with `new` an Listing 8-20, we're keeping track of the scores of and Yellow. The Blue team starts with 10 points,

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

Listing 8-20: Creating a new hash map and inser

Note that we need to first `use` the `HashMap` fro standard library. Of our three common collectio it's not included in the features brought into sco maps also have less support from the standard construct them, for example.

Just like vectors, hash maps store their data on t type `String` and values of type `i32`. Like vecto of the keys must have the same type, and all of t

Another way of constructing a hash map is by us
of tuples, where each tuple consists of a key and
gathers data into a number of collection types, i
had the team names and initial scores in two se
method to create a vector of tuples where “Blue
we could use the `collect` method to turn that v
shown in Listing 8-21:

```
use std::collections::HashMap;

let teams  = vec![String::from("Blue"), St
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> =
teams.iter().zip(initial_scores.iter()).co
```

Listing 8-21: Creating a hash map from a list of t

The type annotation `HashMap<_, _>` is needed h
into many different data structures and Rust do
specify. For the parameters for the key and valu
underscores, and Rust can infer the types that th
types of the data in the vectors.

Hash Maps and Ownership

For types that implement the `Copy` trait, like `i3`
map. For owned values like `String`, the values w
be the owner of those values, as demonstrated

```
use std::collections::HashMap;

let field_name = String::from("Favorite co
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid
and
// see what compiler error you get!
```

Listing 8-22: Showing that keys and values are o
inserted

We aren't able to use the variables `field_name`
moved into the hash map with the call to `inser`

If we insert references to values into the hash m
the hash map. The values that the references pc
long as the hash map is valid. We'll talk more ab
References with Lifetimes" section in Chapter 1C

Accessing Values in a Hash Map

We can get a value out of the hash map by provi
shown in Listing 8-23:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

Listing 8-23: Accessing the score for the Blue tea

Here, `score` will have the value that's associate
will be `Some(&10)`. The result is wrapped in `Som`
`Option<&V>`; if there's no value for that key in th
The program will need to handle the `Option` in
Chapter 6.

We can iterate over each key/value pair in a hasl
with vectors, using a `for` loop:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

This code will print each pair in an arbitrary orde

```
Yellow: 50
Blue: 10
```

Updating a Hash Map

Although the number of keys and values is grow value associated with it at a time. When you war you have to decide how to handle the case wher You could replace the old value with the new val value. You could keep the old value and ignore t value if the key *doesn't* already have a value. Or y the new value. Let's look at how to do each of th

Overwriting a Value

If we insert a key and a value into a hash map ar different value, the value associated with that ke code in Listing 8-24 calls `insert` twice, the hash pair because we're inserting the value for the Blu

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

println!("{:?}", scores);
```

Listing 8-24: Replacing a value stored with a part

This code will print `{"Blue": 25}`. The original v

Only Inserting a Value If the Key Has No Valu

It's common to check whether a particular key h value for it. Hash maps have a special API for thi want to check as a parameter. The return value called `Entry` that represents a value that might to check whether the key for the Yellow team ha

doesn't, we want to insert the value 50, and the
entry API, the code looks like Listing 8-25:

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_ir
scores.entry(String::from("Blue")).or_inse

println!("{:?}", scores);
```

Listing 8-25: Using the `entry` method to only in:
value

The `or_insert` method on `Entry` is defined to
value for the corresponding `Entry` key if that ke
parameter as the new value for this key and retu
value. This technique is much cleaner than writi
plays more nicely with the borrow checker.

Running the code in Listing 8-25 will print `{"Yel`
to `entry` will insert the key for the Yellow team
team doesn't have a value already. The second c
map because the Blue team already has the valu

Updating a Value Based on the Old Value

Another common use case for hash maps is to l
it based on the old value. For instance, Listing 8-
times each word appears in some text. We use a
and increment the value to keep track of how m
the first time we've seen a word, we'll first insert

```
use std::collections::HashMap;

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(
    *count += 1;
}

println!("{:?}", map);
```

Listing 8-26: Counting occurrences of words usir
counts

This code will print `{"world": 2, "hello": 1,` method actually returns a mutable reference (`&` we store that mutable reference in the `count` v value, we must first dereference `count` using th goes out of scope at the end of the `for` loop, sc allowed by the borrowing rules.

Hashing Functions

By default, `HashMap` uses a cryptographically se provide resistance to Denial of Service (DoS) atta algorithm available, but the trade-off for better s performance is worth it. If you profile your code function is too slow for your purposes, you can s specifying a different *hasher*. A hasher is a type t trait. We'll talk about traits and how to implemer necessarily have to implement your own hasher shared by other Rust users that provide hashers hashing algorithms.

Summary

Vectors, strings, and hash maps will provide a la in programs when you need to store, access, an exercises you should now be equipped to solve:

- Given a list of integers, use a vector and re
median (when sorted, the value in the mid
that occurs most often; a hash map will be
- Convert strings to pig latin. The first conso
end of the word and “ay” is added, so “first
with a vowel have “hay” added to the end i
Keep in mind the details about UTF-8 enco
- Using a hash map and vectors, create a tex
employee names to a department in a com
Engineering” or “Add Amir to Sales.” Then l
in a department or all people in the compa
alphabetically.

The standard library API documentation describ
hash maps have that will be helpful for these ex

We’re getting into more complex programs in wl
perfect time to discuss error handling. We’ll do t

Error Handling

Rust’s commitment to reliability extends to erro
software, so Rust has a number of features for h
something goes wrong. In many cases, Rust requ
possibility of an error and take some action befo
requirement makes your program more robust
and handle them appropriately before you’ve de

Rust groups errors into two major categories: *re*
For a recoverable error, such as a file not found
problem to the user and retry the operation. Un
symptoms of bugs, like trying to access a locatio

Most languages don’t distinguish between these
in the same way, using mechanisms such as exc
Instead, it has the type `Result<T, E>` for recove
that stops execution when the program encoun
chapter covers calling `panic!` first and then talk
values. Additionally, we’ll explore considerations
recover from an error or to stop execution.

Unrecoverable Errors with `pa

Sometimes, bad things happen in your code, an
In these cases, Rust has the `panic!` macro. Wh
program will print a failure message, unwind an
This most commonly occurs when a bug of som
clear to the programmer how to handle the erro

Unwinding the Stack or Aborting in

By default, when a panic occurs, the program
Rust walks back up the stack and cleans up th
encounters. But this walking back and cleanu
to immediately *abort*, which ends the progra
that the program was using will then need to
system. If in your project you need to make th
possible, you can switch from unwinding to a
`panic = 'abort'` to the appropriate `[profi`
For example, if you want to abort on panic in

```
[profile.release]
panic = 'abort'
```

Let's try calling `panic!` in a simple program:

Filename: src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

When you run the program, you'll see somethin

```
$ cargo run
   Compiling panic v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn'
note: Run with `RUST_BACKTRACE=1` for a ba
```

The call to `panic!` causes the error message co
line shows our panic message and the place in c

occurred: *src/main.rs:2:4* indicates that it's the se *src/main.rs* file.

In this case, the line indicated is part of our code `panic!` macro call. In other cases, the `panic!` c calls, and the filename and line number reporte someone else's code where the `panic!` macro i eventually led to the `panic!` call. We can use th `panic!` call came from to figure out the part of We'll discuss what a backtrace is in more detail r

Using a `panic!` Backtrace

Let's look at another example to see what it's lik library because of a bug in our code instead of f directly. Listing 9-1 has some code that attempts vector:

Filename: src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

Listing 9-1: Attempting to access an element bey cause a `panic!`

Here, we're attempting to access the hundredth index 99 because indexing starts at zero), but it situation, Rust will panic. Using `[]` is supposed an invalid index, there's no element that Rust co

Other languages, like C, will attempt to give you situation, even though it isn't what you want: yo memory that would correspond to that element memory doesn't belong to the vector. This is call security vulnerabilities if an attacker is able to m to read data they shouldn't be allowed to that is

To protect your program from this sort of vulne at an index that doesn't exist, Rust will stop exec it and see:

```
$ cargo run
   Compiling panic v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bo
is
99', /checkout/src/liballoc/vec.rs:1555:10
note: Run with `RUST_BACKTRACE=1` for a ba
```

This error points at a file we didn't write, *vec.rs*. T
in the standard library. The code that gets run w
vec.rs, and that is where the `panic!` is actually h

The next note line tells us that we can set the `RU
to get a backtrace of exactly what happened to c
all the functions that have been called to get to t
they do in other languages: the key to reading th
and read until you see files you wrote. That's the
The lines above the lines mentioning your files a
lines below are code that called your code. Thes
standard library code, or crates that you're using
setting the `RUST_BACKTRACE` environment variab
shows output similar to what you'll see:

```
$ RUST_BACKTRACE=1 cargo run
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bo
is 99', /checkout/src/liballoc/vec.rs:1555
stack backtrace:
   0: std::sys::imp::backtrace::tracing::i
             at /checkout/src/libstd/sys/u
/gcc_s.rs:49
   1: std::sys_common::backtrace::_print
             at /checkout/src/libstd/sys_c
   2: std::panicking::default_hook::{{clos
             at /checkout/src/libstd/sys_c
             at /checkout/src/libstd/panic
   3: std::panicking::default_hook
             at /checkout/src/libstd/panic
   4: std::panicking::rust_panic_with_hook
             at /checkout/src/libstd/panic
   5: std::panicking::begin_panic
             at /checkout/src/libstd/panic
   6: std::panicking::begin_panic_fmt
             at /checkout/src/libstd/panic
   7: rust_begin_unwind
             at /checkout/src/libstd/panic
   8: core::panicking::panic_fmt
             at /checkout/src/libcore/pani
   9: core::panicking::panic_bounds_check
             at /checkout/src/libcore/pani
  10: <alloc::vec::Vec<T> as core::ops::in
             at /checkout/src/liballoc/vec
  11: panic::main
             at src/main.rs:4
  12: __rust_maybe_catch_panic
             at /checkout/src/libpanic_unw
  13: std::rt::lang_start
             at /checkout/src/libstd/panic
             at /checkout/src/libstd/panic
             at /checkout/src/libstd/rt.rs
  14: main
  15: __libc_start_main
  16: <unknown>
```

Listing 9-2: The backtrace generated by a call to environment variable `RUST_BACKTRACE` is set

That’s a lot of output! The exact output you see operating system and Rust version. In order to g debug symbols must be enabled. Debug symbol `cargo build` or `cargo run` without the `--rele

In the output in Listing 9-2, line 11 of the backtra that's causing the problem: line 4 of *src/main.rs*. panic, the location pointed to by the first line me should start investigating. In Listing 9-1, where w panic in order to demonstrate how to use backt request an element at index 99 from a vector th code panics in the future, you'll need to figure ou what values to cause the panic and what the coc

We'll come back to `panic!` and when we should handle error conditions in the "To `panic!` or No chapter. Next, we'll look at how to recover from

Recoverable Errors with `Resul

Most errors aren't serious enough to require the Sometimes, when a function fails, it's for a reaso respond to. For example, if you try to open a file file doesn't exist, you might want to create the fi

Recall from "Handling Potential Failure with the `Result` enum is defined as having two variants,

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The `T` and `E` are generic type parameters: we'll Chapter 10. What you need to know right now is value that will be returned in a success case with the type of the error that will be returned in a fa Because `Result` has these generic type parame the functions that the standard library has defin where the successful value and error value we w

Let's call a function that returns a `Result` value Listing 9-3 we try to open a file:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
}
```

Listing 9-3: Opening a file

How do we know `File::open` returns a `Result`
library API documentation, or we could ask the c
annotation that we know is *not* the return type c
the code, the compiler will tell us that the types
then tell us what the type of `f` *is*. Let's try it! We
`File::open` isn't of type `u32`, so let's change th

```
let f: u32 = File::open("hello.txt");
```

Attempting to compile now gives us the followin

```
error[E0308]: mismatched types
 --> src/main.rs:4:18
  |
4 |     let f: u32 = File::open("hello.txt
  |                  ^^^^^^^^^^^^^^^^^^^^^
`std::result::Result`
  |
  = note: expected type `u32`
             found type `std::result::Resu
std::io::Error>`
```

This tells us the return type of the `File::open` f
generic parameter `T` has been filled in here wit
`std::fs::File`, which is a file handle. The type
`std::io::Error`.

This return type means the call to `File::open` r
that we can read from or write to. The function c
file might not exist, or we might not have permis
`File::open` function needs to have a way to tel
and at the same time give us either the file hanc
information is exactly what the `Result` enum co

In the case where `File::open` succeeds, the val
instance of `Ok` that contains a file handle. In the
be an instance of `Err` that contains more inforr
happened.

We need to add to the code in Listing 9-3 to take
value `File::open` returns. Listing 9-4 shows on
basic tool, the `match` expression that we discus

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem op
        },
    };
}
```

Listing 9-4: Using a `match` expression to handle
returned

Note that, like the `Option` enum, the `Result` en
imported in the prelude, so we don't need to spe
`Err` variants in the `match` arms.

Here we tell Rust that when the result is `Ok`, ret
`Ok` variant, and we then assign that file handle v
`match`, we can use the file handle for reading or

The other arm of the `match` handles the case w
`File::open`. In this example, we've chosen to ca
named *hello.txt* in our current directory and we r
output from the `panic!` macro:

```
thread 'main' panicked at 'There was a pro
repr:
Os { code: 2, message: "No such file or di
```

As usual, this output tells us exactly what has go

Matching on Different Errors

The code in Listing 9-4 will `panic!` no matter wh
to do instead is take different actions for differe

failed because the file doesn't exist, we want to c
to the new file. If `File::open` failed for any othe
didn't have permission to open the file—we still
way as it did in Listing 9-4. Look at Listing 9-5, wl

Filename: src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match F
                Ok(fc) => fc,
                Err(e) => panic!("Tried to
problem: {:?}", e),
            },
            other_error => panic!("There w
{:?}", other_error),
        },
    };
}
```

Listing 9-5: Handling different kinds of errors in

The type of the value that `File::open` returns i
which is a struct provided by the standard librar
we can call to get an `io::ErrorKind` value. The
the standard library and has variants representi
might result from an `io` operation. The variant
`ErrorKind::NotFound`, which indicates the file w
So, we `match` on `f`, but we also then have an ir

The condition we want to check in the match gu
`error.kind()` is the `NotFound` variant of the `E`
create the file with `File::create`. However, bec
we need to add another inner `match` statement
opened, a different error message will be printe
stays the same so the program panics on any er

That's a lot of `match`! `match` is very powerful, bu
Chapter 13, we'll learn about closures. The `Resu`
that accept a closure, and are implemented as `m`

Rustacean might write this:

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt").map_er
        if error.kind() == ErrorKind::NotF
            File::create("hello.txt").unwr
                panic!("Tried to create fi
{:?}", error);
            })
        } else {
            panic!("There was a problem op
        }
    });
}
```

Come back to this example after you've read Ch `map_err` and `unwrap_or_else` methods do in th There's many more of these methods that can c dealing with errors. We'll be looking at some oth

Shortcuts for Panic on Error: `unwrap` a

Using `match` works well enough, but it can be a communicate intent well. The `Result<T, E>` typ on it to do various tasks. One of those methods, that is implemented just like the `match` stateme `Result` value is the `Ok` variant, `unwrap` will retu `Result` is the `Err` variant, `unwrap` will call the example of `unwrap` in action:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap
}
```

If we run this code without a *hello.txt* file, we'll se call that the `unwrap` method makes:

```
thread 'main' panicked at 'called `Result:
Error {
repr: Os { code: 2, message: "No such file
src/libcore/result.rs:906:4
```

Another method, `expect`, which is similar to `un` error message. Using `expect` instead of `unwrap` can convey your intent and make tracking down syntax of `expect` looks like this:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect
}
```

We use `expect` in the same way as `unwrap`: to `panic!` macro. The error message used by `exp` parameter that we pass to `expect`, rather than `unwrap` uses. Here's what it looks like:

```
thread 'main' panicked at 'Failed to open
code:
2, message: "No such file or directory" }
```

Because this error message starts with the text `Failed to open hello.txt`, it will be easier to f message is coming from. If we use `unwrap` in m figure out exactly which `unwrap` is causing the p panic print the same message.

Propagating Errors

When you're writing a function whose implemer instead of handling the error within this functior calling code so that it can decide what to do. Thi and gives more control to the calling code, wher logic that dictates how the error should be hand the context of your code.

For example, Listing 9-6 shows a function that re doesn't exist or can't be read, this function will r called this function:

Filename: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<Str
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

Listing 9-6: A function that returns errors to the

This function can be written in a much shorter w
lot of it manually in order to explore error handl
way. Let's look at the return type of the function
This means the function is returning a value of t
generic parameter `T` has been filled in with the
generic type `E` has been filled in with the concre
succeeds without any problems, the code that c
value that holds a `String` —the username that t
function encounters any problems, the code tha
`Err` value that holds an instance of `io::Error`
what the problems were. We chose `io::Error`
because that happens to be the type of the erro
operations we're calling in this function's body th
function and the `read_to_string` method.

The body of the function starts by calling the `Fi`
the `Result` value returned with a `match` similar
instead of calling `panic!` in the `Err` case, we re
pass the error value from `File::open` back to th
value. If `File::open` succeeds, we store the file

continue.

Then we create a new `String` in variable `s` anc the file handle in `f` to read the contents of the f method also returns a `Result` because it might succeeded. So we need another `match` to handl succeeds, then our function has succeeded, and that's now in `s` wrapped in an `Ok`. If `read_to_s` in the same way that we returned the error valu return value of `File::open`. However, we don't because this is the last expression in the functio

The code that calls this code will then handle ge a username or an `Err` value that contains an `i` calling code will do with those values. If the callir call `panic!` and crash the program, use a defau from somewhere other than a file, for example. what the calling code is actually trying to do, so v information upward for it to handle appropriate

This pattern of propagating errors is so commor question mark operator `?` to make this easier.

A Shortcut for Propagating Errors: the `?` Ope

Listing 9-7 shows an implementation of `read_us` functionality as it had in Listing 9-6, but this imp operator:

Filename: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<Str
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

Listing 9-7: A function that returns errors to the

The `?` placed after a `Result` value is defined to

match expressions we defined to handle the Re
of the Result is an Ok , the value inside the Ok
expression, and the program will continue. If the
returned from the whole function as if we had u
value gets propagated to the calling code.

There is a difference between what the match e
error values taken by ? go through the from fu
the standard library, which is used to convert er
When ? calls the from function, the error type
type defined in the return type of the current fu
returns one error type to represent all the ways
might fail for many different reasons. As long as
from function to define how to convert itself to
of the conversion automatically.

In the context of Listing 9-7, the ? at the end of
value inside an Ok to the variable f . If an error
whole function and give any Err value to the ca
the ? at the end of the read_to_string call.

The ? operator eliminates a lot of boilerplate ar
implementation simpler. We could even shorter
calls immediately after the ? , as shown in Listin

Filename: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<Str
    let mut s = String::new();

    File::open("hello.txt")?.read_to_strin

    Ok(s)
}
```

Listing 9-8: Chaining method calls after ?

We've moved the creation of the new String in
that part hasn't changed. Instead of creating a va
read_to_string directly onto the result of File
a ? at the end of the read_to_string call, and

the username in `s` when both `File::open` and
returning errors. The functionality is again the s
this is just a different, more ergonomic way to w

Speaking of different ways to write this function
shorter:

Filename: src/main.rs

```
use std::io;
use std::io::Read;
use std::fs;

fn read_username_from_file() -> Result<Str
    fs::read_to_string("hello.txt")
}
```

Listing 9-9: Using `fs::read_to_string`

Reading a file into a string is a fairly common op
convenience function called `fs::read_to_strin`
`String`, read the contents of the file, and put th
then return it. Of course, this doesn't give us the
error handling, so we did it the hard way at first.

The `?` Operator Can Only Be Used in Functio

The `?` operator can only be used in functions th
because it is defined to work in the same way as
Listing 9-6. The part of the `match` that requires
`return Err(e)`, so the return type of the functi
compatible with this `return`.

Let's look at what happens if we use `?` in the `ma`
return type of `()`:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

When we compile this code, we get the followin

```
error[E0277]: the trait bound `(): std::op
 --> src/main.rs:4:13
  |
4 |     let f = File::open("hello.txt")?;
  |             ------------------------
  |             |
  |             the `?` operator can only
returns
  `Result` (or another type that implement
  |             in this macro invocation
  |
  = help: the trait `std::ops::Try` is not
  = note: required by `std::ops::Try::from
```

This error points out that we're only allowed to u
`Result`. In functions that don't return `Result`, w
return `Result`, you'll need to use a `match` or or
the `Result` instead of using `?` to potentially pr

Now that we've discussed the details of calling p
return to the topic of how to decide which is app

To `panic!` or Not to `panic!`

So how do you decide when you should call `pan
`Result`? When code panics, there's no way to re
error situation, whether there's a possible way t
making the decision on behalf of the code callin
unrecoverable. When you choose to return a `Re
options rather than making the decision for it. T
attempt to recover in a way that's appropriate fo
an `Err` value in this case is unrecoverable, so it
recoverable error into an unrecoverable one. Th
default choice when you're defining a function th

In rare situations, it's more appropriate to write
a `Result`. Let's explore why it's appropriate to p
and tests. Then we'll discuss situations in which
impossible, but you as a human can. The chapte
guidelines on how to decide whether to panic in

Examples, Prototype Code, and Tests

When you're writing an example to illustrate sor
handling code in the example as well can make t
it's understood that a call to a method like `unwr`
placeholder for the way you'd want your applica
based on what the rest of your code is doing.

Similarly, the `unwrap` and `expect` methods are
you're ready to decide how to handle errors. The
for when you're ready to make your program m

If a method call fails in a test, you'd want the wh
isn't the functionality under test. Because `panic`
calling `unwrap` or `expect` is exactly what should

Cases in Which You Have More Informa

It would also be appropriate to call `unwrap` whe
ensures the `Result` will have an `Ok` value, but t
understands. You'll still have a `Result` value tha
operation you're calling still has the possibility o
logically impossible in your particular situation. I
inspecting the code that you'll never have an `Er`
call `unwrap`. Here's an example:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1".parse().unw
```

We're creating an `IpAddr` instance by parsing a
`127.0.0.1` is a valid IP address, so it's acceptabl
having a hardcoded, valid string doesn't change
we still get a `Result` value, and the compiler wil
the `Err` variant is a possibility because the com
this string is always a valid IP address. If the IP a
than being hardcoded into the program and the
we'd definitely want to handle the `Result` in a r

Guidelines for Error Handling

It's advisable to have your code panic when it's p
in a bad state. In this context, a *bad state* is when

contract, or invariant has been broken, such as v
values, or missing values are passed to your cod

- The bad state is not something that's *expec*
- Your code after this point needs to rely on
- There's not a good way to encode this infor

If someone calls your code and passes in values
choice might be to call `panic!` and alert the per
their code so they can fix it during development
appropriate if you're calling external code that is
invalid state that you have no way of fixing.

However, when failure is expected, it is more ap
make a `panic!` call. Examples include a parser b
HTTP request returning a status that indicates y
returning a `Result` indicates that failure is an e
code must decide how to handle.

When your code performs operations on values
are valid first and panic if the values aren't valid.
attempting to operate on invalid data can expos
the main reason the standard library will call `pa`
memory access: trying to access memory that d
structure is a common security problem. Functio
behavior is only guaranteed if the inputs meet p
when the contract is violated makes sense becau
indicates a caller-side bug and it's not a kind of e
have to explicitly handle. In fact, there's no reaso
the calling *programmers* need to fix the code. Co
a violation will cause a panic, should be explaine
function.

However, having lots of error checks in all of you
annoying. Fortunately, you can use Rust's type sy
compiler does) to do many of the checks for you
as a parameter, you can proceed with your code
already ensured you have a valid value. For exam
`Option`, your program expects to have *somethin*
doesn't have to handle two cases for the `Some` a
one case for definitely having a value. Code tryin
won't even compile, so your function doesn't hav
Another example is using an unsigned integer ty
parameter is never negative.

Let's take the idea of using Rust's type system to
step further and look at creating a custom type
game in Chapter 2 in which our code asked the
and 100. We never validated that the user's gue
before checking it against our secret number; w
positive. In this case, the consequences were no
or "Too low" would still be correct. But it would
user toward valid guesses and have different be
that's out of range versus when a user types, fo

One way to do this would be to parse the guess
allow potentially negative numbers, and then ad
range, like so:

```
loop {
    // --snip--

    let guess: i32 = match guess.trim().pa
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will b
        continue;
    }

    match guess.cmp(&secret_number) {
    // --snip--
}
```

The `if` expression checks whether our value is
problem, and calls `continue` to start the next it
another guess. After the `if` expression, we can
between `guess` and the secret number knowin

However, this is not an ideal solution: if it was a
operated on values between 1 and 100, and it h
requirement, having a check like this in every fu
impact performance).

Instead, we can make a new type and put the va
instance of the type rather than repeating the va
safe for functions to use the new type in their si
values they receive. Listing 9-9 shows one way t
create an instance of `Guess` if the `new` function

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be be
value);
        }

        Guess {
            value
        }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Listing 9-10: A `Guess` type that will only continu

First, we define a struct named `Guess` that has a
. This is where the number will be stored.

Then we implement an associated function nam
instances of `Guess` values. The `new` function is
named `value` of type `i32` and to return a `Gues`
function tests `value` to make sure it's between
test, we make a `panic!` call, which will alert the
calling code that they have a bug they need to fi
`value` outside this range would violate the cont
The conditions in which `Guess::new` might pani
facing API documentation; we'll cover document
possibility of a `panic!` in the API documentation
`value` does pass the test, we create a new `Gues`
`value` parameter and return the `Guess`.

Next, we implement a method named `value` th
other parameters, and returns a `i32`. This kind
getter, because its purpose is to get some data fr
method is necessary because the `value` field of
important that the `value` field be private so coc
allowed to set `value` directly: code outside the
function to create an instance of `Guess`, thereby

to have a `value` that hasn't been checked by th
function.

A function that has a parameter or returns only
then declare in its signature that it takes or retu
wouldn't need to do any additional checks in its

Summary

Rust's error handling features are designed to h
`panic!` macro signals that your program is in a
the process to stop instead of trying to proceed
`Result` enum uses Rust's type system to indicat
that your code could recover from. You can use
code that it needs to handle potential success o
`Result` in the appropriate situations will make y
inevitable problems.

Now that you've seen useful ways that the stand
`Option` and `Result` enums, we'll talk about how
them in your code.

Generic Types, Traits, a

Every programming language has tools for effec
concepts. In Rust, one such tool is *generics*. Gene
concrete types or other properties. When we're
behavior of generics or how they relate to other
be in their place when compiling and running th

Similar to the way a function takes parameters w
code on multiple concrete values, functions can
type instead of a concrete type, like `i32` or `Str`
generics in Chapter 6 with `Option<T>`, Chapter
and Chapter 9 with `Result<T, E>`. In this chapt
own types, functions, and methods with generic

First, we'll review how to extract a function to re
the same technique to make a generic function
the types of their parameters. We'll also explain
and enum definitions.

Then you'll learn how to use *traits* to define beha
combine traits with generic types to constrain a
have a particular behavior, as opposed to just ar

Finally, we'll discuss *lifetimes*, a variety of generic
about how references relate to each other. Lifet
many situations while still enabling the compiler
valid.

Removing Duplication by Extra

Before diving into generics syntax, let's first look
doesn't involve generic types by extracting a fun
to extract a generic function! In the same way th
extract into a function, you'll start to recognize d

Consider a short program that finds the largest
10-1.

Filename: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100
    
    let mut largest = number_list[0];
    
    for number in number_list {
        if number > largest {
            largest = number;
        }
    }
    
    println!("The largest number is {}", l
}
```

Listing 10-1: Code to find the largest number in a

This code stores a list of integers in the variable
number in the list in a variable named `largest`.
numbers in the list, and if the current number is
`largest`, it replaces the number in that variable
less than the largest number seen so far, the var
moves on to the next number in the list. After co
`largest` should hold the largest number, which

To find the largest number in two different lists
code in Listing 10-1 and use the same logic at tw
shown in Listing 10-2.

Filename: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 10(

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", l

    let number_list = vec![102, 34, 6000,

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", l
}
```

Listing 10-2: Code to find the largest number in

Although this code works, duplicating code is te
to update the code in multiple places when we w

To eliminate this duplication, we can create an a
operates on any list of integers given to it in a pa
code clearer and lets us express the concept of
abstractly.

In Listing 10-3, we extracted the code that finds
named `largest`. Unlike the code in Listing 10-1,
only one particular list, this program can find th

Filename: src/main.rs

```
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 10(

    let result = largest(&number_list);
    println!("The largest number is {}", r

    let number_list = vec![102, 34, 6000,

    let result = largest(&number_list);
    println!("The largest number is {}", r
}
```

Listing 10-3: Abstracted code to find the largest

The `largest` function has a parameter called `l` slice of `i32` values that we might pass into the f function, the code runs on the specific values th

In sum, here are the steps we took to change th 10-3:

1. Identify duplicate code.
2. Extract the duplicate code into the body of and return values of that code in the functi
3. Update the two instances of duplicated co

Next, we'll use these same steps with generics to ways. In the same way that the function body ca instead of specific values, generics allow code to

For example, say we had two functions: one that `i32` values and one that finds the largest item i we eliminate that duplication? Let's find out!

Generic Data Types

We can use generics to create definitions for ite which we can then use with many different conc to define functions, structs, enums, and method how generics affect code performance.

In Function Definitions

When defining a function that uses generics, we the function where we would usually specify the return value. Doing so makes our code more fle to callers of our function while preventing code

Continuing with our `largest` function, Listing 1 find the largest value in a slice.

Filename: src/main.rs

```
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 10(

    let result = largest_i32(&number_list)
    println!("The largest number is {}", r

    let char_list = vec!['y', 'm', 'a', 'c

    let result = largest_char(&char_list);
    println!("The largest char is {}", res
}
```

Listing 10-4: Two functions that differ only in the signatures

The `largest_i32` function is the one we extract largest `i32` in a slice. The `largest_char` functic The function bodies have the same code, so let's introducing a generic type parameter in a single

To parameterize the types in the new function w parameter, just as we do for the value paramete identifier as a type parameter name. But we'll us parameter names in Rust are short, often just a convention is CamelCase. Short for "type," `T` is t programmers.

When we use a parameter in the body of the fur
parameter name in the signature so the compile
Similarly, when we use a type parameter name i
declare the type parameter name before we use
function, place type name declarations inside ar
of the function and the parameter list, like this:

```
fn largest<T>(list: &[T]) -> T {
```

We read this definition as: the function `largest`
function has one parameter named `list`, whicl
`largest` function will return a value of the sam

Listing 10-5 shows the combined `largest` funct
type in its signature. The listing also shows how
slice of `i32` values or `char` values. Note that th
it later in this chapter.

Filename: src/main.rs

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 10(

    let result = largest(&number_list);
    println!("The largest number is {}", r

    let char_list = vec!['y', 'm', 'a', 'c

    let result = largest(&char_list);
    println!("The largest char is {}", res
}
```

Listing 10-5: A definition of the `largest` functio
but doesn't compile yet

If we compile this code right now, we'll get this e

```
error[E0369]: binary operation `>` cannot
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::
`T`
```

The note mentions `std::cmp::PartialOrd`, whi
the next section. For now, this error states that t
all possible types that `T` could be. Because we w
the body, we can only use types whose values ca
comparisons, the standard library has the `std:
implement on types (see Appendix C for more o
that a generic type has a particular trait in the "T
explore other ways of using generic type param

In Struct Definitions

We can also define structs to use a generic type
the `<>` syntax. Listing 10-6 shows how to define
coordinate values of any type.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Listing 10-6: A `Point<T>` struct that holds `x` an

The syntax for using generics in struct definition
definitions. First, we declare the name of the typ
just after the name of the struct. Then we can us
definition where we would otherwise specify con

Note that because we've used only one generic t
definition says that the `Point<T>` struct is gene

`x` and `y` are *both* that same type, whatever tha
instance of a `Point<T>` that has values of differ
won't compile.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 }
}
```

Listing 10-7: The fields `x` and `y` must be the sa
generic data type `T`.

In this example, when we assign the integer valu
that the generic type `T` will be an integer for thi
we specify 4.0 for `y`, which we've defined to hav
mismatch error like this:

```
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4
  |                                      ^
found
floating-point variable
  |
  = note: expected type `{integer}`
             found type `{float}`
```

To define a `Point` struct where `x` and `y` are bo
types, we can use multiple generic type paramet
can change the definition of `Point` to be generi
type `T` and `y` is of type `U`.

Filename: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10
    let both_float = Point { x: 1.0, y: 4.
    let integer_and_float = Point { x: 5,
}
```

Listing 10-8: A `Point<T, U>` generic over two ty
different types

Now all the instances of `Point` shown are allow
parameters in a definition as you want, but usin
hard to read. When you need lots of generic typ
your code needs restructuring into smaller piece

In Enum Definitions

As we did with structs, we can define enums to h
variants. Let's take another look at the `Option<T`
provides, which we used in Chapter 6:

```
enum Option<T> {
    Some(T),
    None,
}
```

This definition should now make more sense to
enum that is generic over type `T` and has two v
of type `T`, and a `None` variant that doesn't hold
enum, we can express the abstract concept of h
`Option<T>` is generic, we can use this abstractio
optional value is.

Enums can use multiple generic types as well. Th
we used in Chapter 9 is one example:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The `Result` enum is generic over two types, `T` which holds a value of type `T`, and `Err`, which l definition makes it convenient to use the `Resul` operation that might succeed (return a value of of some type `E`). In fact, this is what we used to was filled in with the type `std::fs::File` when `E` was filled in with the type `std::io::Error` w file.

When you recognize situations in your code witl that differ only in the types of the values they hc generic types instead.

In Method Definitions

We can implement methods on structs and enu generic types in their definitions, too. Listing 10- defined in Listing 10-6 with a method named `x`

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Listing 10-9: Implementing a method named `x` return a reference to the `x` field of type `T`

Here, we've defined a method named `x` on `Poi`
data in the field `x` .

Note that we have to declare `T` just after `impl`
implementing methods on the type `Point<T>` . E
`impl` , Rust can identify that the type in the angl
rather than a concrete type.

We could, for example, implement methods onl
than on `Point<T>` instances with any generic ty
concrete type `f32` , meaning we don't declare ar

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32
        (self.x.powi(2) + self.y.powi(2)).
    }
}
```

Listing 10-10: An `impl` block that only applies tc
type for the generic type parameter `T`

This code means the type `Point<f32>` will have
`distance_from_origin` and other instances of
will not have this method defined. The method r
the point at coordinates (0.0, 0.0) and uses math
only for floating point types.

Generic type parameters in a struct definition ar
use in that struct's method signatures. For exam
`mixup` on the `Point<T, U>` struct from Listing 1
`Point` as a parameter, which might have differe
calling `mixup` on. The method creates a new `Po`
the `self` `Point` (of type `T`) and the `y` value fr

Filename: src/main.rs

```rust
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c'};

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x,
}
```

Listing 10-11: A method that uses different gene

In `main`, we've defined a `Point` that has an `i32`
`y` (with value `10.4`). The `p2` variable is a `Point`
(with value `"Hello"`) and a `char` for `y` (with va
argument `p2` gives us `p3`, which will have an `i`
The `p3` variable will have a `char` for `y`, becaus
macro call will print `p3.x = 5, p3.y = c`.

The purpose of this example is to demonstrate a
parameters are declared with `impl` and some a
definition. Here, the generic parameters `T` and
they go with the struct definition. The generic pa
`fn mixup`, because they're only relevant to the r

Performance of Code Using Generics

You might be wondering whether there is a runt
type parameters. The good news is that Rust im
your code doesn't run any slower using generic
types.

Rust accomplishes this by performing monomor

generics at compile time. *Monomorphization* is th
into specific code by filling in the concrete types

In this process, the compiler does the opposite o
generic function in Listing 10-5: the compiler loo
code is called and generates code for the concre
with.

Let's look at how this works with an example tha
`Option<T>` enum:

```
let integer = Some(5);
let float = Some(5.0);
```

When Rust compiles this code, it performs mono
the compiler reads the values that have been us
identifies two kinds of `Option<T>`: one is `i32` a
expands the generic definition of `Option<T>` int
thereby replacing the generic definition with the

The monomorphized version of the code looks l
`Option<T>` is replaced with the specific definitio

Filename: src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Because Rust compiles generic code into code th
we pay no runtime cost for using generics. When
would if we had duplicated each definition by ha
monomorphization makes Rust's generics extre

Traits: Defining Shared Behavi

A *trait* tells the Rust compiler about functionality
with other types. We can use traits to define sha
can use trait bounds to specify that a generic ca
behavior.

> Note: Traits are similar to a feature often call
> although with some differences.

Defining a Trait

A type's behavior consists of the methods we ca
share the same behavior if we can call the same
definitions are a way to group method signature
behaviors necessary to accomplish some purpo

For example, let's say we have multiple structs t
of text: a `NewsArticle` struct that holds a news
a `Tweet` that can have at most 280 characters a
whether it was a new tweet, a retweet, or a reply

We want to make a media aggregator library tha
might be stored in a `NewsArticle` or `Tweet` ins
from each type, and we need to request that su
method on an instance. Listing 10-12 shows the
expresses this behavior.

Filename: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Listing 10-12: A `Summary` trait that consists of th
method

Here, we declare a trait using the `trait` keywor
`Summary` in this case. Inside the curly brackets, v
describe the behaviors of the types that implem
`fn summarize(&self) -> String`.

After the method signature, instead of providing brackets, we use a semicolon. Each type implem custom behavior for the body of the method. Th that has the `Summary` trait will have the method signature exactly.

A trait can have multiple methods in its body: th per line and each line ends in a semicolon.

Implementing a Trait on a Type

Now that we've defined the desired behavior us implement it on the types in our media aggregat implementation of the `Summary` trait on the `New` headline, the author, and the location to create the `Tweet` struct, we define `summarize` as the u of the tweet, assuming that tweet content is alre

Filename: src/lib.rs

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.hea
self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, s
    }
}
```

Listing 10-13: Implementing the `Summary` trait o

Implementing a trait on a type is similar to imple
difference is that after `impl`, we put the trait na
use the `for` keyword, and then specify the nam
the trait for. Within the `impl` block, we put the r
definition has defined. Instead of adding a semi
curly brackets and fill in the method body with t
methods of the trait to have for the particular ty

After implementing the trait, we can call the me
and `Tweet` in the same way we call regular meth

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as y
people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summariz
```

This code prints

`1 new tweet: horse_ebooks: of course, as y`

Note that because we defined the `Summary` trait
types in the same *lib.rs* in Listing 10-13, they're a
lib.rs is for a crate we've called `aggregator` and
crate's functionality to implement the `Summary` t
library's scope. They would need to import the tr
do so by specifying `use aggregator::Summary;`
implement `Summary` for their type. The `Summary`
trait for another crate to implement it, which it is
before `trait` in Listing 10-12.

One restriction to note with trait implementation
a type only if either the trait or the type is local t
implement standard library traits like `Display` o
our `aggregator` crate functionality, because the
`aggregator` crate. We can also implement `Summ`
crate, because the trait `Summary` is local to our

But we can't implement external traits on extern
implement the `Display` trait on `Vec<T>` within

`Display` and `Vec<T>` are defined in the standar
`aggregator` crate. This restriction is part of a pr
and more specifically the *orphan rule*, so named
present. This rule ensures that other people's co
versa. Without the rule, two crates could implem
and Rust wouldn't know which implementation t

Default Implementations

Sometimes it's useful to have default behavior fo
trait instead of requiring implementations for al
implement the trait on a particular type, we can
default behavior.

Listing 10-14 shows how to specify a default stri
`Summary` trait instead of only defining the meth
10-12.

Filename: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Listing 10-14: Definition of a `Summary` trait with
`summarize` method

To use a default implementation to summarize i
defining a custom implementation, we specify a
`impl Summary for NewsArticle {}`.

Even though we're no longer defining the `summa`
directly, we've provided a default implementatio
implements the `Summary` trait. As a result, we ca
an instance of `NewsArticle`, like this:

```
let article = NewsArticle {
    headline: String::from("Penguins win t
    location: String::from("Pittsburgh, PA
    author: String::from("Iceburgh"),
    content: String::from("The Pittsburgh
    hockey team in the NHL."),
};

println!("New article available! {}", arti
```

This code prints `New article available! (Read`

Creating a default implementation for `summariz`
anything about the implementation of `Summary`
reason is that the syntax for overriding a default
syntax for implementing a trait method that doe

Default implementations can call other methods
methods don't have a default implementation. I
useful functionality and only require implement
example, we could define the `Summary` trait to h
whose implementation is required, and then de
default implementation that calls the `summarize`

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)",
    }
}
```

To use this version of `Summary`, we only need to
implement the trait on a type:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

After we define `summarize_author`, we can call
struct, and the default implementation of `summa`
`summarize_author` that we've provided. Becaus
`summarize_author`, the `Summary` trait has given
method without requiring us to write any more

```
let tweet = Tweet {
    username: String::from("horse_ebooks")
    content: String::from("of course, as y
people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summari
```

This code prints `1 new tweet: (Read more fro`

Note that it isn't possible to call the default impl
implementation of that same method.

Traits as arguments

Now that you know how to define traits and imp
explore how to use traits to accept arguments o

For example, in Listing 10-13, we implemented t
`NewsArticle` and `Tweet`. We can define a funct
method on its parameter `item`, which is of som
trait. To do this, we can use the '`impl Trait`' sy

```
pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.sum
}
```

In the body of `notify`, we can call any methods
`Summary` trait, like `summarize`.

Trait Bounds

The `impl Trait` syntax works for short exampl
form. This is called a 'trait bound', and it looks lil

```
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.sum
}
```

This is equivalent to the example above, but is a
bounds with the declaration of the generic type
angle brackets. Because of the trait bound on `T`

instance of `NewsArticle` or `Tweet`. Code that c
like a `String` or an `i32`, won't compile, becaus
`Summary`.

When should you use this form over `impl Trai`
shorter examples, trait bounds are nice for mor
wanted to take two things that implement `Summ`

```
pub fn notify(item1: impl Summary, item2:
pub fn notify<T: Summary>(item1: T, item2:
```

The version with the bound is a bit easier. In ger
makes your code the most understandable.

Multiple trait bounds with `+`

We can specify multiple trait bounds on a gener
example, to use display formatting on the type
`summarize` method, we can use `T: Summary +`
that implements `Summary` and `Display`. This ca

`where` clauses for clearer code

However, there are downsides to using too man
own trait bounds, so functions with multiple ger
trait bound information between a function's na
function signature hard to read. For this reason,
specifying trait bounds inside a `where` clause af
of writing this:

```
fn some_function<T: Display + Clone, U: Cl
{
```

we can use a `where` clause, like this:

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

This function's signature is less cluttered in that
and return type are close together, similar to a f

Returning Traits

We can use the `impl Trait` syntax in return po
that implements a trait:

```
fn returns_summarizable() -> impl Summary
    Tweet {
        username: String::from("horse_eboo
        content: String::from("of course,
people"),
        reply: false,
        retweet: false,
    }
}
```

This signature says, "I'm going to return someth
trait, but I'm not going to tell you the exact type.
, but the caller doesn't know that.

Why is this useful? In chapter 13, we're going to
heavily on traits: closures, and iterators. These f
compiler knows, or types that are very, very long
"this returns an `Iterator` " without needing to w

This only works if you have a single type that you
this would *not* work:

```
fn returns_summarizable(switch: bool) -> i
    if switch {
        NewsArticle {
            headline: String::from("Pengui
Championship!"),
            location: String::from("Pittsb
            author: String::from("Iceburgh
            content: String::from("The Pit
the best
            hockey team in the NHL."),
        }
    } else {
        Tweet {
            username: String::from("horse_
            content: String::from("of cour
know, people"),
            reply: false,
            retweet: false,
        }
    }
}
```

Here, we try to return either a `NewsArticle` or a
restrictions around how `impl Trait` works. To
until Chapter 17, "trait objects".

Fixing the `largest` Function with Trait

Now that you know how to specify the behavior
type parameter's bounds, let's return to Listing 1
`largest` function that uses a generic type para
code, we received this error:

```
error[E0369]: binary operation `>` cannot
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::
`T`
```

In the body of `largest` we wanted to compare
greater than (`>`) operator. Because that operat
the standard library trait `std::cmp::PartialOrc`
the trait bounds for `T` so the `largest` function
can compare. We don't need to bring `PartialOr`
prelude. Change the signature of `largest` to lo

```
fn largest<T: PartialOrd>(list: &[T]) -> T
```

This time when we compile the code, we get a d

```
error[E0508]: cannot move out of type `[T]
 --> src/main.rs:2:23
  |
2 |     let mut largest = list[0];
  |                       ^^^^^^^
  |                       |
  |                       cannot move out
  |                       help: consider u
`&list[0]`

error[E0507]: cannot move out of borrowed
 --> src/main.rs:4:9
  |
4 |     for &item in list.iter() {
  |         ^----
  |         ||
  |         |hint: to prevent move, use `r
  |         cannot move out of borrowed co
```

The key line in this error is `cannot move out of` our non-generic versions of the `largest` functic largest `i32` or `char`. As discussed in the "Stack- 4, types like `i32` and `char` that have a known s implement the `Copy` trait. But when we made tl became possible for the `list` parameter to hav `Copy` trait. Consequently, we wouldn't be able t into the `largest` variable, resulting in this error

To call this code with only those types that imple `Copy` to the trait bounds of `T`! Listing 10-15 sho `largest` function that will compile as long as th we pass into the function implement the `Partia` `char` do.

Filename: src/main.rs

```
fn largest<T: PartialOrd + Copy>(list: &[T
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 10(

    let result = largest(&number_list);
    println!("The largest number is {}", r

    let char_list = vec!['y', 'm', 'a', 'c

    let result = largest(&char_list);
    println!("The largest char is {}", res
}
```

Listing 10-15: A working definition of the `larges` type that implements the `PartialOrd` and `Copy`

If we don't want to restrict the `largest` functior `Copy` trait, we could specify that `T` has the trait we could clone each value in the slice when we v ownership. Using the `clone` function means we allocations in the case of types that own heap da can be slow if we're working with large amounts

Another way we could implement `largest` is fo a `T` value in the slice. If we change the return ty changing the body of the function to return a re or `Copy` trait bounds and we could avoid heap a alternate solutions on your own!

Using Trait Bounds to Conditionally Im

By using a trait bound with an `impl` block that u implement methods conditionally for types that example, the type `Pair<T>` in Listing 10-16 alwa `Pair<T>` only implements the `cmp_display` me

the `PartialOrd` trait that enables comparison a
printing.

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member i
        } else {
            println!("The largest member i
        }
    }
}
```

Listing 10-16: Conditionally implement methods
bounds

We can also conditionally implement a trait for a
trait. Implementations of a trait on any type that
blanket implementations and are extensively use
example, the standard library implements the `T`
implements the `Display` trait. The `impl` block i
this code:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Because the standard library has this blanket im
`to_string` method defined by the `ToString` tr
`Display` trait. For example, we can turn integer
values like this because integers implement `Dis`

```
let s = 3.to_string();
```

Blanket implementations appear in the docume
“Implementors” section.

Traits and trait bounds let us write code that use
duplication but also specify to the compiler that
particular behavior. The compiler can then use t
that all the concrete types used with our code pr
dynamically typed languages, we would get an e
on a type that the type didn’t implement. But Ru
so we’re forced to fix the problems before our co
we don’t have to write code that checks for beha
already checked at compile time. Doing so impro
give up the flexibility of generics.

Another kind of generic that we’ve already been
ensuring that a type has the behavior we want, l
valid as long as we need them to be. Let’s look a

Validating References with Lif

One detail we didn’t discuss in the “References a
that every reference in Rust has a *lifetime*, which
is valid. Most of the time, lifetimes are implicit an
types are inferred. We must annotate types whe
similar way, we must annotate lifetimes when th
related in a few different ways. Rust requires us
generic lifetime parameters to ensure the actua
definitely be valid.

The concept of lifetimes is somewhat different fr
languages, arguably making lifetimes Rust’s mos
won’t cover lifetimes in their entirety in this chap
might encounter lifetime syntax so you can beco
the “Advanced Lifetimes” section in Chapter 19 f

Preventing Dangling References with L

The main aim of lifetimes is to prevent dangling

reference data other than the data it's intended
Listing 10-17, which has an outer scope and an i

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

Listing 10-17: An attempt to use a reference who

> Note: The examples in Listings 10-17, 10-18, a
> giving them an initial value, so the variable na
> first glance, this might appear to be in conflict
> However, if we try to use a variable before giv
> time error, which shows that Rust indeed doe

The outer scope declares a variable named `r` w
scope declares a variable named `x` with the init
we attempt to set the value of `r` as a reference
we attempt to print the value in `r`. This code wo
referring to has gone out of scope before we try

```
error[E0597]: `x` does not live long enoug
  --> src/main.rs:7:5
   |
6  |         r = &x;
   |              - borrow occurs here
7  |     }
   |     ^ `x` dropped here while still bo
...
10 | }
   | - borrowed value needs to live until
```

The variable `x` doesn't "live long enough." The r
when the inner scope ends on line 7. But `r` is st
its scope is larger, we say that it "lives longer." If
would be referencing memory that was dealloca
anything we tried to do with `r` wouldn't work co
that this code is invalid? It uses a borrow checke

The Borrow Checker

The Rust compiler has a *borrow checker* that con
all borrows are valid. Listing 10-18 shows the sa
annotations showing the lifetimes of the variabl

```
{
    let r;                // ---------+--
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

Listing 10-18: Annotations of the lifetimes of `r`
respectively

Here, we've annotated the lifetime of `r` with `'a`
you can see, the inner `'b` block is much smaller
compile time, Rust compares the size of the two
lifetime of `'a` but that it refers to memory with
rejected because `'b` is shorter than `'a`: the su
long as the reference.

Listing 10-19 fixes the code so it doesn't have a
without any errors.

```
{
    let x = 5;            // ----------+-
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Listing 10-19: A valid reference because the data
reference

Here, `x` has the lifetime `'b`, which in this case i
reference `x` because Rust knows that the refere
is valid.

Now that you know where the lifetimes of refere
lifetimes to ensure references will always be vali
parameters and return values in the context of f

Generic Lifetimes in Functions

Let's write a function that returns the longer of t
two string slices and return a string slice. After w
function, the code in Listing 10-20 should print

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(),
    println!("The longest string is {}", r
}
```

Listing 10-20: A `main` function that calls the `lon`
two string slices

Note that we want the function to take string sli
don't want the `longest` function to take owners
allow the function to accept slices of a `String` (
`string1`) as well as string literals (which is what

Refer to the "String Slices as Parameters" section
about why the parameters we use in Listing 10-2

If we try to implement the `longest` function as
compile.

Filename: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-21: An implementation of the `longest`
two string slices but does not yet compile

Instead, we get the following error that talks abo

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expe
  |
  = help: this function's return type cont
signature does not say whether it is borro
```

The help text reveals that the return type needs because Rust can't tell whether the reference be Actually, we don't know either, because the `if` b returns a reference to `x` and the `else` block re

When we're defining this function, we don't know passed into this function, so we don't know whe will execute. We also don't know the concrete lif passed in, so we can't look at the scopes as we c determine whether the reference we return will can't determine this either, because it doesn't kn relate to the lifetime of the return value. To fix th parameters that define the relationship between checker can perform its analysis.

Lifetime Annotation Syntax

Lifetime annotations don't change how long any functions can accept any type when the signatur functions can accept references with any lifetim parameter. Lifetime annotations describe the re references to each other without affecting the li

Lifetime annotations have a slightly unusual syn must start with an apostrophe (`'`) and are usua generic types. Most people use the name `'a` . W annotations after the `&` of a reference, using a s the reference's type.

Here are some examples: a reference to an `i32` reference to an `i32` that has a lifetime paramet reference to an `i32` that also has the lifetime `'`

```
&i32        // a reference
&'a i32     // a reference with an explic
&'a mut i32 // a mutable reference with an
```

One lifetime annotation by itself doesn't have m annotations are meant to tell Rust how generic l references relate to each other. For example, let parameter `first` that is a reference to an `i32` has another parameter named `second` that is a has the lifetime `'a`. The lifetime annotations in `second` must both live as long as that generic lif

Lifetime Annotations in Function Signa

Now let's examine lifetime annotations in the co with generic type parameters, we need to declar angle brackets between the function name and want to express in this signature is that all the re return value must have the same lifetime. We'll to each reference, as shown in Listing 10-22.

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) ->
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-22: The `longest` function definition s signature must have the same lifetime `'a`

This code should compile and produce the resul `main` function in Listing 10-20.

The function signature now tells Rust that for so two parameters, both of which are string slices t . The function signature also tells Rust that the s will live at least as long as lifetime `'a`. These co enforce. Remember, when we specify the lifetim signature, we're not changing the lifetimes of an

Rather, we're specifying that the borrow checker
adhere to these constraints. Note that the `long
exactly how long `x` and `y` will live, only that so
that will satisfy this signature.

When annotating lifetimes in functions, the ann
not in the function body. Rust can analyze the c
help. However, when a function has references t
it becomes almost impossible for Rust to figure
return values on its own. The lifetimes might be
called. This is why we need to annotate the lifeti

When we pass concrete references to `longest`,
substituted for `'a` is the part of the scope of `x`
other words, the generic lifetime `'a` will get the
smaller of the lifetimes of `x` and `y`. Because we
with the same lifetime parameter `'a`, the return
length of the smaller of the lifetimes of `x` and `y

Let's look at how the lifetime annotations restric
references that have different concrete lifetimes
example.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long strin

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_st
        println!("The longest string is {}
    }
}
```

Listing 10-23: Using the `longest` function with r
different concrete lifetimes

In this example, `string1` is valid until the end o
until the end of the inner scope, and `result` ref
the end of the inner scope. Run this code, and yo
approves of this code; it will compile and print
`The longest string is long string is long`.

Next, let's try an example that shows that the life
must be the smaller lifetime of the two argumen

`result` variable outside the inner scope but lea
`result` variable inside the scope with `string2`
uses `result` outside the inner scope, after the i
Listing 10-24 will not compile.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long strin
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(),
    }
    println!("The longest string is {}", r
}
```

Listing 10-24: Attempting to use `result` after `s

When we try to compile this code, we'll get this e

```
error[E0597]: `string2` does not live long
  --> src/main.rs:15:5
   |
14 |         result = longest(string1.as_s
   |
15 |     }
   |     ^ `string2` dropped here while st
16 |     println!("The longest string is {
17 | }
   | - borrowed value needs to live until
```

The error shows that for `result` to be valid for
would need to be valid until the end of the oute
annotated the lifetimes of the function paramet
lifetime parameter `'a`.

As humans, we can look at this code and see tha
and therefore `result` will contain a reference to
gone out of scope yet, a reference to `string1` w
statement. However, the compiler can't see that
We've told Rust that the lifetime of the reference
the same as the smaller of the lifetimes of the re
borrow checker disallows the code in Listing 10-
reference.

Try designing more experiments that vary the va

passed in to the `longest` function and how the
hypotheses about whether or not your experim
before you compile; then check to see if you're r

Thinking in Terms of Lifetimes

The way in which you need to specify lifetime pa
function is doing. For example, if we changed th
function to always return the first parameter rat
wouldn't need to specify a lifetime on the `y` par
compile:

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a
    x
}
```

In this example, we've specified a lifetime param
return type, but not for the parameter `y`, becau
any relationship with the lifetime of `x` or the ret

When returning a reference from a function, the
type needs to match the lifetime parameter for
reference returned does *not* refer to one of the
created within this function, which would be a d
will go out of scope at the end of the function. C
implementation of the `longest` function that w

Filename: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a st
    let result = String::from("really long
    result.as_str()
}
```

Here, even though we've specified a lifetime par
implementation will fail to compile because the
the lifetime of the parameters at all. Here is the

```
error[E0597]: `result` does not live long
 --> src/main.rs:3:5
  |
3 |     result.as_str()
  |     ^^^^^^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
note: borrowed value must be valid for the
function body at 1:1...
 --> src/main.rs:1:1
  |
1 | / fn longest<'a>(x: &str, y: &str) ->
2 | |     let result = String::from("reall
3 | |     result.as_str()
4 | | }
  | |_^
```

The problem is that `result` goes out of scope a `longest` function. We're also trying to return a r function. There is no way we can specify lifetime dangling reference, and Rust won't let us create best fix would be to return an owned data type r function is then responsible for cleaning up the

Ultimately, lifetime syntax is about connecting th and return values of functions. Once they're con to allow memory-safe operations and disallow o pointers or otherwise violate memory safety.

Lifetime Annotations in Struct Definiti

So far, we've only defined structs to hold owned references, but in that case we would need to ad reference in the struct's definition. Listing 10-25 `ImportantExcerpt` that holds a string slice.

Filename: src/main.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishm
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first
}
```

Listing 10-25: A struct that holds a reference, so annotation

This struct has one field, `part`, that holds a strir generic data types, we declare the name of the g angle brackets after the name of the struct so w the body of the struct definition. This annotatior `ImportantExcerpt` can't outlive the reference it

The `main` function here creates an instance of t holds a reference to the first sentence of the `St` The data in `novel` exists before the `ImportantE` addition, `novel` doesn't go out of scope until aft scope, so the reference in the `ImportantExcerp`

Lifetime Elision

You've learned that every reference has a lifetim lifetime parameters for functions or structs that 4 we had a function in Listing 4-9, which is show without lifetime annotations.

Filename: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumera
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Listing 10-26: A function we defined in Listing 4-! annotations, even though the parameter and re

The reason this function compiles without lifetin versions (pre-1.0) of Rust, this code wouldn't hav needed an explicit lifetime. At that time, the fun written like this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

After writing a lot of Rust code, the Rust team fo entering the same lifetime annotations over and situations were predictable and followed a few c programmed these patterns into the compiler's infer the lifetimes in these situations and would

This piece of Rust history is relevant because it's patterns will emerge and be added to the compi annotations might be required.

The patterns programmed into Rust's analysis o *elision rules*. These aren't rules for programmers cases that the compiler will consider, and if your to write the lifetimes explicitly.

The elision rules don't provide full inference. If R but there is still ambiguity as to what lifetimes th won't guess what the lifetime of the remaining r instead of guessing, the compiler will give you ar the lifetime annotations that specify how the ref

Lifetimes on function or method parameters are return values are called *output lifetimes*.

The compiler uses three rules to figure out what
aren't explicit annotations. The first rule applies
third rules apply to output lifetimes. If the comp
and there are still references for which it can't fi
stop with an error.

The first rule is that each parameter that is a ref
parameter. In other words, a function with one p
parameter: `fn foo<'a>(x: &'a i32)` ; a functio
separate lifetime parameters: `fn foo<'a, 'b>(`

The second rule is if there is exactly one input lif
assigned to all output lifetime parameters: `fn f`

The third rule is if there are multiple input lifetin
`&self` or `&mut self` because this is a method,
output lifetime parameters. This third rule make
write because fewer symbols are necessary.

Let's pretend we're the compiler. We'll apply the
lifetimes of the references in the signature of th
10-26 are. The signature starts without any lifeti

```
fn first_word(s: &str) -> &str {
```

Then the compiler applies the first rule, which sp
own lifetime. We'll call it `'a` as usual, so now the

```
fn first_word<'a>(s: &'a str) -> &str {
```

The second rule applies because there is exactly
specifies that the lifetime of the one input paran
lifetime, so the signature is now this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Now all the references in this function signature
continue its analysis without needing the progra
this function signature.

Let's look at another example, this time using th
lifetime parameters when we started working w

```
fn longest(x: &str, y: &str) -> &str {
```

Let's apply the first rule: each parameter gets its
parameters instead of one, so we have two lifeti

```
fn longest<'a, 'b>(x: &'a str, y: &'b str)
```

You can see that the second rule doesn't apply b
lifetime. The third rule doesn't apply either, beca
than a method, so none of the parameters are
rules, we still haven't figured out what the return
an error trying to compile the code in Listing 10-
lifetime elision rules but still couldn't figure out a
the signature.

Because the third rule really only applies in meth
in that context next to see why the third rule me
lifetimes in method signatures very often.

Lifetime Annotations in Method Defini

When we implement methods on a struct with li
that of generic type parameters shown in Listing
lifetime parameters depends on whether they're
method parameters and return values.

Lifetime names for struct fields always need to b
and then used after the struct's name, because t
type.

In method signatures inside the `impl` block, ref
of references in the struct's fields, or they might
lifetime elision rules often make it so that lifetim
method signatures. Let's look at some examples
`ImportantExcerpt` that we defined in Listing 10

First, we'll use a method named `level` whose o
and whose return value is an `i32`, which is not

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

The lifetime parameter declaration after `impl` a
required, but we're not required to annotate the
because of the first elision rule.

Here is an example where the third lifetime elisi

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, ann
        println!("Attention please: {}", a
        self.part
    }
}
```

There are two input lifetimes, so Rust applies th
both `&self` and `announcement` their own lifetim
parameters is `&self`, the return type gets the li
have been accounted for.

The Static Lifetime

One special lifetime we need to discuss is `'stat
duration of the program. All string literals have t
annotate as follows:

```
let s: &'static str = "I have a static lif
```

The text of this string is stored directly in the bir
available. Therefore, the lifetime of all string lite

You might see suggestions to use the `'static` l
specifying `'static` as the lifetime for a referen
you have actually lives the entire lifetime of you
whether you want it to live that long, even if it co
results from attempting to create a dangling ref
lifetimes. In such cases, the solution is fixing tho
`'static` lifetime.

Generic Type Parameters, Tra
Lifetimes Together

Let's briefly look at the syntax of specifying gene
lifetimes all in one function!

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x:
&'a str
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

This is the `longest` function from Listing 10-22
slices. But now it has an extra parameter namec
can be filled in by any type that implements the
`where` clause. This extra parameter will be print
lengths of the string slices, which is why the `Dis`
Because lifetimes are a type of generic, the decl
and the generic type parameter `T` go in the san
the function name.

Summary

We covered a lot in this chapter! Now that you k
traits and trait bounds, and generic lifetime para
without repetition that works in many different
let you apply the code to different types. Traits a
though the types are generic, they'll have the be
how to use lifetime annotations to ensure that t
dangling references. And all of this analysis hap
affect runtime performance!

Believe it or not, there is much more to learn on
chapter: Chapter 17 discusses trait objects, whic
Chapter 19 covers more complex scenarios invo
some advanced type system features. But next,
so you can make sure your code is working the

Writing Automated Tes

In his 1972 essay "The Humble Programmer," Ec testing can be a very effective way to show the p inadequate for showing their absence." That doe much as we can!

Correctness in our programs is the extent to wh do. Rust is designed with a high degree of conce programs, but correctness is complex and not e shoulders a huge part of this burden, but the typ incorrectness. As such, Rust includes support fo within the language.

As an example, say we write a function called `ac` number is passed to it. This function's signature and returns an integer as a result. When we imp Rust does all the type checking and borrow chec ensure that, for instance, we aren't passing a `St` this function. But Rust *can't* check that this funct which is return the parameter plus 2 rather than parameter minus 50! That's where tests come in

We can write tests that assert, for example, that function, the returned value is `5`. We can run th to our code to make sure any existing correct be

Testing is a complex skill: although we can't cove good tests in one chapter, we'll discuss the mech talk about the annotations and macros available default behavior and options provided for runni tests into unit tests and integration tests.

How to Write Tests

Tests are Rust functions that verify that the non- expected manner. The bodies of test functions t

1. Set up any needed data or state.
2. Run the code you want to test.
3. Assert the results are what you expect.

Let's look at the features Rust provides specifica
actions, which include the `test` attribute, a few
attribute.

The Anatomy of a Test Function

At its simplest, a test in Rust is a function that's a
Attributes are metadata about pieces of Rust co
attribute we used with structs in Chapter 5. To c
add `#[test]` on the line before `fn` . When you r
command, Rust builds a test runner binary that
`test` attribute and reports on whether each tes

In Chapter 7, we saw that when we make a new
module with a test function in it is automatically
you start writing your tests so you don't have to
syntax of test functions every time you start a ne
additional test functions and as many test modu

We'll explore some aspects of how tests work by
generated for us without actually testing any cod
tests that call some code that we've written and

Let's create a new library project called `adder` :

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

The contents of the *src/lib.rs* file in your `adder` l

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Listing 11-1: The test module and function gene

For now, let's ignore the top two lines and focus
Note the `#[test]` annotation before the `fn` lin

function, so the test runner knows to treat this f
non-test functions in the `tests` module to help
common operations, so we need to indicate whi
`#[test]` attribute.

The function body uses the `assert_eq!` macro
assertion serves as an example of the format fo
this test passes.

The `cargo test` command runs all tests in our

```
$ cargo test
   Compiling adder v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/adder-ce99b

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 igr

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 igr
```

Listing 11-2: The output from running the autom

Cargo compiled and ran the test. After the `Comp`
is the line `running 1 test`. The next line shows
function, called `it_works`, and the result of run
summary of running the tests appears next. The
all the tests passed, and the portion that reads
number of tests that passed or failed.

Because we don't have any tests we've marked a
`0 ignored`. We also haven't filtered the tests be
shows `0 filtered out`. We'll talk about ignorin
section, "Controlling How Tests Are Run."

The `0 measured` statistic is for benchmark tests
Benchmark tests are, as of this writing, only avai
documentation about benchmark tests to learn

The next part of the test output, which starts wi
of any documentation tests. We don't have any

compile any code examples that appear in our A
us keep our docs and our code in sync! We'll dis
tests in the "Documentation Comments" section
the `Doc-tests` output.

Let's change the name of our test to see how th
the `it_works` function to a different name, such

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

Then run `cargo test` again. The output now sh
`it_works`:

```
running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

Let's add another test, but this time we'll make a
something in the test function panics. Each test
main thread sees that a test thread has died, the
about the simplest way to cause a panic in Chap
macro. Enter the new test, `another`, so your *src*

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

Listing 11-3: Adding a second test that will fail be

Run the tests again using `cargo test`. The outp
shows that our `exploration` test passed and `a

```
running 2 tests
test tests::exploration ... ok
test tests::another ... FAILED

failures:

---- tests::another stdout ----
    thread 'tests::another' panicked at 'M
src/lib.rs:10:8
note: Run with `RUST_BACKTRACE=1` for a ba

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0
out

error: test failed
```

Listing 11-4: Test results when one test passes a

Instead of `ok`, the line `test tests::another` sh
appear between the individual results and the su
detailed reason for each test failure. In this case
`panicked at 'Make this test fail'`, which ha
The next section lists just the names of all the fa
are lots of tests and lots of detailed failing test o
failing test to run just that test to more easily de
run tests in the "Controlling How Tests Are Run"

The summary line displays at the end: overall, ou
test pass and one test fail.

Now that you've seen what the test results look
some macros other than `panic!` that are usefu

Checking Results with the `assert!` Ma

The `assert!` macro, provided by the standard l
ensure that some condition in a test evaluates to
an argument that evaluates to a Boolean. If the v
and the test passes. If the value is `false`, the `a
macro, which causes the test to fail. Using the `a

code is functioning in the way we intend.

In Chapter 5, Listing 5-15, we used a `Rectangle` which are repeated here in Listing 11-5. Let's pu write some tests for it using the `assert!` macro

Filename: src/lib.rs

```
#[derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    pub fn can_hold(&self, other: &Rectang
        self.length > other.length && self
    }
}
```

Listing 11-5: Using the `Rectangle` struct and its

The `can_hold` method returns a Boolean, which `assert!` macro. In Listing 11-6, we write a test t by creating a `Rectangle` instance that has a len asserting that it can hold another `Rectangle` ins width of 1:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle { length: 8
        let smaller = Rectangle { length:

        assert!(larger.can_hold(&smaller))
    }
}
```

Listing 11-6: A test for `can_hold` that checks wh hold a smaller rectangle

Note that we've added a new line inside the `tes` `tests` module is a regular module that follows

Chapter 7 in the “Privacy Rules” section. Becaus module, we need to bring the code under test ir the inner module. We use a glob here so anythir available to this `tests` module.

We’ve named our test `larger_can_hold_smalle` `Rectangle` instances that we need. Then we cal the result of calling `larger.can_hold(&smaller)` return `true`, so our test should pass. Let’s find

```
running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

It does pass! Let’s add another test, this time ass hold a larger rectangle:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle { length: 8
        let smaller = Rectangle { length:

        assert!(!smaller.can_hold(&larger)
    }
}
```

Because the correct result of the `can_hold` func negate that result before we pass it to the `asse` pass if `can_hold` returns `false`:

```
running 2 tests
test tests::smaller_cannot_hold_larger ...
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 igr
```

Two tests that pass! Now let's see what happens introduce a bug in our code. Let's change the im method by replacing the greater-than sign with lengths:

```
// --snip--

impl Rectangle {
    pub fn can_hold(&self, other: &Rectang
        self.length < other.length && sel
    }
}
```

Running the tests now produces the following:

```
running 2 tests
test tests::smaller_cannot_hold_larger ...
test tests::larger_can_hold_smaller ... F

failures:

---- tests::larger_can_hold_smaller stdout
    thread 'tests::larger_can_hold_smalle
    larger.can_hold(&smaller)', src/lib.r
note: Run with `RUST_BACKTRACE=1` for a b

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed;
out
```

Our tests caught the bug! Because `larger.leng` comparison of the lengths in `can_hold` now ret

Testing Equality with the `assert_eq!` a

A common way to test functionality is to compa the value you expect the code to return to make using the `assert!` macro and passing it an exp However, this is such a common test that the st macros— `assert_eq!` and `assert_ne!` —to per These macros compare two arguments for equa also print the two values if the assertion fails, w test failed; conversely, the `assert!` macro only the `==` expression, not the values that lead to t

In Listing 11-7, we write a function named add_t
returns the result. Then we test this function usi

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

Listing 11-7: Testing the function add_two using

Let's check that it passes!

```
running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

The first argument we gave to the assert_eq! r
calling add_two(2) . The line for this test is test
the ok text indicates that our test passed!

Let's introduce a bug into our code to see what i
assert_eq! fails. Change the implementation o
3 :

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

Run the tests again:

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
        thread 'tests::it_adds_two' panick
== right)`
  left: `4`,
 right: `5`', src/lib.rs:11:8
note: Run with `RUST_BACKTRACE=1` for a ba

failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0
out
```

Our test caught the bug! The `it_adds_two` test
`assertion failed: `(left == right)`` and sh
was `5`. This message is useful and helps us star
argument to `assert_eq!` was `4` but the `right`
`add_two(2)`, was `5`.

Note that in some languages and test framewor
that assert two values are equal are called `expe`
which we specify the arguments matters. Howe
`right`, and the order in which we specify the va
code under test produces doesn't matter. We co
`assert_eq!(add_two(2), 4)`, which would resu
`assertion failed: `(left == right)`` and th

The `assert_ne!` macro will pass if the two value
they're equal. This macro is most useful for case
will be, but we know what the value definitely *w*
intend. For example, if we're testing a function t
in some way, but the way in which the input is c
week that we run our tests, the best thing to ass
function is not equal to the input.

Under the surface, the `assert_eq!` and `assert`
and `!=`, respectively. When the assertions fail, t
using debug formatting, which means the values
the `PartialEq` and `Debug` traits. All the primiti
library types implement these traits. For structs
need to implement `PartialEq` to assert that va

equal. You'll need to implement `Debug` to print t
Because both traits are derivable traits, as ment
is usually as straightforward as adding the `#[de`
to your struct or enum definition. See Appendix
about these and other derivable traits.

Adding Custom Failure Messages

You can also add a custom message to be printe
optional arguments to the `assert!`, `assert_eq`
arguments specified after the one required argu
arguments to `assert_eq!` and `assert_ne!` are
(discussed in Chapter 8 in the "Concatenation wi
Macro" section), so you can pass a format string
values to go in those placeholders. Custom mes
assertion means; when a test fails, you'll have a
with the code.

For example, let's say we have a function that gr
test that the name we pass into the function app

Filename: src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

The requirements for this program haven't beer
sure the `Hello` text at the beginning of the gree
want to have to update the test when the requir
checking for exact equality to the value returnec
just assert that the output contains the text of th

Let's introduce a bug into this code by changing what this test failure looks like:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

Running this test produces the following:

```
running 1 test
test tests::greeting_contains_name ... FAI

failures:

---- tests::greeting_contains_name stdout
        thread 'tests::greeting_contains_n
failed:
result.contains("Carol")', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a ba

failures:
    tests::greeting_contains_name
```

This result just indicates that the assertion failec
more useful failure message in this case would p
`greeting` function. Let's change the test functio
made from a format string with a placeholder fil
from the `greeting` function:

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, va
    );
}
```

Now when we run the test, we'll get a more infor

```
---- tests::greeting_contains_name stdout
        thread 'tests::greeting_contains_n
not
contain name, value was `Hello!`', src/lib
note: Run with `RUST_BACKTRACE=1` for a ba
```

We can see the value we actually got in the test o
what happened instead of what we were expect

Checking for Panics with `should_panic`

In addition to checking that our code returns the important to check that our code handles error consider the `Guess` type that we created in Cha uses `Guess` depends on the guarantee that `Gue` between 1 and 100. We can write a test that ens `Guess` instance with a value outside that range

We do this by adding another attribute, `should_` attribute makes a test pass if the code inside the the code inside the function doesn't panic.

Listing 11-8 shows a test that checks that the err when we expect them to:

Filename: src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be be
value);
        }

        Guess {
            value
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Listing 11-8: Testing that a condition will cause a

We place the `#[should_panic]` attribute after tl test function it applies to. Let's look at the result

```
running 1 test
test tests::greater_than_100 ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

Looks good! Now let's introduce a bug in our cod
`new` function will panic if the value is greater tha

```
// --snip--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1  {
            panic!("Guess value must be be
value);
        }

        Guess {
            value
        }
    }
}
```

When we run the test in Listing 11-8, it will fail:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0
out
```

We don't get a very helpful message in this case,
function, we see that it's annotated with `#[shou`
that the code in the test function did not cause a

Tests that use `should_panic` can be imprecise b
code has caused some panic. A `should_panic` t
for a different reason than the one we were exp
`should_panic` tests more precise, we can add a
the `should_panic` attribute. The test harness w
contains the provided text. For example, conside

Listing 11-9 where the `new` function panics with whether the value is too small or too large:

Filename: src/lib.rs

```
// --snip--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value must be gr
{}.",
                   value);
        } else if value > 100 {
            panic!("Guess value must be le
{}.",
                   value);
        }

        Guess {
            value
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "Guess value
100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Listing 11-9: Testing that a condition will cause a message

This test will pass because the value we put in th `expected` parameter is a substring of the messa panics with. We could have specified the entire p in this case would be `Guess value must be les` What you choose to specify in the expected para how much of the panic message is unique or dy test to be. In this case, a substring of the panic m code in the test function executes the `else if`

To see what happens when a `should_panic` tes
let's again introduce a bug into our code by swa
and the `else if value > 100` blocks:

```
if value < 1 {
    panic!("Guess value must be less than
value);
} else if value > 100 {
    panic!("Guess value must be greater th
value);
}
```

This time when we run the `should_panic` test, i

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
        thread 'tests::greater_than_100' p
greater than or equal to 1, got 200.', src
note: Run with `RUST_BACKTRACE=1` for a ba
note: Panic did not include expected strin
or
equal to 100'

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0
out
```

The failure message indicates that this test did i
panic message did not include the expected stri
`'Guess value must be less than or equal to`
did get in this case was
`Guess value must be greater than or equal`
figuring out where our bug is!

Using `Result<T, E>` in tests

So far, we've written tests that panic when they
`Result<T, E>` too! Here's that first example, bu

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two
        }
    }
}
```

Here, we've changed the `it_works` function to r than `assert_eq!`, we return `Ok(())` for the suc inside for the failure case. As before, this test wi based on panics, it will use the `Result<T, E>` to this, you can't use `#[should_panic]` with one o be returning an `Err` instead!

Now that you know several ways to write tests, l we run our tests and explore the different optio

Controlling How Tests Are Run

Just as `cargo run` compiles your code and then `cargo test` compiles your code in test mode an can specify command line options to change the example, the default behavior of the binary proc tests in parallel and capture output generated d from being displayed and making it easier to rea results.

Some command line options go to `cargo test`, binary. To separate these two types of argumen `cargo test` followed by the separator `--` and t binary. Running `cargo test --help` displays th `cargo test`, and running `cargo test -- --hel after the separator `--`.

Running Tests in Parallel or Consecutiv

When you run multiple tests, by default they rur
the tests will finish running faster so you can get
your code is working. Because the tests are runr
tests don't depend on each other or on any shar
environment, such as the current working direct

For example, say each of your tests runs some c
test-output.txt and writes some data to that file. T
file and asserts that the file contains a particular
Because the tests run at the same time, one test
when another test writes and reads the file. The
the code is incorrect but because the tests have
running in parallel. One solution is to make sure
another solution is to run the tests one at a time

If you don't want to run the tests in parallel or if
over the number of threads used, you can send
number of threads you want to use to the test b
example:

```
$ cargo test -- --test-threads=1
```

We set the number of test threads to `1`, telling t
parallelism. Running the tests using one thread
parallel, but the tests won't interfere with each c

Showing Function Output

By default, if a test passes, Rust's test library cap
output. For example, if we call `println!` in a tes
the `println!` output in the terminal; we'll see o
passed. If a test fails, we'll see whatever was prir
of the failure message.

As an example, Listing 11-10 has a silly function
and returns 10, as well as a test that passes and

Filename: src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(
        assert_eq!(5, value);
    }
}
```

Listing 11-10: Tests for a function that calls `prir`

When we run these tests with `cargo test`, we'll

```
running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ---
        I got the value 8
thread 'tests::this_test_will_fail' panick
== right)`
  left: `5`,
 right: `10`', src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a ba

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0
out
```

Note that nowhere in this output do we see `I g` printed when the test that passes runs. That out from the test that failed, `I got the value 8`, ap summary output, which also shows the cause of

If we want to see printed values for passing test capture behavior by using the `--nocapture` flag

```
$ cargo test -- --nocapture
```

When we run the tests in Listing 11-10 again wit following output:

```
running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panick
== right)`
  left: `5`,
 right: `10`', src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a ba
test tests::this_test_will_fail ... FAILED

failures:

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0
out
```

Note that the output for the tests and the test re that the tests are running in parallel, as we talke using the `--test-threads=1` option and the `--` output looks like then!

Running a Subset of Tests by Name

Sometimes, running a full test suite can take a lo a particular area, you might want to run only the can choose which tests to run by passing `cargo` test(s) you want to run as an argument.

To demonstrate how to run a subset of tests, we function, as shown in Listing 11-11, and choose

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

Listing 11-11: Three tests with three different na

If we run the tests without passing any argumer
run in parallel:

```
running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 igr
```

Running Single Tests

We can pass the name of any test function to ca

```
$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/adder-06a75

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

Only the test with the name `one_hundred` ran; t
name. The test output lets us know we had mor
by displaying `2 filtered out` at the end of the

We can't specify the names of multiple tests in tl
`cargo test` will be used. But there is a way to r

Filtering to Run Multiple Tests

We can specify part of a test name, and any test
be run. For example, because two of our tests' n
two by running `cargo test add`:

```
$ cargo test add
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/adder-06a75

running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 igr
```

This command ran all tests with `add` in the nam
`one_hundred`. Also note that the module in whic
test's name, so we can run all the tests in a mod

Ignoring Some Tests Unless Specifically

Sometimes a few specific tests can be very time-
want to exclude them during most runs of `carg
arguments all tests you do want to run, you can
tests using the `ignore` attribute to exclude ther

Filename: src/lib.rs

```rust
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

After `#[test]` we add the `#[ignore]` line to th
we run our tests, `it_works` runs, but `expensive`

```
$ cargo test
   Compiling adder v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/adder-ce99b

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 igr
```

The `expensive_test` function is listed as `ignor`
tests, we can use `cargo test -- --ignored`:

```
$ cargo test -- --ignored
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/adder-ce99b

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

By controlling which tests run, you can make su
fast. When you're at a point where it makes sens
tests and you have time to wait for the results, y
`cargo test -- --ignored` instead.

Test Organization

As mentioned at the start of the chapter, testing

people use different terminology and organizati
tests in terms of two main categories: *unit tests a*
small and more focused, testing one module in i
private interfaces. Integration tests are entirely e
code in the same way any other external code w
and potentially exercising multiple modules per

Writing both kinds of tests is important to ensur
doing what you expect them to separately and t

Unit Tests

The purpose of unit tests is to test each unit of c
code to quickly pinpoint where code is and isn't
tests in the *src* directory in each file with the cod
is to create a module named `tests` in each file
annotate the module with `cfg(test)` .

The Tests Module and `#[cfg(test)]`

The `#[cfg(test)]` annotation on the tests mod
test code only when you run `cargo test` , not w
compile time when you only want to build the lib
compiled artifact because the tests are not inclu
integration tests go in a different directory, they
annotation. However, because unit tests go in th
`#[cfg(test)]` to specify that they shouldn't be

Recall that when we generated the new `adder` p
chapter, Cargo generated this code for us:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

This code is the automatically generated test mo
configuration and tells Rust that the following ite

certain configuration option. In this case, the co
provided by Rust for compiling and running test
compiles our test code only if we actively run th
any helper functions that might be within this m
annotated with `#[test]` .

Testing Private Functions

There's debate within the testing community ab
should be tested directly, and other languages n
private functions. Regardless of which testing id
rules do allow you to test private functions. Con
the private function `internal_adder` :

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2)
    }
}
```

Listing 11-12: Testing a private function

Note that the `internal_adder` function is not m
just Rust code and the `tests` module is just an
`internal_adder` in a test just fine. If you don't t
tested, there's nothing in Rust that will compel y

Integration Tests

In Rust, integration tests are entirely external to

the same way any other code would, which mea
are part of your library's public API. Their purpo
your library work together correctly. Units of co
could have problems when integrated, so test c
important as well. To create integration tests, yo

The *tests* Directory

We create a *tests* directory at the top level of our
knows to look for integration test files in this dir
test files as we want to in this directory, and Car
individual crate.

Let's create an integration test. With the code in
make a *tests* directory, create a new file named *t*
code in Listing 11-13:

Filename: tests/integration_test.rs

```
extern crate adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Listing 11-13: An integration test of a function in

We've added `extern crate adder` at the top of
unit tests. The reason is that each test in the `te`
we need to import our library into each of them

We don't need to annotate any code in *tests/inte*
Cargo treats the `tests` directory specially and c
when we run `cargo test`. Run `cargo test` no

```
$ cargo test
   Compiling adder v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/adder-abcab

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 igr

     Running target/debug/deps/integration

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 igr

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 igr
```

The three sections of output include the unit tes tests. The first section for the unit tests is the sa each unit test (one named `internal` that we ad summary line for the unit tests.

The integration tests section starts with the line `Running target/debug/deps/integration_test` end of your output will be different). Next, there integration test and a summary line for the resu the `Doc-tests adder` section starts.

Similarly to how adding more unit test functions tests section, adding more test functions to the i lines to this integration test file's section. Each ir so if we add more files in the *tests* directory, the sections.

We can still run a particular integration test func name as an argument to `cargo test`. To run all test file, use the `--test` argument of `cargo te

```
$ cargo test --test integration_test
    Finished dev [unoptimized + debuginfo]
     Running target/debug/integration_test

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

This command runs only the tests in the *tests/int*

Submodules in Integration Tests

As you add more integration tests, you might wa *tests* directory to help organize them; for examp the functionality they're testing. As mentioned e compiled as its own separate crate.

Treating each integration test file as its own crat that are more like the way end users will be usir files in the *tests* directory don't share the same b learned in Chapter 7 regarding how to separate

The different behavior of files in the *tests* directo a set of helper functions that would be useful in try to follow the steps in the "Moving Modules to extract them into a common module. For examp place a function named `setup` in it, we can add call from multiple test functions in multiple test

Filename: tests/common.rs

```rust
pub fn setup() {
    // setup code specific to your library
}
```

When we run the tests again, we'll see a new sec *common.rs* file, even though this file doesn't con the `setup` function from anywhere:

```
running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 igr

     Running target/debug/deps/common-b8b(

running 0 tests

test result: ok. 0 passed; 0 failed; 0 igr

     Running target/debug/deps/integratior

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 igr

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 igr
```

Having `common` appear in the test results with `r` what we wanted. We just wanted to share some files.

To avoid having `common` appear in the test outp *tests/common.rs*, we'll create *tests/common/mod.* Filesystems" section of Chapter 7, we used the n *module_name/mod.rs* for files of modules that ha submodules for `common` here, but naming the fi `common` module as an integration test file. Wher into *tests/common/mod.rs* and delete the *tests/cc* output will no longer appear. Files in subdirecto compiled as separate crates or have sections in

After we've created *tests/common/mod.rs*, we car test files as a module. Here's an example of calli `it_adds_two` test in *tests/integration_test.rs*:

Filename: tests/integration_test.rs

```
extern crate adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

Note that the `mod common;` declaration is the sa demonstrated in Listing 7-4. Then in the test fun `common::setup()` function.

Integration Tests for Binary Crates

If our project is a binary crate that only contains *src/lib.rs* file, we can't create integration tests in `extern crate` to import functions defined in th expose functions that other crates can call and u on their own.

This is one of the reasons Rust projects that pro *src/main.rs* file that calls logic that lives in the *src* integration tests *can* test the library crate by usi important functionality. If the important functio code in the *src/main.rs* file will work as well, and need to be tested.

Summary

Rust's testing features provide a way to specify h continues to work as you expect, even as you m different parts of a library separately and can te Integration tests check that many parts of the lib they use the library's public API to test the code use it. Even though Rust's type system and owne of bugs, tests are still important to reduce logic l is expected to behave.

Let's combine the knowledge you learned in this work on a project!

An I/O Project: Buildin
Line Program

This chapter is a recap of the many skills you've few more standard library features. We'll build a with file and command line input/output to prac now have under your belt.

Rust's speed, safety, single binary output, and cr language for creating command line tools, so fo version of the classic command line tool `grep` (g and **p**rint). In the simplest use case, `grep` searc string. To do so, `grep` takes as its arguments a f the file, finds lines in that file that contain the st

Along the way, we'll show how to make our com terminal that many command line tools use. We variable to allow the user to configure the beha standard error console stream (`stderr`) insteac example, the user can redirect successful outpu messages onscreen.

One Rust community member, Andrew Gallant, very fast version of `grep`, called `ripgrep`. By cc fairly simple, but this chapter will give you some need to understand a real-world project such as

Our `grep` project will combine a number of con

- Organizing code (using what you learned ir
- Using vectors and strings (collections, Chap
- Handling errors (Chapter 9)
- Using traits and lifetimes where appropria
- Writing tests (Chapter 11)

We'll also briefly introduce closures, iterators, ar and 17 will cover in detail.

Accepting Command Line Arg

Let's create a new project with, as always, `cargc` to distinguish it from the `grep` tool that you mig

```
$ cargo new minigrep
     Created binary (application) `minigre
$ cd minigrep
```

The first task is to make `minigrep` accept its tw
filename and a string to search for. That is, we w
with `cargo run`, a string to search for, and a pa

```
$ cargo run searchstring example-filename.
```

Right now, the program generated by `cargo ne`
it. Some existing libraries on Crates.io can help v
command line arguments, but because you're ju
implement this capability ourselves.

Reading the Argument Values

To enable `minigrep` to read the values of comm
we'll need a function provided in Rust's standard
This function returns an *iterator* of the comman
`minigrep`. We haven't discussed iterators yet (w
but for now, you only need to know two details a
series of values, and we can call the `collect` m
collection, such as a vector, containing all the ele

Use the code in Listing 12-1 to allow your `minig`
line arguments passed to it and then collect the

Filename: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().co
    println!("{:?}", args);
}
```

Listing 12-1: Collecting the command line argum

First, we bring the `std::env` module into scope
its `args` function. Notice that the `std::env::ar`
modules. As we discussed in Chapter 7, in cases

in more than one module, it's conventional to br
rather than the function. By doing so, we can ea
`std::env`. It's also less ambiguous than adding
the function with just `args`, because `args` migh
that's defined in the current module.

The `args` Function and Invalid Unico

Note that `std::env::args` will panic if any ar
your program needs to accept arguments con
`std::env::args_os` instead. That function re
`OsString` values instead of `String` values. W
`std::env::args` here for simplicity, because
platform and are more complex to work with

On the first line of `main`, we call `env::args`, and
turn the iterator into a vector containing all the v
can use the `collect` function to create many ki
annotate the type of `args` to specify that we wa
very rarely need to annotate types in Rust, `coll`
need to annotate because Rust isn't able to infer

Finally, we print the vector using the debug form
first with no arguments and then with two argum

```
$ cargo run
--snip--
["target/debug/minigrep"]

$ cargo run needle haystack
--snip--
["target/debug/minigrep", "needle", "hayst
```

Notice that the first value in the vector is `"targe`
name of our binary. This matches the behavior o
programs use the name by which they were invo
convenient to have access to the program name
messages or change behavior of the program ba
used to invoke the program. But for the purpose
save only the two arguments we need.

Saving the Argument Values in Variabl

Printing the value of the vector of arguments illu
access the values specified as command line arg
values of the two arguments in variables so we c
rest of the program. We do that in Listing 12-2:

Filename: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().cc

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

Listing 12-2: Creating variables to hold the query

As we saw when we printed the vector, the prog
the vector at `args[0]` , so we're starting at index
takes is the string we're searching for, so we put
the variable `query` . The second argument will b
to the second argument in the variable `filenam`

We temporarily print the values of these variabl
as we intend. Let's run this program again with t
:

```
$ cargo run test sample.txt
   Compiling minigrep v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep test s
Searching for test
In file sample.txt
```

Great, the program is working! The values of the
into the right variables. Later we'll add some err
potential erroneous situations, such as when the
now, we'll ignore that situation and work on add

Reading a File

Now we'll add functionality to read the file that i
line argument. First, we need a sample file to tes
make sure `minigrep` is working is one with a sm
with some repeated words. Listing 12-3 has an E
well! Create a file called *poem.txt* at the root leve
"I'm Nobody! Who are you?"

Filename: poem.txt

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Listing 12-3: A poem by Emily Dickinson makes a

With the text in place, edit *src/main.rs* and add c
Listing 12-4:

Filename: src/main.rs

```
use std::env;
use std::fs;

fn main() {
    // --snip--
    println!("In file {}", filename);

    let contents = fs::read_to_string(file
        .expect("Something went wrong read

    println!("With text:\n{}", contents);
}
```

Listing 12-4: Reading the contents of the file spe

First, we add another `use` statement to bring in
library: we need `std::fs` to handle files.

In `main`, we've added a new statement: `fs::rea`
open that file, and then produce a new `String`

After that line, we've again added a temporary p
value of contents after the file is read, so we ca
so far.

Let's run this code with any string as the first co
haven't implemented the searching part yet) and
argument:

```
$ cargo run the poem.txt
   Compiling minigrep v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep the po
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us — don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Great! The code read and then printed the conte
flaws. The main function has multiple responsib
and easier to maintain if each function is respon
problem is that we're not handling errors as wel
small, so these flaws aren't a big problem, but as
to fix them cleanly. It's good practice to begin re
program, because it's much easier to refactor sm
next.

Refactoring to Improve Modul
Handling

To improve our program, we'll fix four problems
structure and how it's handling potential errors.

First, our main function now performs two task
files. For such a small function, this isn't a major
grow our program inside main , the number of s
handles will increase. As a function gains respon

reason about, harder to test, and harder to char
It's best to separate functionality so each functic

This issue also ties into the second problem: altl
configuration variables to our program, variable
the program's logic. The longer `main` becomes,
into scope; the more variables we have in scope
the purpose of each. It's best to group the confi
to make their purpose clear.

The third problem is that we've used `expect` to
the file fails, but the error message just prints `s
can fail in a number of ways: for example, the fil
have permission to open it. Right now, regardles
`something went wrong` error message, which w

Fourth, we use `expect` repeatedly to handle dif
program without specifying enough arguments,
error from Rust that doesn't clearly explain the p
error-handling code were in one place so future
consult in the code if the error-handling logic ne
handling code in one place will also ensure that
meaningful to our end users.

Let's address these four problems by refactoring

Separation of Concerns for Binary Proj

The organizational problem of allocating respon
function is common to many binary projects. As
developed a process to use as a guideline for sp
binary program when `main` starts getting large.

- Split your program into a *main.rs* and a *lib.*
  lib.rs.
- As long as your command line parsing logi
- When the command line parsing logic start
  main.rs and move it to *lib.rs*.
- The responsibilities that remain in the `mai`
  be limited to the following:

- Calling the command line parsing log
- Setting up any other configuration
- Calling a `run` function in *lib.rs*
- Handling the error if `run` returns an

This pattern is about separating concerns: *main.* *lib.rs* handles all the logic of the task at hand. Be function directly, this structure lets you test all o into functions in *lib.rs*. The only code that remai verify its correctness by reading it. Let's rework

Extracting the Argument Parser

We'll extract the functionality for parsing argum to prepare for moving the command line parsin the new start of `main` that calls a new function *src/main.rs* for the moment.

Filename: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().co

    let (query, filename) = parse_config(&

    // --snip--
}

fn parse_config(args: &[String]) -> (&str,
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

Listing 12-5: Extracting a `parse_config` functior

We're still collecting the command line argumen assigning the argument value at index `1` to the value at index `2` to the variable `filename` withi whole vector to the `parse_config` function. The the logic that determines which argument goes values back to `main`. We still create the `query` a `main` no longer has the responsibility of determ arguments and variables correspond.

This rework may seem like overkill for our small
small, incremental steps. After making this chan
that the argument parsing still works. It's good t
identify the cause of problems when they occur.

Grouping Configuration Values

We can take another small step to improve the
moment, we're returning a tuple, but then we im
individual parts again. This is a sign that perhaps
yet.

Another indicator that shows there's room for in
`parse_config`, which implies that the two value
part of one configuration value. We're not curre
structure of the data other than by grouping the
the two values into one struct and give each of t
Doing so will make it easier for future maintaine
different values relate to each other and what th

> Note: Some people call this anti-pattern of us
> complex type would be more appropriate *pri*

Listing 12-6 shows the addition of a struct name
named `query` and `filename`. We've also chang
return an instance of the `Config` struct and upd
rather than having separate variables:

Filename: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().co

    let config = parse_config(&args);

    println!("Searching for {}", config.qu
    println!("In file {}", config.filename

    let contents = fs::read_to_string(conf
        .expect("Something went wrong read

    // --snip--
}

struct Config {
    query: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config
    let query = args[1].clone();
    let filename = args[2].clone();

    Config { query, filename }
}
```

Listing 12-6: Refactoring `parse_config` to return

The signature of `parse_config` now indicates th body of `parse_config`, where we used to return values in `args`, we now define `Config` to conta variable in `main` is the owner of the argument v `parse_config` function borrow them, which me rules if `Config` tried to take ownership of the va

We could manage the `String` data in a number though somewhat inefficient, route is to call the will make a full copy of the data for the `Config` time and memory than storing a reference to th data also makes our code very straightforward b lifetimes of the references; in this circumstance, simplicity is a worthwhile trade-off.

The Trade-Offs of Using `clone`

> There's a tendency among many Rustaceans t
> ownership problems because of its runtime c
> to use more efficient methods in this type of s
> copy a few strings to continue making progre
> copies only once and your filename and quer
> have a working program that's a bit inefficien
> on your first pass. As you become more expe
> start with the most efficient solution, but for r
> `clone`.

We've updated `main` so it places the instance of into a variable named `config`, and we updated separate `query` and `filename` variables so it no struct instead.

Now our code more clearly conveys that `query` their purpose is to configure how the program v values knows to find them in the `config` instan purpose.

Creating a Constructor for `Config`

So far, we've extracted the logic responsible for from `main` and placed it in the `parse_config` fu that the `query` and `filename` values were relat conveyed in our code. We then added a `Config` of `query` and `filename` and to be able to retur names from the `parse_config` function.

So now that the purpose of the `parse_config` f instance, we can change `parse_config` from a p `new` that is associated with the `Config` struct. M more idiomatic. We can create instances of type `String`, by calling `String::new`. Similarly, by ch function associated with `Config`, we'll be able to calling `Config::new`. Listing 12-7 shows the cha

Filename: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().co

    let config = Config::new(&args);

    // --snip--
}

// --snip--

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}
```

Listing 12-7: Changing `parse_config` into `Conf`

We've updated `main` where we were calling `par` `Config::new`. We've changed the name of `pars` an `impl` block, which associates the `new` functi code again to make sure it works.

Fixing the Error Handling

Now we'll work on fixing our error handling. Rec values in the `args` vector at index `1` or index `2` the vector contains fewer than three items. Try r arguments; it will look like this:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bo
but the index is 1', src/main.rs:29:21
note: Run with `RUST_BACKTRACE=1` for a ba
```

The line `index out of bounds: the len is 1 b` message intended for programmers. It won't he happened and what they should do instead. Let'

Improving the Error Message

In Listing 12-8, we add a check in the `new` functi
enough before accessing index `1` and `2`. If the
panics and displays a better error message than

Filename: src/main.rs

```
// --snip--
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
    // --snip--
```

Listing 12-8: Adding a check for the number of a

This code is similar to the `Guess::new` function
called `panic!` when the `value` argument was c
Instead of checking for a range of values here, w
is at least `3` and the rest of the function can ope
condition has been met. If `args` has fewer than
true, and we call the `panic!` macro to end the p

With these extra few lines of code in `new`, let's r
arguments again to see what the error looks like

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep`
thread 'main' panicked at 'not enough argu
note: Run with `RUST_BACKTRACE=1` for a ba
```

This output is better: we now have a reasonable
have extraneous information we don't want to g
technique we used in Listing 9-9 isn't the best to
appropriate for a programming problem rather
Chapter 9. Instead, we can use the other technic
9—returning a `Result` that indicates either succ

Returning a `Result` from `new` Instead of Call

We can instead return a `Result` value that will c
successful case and will describe the problem in
communicating to `main`, we can use the `Result`
Then we can change `main` to convert an `Err` va

our users without the surrounding text about `t`
that a call to `panic!` causes.

Listing 12-9 shows the changes we need to mak
and the body of the function needed to return a
until we update `main` as well, which we'll do in t

Filename: src/main.rs

```
impl Config {
    fn new(args: &[String]) -> Result<Con
        if args.len() < 3 {
            return Err("not enough argumen
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```

Listing 12-9: Returning a `Result` from `Config::`

Our `new` function now returns a `Result` with a
and a `&'static str` in the error case. Recall fro
Chapter 10 that `&'static str` is the type of stri
message type for now.

We've made two changes in the body of the `new`
when the user doesn't pass enough arguments,
we've wrapped the `Config` return value in an `O`
conform to its new type signature.

Returning an `Err` value from `Config::new` allo
`Result` value returned from the `new` function a
the error case.

Calling `Config::new` and Handling Errors

To handle the error case and print a user-friend
to handle the `Result` being returned by `Config`
We'll also take the responsibility of exiting the c
error code from `panic!` and implement it by ha
convention to signal to the process that called o
with an error state.

Filename: src/main.rs

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().co

    let config = Config::new(&args).unwrap
        println!("Problem parsing argument
        process::exit(1);
    });

    // --snip--
```

Listing 12-10: Exiting with an error code if creati

In this listing, we've used a method we haven't c which is defined on `Result<T, E>` by the stand allows us to define some custom, non- `panic!` e value, this method's behavior is similar to `unwra` wrapping. However, if the value is an `Err` value *closure*, which is an anonymous function we defi `unwrap_or_else`. We'll cover closures in more d need to know that `unwrap_or_else` will pass the case is the static string `not enough arguments` t closure in the argument `err` that appears betw closure can then use the `err` value when it runs

We've added a new `use` line to import `process` in the closure that will be run in the error case is value and then call `process::exit`. The `proces` program immediately and return the number th This is similar to the `panic!` -based handling we get all the extra output. Let's try it:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep`
Problem parsing arguments: not enough argu
```

Great! This output is much friendlier for our use

Extracting Logic from `main`

Now that we've finished refactoring the configur
program's logic. As we stated in "Separation of C
extract a function named `run` that will hold all t
function that isn't involved with setting up config
we're done, `main` will be concise and easy to ve
write tests for all the other logic.

Listing 12-11 shows the extracted `run` function.
incremental improvement of extracting the func
in *src/main.rs*.

Filename: src/main.rs

```
fn main() {
    // --snip--

    println!("Searching for {}", config.qu
    println!("In file {}", config.filename

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(conf
        .expect("something went wrong reac

    println!("With text:\n{}", contents);
}

// --snip--
```

Listing 12-11: Extracting a `run` function containi

The `run` function now contains all the remainin
reading the file. The `run` function takes the `Con

Returning Errors from the `run` Function

With the remaining program logic separated int
the error handling, as we did with `Config::new`
program to panic by calling `expect`, the `run` fu
when something goes wrong. This will let us furt
around handling errors in a user-friendly way. Li
need to make to the signature and body of `run`

Filename: src/main.rs

```
use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<d
    let contents = fs::read_to_string(conf

    println!("With text:\n{}", contents);

    Ok(())
}
```

Listing 12-12: Changing the `run` function to retu

We've made three significant changes here. First `run` function to `Result<(), Box<dyn Error>>`. unit type, `()`, and we keep that as the value ret

For the error type, we used the *trait object* `Box<d` `std::error::Error` into scope with a `use` state objects in Chapter 17. For now, just know that `B` will return a type that implements the `Error` tra particular type the return value will be. This give that may be of different types in different error it's short for "dynamic."

Second, we've removed the call to `expect` in fav Chapter 9. Rather than `panic!` on an error, `?` w current function for the caller to handle.

Third, the `run` function now returns an `Ok` valu the `run` function's success type as `()` in the sig wrap the unit type value in the `Ok` value. This `O` at first, but using `()` like this is the idiomatic wa for its side effects only; it doesn't return a value

When you run this code, it will compile but will d

```
warning: unused `std::result::Result` whic
  --> src/main.rs:18:5
   |
18 |     run(config);
   |     ^^^^^^^^^^^^
= note: #[warn(unused_must_use)] on by def
```

Rust tells us that our code ignored the `Result` v indicate that an error occurred. But we're not ch

was an error, and the compiler reminds us that
error handling code here! Let's rectify that probl

Handling Errors Returned from `run` in `main`

We'll check for errors and handle them using a t
`Config::new` in Listing 12-10, but with a slight d

Filename: src/main.rs

```
fn main() {
    // --snip--

    println!("Searching for {}", config.qu
    println!("In file {}", config.filename

    if let Err(e) = run(config) {
        println!("Application error: {}",

        process::exit(1);
    }
}
```

We use `if let` rather than `unwrap_or_else` to
value and call `process::exit(1)` if it does. The
that we want to `unwrap` in the same way that `C`
instance. Because `run` returns `()` in the succes
an error, so we don't need `unwrap_or_else` to r
would only be `()`.

The bodies of the `if let` and the `unwrap_or_e`
cases: we print the error and exit.

Splitting Code into a Library Crate

Our `minigrep` project is looking good so far! No
put some code into the *src/lib.rs* file so we can te
fewer responsibilities.

Let's move all the code that isn't the `main` functi

- The `run` function definition
- The relevant `use` statements
- The definition of `Config`

- The `Config::new` function definition

The contents of *src/lib.rs* should have the signatu
omitted the bodies of the functions for brevity).
modify *src/main.rs* in the listing after this one.

Filename: src/lib.rs

```
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub filename: String,
}

impl Config {
    pub fn new(args: &[String]) -> Result<
        // --snip--
    }
}

pub fn run(config: Config) -> Result<(), E
    // --snip--
}
```

Listing 12-13: Moving `Config` and `run` into *src/*

We've made liberal use of the `pub` keyword: on
method, and on the `run` function. We now have
that we can test!

Now we need to bring the code we moved to *src*
crate in *src/main.rs*, as shown in Listing 12-14:

Filename: src/main.rs

```
extern crate minigrep;

use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config)
        // --snip--
    }
}
```

Listing 12-14: Bringing the `minigrep` crate into t

To bring the library crate into the binary crate, w
we add a `use minigrep::Config` line to bring th
prefix the `run` function with our crate name. No
connected and should work. Run the program w
everything works correctly.

Whew! That was a lot of work, but we've set ours
Now it's much easier to handle errors, and we've
Almost all of our work will be done in *src/lib.rs* fr

Let's take advantage of this newfound modularit
have been difficult with the old code but is easy
tests!

Developing the Library's Funct Driven Development

Now that we've extracted the logic into *src/lib.rs*
error handling in *src/main.rs*, it's much easier to
of our code. We can call functions directly with v
values without having to call our binary from the
some tests for the functionality in the `Config::`

In this section, we'll add the searching logic to th
Test-driven development (TDD) process. This so
follows these steps:

1. Write a test that fails and run it to make su
2. Write or modify just enough code to make
3. Refactor the code you just added or chang to pass.
4. Repeat from step 1!

This process is just one of many ways to write so
design as well. Writing the test before you write
helps to maintain high test coverage throughout

We'll test drive the implementation of the functi
searching for the query string in the file content
match the query. We'll add this functionality in a

Writing a Failing Test

Because we don't need them anymore, let's rem *src/lib.rs* and *src/main.rs* that we used to check tl *src/lib.rs*, we'll add a `tests` module with a test fu test function specifies the behavior we want the query and the text to search for the query in, an text that contain the query. Listing 12-15 shows

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive.'
            search(query, contents)
        );
    }
}
```

Listing 12-15: Creating a failing test for the `sear`

This test searches for the string `"duct"`. The te› one of which contains `"duct"`. We assert that tl function contains only the line we expect.

We aren't able to run this test and watch it fail b the `search` function doesn't exist yet! So now w test to compile and run by adding a definition of returns an empty vector, as shown in Listing 12- fail because an empty vector doesn't match a ve `"safe, fast, productive."`

Filename: src/lib.rs

```
fn search<'a>(query: &str, contents: &'a s
    vec![]
}
```

Listing 12-16: Defining just enough of the `searc`

Notice that we need an explicit lifetime `'a` defir used with the `contents` argument and the retu lifetime parameters specify which argument life the return value. In this case, we indicate that th string slices that reference slices of the argumer argument `query`).

In other words, we tell Rust that the data return long as the data passed into the `search` functio important! The data referenced *by* a slice needs valid; if the compiler assumes we're making strir `contents` , it will do its safety checking incorrect

If we forget the lifetime annotations and try to c error:

```
error[E0106]: missing lifetime specifier
 --> src/lib.rs:5:51
  |
5 | fn search(query: &str, contents: &str)
  |
parameter
  |
  = help: this function's return type cont
  signature does not say whether it is bor
```

Rust can't possibly know which of the two argum Because `contents` is the argument that contain the parts of that text that match, we know `cont` connected to the return value using the lifetime

Other programming languages don't require you values in the signature. So although this might s time. You might want to compare this example v Lifetimes" section in Chapter 10.

Now let's run the test:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///proj
--warnings--
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/minigrep-ab

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
        thread 'tests::one_result' panicke
==
right)`
left: `["safe, fast, productive."]`,
right: `[]`)', src/lib.rs:48:8
note: Run with `RUST_BACKTRACE=1` for a ba


failures:
    tests::one_result

test result: FAILED. 0 passed; 1 failed; 0
out

error: test failed, to rerun pass '--lib'
```

Great, the test fails, exactly as we expected. Let's

Writing Code to Pass the Test

Currently, our test is failing because we always r
implement `search`, our program needs to follow

- Iterate through each line of the contents.
- Check whether the line contains our query
- If it does, add it to the list of values we're re
- If it doesn't, do nothing.
- Return the list of results that match.

Let's work through each step, starting with iterat

Iterating Through Lines with the `lines` Meth

Rust has a helpful method to handle line-by-line
named `lines`, that works as shown in Listing 12

Filename: src/lib.rs

```
fn search<'a>(query: &str, contents: &'a s
    for line in contents.lines() {
        // do something with line
    }
}
```

Listing 12-17: Iterating through each line in `cont`

The `lines` method returns an iterator. We'll tal
13, but recall that you saw this way of using an i
`for` loop with an iterator to run some code on e

Searching Each Line for the Query

Next, we'll check whether the current line contai
strings have a helpful method named `contains`
`contains` method in the `search` function, as sh
won't compile yet:

Filename: src/lib.rs

```
fn search<'a>(query: &str, contents: &'a s
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

Listing 12-18: Adding functionality to see whethe
`query`

Storing Matching Lines

We also need a way to store the lines that conta
make a mutable vector before the `for` loop and
`line` in the vector. After the `for` loop, we retur
12-19:

Filename: src/lib.rs

```
fn search<'a>(query: &str, contents: &'a s
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Listing 12-19: Storing the lines that match so we

Now the `search` function should return only th test should pass. Let's run the test:

```
$ cargo test
--snip--
running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

Our test passed, so we know it works!

At this point, we could consider opportunities fo the search function while keeping the tests pass functionality. The code in the search function isr advantage of some useful features of iterators. Chapter 13, where we'll explore iterators in deta

Using the `search` Function in the `run` Functi

Now that the `search` function is working and te our `run` function. We need to pass the `config.` `run` reads from the file to the `search` function. returned from `search`:

Filename: src/lib.rs

```
pub fn run(config: Config) -> Result<(), E
    let contents = fs::read_to_string(conf

    for line in search(&config.query, &con
        println!("{}", line);
    }

    Ok(())
}
```

We're still using a `for` loop to return each line f

Now the entire program should work! Let's try it
return exactly one line from the Emily Dickinson

```
$ cargo run frog poem.txt
   Compiling minigrep v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep frog p
How public, like a frog
```

Cool! Now let's try a word that will match multip

```
$ cargo run body poem.txt
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep body p
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

And finally, let's make sure that we don't get any
isn't anywhere in the poem, such as "monomorp

```
$ cargo run monomorphization poem.txt
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep monomo
```

Excellent! We've built our own mini version of a
how to structure applications. We've also learne
lifetimes, testing, and command line parsing.

To round out this project, we'll briefly demonstr
variables and how to print to standard error, bo
writing command line programs.

Working with Environment Va

We'll improve `minigrep` by adding an extra feat
searching that the user can turn on via an enviro
feature a command line option and require that
to apply, but instead we'll use an environment v
set the environment variable once and have all t
that terminal session.

Writing a Failing Test for the Case-Inse

We want to add a new `search_case_insensitiv`
environment variable is on. We'll continue to foll
is again to write a failing test. We'll add a new te
`search_case_insensitive` function and renam
`case_sensitive` to clarify the differences betwe
12-20:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive.'
            search(query, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query,
        );
    }
}
```

Listing 12-20: Adding a new failing test for the ca
add

Note that we've edited the old test's `contents` t
text `"Duct tape."` using a capital D that should
searching in a case-sensitive manner. Changing
that we don't accidentally break the case-sensiti
already implemented. This test should pass now
work on the case-insensitive search.

The new test for the case-*insensitive* search uses
`search_case_insensitive` function we're about
match the line containing `"Rust:"` with a capita

even though both have different casing than the
will fail to compile because we haven't yet define
function. Feel free to add a skeleton implementa
vector, similar to the way we did for the `search`
test compile and fail.

Implementing the `search_case_insens`

The `search_case_insensitive` function, shown
same as the `search` function. The only differen
and each `line` so whatever the case of the inpu
when we check whether the line contains the qu

Filename: src/lib.rs

```
fn search_case_insensitive<'a>(query: &str
str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&q
            results.push(line);
        }
    }

    results
}
```

Listing 12-21: Defining the `search_case_insens`
and the line before comparing them

First, we lowercase the `query` string and store i
same name. Calling `to_lowercase` on the query
the user's query is `"rust"`, `"RUST"`, `"Rust"`, or
were `"rust"` and be insensitive to the case.

Note that `query` is now a `String` rather than a
`to_lowercase` creates new data rather than ref
is `"rUsT"`, as an example: that string slice doesn
to use, so we have to allocate a new `String` con
`query` as an argument to the `contains` method
because the signature of `contains` is defined to

Next, we add a call to `to_lowercase` on each `l
contains `query` to lowercase all characters. Nov
`query` to lowercase, we'll find matches no matte

Let's see if this implementation passes the tests:

```
running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 igr
```

Great! They passed. Now, let's call the new `sear
the `run` function. First, we'll add a configuration
between case-sensitive and case-insensitive sea
compiler errors since we aren't initializing this fi

Filename: src/lib.rs

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

Note that we added the `case_sensitive` field tl
the `run` function to check the `case_sensitive`
whether to call the `search` function or the `sear
shown in Listing 12-22. Note this still won't com

Filename: src/lib.rs

```
pub fn run(config: Config) -> Result<(), E
    let contents = fs::read_to_string(conf

    let results = if config.case_sensitive
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.qu
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

Listing 12-22: Calling either `search` or `search_c` in `config.case_sensitive`

Finally, we need to check for the environment va environment variables are in the `env` module ir bring that module into scope with a `use std::e` we'll use the `var` method from the `env` module named `CASE_INSENSITIVE`, as shown in Listing 1

Filename: src/lib.rs

```
use std::env;

// --snip--

impl Config {
    pub fn new(args: &[String]) -> Result<
        if args.len() < 3 {
            return Err("not enough argumen
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CAS

        Ok(Config { query, filename, case_
    }
}
```

Listing 12-23: Checking for an environment varia

Here, we create a new variable `case_sensitive` `env::var` function and pass it the name of the variable. The `env::var` method returns a `Resul` variant that contains the value of the environme variable is set. It will return the `Err` variant if th

We're using the `is_err` method on the `Result` therefore unset, which means it *should* do a case `CASE_INSENSITIVE` environment variable is set t and the program will perform a case-insensitive of the environment variable, just whether it's set rather than using `unwrap`, `expect`, or any of the `Result`.

We pass the value in the `case_sensitive` variab function can read that value and decide whethe `search_case_insensitive`, as we implemented

Let's give it a try! First, we'll run our program wit and with the query `to`, which should match any lowercase:

```
$ cargo run to poem.txt
   Compiling minigrep v0.1.0 (file:///proj
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep to poe
Are you nobody, too?
How dreary to be somebody!
```

Looks like that still works! Now, let's run the prog `1` but with the same query `to`.

If you're using PowerShell, you will need to set th program in two commands rather than one:

```
$ $env:CASE_INSENSITIVE=1
$ cargo run to poem.txt
```

We should get lines that contain "to" that might

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/minigrep to poe
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

Excellent, we also got lines containing "To"! Our
insensitive searching controlled by an environm
manage options set using either command line

Some programs allow arguments *and* environm
configuration. In those cases, the programs deci
precedence. For another exercise on your own,
through either a command line argument or an
whether the command line argument or the env
precedence if the program is run with one set to
insensitive.

The `std::env` module contains many more use
environment variables: check out its documenta

Writing Error Messages to Sta
Standard Output

At the moment, we're writing all of our output to
function. Most terminals provide two kinds of ou
general information and *standard error* (`stderr`
enables users to choose to direct the successful
print error messages to the screen.

The `println!` function is only capable of printir
use something else to print to standard error.

Checking Where Errors Are Written

First, let's observe how the content printed by `m`
standard output, including any error messages w
instead. We'll do that by redirecting the standard
intentionally causing an error. We won't redirect

content sent to standard error will continue to d

Command line programs are expected to send e
stream so we can still see error messages on the
standard output stream to a file. Our program is
about to see that it saves the error message out

The way to demonstrate this behavior is by runr
filename, *output.txt*, that we want to redirect the
pass any arguments, which should cause an erro

```
$ cargo run > output.txt
```

The `>` syntax tells the shell to write the content
instead of the screen. We didn't see the error m
the screen, so that means it must have ended u
contains:

```
Problem parsing arguments: not enough argu
```

Yup, our error message is being printed to stand
error messages like this to be printed to standar
run ends up in the file. We'll change that.

Printing Errors to Standard Error

We'll use the code in Listing 12-24 to change how
Because of the refactoring we did earlier in this
messages is in one function, `main`. The standard
macro that prints to the standard error stream,
were calling `println!` to print errors to use `epr`

Filename: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().co

    let config = Config::new(&args).unwrap
        eprintln!("Problem parsing argumen
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config)
        eprintln!("Application error: {}",

        process::exit(1);
    }
}
```

Listing 12-24: Writing error messages to standar using `eprintln!`

After changing `println!` to `eprintln!`, let's ru without any arguments and redirecting standar

```
$ cargo run > output.txt
Problem parsing arguments: not enough argu
```

Now we see the error onscreen and *output.txt* co we expect of command line programs.

Let's run the program again with arguments tha standard output to a file, like so:

```
$ cargo run to poem.txt > output.txt
```

We won't see any output to the terminal, and *ou*

Filename: output.txt

```
Are you nobody, too?
How dreary to be somebody!
```

This demonstrates that we're now using standar standard error for error output as appropriate.

Summary

This chapter recapped some of the major conce

how to perform common I/O operations in Rust
files, environment variables, and the `eprintln!`
prepared to write command line applications. By
chapters, your code will be well organized, store
data structures, handle errors nicely, and be wel

Next, we'll explore some Rust features that were
closures and iterators.

Functional Language F
and Closures

Rust's design has taken inspiration from many e
one significant influence is *functional programmi*
often includes using functions as values by pass
them from other functions, assigning them to va
forth.

In this chapter, we won't debate the issue of wha
but will instead discuss some features of Rust th
languages often referred to as functional.

More specifically, we'll cover:

- *Closures*, a function-like construct you can s
- *Iterators*, a way of processing a series of ele
- How to use these two features to improve
- The performance of these two features (Sp
  might think!)

Other Rust features, such as pattern matching a
other chapters, are influenced by the functional
iterators is an important part of writing idiomati
entire chapter to them.

Closures: Anonymous Functio
Their Environment

Rust's closures are anonymous functions you ca
arguments to other functions. You can create th

the closure to evaluate it in a different context. U
values from the scope in which they're called. W
features allow for code reuse and behavior custo

Creating an Abstraction of Behavior w

Let's work on an example of a situation in which
executed later. Along the way, we'll talk about th
and traits.

Consider this hypothetical situation: we work at
generate custom exercise workout plans. The ba
algorithm that generates the workout plan takes
the app user's age, body mass index, exercise pr
intensity number they specify. The actual algorit
example; what's important is that this calculation
this algorithm only when we need to and only ca
wait more than necessary.

We'll simulate calling this hypothetical algorithm
`simulated_expensive_calculation` shown in Li
`calculating slowly...`, wait for two seconds, a
passed in:

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn simulated_expensive_calculation(intensi
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

Listing 13-1: A function to stand in for a hypothe
seconds to run

Next is the `main` function, which contains the pa
this example. This function represents the code
for a workout plan. Because the interaction with
the use of closures, we'll hardcode values repres
print the outputs.

The required inputs are these:

- An intensity number from the user, which i
workout to indicate whether they want a lo
intensity workout
- A random number that will generate some

The output will be the recommended workout p
function we'll use:

Filename: src/main.rs

```
fn main() {
    let simulated_user_specified_value = 1
    let simulated_random_number = 7;

    generate_workout(
        simulated_user_specified_value,
        simulated_random_number
    );
}
```

Listing 13-2: A `main` function with hardcoded va
random number generation

We've hardcoded the variable `simulated_user_`
variable `simulated_random_number` as 7 for sim
we'd get the intensity number from the app fror
generate a random number, as we did in the Gu
The `main` function calls a `generate_workout` fu
values.

Now that we have the context, let's get to the alg
`generate_workout` in Listing 13-3 contains the b
most concerned with in this example. The rest o
will be made to this function.

Filename: src/main.rs

```rust
fn generate_workout(intensity: u32, random
    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            simulated_expensive_calculatio
        );
        println!(
            "Next, do {} situps!",
            simulated_expensive_calculatio
        );
    } else {
        if random_number == 3 {
            println!("Take a break today!
        } else {
            println!(
                "Today, run for {} minutes
                simulated_expensive_calcul
            );
        }
    }
}
```

Listing 13-3: The business logic that prints the w calls to the `simulated_expensive_calculation`

The code in Listing 13-3 has multiple calls to the `if` block calls `simulated_expensive_calculati` `else` doesn't call it at all, and the code inside th

The desired behavior of the `generate_workout` user wants a low-intensity workout (indicated by intensity workout (a number of 25 or greater).

Low-intensity workout plans will recommend a r on the complex algorithm we're simulating.

If the user wants a high-intensity workout, there the random number generated by the app happ a break and hydration. If not, the user will get a on the complex algorithm.

This code works the way the business wants it to team decides that we need to make some chang `simulated_expensive_calculation` function in when those changes happen, we want to refacto `simulated_expensive_calculation` function or where we're currently unnecessarily calling the f

other calls to that function in the process. That i
isn't needed, and we still want to call it only once

Refactoring Using Functions

We could restructure the workout program in m
duplicated call to the `simulated_expensive_cal`
shown in Listing 13-4:

Filename: src/main.rs

```
fn generate_workout(intensity: u32, random
    let expensive_result =
        simulated_expensive_calculation(in

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result
        );
        println!(
            "Next, do {} situps!",
            expensive_result
        );
    } else {
        if random_number == 3 {
            println!("Take a break today!
        } else {
            println!(
                "Today, run for {} minutes
                expensive_result
            );
        }
    }
}
```

Listing 13-4: Extracting the calls to `simulated_ex`
and storing the result in the `expensive_result`

This change unifies all the calls to `simulated_ex`
problem of the first `if` block unnecessarily calli
we're now calling this function and waiting for th
the inner `if` block that doesn't use the result va

We want to define code in one place in our prog
we actually need the result. This is a use case fo

Refactoring with Closures to Store Code

Instead of always calling the `simulated_expensi` `if` blocks, we can define a closure and store th storing the result of the function call, as shown i the whole body of `simulated_expensive_calcu` introducing here:

Filename: src/main.rs

```
let expensive_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

Listing 13-5: Defining a closure and storing it in t

The closure definition comes after the `=` to assi `expensive_closure`. To define a closure, we sta inside which we specify the parameters to the cl because of its similarity to closure definitions in one parameter named `num`: if we had more tha them with commas, like `|param1, param2|`.

After the parameters, we place curly brackets th —these are optional if the closure body is a sing after the curly brackets, needs a semicolon to cc returned from the last line in the closure body (the closure when it's called, because that line dc function bodies.

Note that this `let` statement means `expensive` anonymous function, not the *resulting value* of ca that we're using a closure because we want to d store that code, and call it at a later point; the cc `expensive_closure`.

With the closure defined, we can change the coc to execute the code and get the resulting value. function: we specify the variable name that hold with parentheses containing the argument value 13-6:

Filename: src/main.rs

```
fn generate_workout(intensity: u32, random
    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today!
        } else {
            println!(
                "Today, run for {} minutes
                expensive_closure(intensit
            );
        }
    }
}
```

Listing 13-6: Calling the `expensive_closure` we'

Now the expensive calculation is called in only o
that code where we need the results.

However, we've reintroduced one of the probler
the closure twice in the first `if` block, which wil
make the user wait twice as long as they need to
creating a variable local to that `if` block to holc
closures provide us with another solution. We'll
first let's talk about why there aren't type annota
traits involved with closures.

Closure Type Inference and Annotatio

Closures don't require you to annotate the types
value like `fn` functions do. Type annotations are
they're part of an explicit interface exposed to y

rigidly is important for ensuring that everyone a function uses and returns. But closures aren't us they're stored in variables and used without nar users of our library.

Closures are usually short and relevant only witl any arbitrary scenario. Within these limited cont infer the types of the parameters and the returr the types of most variables.

Making programmers annotate the types in thes be annoying and largely redundant with the info available.

As with variables, we can add type annotations i clarity at the cost of being more verbose than is types for the closure we defined in Listing 13-5 v Listing 13-7:

Filename: src/main.rs

```
let expensive_closure = |num: u32| -> u32
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

Listing 13-7: Adding optional type annotations o types in the closure

With type annotations added, the syntax of clos of functions. The following is a vertical comparis function that adds 1 to its parameter and a closu added some spaces to line up the relevant parts similar to function syntax except for the use of p optional:

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 }
let add_one_v3 = |x|             { x + 1 }
let add_one_v4 = |x|               x + 1
```

The first line shows a function definition, and th closure definition. The third line removes the typ definition, and the fourth line removes the bracl

closure body has only one expression. These are
the same behavior when they're called.

Closure definitions will have one concrete type i
and for their return value. For instance, Listing 1
closure that just returns the value it receives as
useful except for the purposes of this example.
annotations to the definition: if we then try to ca
as an argument the first time and a `u32` the sec

Filename: src/main.rs

```
let example_closure = |x| x;

let s = example_closure(String::from("hell
let n = example_closure(5);
```

Listing 13-8: Attempting to call a closure whose t
types

The compiler gives us this error:

```
error[E0308]: mismatched types
 --> src/main.rs
  |
  | let n = example_closure(5);
  |                         ^ expected str
integral variable
  |
  = note: expected type `std::string::Str
             found type `{integer}`
```

The first time we call `example_closure` with the
the type of `x` and the return type of the closure
locked in to the closure in `example_closure`, an
different type with the same closure.

Storing Closures Using Generic Parame

Let's return to our workout generation app. In Li
the expensive calculation closure more times th
this issue is to save the result of the expensive c
the variable in each place we need the result, in
However, this method could result in a lot of rep

Fortunately, another solution is available to us. \
the closure and the resulting value of calling the
closure only if we need the resulting value, and i
rest of our code doesn't have to be responsible
may know this pattern as *memoization* or *lazy ev*

To make a struct that holds a closure, we need t
because a struct definition needs to know the ty
instance has its own unique anonymous type: th
same signature, their types are still considered c
function parameters that use closures, we use g
discussed in Chapter 10.

The `Fn` traits are provided by the standard libra
one of the traits: `Fn`, `FnMut`, or `FnOnce`. We'll d
traits in the "Capturing the Environment with Clo
can use the `Fn` trait.

We add types to the `Fn` trait bound to represen
return values the closures must have to match t
closure has a parameter of type `u32` and return
specify is `Fn(u32) -> u32`.

Listing 13-9 shows the definition of the `Cacher`
optional result value:

Filename: src/main.rs

```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```

Listing 13-9: Defining a `Cacher` struct that holds
optional result in `value`

The `Cacher` struct has a `calculation` field of th
on `T` specify that it's a closure by using the `Fn`
the `calculation` field must have one `u32` para
parentheses after `Fn`) and must return a `u32` (s

Note: Functions can implement all three of th do doesn't require capturing a value from the function rather than a closure where we neec `Fn` trait.

The `value` field is of type `Option<u32>` . Before `None` . When code using a `Cacher` asks for the *r* execute the closure at that time and store the re `value` field. Then if the code asks for the result executing the closure again, the `Cacher` will retu variant.

The logic around the `value` field we've just desc

Filename: src/main.rs

```
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)
                self.value = Some(v);
                v
            },
        }
    }
}
```

Listing 13-10: The caching logic of `Cacher`

We want `Cacher` to manage the struct fields' va code potentially change the values in these field

The `Cacher::new` function takes a generic parar having the same trait bound as the `Cacher` stru `Cacher` instance that holds the closure specifie `None` value in the `value` field, because we have

When the calling code needs the result of evalua
closure directly, it will call the `value` method. Th
already have a resulting value in `self.value` in
within the `Some` without executing the closure a

If `self.value` is `None`, the code calls the closur
the result in `self.value` for future use, and ret

Listing 13-11 shows how we can use this `Cacher`
`generate_workout` from Listing 13-6:

Filename: src/main.rs

```
fn generate_workout(intensity: u32, random
    let mut expensive_result = Cacher::new
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(
        num
    });

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensi
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensi
        );
    } else {
        if random_number == 3 {
            println!("Take a break today!
        } else {
            println!(
                "Today, run for {} minutes
                expensive_result.value(int
            );
        }
    }
}
```

Listing 13-11: Using `Cacher` in the `generate_wo`
caching logic

Instead of saving the closure in a variable directl
that holds the closure. Then, in each place we w
method on the `Cacher` instance. We can call the
want, or not call it at all, and the expensive calcu

Try running this program with the `main` functio
in the `simulated_user_specified_value` and `s
verify that in all the cases in the various `if` and
`calculating slowly...` appears only once and
takes care of the logic necessary to ensure we a
more than we need to so `generate_workout` ca

Limitations of the `Cacher` Implementa

Caching values is a generally useful behavior tha
of our code with different closures. However, th
implementation of `Cacher` that would make re

The first problem is that a `Cacher` instance assu
for the parameter `arg` to the `value` method. T

```
#[test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);

    assert_eq!(v2, 2);
}
```

This test creates a new `Cacher` instance with a c
into it. We call the `value` method on this `Cache
and then an `arg` value of 2, and we expect the c
should return 2.

Run this test with the `Cacher` implementation i
the test will fail on the `assert_eq!` with this me

```
thread 'call_with_different_values' panick
== right)`
  left: `1`,
 right: `2`', src/main.rs
```

The problem is that the first time we called `c.va
saved `Some(1)` in `self.value`. Thereafter, no n
method, it will always return 1.

Try modifying `Cacher` to hold a hash map rathe

hash map will be the `arg` values that are passe
will be the result of calling the closure on that ke
`self.value` directly has a `Some` or a `None` valu
`arg` in the hash map and return the value if it's
`Cacher` will call the closure and save the resulti
with its `arg` value.

The second problem with the current `Cacher` in
closures that take one parameter of type `u32` a
cache the results of closures that take a string sl
example. To fix this issue, try introducing more g
flexibility of the `Cacher` functionality.

Capturing the Environment with Closu

In the workout generator example, we only used
functions. However, closures have an additional
they can capture their environment and access v
they're defined.

Listing 13-12 has an example of a closure stored
the `x` variable from the closure's surrounding e

Filename: src/main.rs

```
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;

    let y = 4;

    assert!(equal_to_x(y));
}
```

Listing 13-12: Example of a closure that refers to

Here, even though `x` is not one of the paramete
closure is allowed to use the `x` variable that's de
`equal_to_x` is defined in.

We can't do the same with functions; if we try wi
won't compile:

Filename: src/main.rs

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x

    let y = 4;

    assert!(equal_to_x(y));
}
```

We get an error:

```
error[E0434]: can't capture dynamic enviro
...
} closure form instead
 --> src/main.rs
  |
4 |     fn equal_to_x(z: i32) -> bool { z
  |
```

The compiler even reminds us that this only wor

When a closure captures a value from its enviro values for use in the closure body. This use of m to pay in more common cases where we want to environment. Because functions are never allow defining and using functions will never incur this

Closures can capture values from their environm to the three ways a function can take a paramet mutably, and borrowing immutably. These are e follows:

- `FnOnce` consumes the variables it capture the closure's *environment*. To consume the take ownership of these variables and mov defined. The `Once` part of the name repre take ownership of the same variables more once.
- `FnMut` can change the environment becau
- `Fn` borrows values from the environment

When you create a closure, Rust infers which tra uses the values from the environment. All closu can all be called at least once. Closures that don implement `FnMut`, and closures that don't need variables also implement `Fn`. In Listing 13-12, th

immutably (so `equal_to_x` has the `Fn` trait) be
needs to read the value in `x` .

If you want to force the closure to take ownersh
environment, you can use the `move` keyword be
technique is mostly useful when passing a closu
so it's owned by the new thread.

We'll have more examples of `move` closures in C
concurrency. For now, here's the code from Listi
added to the closure definition and using vector
integers can be copied rather than moved; note

Filename: src/main.rs

```
fn main() {
    let x = vec![1, 2, 3];

    let equal_to_x = move |z| z == x;

    println!("can't use x here: {:?}", x);

    let y = vec![1, 2, 3];

    assert!(equal_to_x(y));
}
```

We receive the following error:

```
error[E0382]: use of moved value: `x`
 --> src/main.rs:6:40
  |
4 |     let equal_to_x = move |z| z == x;
  |                      -------- value mo
5 |
6 |     println!("can't use x here: {:?}",
  |
  |
  = note: move occurs because `x` has type
does not
  implement the `Copy` trait
```

The `x` value is moved into the closure when the
the `move` keyword. The closure then has owner
use `x` anymore in the `println!` statement. Re
example.

Most of the time when specifying one of the `Fn`

and the compiler will tell you if you need `FnMut`
the closure body.

To illustrate situations where closures that can c
as function parameters, let's move on to our nex

Processing a Series of Items w

The iterator pattern allows you to perform some
An iterator is responsible for the logic of iteratin
when the sequence has finished. When you use
reimplement that logic yourself.

In Rust, iterators are *lazy*, meaning they have no
consume the iterator to use it up. For example, t
iterator over the items in the vector `v1` by callin
This code by itself doesn't do anything useful.

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
```

Listing 13-13: Creating an iterator

Once we've created an iterator, we can use it in a
Chapter 3, we used iterators with `for` loops to e
although we glossed over what the call to `iter`

The example in Listing 13-14 separates the creat
iterator in the `for` loop. The iterator is stored in
iteration takes place at that time. When the `for`
`v1_iter`, each element in the iterator is used in
prints out each value.

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

Listing 13-14: Using an iterator in a `for` loop

In languages that don't have iterators provided b
likely write this same functionality by starting a v
to index into the vector to get a value, and incre
until it reached the total number of items in the

Iterators handle all that logic for you, cutting dov
potentially mess up. Iterators give you more flex
many different kinds of sequences, not just data
vectors. Let's examine how iterators do that.

The `Iterator` Trait and the `next` Meth

All iterators implement a trait named `Iterator`
library. The definition of the trait looks like this:

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Ite

    // methods with default implementatio
}
```

Notice this definition uses some new syntax: `ty`
defining an *associated type* with this trait. We'll ta
Chapter 19. For now, all you need to know is tha
`Iterator` trait requires that you also define an
used in the return type of the `next` method. In
the type returned from the iterator.

The `Iterator` trait only requires implementors
method, which returns one item of the iterator a
iteration is over, returns `None` .

We can call the `next` method on iterators direct
values are returned from repeated calls to `next`
vector:

Filename: src/lib.rs

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

Listing 13-15: Calling the `next` method on an ite

Note that we needed to make `v1_iter` mutable
iterator changes internal state that the iterator u
sequence. In other words, this code *consumes*, o
`next` eats up an item from the iterator. We didr
when we used a `for` loop because the loop too
mutable behind the scenes.

Also note that the values we get from the calls to
the values in the vector. The `iter` method prod
references. If we want to create an iterator that
owned values, we can call `into_iter` instead of
over mutable references, we can call `iter_mut`

Methods that Consume the Iterator

The `Iterator` trait has a number of different m
provided by the standard library; you can find ou
the standard library API documentation for the
methods call the `next` method in their definitio
implement the `next` method when implementi

Methods that call `next` are called *consuming ad*
the iterator. One example is the `sum` method, w
and iterates through the items by repeatedly cal
iterator. As it iterates through, it adds each item
total when iteration is complete. Listing 13-16 ha
method:

Filename: src/lib.rs

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

Listing 13-16: Calling the `sum` method to get the

We aren't allowed to use `v1_iter` after the call
of the iterator we call it on.

Methods that Produce Other Iterators

Other methods defined on the `Iterator` trait, k
to change iterators into different kinds of iterato
iterator adaptors to perform complex actions in
iterators are lazy, you have to call one of the cor
results from calls to iterator adaptors.

Listing 13-17 shows an example of calling the ite
takes a closure to call on each item to produce a
creates a new iterator in which each item from t
However, this code produces a warning:

Filename: src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

Listing 13-17: Calling the iterator adaptor `map` to

The warning we get is this:

```
warning: unused `std::iter::Map` which mus
lazy
and do nothing unless consumed
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(unused_must_use)] on by d
```

The code in Listing 13-17 doesn't do anything; th
called. The warning reminds us why: iterator ada
consume the iterator here.

To fix this and consume the iterator, we'll use th
Chapter 12 with `env::args` in Listing 12-1. This
collects the resulting values into a collection dat

In Listing 13-18, we collect the results of iterating
from the call to `map` into a vector. This vector wi
the original vector incremented by 1.

Filename: src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).

assert_eq!(v2, vec![2, 3, 4]);
```

Listing 13-18: Calling the `map` method to create
`collect` method to consume the new iterator a

Because `map` takes a closure, we can specify any
each item. This is a great example of how closur
while reusing the iteration behavior that the `Ite

Using Closures that Capture Their Envi

Now that we've introduced iterators, we can der
that capture their environment by using the `fil
method on an iterator takes a closure that takes
returns a Boolean. If the closure returns `true`, t
iterator produced by `filter`. If the closure retu

included in the resulting iterator.

In Listing 13-19, we use `filter` with a closure tl
from its environment to iterate over a collection
return only shoes that are the specified size.

Filename: src/lib.rs

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_my_size(shoes: Vec<Shoe>, shoe
    shoes.into_iter()
        .filter(|s| s.size == shoe_size)
        .collect()
}

#[test]
fn filters_by_size() {
    let shoes = vec![
        Shoe { size: 10, style: String::fr
        Shoe { size: 13, style: String::fr
        Shoe { size: 10, style: String::fr
    ];

    let in_my_size = shoes_in_my_size(shoe

    assert_eq!(
        in_my_size,
        vec![
            Shoe { size: 10, style: String
            Shoe { size: 10, style: String
        ]
    );
}
```

Listing 13-19: Using the `filter` method with a c

The `shoes_in_my_size` function takes ownershi
as parameters. It returns a vector containing on

In the body of `shoes_in_my_size`, we call `into_`
ownership of the vector. Then we call `filter` to
iterator that only contains elements for which th

The closure captures the `shoe_size` parameter the value with each shoe's size, keeping only sho `collect` gathers the values returned by the ada returned by the function.

The test shows that when we call `shoes_in_my_` have the same size as the value we specified.

Creating Our Own Iterators with the `I`

We've shown that you can create an iterator by `iter_mut` on a vector. You can create iterators f standard library, such as hash map. You can also want by implementing the `Iterator` trait on yo mentioned, the only method you're required to method. Once you've done that, you can use all implementations provided by the `Iterator` trai

To demonstrate, let's create an iterator that will create a struct to hold some values. Then we'll m implementing the `Iterator` trait and using the

Listing 13-20 has the definition of the `Counter` s to create instances of `Counter`:

Filename: src/lib.rs

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

Listing 13-20: Defining the `Counter` struct and a of `Counter` with an initial value of 0 for `count`

The `Counter` struct has one field named `count` keep track of where we are in the process of iter private because we want the implementation of `new` function enforces the behavior of always st in the `count` field.

Next, we'll implement the `Iterator` trait for our
of the `next` method to specify what we want to
shown in Listing 13-21:

Filename: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Ite
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

Listing 13-21: Implementing the `Iterator` trait

We set the associated `Item` type for our iterato
return `u32` values. Again, don't worry about ass
Chapter 19.

We want our iterator to add 1 to the current sta
would return 1 first. If the value of `count` is less
value wrapped in `Some`, but if `count` is 6 or high

Using Our `Counter` Iterator's `next` Method

Once we've implemented the `Iterator` trait, we
a test demonstrating that we can use the iterato
by calling the `next` method on it directly, just as
a vector in Listing 13-15.

Filename: src/lib.rs

```
#[test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

Listing 13-22: Testing the functionality of the `ne`

This test creates a new `Counter` instance in the `next` repeatedly, verifying that we have implem iterator to have: returning the values from 1 to 5

Using Other `Iterator` Trait Methods

We implemented the `Iterator` trait by defining use any `Iterator` trait method's default implen library, because they all use the `next` method's

For example, if for some reason we wanted to ta instance of `Counter`, pair them with values proc after skipping the first value, multiply each pair t are divisible by 3, and add all the resulting value in the test in Listing 13-23:

Filename: src/lib.rs

```
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Cour
                                 .map(|(a,
                                 .filter(|
                                 .sum();
    assert_eq!(18, sum);
}
```

Listing 13-23: Using a variety of `Iterator` trait r

Note that `zip` produces only four pairs; the the produced because `zip` returns `None` when eith

All of these method calls are possible because w
works, and the standard library provides default
that call `next`.

Improving Our I/O Project

With this new knowledge about iterators, we car
12 by using iterators to make places in the code
at how iterators can improve our implementatic
the `search` function.

Removing a `clone` Using an Iterator

In Listing 12-6, we added code that took a slice c
instance of the `Config` struct by indexing into tl
allowing the `Config` struct to own those values.
the implementation of the `Config::new` functio

Filename: src/lib.rs

```
impl Config {
    pub fn new(args: &[String]) -> Result<
        if args.len() < 3 {
            return Err("not enough argumen
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CAS

        Ok(Config { query, filename, case_
    }
}
```

Listing 13-24: Reproduction of the `Config::new`

At the time, we said not to worry about the ineff
remove them in the future. Well, that time is nov

We needed `clone` here because we have a slice
parameter `args`, but the `new` function doesn't
`Config` instance, we had to clone the values fro

`Config` so the `Config` instance can own its valu

With our new knowledge about iterators, we car ownership of an iterator as its argument insteac iterator functionality instead of the code that ch indexes into specific locations. This will clarify wl because the iterator will access the values.

Once `Config::new` takes ownership of the itera operations that borrow, we can move the `Strir` `Config` rather than calling `clone` and making a

Using the Returned Iterator Directly

Open your I/O project's *src/main.rs* file, which sh

Filename: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().cc

    let config = Config::new(&args).unwrap
        eprintln!("Problem parsing argumer
        process::exit(1);
    });

    // --snip--
}
```

We'll change the start of the `main` function that in Listing 13-25. This won't compile until we upd

Filename: src/main.rs

```
fn main() {
    let config = Config::new(env::args()).
        eprintln!("Problem parsing argumer
        process::exit(1);
    });

    // --snip--
}
```

Listing 13-25: Passing the return value of `env::a`

The `env::args` function returns an iterator! Rat into a vector and then passing a slice to `Config:`

of the iterator returned from `env::args` to `Con

Next, we need to update the definition of `Confi
file, let's change the signature of `Config::new` t
won't compile because we need to update the fu

Filename: src/lib.rs

```
impl Config {
    pub fn new(mut args: std::env::Args) -
        // --snip--
```

Listing 13-26: Updating the signature of `Config:

The standard library documentation for the `env
of the iterator it returns is `std::env::Args`. We'
`Config::new` function so the parameter `args` h
of `&[String]`. Because we're taking ownership
by iterating over it, we can add the `mut` keywor
parameter to make it mutable.

Using `Iterator` Trait Methods Instead of Ind

Next, we'll fix the body of `Config::new`. The sta
mentions that `std::env::Args` implements the
call the `next` method on it! Listing 13-27 update
the `next` method:

Filename: src/lib.rs

```rust
impl Config {
    pub fn new(mut args: std::env::Args) -
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get
        };

        let filename = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get
        };

        let case_sensitive = env::var("CAS

        Ok(Config { query, filename, case_
    }
}
```

Listing 13-27: Changing the body of `Config::ne`

Remember that the first value in the return valu program. We want to ignore that and get to the do nothing with the return value. Second, we ca put in the `query` field of `Config`. If `next` returr the value. If it returns `None`, it means not enoug return early with an `Err` value. We do the same

Making Code Clearer with Iterator Ada

We can also take advantage of iterators in the `s` which is reproduced here in Listing 13-28 as it w

Filename: src/lib.rs

```rust
pub fn search<'a>(query: &str, contents: &
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Listing 13-28: The implementation of the `searc

We can write this code in a more concise way us
so also lets us avoid having a mutable intermedi
programming style prefers to minimize the amo
clearer. Removing the mutable state might enab
searching happen in parallel, because we would
to the `results` vector. Listing 13-29 shows this

Filename: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &
    contents.lines()
        .filter(|line| line.contains(query
        .collect()
}
```

Listing 13-29: Using iterator adaptor methods in
function

Recall that the purpose of the `search` function i
contain the `query`. Similar to the `filter` exam
`filter` adaptor to keep only the lines that `line`
We then collect the matching lines into another
Feel free to make the same change to use iterat
`search_case_insensitive` function as well.

The next logical question is which style you shou
the original implementation in Listing 13-28 or t
13-29. Most Rust programmers prefer to use the
the hang of at first, but once you get a feel for th
they do, iterators can be easier to understand. I
of looping and building new vectors, the code fo
the loop. This abstracts away some of the comm
concepts that are unique to this code, such as th
the iterator must pass.

But are the two implementations truly equivaler
that the more low-level loop will be faster. Let's

Comparing Performance: Loop

To determine whether to use loops or iterators,

our `search` functions is faster: the version with
with iterators.

We ran a benchmark by loading the entire conte
Holmes by Sir Arthur Conan Doyle into a `String`
contents. Here are the results of the benchmark
`for` loop and the version using iterators:

```
test bench_search_for  ... bench:  19,620,
test bench_search_iter ... bench:  19,234,
```

The iterator version was slightly faster! We won'
because the point is not to prove that the two ve
general sense of how these two implementation

For a more comprehensive benchmark, you sho
various sizes as the `contents`, different words a
`query`, and all kinds of other variations. The poi
level abstraction, get compiled down to roughly
lower-level code yourself. Iterators are one of Ru
we mean using the abstraction imposes no addi
analogous to how Bjarne Stroustrup, the origina
defines *zero-overhead* in "Foundations of C++" (2

> In general, C++ implementations obey the zer
> don't use, you don't pay for. And further: Wha
> code any better.

As another example, the following code is taken
algorithm uses the linear prediction mathematic
based on a linear function of the previous samp
to do some math on three variables in scope: a
`coefficients`, and an amount by which to shift
the variables within this example but not given t
doesn't have much meaning outside of its conte
example of how Rust translates high-level ideas

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
                                 .zip(&buf
                                 .map(|(&c
                                 .sum::<i6
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

To calculate the value of `prediction`, this code in `coefficients` and uses the `zip` method to p previous 12 values in `buffer`. Then, for each pa sum all the results, and shift the bits in the sum

Calculations in applications like audio decoders highly. Here, we're creating an iterator, using tw value. What assembly code would this Rust code compiles down to the same assembly you'd writ corresponding to the iteration over the values ir there are 12 iterations, so it "unrolls" the loop. *U* removes the overhead of the loop controlling cc code for each iteration of the loop.

All of the coefficients get stored in registers, whi fast. There are no bounds checks on the array a optimizations that Rust is able to apply make the Now that you know this, you can use iterators ar code seem like it's higher level but don't impose doing so.

Summary

Closures and iterators are Rust features inspired language ideas. They contribute to Rust's capabi at low-level performance. The implementations runtime performance is not affected. This is par zero-cost abstractions.

Now that we've improved the expressiveness of more features of `cargo` that will help us share t

More About Cargo and

So far we've used only the most basic features o code, but it can do a lot more. In this chapter, w advanced features to show you how to do the fc

- Customize your build through release prof
- Publish libraries on crates.io
- Organize large projects with workspaces
- Install binaries from crates.io
- Extend Cargo using custom commands

Cargo can do even more than what we cover in of all its features, see its documentation.

Customizing Builds with Relea

In Rust, *release profiles* are predefined and custo configurations that allow a programmer to have for compiling code. Each profile is configured in

Cargo has two main profiles: the `dev` profile Ca and the `release` profile Cargo uses when you r profile is defined with good defaults for develop good defaults for release builds.

These profile names might be familiar from the

```
$ cargo build
    Finished dev [unoptimized + debuginfo]
$ cargo build --release
    Finished release [optimized] target(s)
```

The `dev` and `release` shown in this build outpu different profiles.

Cargo has default settings for each of the profile `[profile.*]` sections in the project's *Cargo.tom* sections for any profile you want to customize, y default settings. For example, here are the defau for the `dev` and `release` profiles:

Filename: Cargo.toml

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

The `opt-level` setting controls the number of c
code, with a range of 0 to 3. Applying more optir
if you're in development and compiling your coc
even if the resulting code runs slower. That is th
`dev` is `0`. When you're ready to release your co
compiling. You'll only compile in release mode o
program many times, so release mode trades lo
faster. That is why the default `opt-level` for th

You can override any default setting by adding a
For example, if we want to use optimization leve
add these two lines to our project's *Cargo.toml* fi

Filename: Cargo.toml

```
[profile.dev]
opt-level = 1
```

This code overrides the default setting of `0`. No
will use the defaults for the `dev` profile plus our
Because we set `opt-level` to `1`, Cargo will app
default, but not as many as in a release build.

For the full list of configuration options and defa
documentation.

Publishing a Crate to Crates.io

We've used packages from crates.io as depende
share your code with other people by publishing
registry at crates.io distributes the source code
code that is open source.

Rust and Cargo have features that help make yo
people to use and to find in the first place. We'll

next and then explain how to publish a package

Making Useful Documentation Comme

Accurately documenting your packages will help
use them, so it's worth investing the time to writ
discussed how to comment Rust code using two
particular kind of comment for documentation,
documentation comment, that will generate HTM
the contents of documentation comments for pu
programmers interested in knowing how to *use*
crate is *implemented*.

Documentation comments use three slashes, `/`
Markdown notation for formatting the text. Plac
before the item they're documenting. Listing 14-
for an `add_one` function in a crate named `my_c`

Filename: src/lib.rs

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, my_crate::add_one(5));
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Listing 14-1: A documentation comment for a fu

Here, we give a description of what the `add_one`
the heading `Examples`, and then provide code t
`add_one` function. We can generate the HTML d
documentation comment by running `cargo doc`
tool distributed with Rust and puts the generate
target/doc directory.

For convenience, running `cargo doc --open` wi
crate's documentation (as well as the document
dependencies) and open the result in a web bro

function and you'll see how the text in the docur
shown in Figure 14-1:

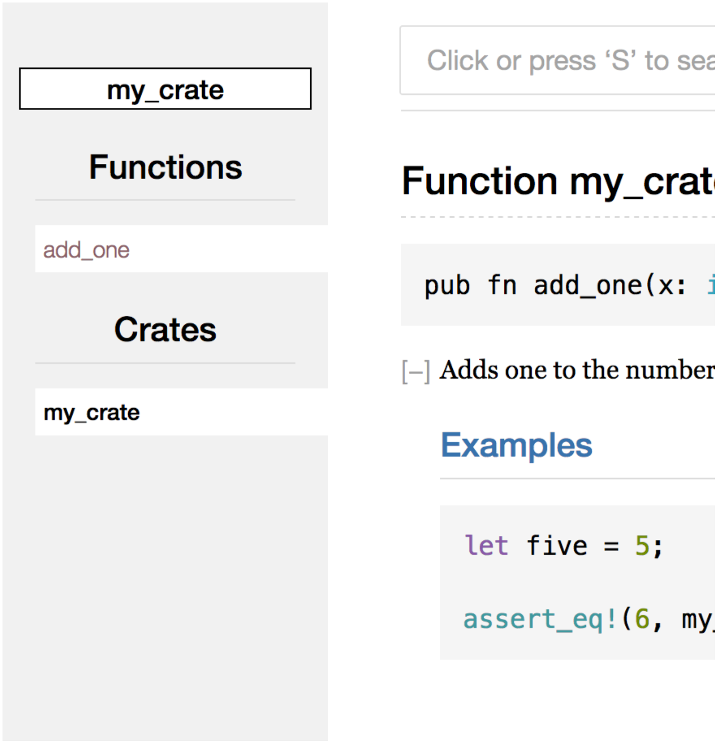

Figure 14-1: HTML documentation for the `add_c`

Commonly Used Sections

We used the `# Examples` Markdown heading in
HTML with the title "Examples." Here are some c
commonly use in their documentation:

- **Panics**: The scenarios in which the functio
  Callers of the function who don't want thei
  they don't call the function in these situatic
- **Errors**: If the function returns a `Result`, d
  might occur and what conditions might cau
  be helpful to callers so they can write code
  errors in different ways.
- **Safety**: If the function is `unsafe` to call (we
  there should be a section explaining why tl
  invariants that the function expects callers

Most documentation comments don't need all o
checklist to remind you of the aspects of your cc
be interested in knowing about.

Documentation Comments as Tests

Adding example code blocks in your documenta
how to use your library, and doing so has an add
will run the code examples in your documentati
documentation with examples. But nothing is w
because the code has changed since the docum
`cargo test` with the documentation for the `ad`
will see a section in the test results like this:

```
   Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 igr
```

Now if we change either the function or the exa
example panics and run `cargo test` again, we'l
example and the code are out of sync with each

Commenting Contained Items

Another style of doc comment, `//!`, adds docu
the comments rather than adding documentatic
comments. We typically use these doc comment
convention) or inside a module to document the

For example, if we want to add documentation t
`my_crate` crate that contains the `add_one` func
comments that start with `//!` to the beginning
14-2:

Filename: src/lib.rs

```
//! # My Crate
//!
//! `my_crate` is a collection of utilitie
//! calculations more convenient.

/// Adds one to the number given.
// --snip--
```

Listing 14-2: Documentation for the `my_crate` c

Notice there isn't any code after the last line tha

started the comments with `//!` instead of `///`.
contains this comment rather than an item that
the item that contains this comment is the *src/li*
comments describe the entire crate.

When we run `cargo doc --open`, these comme
the documentation for `my_crate` above the list
in Figure 14-2:

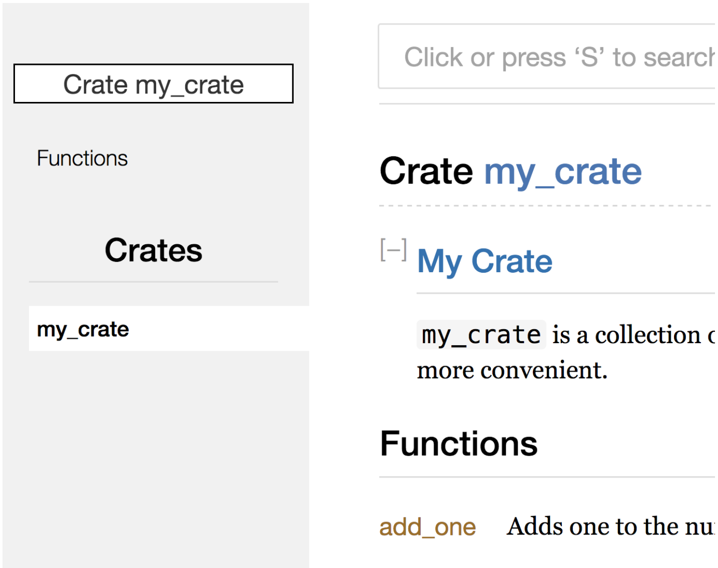

Figure 14-2: Rendered documentation for `my_cr`
describing the crate as a whole

Documentation comments within items are usef
especially. Use them to explain the overall purp
users understand the crate's organization.

Exporting a Convenient Public API with

In Chapter 7, we covered how to organize our co
keyword, how to make items public using the `pu`
into a scope with the `use` keyword. However, th
while you're developing a crate might not be ver
might want to organize your structs in a hierarch
people who want to use a type you've defined d
trouble finding out that type exists. They might a
`use my_crate::some_module::another_module`
`my_crate::UsefulType;`.

The structure of your public API is a major consi

People who use your crate are less familiar with
have difficulty finding the pieces they want to us
hierarchy.

The good news is that if the structure *isn't* conve
library, you don't have to rearrange your interna
export items to make a public structure that's di
using `pub use`. Re-exporting takes a public item
another location, as if it were defined in the othe

For example, say we made a library named `art`
this library are two modules: a `kinds` module c
`PrimaryColor` and `SecondaryColor` and a `util`
named `mix`, as shown in Listing 14-3:

Filename: src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concep

pub mod kinds {
    /// The primary colors according to th
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use kinds::*;

    /// Combines two primary colors in equ
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: Prima
        // --snip--
    }
}
```

Listing 14-3: An `art` library with items organize

Figure 14-3 shows what the front page of the do
by `cargo doc` would look like:

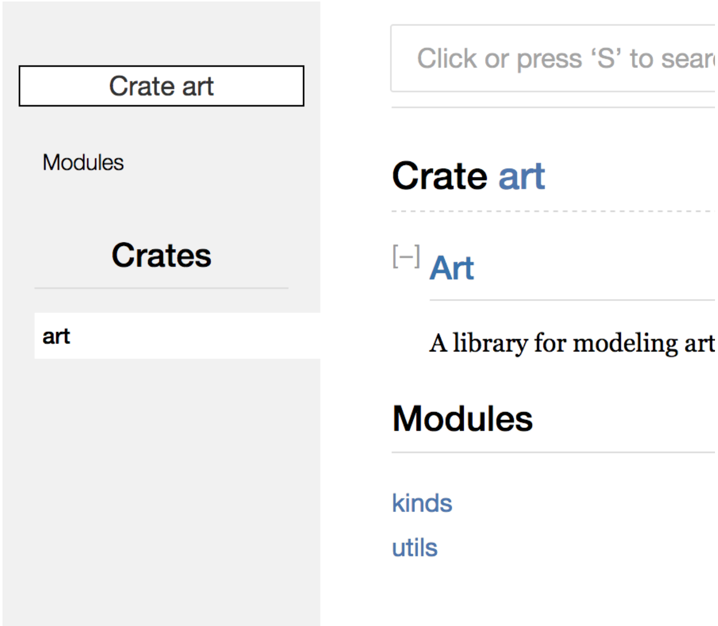

Figure 14-3: Front page of the documentation fo
modules

Note that the `PrimaryColor` and `SecondaryCol`
page, nor is the `mix` function. We have to click

Another crate that depends on this library woul
the items from `art`, specifying the module stru
14-4 shows an example of a crate that uses the
the `art` crate:

Filename: src/main.rs

```
extern crate art;

use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

Listing 14-4: A crate using the `art` crate's items

The author of the code in Listing 14-4, which use
that `PrimaryColor` is in the `kinds` module and

module structure of the `art` crate is more relev
`art` crate than to developers using the `art` cra
organizes parts of the crate into the `kinds` moc
contain any useful information for someone tryi
crate. Instead, the `art` crate's module structure
developers have to figure out where to look, anc
because developers must specify the module na

To remove the internal organization from the pu
code in Listing 14-3 to add `pub use` statements
level, as shown in Listing 14-5:

Filename: src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concep

pub use kinds::PrimaryColor;
pub use kinds::SecondaryColor;
pub use utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

Listing 14-5: Adding `pub use` statements to re-e

The API documentation that `cargo doc` generat
re-exports on the front page, as shown in Figure
`SecondaryColor` types and the `mix` function ea

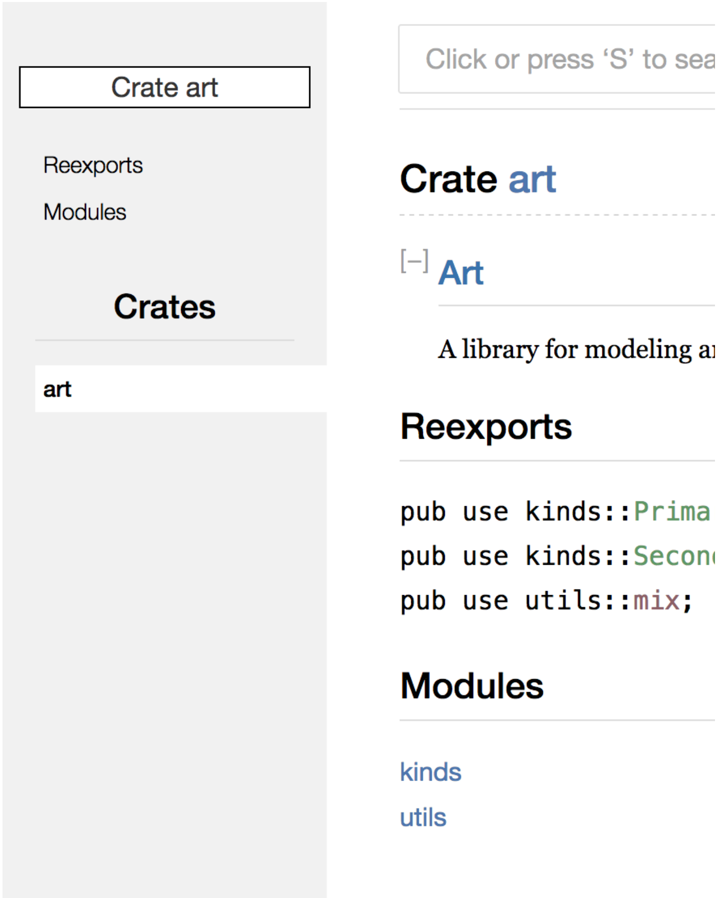

Figure 14-4: The front page of the documentatio

The `art` crate users can still see and use the int demonstrated in Listing 14-4, or they can use th 14-5, as shown in Listing 14-6:

Filename: src/main.rs

```
extern crate art;

use art::PrimaryColor;
use art::mix;

fn main() {
    // --snip--
}
```

Listing 14-6: A program using the re-exported ite

In cases where there are many nested modules, level with `pub use` can make a significant differ use the crate.

Creating a useful public API structure is more of iterate to find the API that works best for your u

flexibility in how you structure your crate intern
structure from what you present to your users. I
you've installed to see if their internal structure

Setting Up a Crates.io Account

Before you can publish any crates, you need to c
an API token. To do so, visit the home page at cr
account. (The GitHub account is currently a requ
other ways of creating an account in the future.)
account settings at https://crates.io/me/ and ret
`cargo login` command with your API key, like t

```
$ cargo login abcdefghijklmnopqrstuvwxyz01
```

This command will inform Cargo of your API tok
/credentials. Note that this token is a *secret*: do n
share it with anyone for any reason, you should
on crates.io.

Adding Metadata to a New Crate

Now that you have an account, let's say you hav
publishing, you'll need to add some metadata to
`[package]` section of the crate's *Cargo.toml* file.

Your crate will need a unique name. While you'r
name a crate whatever you'd like. However, crat
a first-come, first-served basis. Once a crate nam
crate with that name. Search for the name you w
whether it has been used. If it hasn't, edit the na
`[package]` to use the name for publishing, like

Filename: Cargo.toml

```
[package]
name = "guessing_game"
```

Even if you've chosen a unique name, when you
crate at this point, you'll get a warning and then

```
$ cargo publish
    Updating registry `https://github.com/
warning: manifest has no description, lice
documentation,
homepage or repository.
--snip--
error: api errors: missing or empty metada
```

The reason is that you're missing some crucial ir are required so people will know what your crat can use it. To rectify this error, you need to inclu file.

Add a description that is just a sentence or two, in search results. For the `license` field, you nee Linux Foundation's Software Package Data Exch can use for this value. For example, to specify th the MIT License, add the `MIT` identifier:

Filename: Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

If you want to use a license that doesn't appear text of that license in a file, include the file in you `license-file` to specify the name of that file in

Guidance on which license is appropriate for you book. Many people in the Rust community licen Rust by using a dual license of `MIT OR Apache-2` you can also specify multiple license identifiers s licenses for your project.

With a unique name, the version, the author det created the crate, your description, and a license project that is ready to publish might look like th

Filename: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
description = "A fun game where you guess
chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

Cargo's documentation describes other metada
can discover and use your crate more easily.

Publishing to Crates.io

Now that you've created an account, saved your
crate, and specified the required metadata, you'
uploads a specific version to crates.io for others

Be careful when publishing a crate because a pu
never be overwritten, and the code cannot be de
to act as a permanent archive of code so that bu
crates from crates.io will continue to work. Allow
fulfilling that goal impossible. However, there is
versions you can publish.

Run the `cargo publish` command again. It sho

```
$ cargo publish
 Updating registry `https://github.com/rus
Packaging guessing_game v0.1.0 (file:///pr
Verifying guessing_game v0.1.0 (file:///pr
Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/pac
 Finished dev [unoptimized + debuginfo] ta
Uploading guessing_game v0.1.0 (file:///pr
```

Congratulations! You've now shared your code w
can easily add your crate as a dependency of the

Publishing a New Version of an Existin

When you've made changes to your crate and ar
change the `version` value specified in your *Carg*

Semantic Versioning rules to decide what an app
based on the kinds of changes you've made. The
new version.

Removing Versions from Crates.io with

Although you can't remove previous versions of
projects from adding them as a new dependenc
is broken for one reason or another. In such situ
crate version.

Yanking a version prevents new projects from st
while allowing all existing projects that depend o
depend on that version. Essentially, a yank mea
will not break, and any future *Cargo.lock* files ge
version.

To yank a version of a crate, run `cargo yank` an
yank:

```
$ cargo yank --vers 1.0.1
```

By adding `--undo` to the command, you can als
start depending on a version again:

```
$ cargo yank --vers 1.0.1 --undo
```

A yank *does not* delete any code. For example, th
deleting accidentally uploaded secrets. If that ha
immediately.

Cargo Workspaces

In Chapter 12, we built a package that included a
your project develops, you might find that the lib
and you want to split up your package further in
situation, Cargo offers a feature called *workspac*
related packages that are developed in tandem.

Creating a Workspace

A *workspace* is a set of packages that share the s
Let's make a project using a workspace—we'll us
on the structure of the workspace. There are mu
we're going to show one common way. We'll hav
and two libraries. The binary, which will provide
the two libraries. One library will provide an `adc`
an `add_two` function. These three crates will be
start by creating a new directory for the workspa

```
$ mkdir add
$ cd add
```

Next, in the *add* directory, we create the *Cargo.tc*
workspace. This file won't have a `[package]` sec
other *Cargo.toml* files. Instead, it will start with a
us to add members to the workspace by specify
this case, that path is *adder*:

Filename: Cargo.toml

```
[workspace]

members = [
    "adder",
]
```

Next, we'll create the `adder` binary crate by run
directory:

```
$ cargo new adder
     Created binary (application) `adder`
```

At this point, we can build the workspace by run
add directory should look like this:

```
├── Cargo.lock
├── Cargo.toml
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

The workspace has one *target* directory at the tc
be placed into; the `adder` crate doesn't have its
to run `cargo build` from inside the *adder* direc

end up in *add/target* rather than *add/adder/targe*
directory in a workspace like this because the cr
depend on each other. If each crate had its own
have to recompile each of the other crates in the
own *target* directory. By sharing one *target* direc
rebuilding.

Creating the Second Crate in the Work

Next, let's create another member crate in the w
the top-level *Cargo.toml* to specify the *add-one* p

Filename: Cargo.toml

```
[workspace]

members = [
    "adder",
    "add-one",
]
```

Then generate a new library crate named `add-o`

```
$ cargo new add-one --lib
     Created library `add-one` project
```

Your *add* directory should now have these direc

```
├── Cargo.lock
├── Cargo.toml
├── add-one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

In the *add-one/src/lib.rs* file, let's add an `add_one`

Filename: add-one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Now that we have a library crate in the workspa adder depend on the library crate add-one . Fir dependency on add-one to *adder/Cargo.toml*.

Filename: adder/Cargo.toml

```
[dependencies]

add-one = { path = "../add-one" }
```

Cargo doesn't assume that crates in a workspac need to be explicit about the dependency relatic

Next, let's use the add_one function from the a Open the *adder/src/main.rs* file and add an exte new add-one library crate into scope. Then chai add_one function, as in Listing 14-7:

Filename: adder/src/main.rs

```
extern crate add_one;

fn main() {
    let num = 10;
    println!("Hello, world! {} plus one is
add_one::add_one(num));
}
```

Listing 14-7: Using the add-one library crate fro

Let's build the workspace by running cargo bui

```
$ cargo build
   Compiling add-one v0.1.0 (file:///proje
   Compiling adder v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
```

To run the binary crate from the *add* directory, v the workspace we want to use by using the -p a cargo run :

```
$ cargo run -p adder
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

This runs the code in *adder/src/main.rs*, which de

Depending on an External Crate in a Workspa

Notice that the workspace has only one *Cargo.lo* workspace rather than having a *Cargo.lock* in ea all crates are using the same version of all depe the *adder/Cargo.toml* and *add-one/Cargo.toml* file one version of `rand` and record that in the one workspace use the same dependencies means t be compatible with each other. Let's add the `ra` section in the *add-one/Cargo.toml* file to be able crate:

Filename: add-one/Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

We can now add `extern crate rand;` to the *ad* whole workspace by running `cargo build` in th compile the `rand` crate:

```
$ cargo build
    Updating registry `https://github.com/
 Downloading rand v0.3.14
   --snip--
   Compiling rand v0.3.14
   Compiling add-one v0.1.0 (file:///proje
   Compiling adder v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
```

The top-level *Cargo.lock* now contains informatio on `rand`. However, even though `rand` is used s use it in other crates in the workspace unless we well. For example, if we add `extern crate rand` `adder` crate, we'll get an error:

```
$ cargo build
   Compiling adder v0.1.0 (file:///project
error: use of unstable library feature 'ra
(see
issue #27703)
 --> adder/src/main.rs:1:1
  |
1 | extern crate rand;
```

To fix this, edit the *Cargo.toml* file for the `adder` dependency for that crate as well. Building the of dependencies for `adder` in *Cargo.lock*, but no downloaded. Cargo has ensured that every crat crate will be using the same version. Using the s workspace saves space because we won't have r crates in the workspace will be compatible with

Adding a Test to a Workspace

For another enhancement, let's add a test of the the `add_one` crate:

Filename: add-one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

Now run `cargo test` in the top-level *add* direct

```
$ cargo test
   Compiling add-one v0.1.0 (file:///proje
   Compiling adder v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/add_one-f02

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 igr

     Running target/debug/deps/adder-f88af

running 0 tests

test result: ok. 0 passed; 0 failed; 0 igr

   Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 igr
```

The first section of the output shows that the `it
passed. The next section shows that zero tests v
then the last section shows zero documentation
crate. Running `cargo test` in a workspace stru
for all the crates in the workspace.

We can also run tests for one particular crate in
directory by using the `-p` flag and specifying the

```
$ cargo test -p add-one
    Finished dev [unoptimized + debuginfo]
     Running target/debug/deps/add_one-b32

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 igr

   Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 igr
```

This output shows `cargo test` only ran the test
run the `adder` crate tests.

If you publish the crates in the workspace to *htt*
workspace will need to be published separately.
not have an `--all` flag or a `-p` flag, so you mus
run `cargo publish` on each crate in the worksp

For additional practice, add an `add-two` crate tc
the `add-one` crate!

As your project grows, consider using a workspa
individual components than one big blob of cod
a workspace can make coordination between th
the same time.

Installing Binaries from Crate

The `cargo install` command allows you to ins
isn't intended to replace system packages; it's m
developers to install tools that others have shar
only install packages that have binary targets. A
that is created if the crate has a *src/main.rs* file c
opposed to a library target that isn't runnable or
within other programs. Usually, crates have info
whether a crate is a library, has a binary target, c

All binaries installed with `cargo install` are st
folder. If you installed Rust using *rustup.rs* and d
this directory will be *$HOME/.cargo/bin*. Ensure t
able to run programs you've installed with `carg`

For example, in Chapter 12 we mentioned that t
`grep` tool called `ripgrep` for searching files. If
run the following:

```
$ cargo install ripgrep
Updating registry `https://github.com/rust
 Downloading ripgrep v0.3.2
 --snip--
   Compiling ripgrep v0.3.2
    Finished release [optimized + debuginf
  Installing ~/.cargo/bin/rg
```

The last line of the output shows the location an
which in the case of `ripgrep` is `rg`. As long as t

`$PATH`, as mentioned previously, you can then r
faster, rustier tool for searching files!

Extending Cargo with Custom

Cargo is designed so you can extend it with new
modify Cargo. If a binary in your `$PATH` is name
if it was a Cargo subcommand by running `cargo
this are also listed when you run `cargo --list`
install extensions and then run them just like th
convenient benefit of Cargo's design!

Summary

Sharing code with Cargo and crates.io is part of
useful for many different tasks. Rust's standard
are easy to share, use, and improve on a timelin
Don't be shy about sharing code that's useful to
be useful to someone else as well!

Smart Pointers

A *pointer* is a general concept for a variable that
address refers to, or "points at," some other dat
in Rust is a reference, which you learned about i
by the `&` symbol and borrow the value they poi
capabilities other than referring to data. Also, th
the kind of pointer we use most often.

Smart pointers, on the other hand, are data struc
but also have additional metadata and capabiliti
isn't unique to Rust: smart pointers originated ir
well. In Rust, the different smart pointers define
functionality beyond that provided by reference
this chapter is the *reference counting* smart point
have multiple owners of data by keeping track o
owners remain, cleaning up the data.

In Rust, which uses the concept of ownership an

between references and smart pointers is that r
borrow data; in contrast, in many cases, smart p

We've already encountered a few smart pointers
`Vec<T>` in Chapter 8, although we didn't call the
these types count as smart pointers because the
to manipulate it. They also have metadata (such
capabilities or guarantees (such as with `String`
UTF-8).

Smart pointers are usually implemented using s
distinguishes a smart pointer from an ordinary s
implement the `Deref` and `Drop` traits. The `Der`
smart pointer struct to behave like a reference s
either references or smart pointers. The `Drop` t
that is run when an instance of the smart pointe
we'll discuss both traits and demonstrate why th

Given that the smart pointer pattern is a genera
Rust, this chapter won't cover every existing sma
own smart pointers, and you can even write you
smart pointers in the standard library:

- `Box<T>` for allocating values on the heap
- `Rc<T>`, a reference counting type that ena
- `Ref<T>` and `RefMut<T>`, accessed through the borrowing rules at runtime instead of c

In addition, we'll cover the *interior mutability* pat
exposes an API for mutating an interior value. W
they can leak memory and how to prevent them

Let's dive in!

Using `Box<T>` to Point to Data

The most straightforward smart pointer is a *box*
allow you to store data on the heap rather than
is the pointer to the heap data. Refer to Chapter
the stack and the heap.

Boxes don't have performance overhead, other
instead of on the stack. But they don't have mar

them most often in these situations:

- When you have a type whose size can't be to use a value of that type in a context that
- When you have a large amount of data and ensure the data won't be copied when you
- When you want to own a value and you ca a particular trait rather than being of a spe

We'll demonstrate the first situation in the "Enab section. In the second case, transferring owners a long time because the data is copied around o in this situation, we can store the large amount only the small amount of pointer data is copied references stays in one place on the heap. The t and Chapter 17 devotes an entire section, "Using Different Types," just to that topic. So what you l Chapter 17!

Using a `Box<T>` to Store Data on the H

Before we discuss this use case for `Box<T>`, we'l with values stored within a `Box<T>`.

Listing 15-1 shows how to use a box to store an

Filename: src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Listing 15-1: Storing an `i32` value on the heap u

We define the variable `b` to have the value of a is allocated on the heap. This program will print the data in the box similar to how we would if th owned value, when a box goes out of scope, as deallocated. The deallocation happens for the b it points to (stored on the heap).

Putting a single value on the heap isn't very usef themselves in this way very often. Having values

they're stored by default, is more appropriate in
at a case where boxes allow us to define types tl
didn't have boxes.

Enabling Recursive Types with Boxes

At compile time, Rust needs to know how much
whose size can't be known at compile time is a *r*
as part of itself another value of the same type.
theoretically continue infinitely, Rust doesn't kn
recursive type needs. However, boxes have a kn
recursive type definition, you can have recursive

Let's explore the *cons list*, which is a data type co
languages, as an example of a recursive type. Th
straightforward except for the recursion; theref
work with will be useful any time you get into m
recursive types.

More Information About the Cons List

A *cons list* is a data structure that comes from th
dialects. In Lisp, the `cons` function (short for "co
pair from its two arguments, which usually are a
pairs containing pairs form a list.

The cons function concept has made its way into
programming jargon: "to cons *x* onto *y*" informal
container instance by putting the element *x* at th
followed by the container *y*.

Each item in a cons list contains two elements: t
next item. The last item in the list contains only a
item. A cons list is produced by recursively callin
name to denote the base case of the recursion i
as the "null" or "nil" concept in Chapter 6, which

Although functional programming languages us
isn't a commonly used data structure in Rust. M
items in Rust, `Vec<T>` is a better choice to use. C
types *are* useful in various situations, but by star
how boxes let us define a recursive data type wi

Listing 15-2 contains an enum definition for a cc compile yet because the `List` type doesn't have demonstrate.

Filename: src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```

Listing 15-2: The first attempt at defining an enu structure of `i32` values

---

> Note: We're implementing a cons list that hol purposes of this example. We could have imp discussed in Chapter 10, to define a cons list t type.

Using the `List` type to store the list `1, 2, 3` w

Filename: src/main.rs

```
use List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil
}
```

Listing 15-3: Using the `List` enum to store the l

The first `Cons` value holds `1` and another `List` `Cons` value that holds `2` and another `List` val value that holds `3` and a `List` value, which is fi that signals the end of the list.

If we try to compile the code in Listing 15-3, we g

```
error[E0072]: recursive type `List` has ir
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite
2 |     Cons(i32, List),
  |               ----- recursive without
  |
  = help: insert indirection (e.g., a `Box
to
  make `List` representable
```

Listing 15-4: The error we get when attempting t

The error shows this type "has infinite size." The with a variant that is recursive: it holds another can't figure out how much space it needs to stor we get this error a bit. First, let's look at how Rus store a value of a non-recursive type.

Computing the Size of a Non-Recursive Type

Recall the `Message` enum we defined in Listing 6 definitions in Chapter 6:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

To determine how much space to allocate for a each of the variants to see which variant needs t `Message::Quit` doesn't need any space, `Messag` store two `i32` values, and so forth. Because onl space a `Message` value will need is the space it v variants.

Contrast this with what happens when Rust tries recursive type like the `List` enum in Listing 15- looking at the `Cons` variant, which holds a value `List`. Therefore, `Cons` needs an amount of spa the size of a `List`. To figure out how much mer compiler looks at the variants, starting with the

a value of type `i32` and a value of type `List`, a
shown in Figure 15-1.

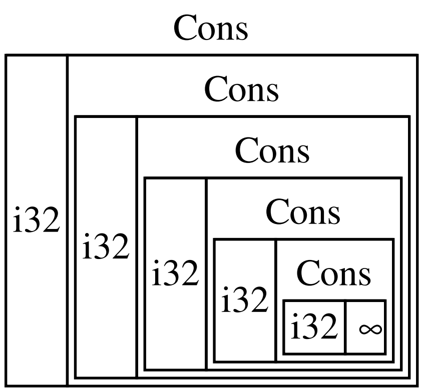

Figure 15-1: An infinite `List` consisting of infinit

Using `Box<T>` to Get a Recursive Type with a

Rust can't figure out how much space to allocate
compiler gives the error in Listing 15-4. But the e
suggestion:

```
  = help: insert indirection (e.g., a `Bo
to
  make `List` representable
```

In this suggestion, "indirection" means that inste
change the data structure to store the value ind
value instead.

Because a `Box<T>` is a pointer, Rust always kno
a pointer's size doesn't change based on the am
means we can put a `Box<T>` inside the `Cons` va
directly. The `Box<T>` will point to the next `List`
than inside the `Cons` variant. Conceptually, we s
"holding" other lists, but this implementation is r
to one another rather than inside one another.

We can change the definition of the `List` enum
`List` in Listing 15-3 to the code in Listing 15-5, w

Filename: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```

Listing 15-5: Definition of `List` that uses `Box<T`

The `Cons` variant will need the size of an `i32` p
pointer data. The `Nil` variant stores no values,
variant. We now know that any `List` value will t
size of a box's pointer data. By using a box, we'v
so the compiler can figure out the size it needs t
shows what the `Cons` variant looks like now.

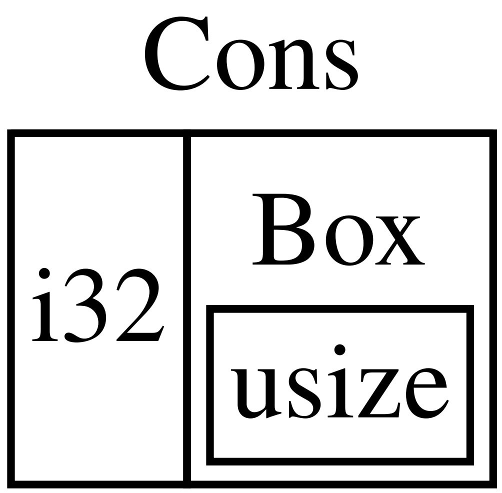

Figure 15-2: A `List` that is not infinitely sized be

Boxes provide only the indirection and heap allc
special capabilities, like those we'll see with the
don't have any performance overhead that thes
can be useful in cases like the cons list where th
need. We'll look at more use cases for boxes in

The `Box<T>` type is a smart pointer because it ir
allows `Box<T>` values to be treated like referenc
scope, the heap data that the box is pointing to
`Drop` trait implementation. Let's explore these t
traits will be even more important to the functio
pointer types we'll discuss in the rest of this cha

Treating Smart Pointers Like R the `Deref` Trait

Implementing the `Deref` trait allows you to cust *operator*, `*` (as opposed to the multiplication or `Deref` in such a way that a smart pointer can be can write code that operates on references and too.

Let's first look at how the dereference operator we'll try to define a custom type that behaves lik dereference operator doesn't work like a referen explore how implementing the `Deref` trait mak work in a similar way as references. Then we'll lo how it lets us work with either references or sma

---

> There's one big difference between the `MyBo` the real `Box<T>`: our version will not store its this example on `Deref`, and so where the dat important than the pointer-like behavior.

Following the Pointer to the Value with

A regular reference is a type of pointer, and one arrow to a value stored somewhere else. In Listi `i32` value and then use the dereference operat

Filename: src/main.rs

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-6: Using the dereference operator to f

The variable `x` holds an `i32` value, `5`. We set `y` assert that `x` is equal to `5`. However, if we wan value in `y`, we have to use `*y` to follow the refe (hence *dereference*). Once we dereference `y`, we pointing to that we can compare with `5`.

If we tried to write `assert_eq!(5, y);` instead,

```
error[E0277]: the trait bound `{integer}:
is
not satisfied
 --> src/main.rs:6:5
  |
6 |     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^^ can't compare `{
  |
  = help: the trait `std::cmp::PartialEq<&
for
  `{integer}`
```

Comparing a number and a reference to a numb different types. We must use the dereference op value it's pointing to.

Using `Box<T>` Like a Reference

We can rewrite the code in Listing 15-6 to use a dereference operator will work as shown in Listi

Filename: src/main.rs

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-7: Using the dereference operator on

The only difference between Listing 15-7 and Lis an instance of a box pointing to the value in `x` r value of `x` . In the last assertion, we can use the box's pointer in the same way that we did when explore what is special about `Box<T>` that enab operator by defining our own box type.

Defining Our Own Smart Pointer

Let's build a smart pointer similar to the `Box<T>` library to experience how smart pointers behav default. Then we'll look at how to add the ability

The `Box<T>` type is ultimately defined as a tuple 15-8 defines a `MyBox<T>` type in the same way. match the `new` function defined on `Box<T>` .

Filename: src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

Listing 15-8: Defining a `MyBox<T>` type

We define a struct named `MyBox` and declare a want our type to hold values of any type. The `My` element of type `T` . The `MyBox::new` function ta returns a `MyBox` instance that holds the value p

Let's try adding the `main` function in Listing 15-7
the `MyBox<T>` type we've defined instead of `Box`
compile because Rust doesn't know how to dere

Filename: src/main.rs

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-9: Attempting to use `MyBox<T>` in the
`Box<T>`

Here's the resulting compilation error:

```
error[E0614]: type `MyBox<{integer}>` cann
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^
```

Our `MyBox<T>` type can't be dereferenced becau
ability on our type. To enable dereferencing with
`Deref` trait.

Treating a Type Like a Reference by Im

As discussed in Chapter 10, to implement a trait
for the trait's required methods. The `Deref` trai
requires us to implement one method named `d`
a reference to the inner data. Listing 15-10 conta
add to the definition of `MyBox` :

Filename: src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

Listing 15-10: Implementing `Deref` on `MyBox<T>`

The `type Target = T;` syntax defines an associ
Associated types are a slightly different way of d
don't need to worry about them for now; we'll co
19.

We fill in the body of the `deref` method with `&s`
to the value we want to access with the `*` opera
that calls `*` on the `MyBox<T>` value now compil

Without the `Deref` trait, the compiler can only c
method gives the compiler the ability to take a v
`Deref` and call the `deref` method to get a `&` re
dereference.

When we entered `*y` in Listing 15-9, behind the

```
*(y.deref())
```

Rust substitutes the `*` operator with a call to th
dereference so we don't have to think about wh
`deref` method. This Rust feature lets us write co
we have a regular reference or a type that imple

The reason the `deref` method returns a referen
dereference outside the parentheses in `*(y.der`
ownership system. If the `deref` method returne
reference to the value, the value would be move
ownership of the inner value inside `MyBox<T>` in
use the dereference operator.

Note that the `*` operator is replaced with a call
to the `*` operator just once, each time we use a
substitution of the `*` operator does not recurse

type `i32`, which matches the `5` in `assert_eq!`

Implicit Deref Coercions with Function

Deref coercion is a convenience that Rust perforr methods. Deref coercion converts a reference tc reference to a type that `Deref` can convert the (happens automatically when we pass a referenc argument to a function or method that doesn't r function or method definition. A sequence of ca type we provided into the type the parameter ne

Deref coercion was added to Rust so that progra calls don't need to add as many explicit referenc The deref coercion feature also lets us write mo references or smart pointers.

To see deref coercion in action, let's use the `MyB` as well as the implementation of `Deref` that we shows the definition of a function that has a stri

Filename: src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

Listing 15-11: A `hello` function that has the par

We can call the `hello` function with a string slic `hello("Rust");` for example. Deref coercion m reference to a value of type `MyBox<String>`, as

Filename: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust'
    hello(&m);
}
```

Listing 15-12: Calling `hello` with a reference to because of deref coercion

Here we're calling the `hello` function with the a

a `MyBox<String>` value. Because we implement
Listing 15-10, Rust can turn `&MyBox<String>` int
standard library provides an implementation of
string slice, and this is in the API documentation
turn the `&String` into `&str` , which matches the

If Rust didn't implement deref coercion, we wou
15-13 instead of the code in Listing 15-12 to call
`&MyBox<String>` .

Filename: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust'
    hello(&(*m)[..]);
}
```

Listing 15-13: The code we would have to write i

The `(*m)` dereferences the `MyBox<String>` into
take a string slice of the `String` that is equal to
signature of `hello` . The code without deref coe
understand with all of these symbols involved. D
these conversions for us automatically.

When the `Deref` trait is defined for the types in
use `Deref::deref` as many times as necessary
parameter's type. The number of times that `Der`
resolved at compile time, so there is no runtime
coercion!

How Deref Coercion Interacts with Mu

Similar to how you use the `Deref` trait to overri
references, you can use the `DerefMut` trait to ov
references.

Rust does deref coercion when it finds types and

- From `&T` to `&U` when `T: Deref<Target=U`
- From `&mut T` to `&mut U` when `T: DerefMu`
- From `&mut T` to `&U` when `T: Deref<Targ`

The first two cases are the same except for muta

have a `&T`, and `T` implements `Deref` to some t
transparently. The second case states that the s
mutable references.

The third case is trickier: Rust will also coerce a r
one. But the reverse is *not* possible: immutable r
mutable references. Because of the borrowing r
that mutable reference must be the only referer
program wouldn't compile). Converting one mut
reference will never break the borrowing rules. (
a mutable reference would require that there is
data, and the borrowing rules don't guarantee tl
assumption that converting an immutable refer
possible.

Running Code on Cleanup witl

The second trait important to the smart pointer
customize what happens when a value is about
an implementation for the `Drop` trait on any typ
used to release resources like files or network c
the context of smart pointers because the functi
always used when implementing a smart pointe
`Drop` to deallocate the space on the heap that t

In some languages, the programmer must call c
every time they finish using an instance of a sma
might become overloaded and crash. In Rust, yo
code be run whenever a value goes out of scope
code automatically. As a result, you don't need t
code everywhere in a program that an instance
you still won't leak resources!

Specify the code to run when a value goes out o
trait. The `Drop` trait requires you to implement
mutable reference to `self`. To see when Rust c
`println!` statements for now.

Listing 15-14 shows a `CustomSmartPointer` stru
that it will print `Dropping CustomSmartPointer!`
scope. This example demonstrates when Rust ru

Filename: src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPoi
self.data);
    }
}

fn main() {
    let c = CustomSmartPointer { data: Str
    let d = CustomSmartPointer { data: Str
    println!("CustomSmartPointers created.
}
```

Listing 15-14: A `CustomSmartPointer` struct that would put our cleanup code

The `Drop` trait is included in the prelude, so we implement the `Drop` trait on `CustomSmartPoint` for the `drop` method that calls `println!`. The b you would place any logic that you wanted to ru out of scope. We're printing some text here to d

In `main`, we create two instances of `CustomSmar` `CustomSmartPointers created.`. At the end of `CustomSmartPointer` will go out of scope, and R `drop` method, printing our final message. Note method explicitly.

When we run this program, we'll see the followi

```
CustomSmartPointers created.
Dropping CustomSmartPointer with data `otl
Dropping CustomSmartPointer with data `my
```

Rust automatically called `drop` for us when our the code we specified. Variables are dropped in `d` was dropped before `c`. This example gives y method works; usually you would specify the cle run rather than a print message.

Dropping a Value Early with `std::mem:`

Unfortunately, it's not straightforward to disable
Disabling `drop` isn't usually necessary; the whol
taken care of automatically. Occasionally, howev
value early. One example is when using smart p
want to force the `drop` method that releases th
same scope can acquire the lock. Rust doesn't le
method manually; instead you have to call the `s`
the standard library if you want to force a value
scope.

If we try to call the `Drop` trait's `drop` method m
function from Listing 15-14, as shown in Listing

Filename: src/main.rs

```
fn main() {
    let c = CustomSmartPointer { data: Str
    println!("CustomSmartPointer created."
    c.drop();
    println!("CustomSmartPointer dropped b
}
```

Listing 15-15: Attempting to call the `drop` meth
clean up early

When we try to compile this code, we'll get this e

```
error[E0040]: explicit use of destructor m
  --> src/main.rs:14:7
   |
14 |     c.drop();
   |       ^^^^ explicit destructor calls
```

This error message states that we're not allowed
message uses the term *destructor*, which is the g
function that cleans up an instance. A *destructor*
creates an instance. The `drop` function in Rust i

Rust doesn't let us call `drop` explicitly because R
`drop` on the value at the end of `main`. This wou
would be trying to clean up the same value twice

We can't disable the automatic insertion of `drop`
we can't call the `drop` method explicitly. So, if w

up early, we can use the `std::mem::drop` functi

The `std::mem::drop` function is different than t
We call it by passing the value we want to force t
The function is in the prelude, so we can modify
`drop` function, as shown in Listing 15-16:

Filename: src/main.rs

```
fn main() {
    let c = CustomSmartPointer { data: Str
    println!("CustomSmartPointer created."
    drop(c);
    println!("CustomSmartPointer dropped b
}
```

Listing 15-16: Calling `std::mem::drop` to explicit
scope

Running this code will print the following:

```
CustomSmartPointer created.
Dropping CustomSmartPointer with data `som
CustomSmartPointer dropped before the end
```

The text `Dropping CustomSmartPointer with d`
the `CustomSmartPointer created.` and
`CustomSmartPointer dropped before the end`
method code is called to drop `c` at that point.

You can use code specified in a `Drop` trait imple
cleanup convenient and safe: for instance, you c
memory allocator! With the `Drop` trait and Rust'
to remember to clean up because Rust does it a

You also don't have to worry about problems re
values still in use: the ownership system that ma
also ensures that `drop` gets called only once wh

Now that we've examined `Box<T>` and some of
let's look at a few other smart pointers defined i

`Rc<T>`, the Reference Countec

In the majority of cases, ownership is clear: you
given value. However, there are cases when a si
owners. For example, in graph data structures, r
same node, and that node is conceptually owne
node shouldn't be cleaned up unless it doesn't h

To enable multiple ownership, Rust has a type c
for *reference counting*. The `Rc<T>` type keeps tra
value which determines whether or not a value i
references to a value, the value can be cleaned u
invalid.

Imagine `Rc<T>` as a TV in a family room. When o
turn it on. Others can come into the room and w
leaves the room, they turn off the TV because it'
turns off the TV while others are still watching it,
remaining TV watchers!

We use the `Rc<T>` type when we want to allocat
parts of our program to read and we can't deter
finish using the data last. If we knew which part
that part the data's owner, and the normal owne
would take effect.

Note that `Rc<T>` is only for use in single-thread
concurrency in Chapter 16, we'll cover how to do
programs.

Using `Rc<T>` to Share Data

Let's return to our cons list example in Listing 15
`Box<T>`. This time, we'll create two lists that bot
Conceptually, this looks similar to Figure 15-3:

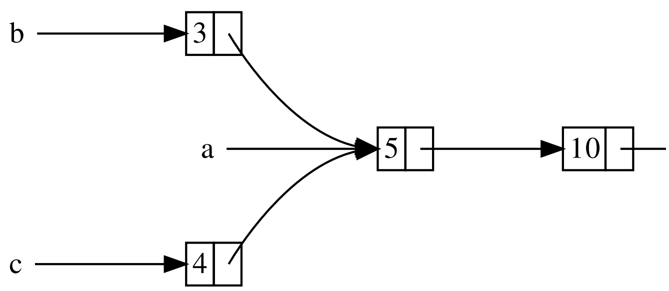

Figure 15-3: Two lists, `b` and `c`, sharing owners

We'll create list `a` that contains 5 and then 10. T starts with 3 and `c` that starts with 4. Both `b` ar first `a` list containing 5 and 10. In other words, b containing 5 and 10.

Trying to implement this scenario using our defi work, as shown in Listing 15-17:

Filename: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

Listing 15-17: Demonstrating we're not allowed t to share ownership of a third list

When we compile this code, we get this error:

```
error[E0382]: use of moved value: `a`
  --> src/main.rs:13:30
   |
12 |     let b = Cons(3, Box::new(a));
   |                              - value
13 |     let c = Cons(4, Box::new(a));
   |                              ^ value
   |
   = note: move occurs because `a` has typ
implement
   the `Copy` trait
```

The `Cons` variants own the data they hold, so w into `b` and `b` owns `a`. Then, when we try to us allowed to because `a` has been moved.

We could change the definition of `Cons` to hold

would have to specify lifetime parameters. By sp
would be specifying that every element in the lis
list. The borrow checker wouldn't let us compile
example, because the temporary `Nil` value wo
reference to it.

Instead, we'll change our definition of `List` to u
shown in Listing 15-18. Each `Cons` variant will n
pointing to a `List`. When we create `b`, instead
the `Rc<List>` that `a` is holding, thereby increas
one to two and letting `a` and `b` share ownershi
also clone `a` when creating `c`, increasing the nu
three. Every time we call `Rc::clone`, the referen
`Rc<List>` will increase, and the data won't be cl
references to it.

Filename: src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(1
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Listing 15-18: A definition of `List` that uses `Rc<`

We need to add a `use` statement to bring `Rc<T`
prelude. In `main`, we create the list holding 5 an
in `a`. Then when we create `b` and `c`, we call th
reference to the `Rc<List>` in `a` as an argument

We could have called `a.clone()` rather than `Rc`
to use `Rc::clone` in this case. The implementat
deep copy of all the data like most types' implen
`Rc::clone` only increments the reference count
copies of data can take a lot of time. By using `R`
can visually distinguish between the deep-copy k
clones that increase the reference count. When

the code, we only need to consider the deep-cop
`Rc::clone`.

Cloning an `Rc<T>` Increases the Refere

Let’s change our working example in Listing 15-1
changing as we create and drop references to th

In Listing 15-19, we’ll change `main` so it has an i
can see how the reference count changes when

Filename: src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(1
    println!("count after creating a = {}"
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}"
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c =
    }
    println!("count after c goes out of sc
Rc::strong_count(&a));
}
```

Listing 15-19: Printing the reference count

At each point in the program where the referen
reference count, which we can get by calling the
function is named `strong_count` rather than `c
has a `weak_count`; we’ll see what `weak_count` is
Reference Cycles” section.

This code prints the following:

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

We can see that the `Rc<List>` in `a` has an initia
we call `clone`, the count goes up by 1. When `c`
down by 1. We don’t have to call a function to de
have to call `Rc::clone` to increase the referenc
`Drop` trait decreases the reference count autom

out of scope.

What we can't see in this example is that when
end of `main`, the count is then 0, and the `Rc<Li`
point. Using `Rc<T>` allows a single value to have
ensures that the value remains valid as long as a

Via immutable references, `Rc<T>` allows you to
your program for reading only. If `Rc<T>` allowed
references too, you might violate one of the bor
multiple mutable borrows to the same place car
inconsistencies. But being able to mutate data is
discuss the interior mutability pattern and the `R`
conjunction with an `Rc<T>` to work with this imm

`RefCell<T>` and the Interior M

Interior mutability is a design pattern in Rust that
when there are immutable references to that da
by the borrowing rules. To mutate data, the patt
structure to bend Rust's usual rules that govern
yet covered unsafe code; we will in Chapter 19. V
mutability pattern when we can ensure that the
runtime, even though the compiler can't guaran
then wrapped in a safe API, and the outer type is

Let's explore this concept by looking at the `RefC`
mutability pattern.

Enforcing Borrowing Rules at Runtime

Unlike `Rc<T>`, the `RefCell<T>` type represents
holds. So, what makes `RefCell<T>` different fro
borrowing rules you learned in Chapter 4:

- At any given time, you can have *either* (but
  or any number of immutable references.
- References must always be valid.

With references and `Box<T>`, the borrowing rule
time. With `RefCell<T>`, these invariants are enf

you break these rules, you'll get a compiler erro
rules, your program will panic and exit.

The advantages of checking the borrowing rules
be caught sooner in the development process, a
performance because all the analysis is complet
checking the borrowing rules at compile time is
cases, which is why this is Rust's default.

The advantage of checking the borrowing rules a
memory-safe scenarios are then allowed, where
compile-time checks. Static analysis, like the Rus
Some properties of code are impossible to dete
famous example is the Halting Problem, which is
an interesting topic to research.

Because some analysis is impossible, if the Rust
complies with the ownership rules, it might reje
conservative. If Rust accepted an incorrect progr
the guarantees Rust makes. However, if Rust rej
programmer will be inconvenienced, but nothin
`RefCell<T>` type is useful when you're sure you
but the compiler is unable to understand and gu

Similar to `Rc<T>`, `RefCell<T>` is only for use in
give you a compile-time error if you try using it i
about how to get the functionality of `RefCell<T`
Chapter 16.

Here is a recap of the reasons to choose `Box<T>`

- `Rc<T>` enables multiple owners of the sam
have single owners.
- `Box<T>` allows immutable or mutable borr
allows only immutable borrows checked at
immutable or mutable borrows checked at
- Because `RefCell<T>` allows mutable borro
mutate the value inside the `RefCell<T>` ev
immutable.

Mutating the value inside an immutable value is
look at a situation in which interior mutability is

Interior Mutability: A Mutable Borrow

A consequence of the borrowing rules is that wh
can't borrow it mutably. For example, this code w

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

If you tried to compile this code, you'd get the fo

```
error[E0596]: cannot borrow immutable loca
 --> src/main.rs:3:18
  |
2 |     let x = 5;
  |         - consider changing this to `m
3 |     let y = &mut x;
  |                  ^ cannot borrow mutab
  |
```

However, there are situations in which it would b
in its methods but appear immutable to other co
methods would not be able to mutate the value.
the ability to have interior mutability. But `RefCe`
borrowing rules completely: the borrow checker
mutability, and the borrowing rules are checked
rules, you'll get a `panic!` instead of a compiler e

Let's work through a practical example where we
immutable value and see why that is useful.

A Use Case for Interior Mutability: Mock Obje

A *test double* is the general programming concep
type during testing. *Mock objects* are specific typ
happens during a test so you can assert that the

Rust doesn't have objects in the same sense as o
Rust doesn't have mock object functionality built
other languages do. However, you can definitely
same purposes as a mock object.

Here's the scenario we'll test: we'll create a librar
maximum value and sends messages based on
current value is. This library could be used to ke
number of API calls they're allowed to make, for

Our library will only provide the functionality of value is and what the messages should be at wh library will be expected to provide the mechanis application could put a message in the applicati message, or something else. The library doesn't is something that implements a trait we'll provid shows the library code:

Filename: src/lib.rs

```
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: 'a + Messer
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
    where T: Messenger {
    pub fn new(messenger: &T, max: usize)
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: us
        self.value = value;

        let percentage_of_max = self.value

        if percentage_of_max >= 0.75 && pe
            self.messenger.send("Warning:
quota!");
        } else if percentage_of_max >= 0.9
            self.messenger.send("Urgent wa
of your quota!");
        } else if percentage_of_max >= 1.0
            self.messenger.send("Error: Yo
        }
    }
}
```

Listing 15-20: A library to keep track of how clos warn when the value is at certain levels

One important part of this code is that the `Mess` `send` that takes an immutable reference to `sel` the interface our mock object needs to have. Th to test the behavior of the `set_value` method c what we pass in for the `value` parameter, but `s` us to make assertions on. We want to be able to with something that implements the `Messenger` when we pass different numbers for `value`, the appropriate messages.

We need a mock object that, instead of sending call `send`, will only keep track of the messages i instance of the mock object, create a `LimitTrac` the `set_value` method on `LimitTracker`, and t the messages we expect. Listing 15-21 shows an to do just that, but the borrow checker won't all

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages:
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning
        let mock_messenger = MockMessenger
        let mut limit_tracker = LimitTrack

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_mes
    }
}
```

Listing 15-21: An attempt to implement a `MockM` borrow checker

This test code defines a `MockMessenger` struct tl `Vec` of `String` values to keep track of the mess an associated function `new` to make it convenie values that start with an empty list of messages. trait for `MockMessenger` so we can give a `MockMe` definition of the `send` method, we take the mes store it in the `MockMessenger` list of `sent_messa`

In the test, we're testing what happens when the to something that is more than 75 percent of th `MockMessenger`, which will start with an empty l `LimitTracker` and give it a reference to the nev 100. We call the `set_value` method on the `Lim`

more than 75 percent of 100. Then we assert tha
`MockMessenger` is keeping track of should now h

However, there's one problem with this test, as s

```
error[E0596]: cannot borrow immutable fiel
mutable
  --> src/lib.rs:52:13
   |
51 |         fn send(&self, message: &str)
   |                 ----- use `&mut self`
52 |             self.sent_messages.push(S
   |             ^^^^^^^^^^^^^^^^^^ cannot
```

We can't modify the `MockMessenger` to keep trac
method takes an immutable reference to `self`.
from the error text to use `&mut self` instead, b
wouldn't match the signature in the `Messenger`
what error message you get).

This is a situation in which interior mutability can
within a `RefCell<T>`, and then the `send` messa
`sent_messages` to store the messages we've see
looks like:

Filename: src/lib.rs

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages:
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut(
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning
        // --snip--

        assert_eq!(mock_messenger.sent_mes
    }
}
```

Listing 15-22: Using `RefCell<T>` to mutate an ir considered immutable

The `sent_messages` field is now of type `RefCel` `Vec<String>`. In the `new` function, we create a r around the empty vector.

For the implementation of the `send` method, th borrow of `self`, which matches the trait definit `RefCell<Vec<String>>` in `self.sent_messages` value inside the `RefCell<Vec<String>>`, which i on the mutable reference to the vector to keep t test.

The last change we have to make is in the assert inner vector, we call `borrow` on the `RefCell<Ve` reference to the vector.

Now that you've seen how to use `RefCell<T>`, l

Keeping Track of Borrows at Runtime with `R`

When creating immutable and mutable referenc respectively. With `RefCell<T>`, we use the `borr` are part of the safe API that belongs to `RefCell` smart pointer type `Ref<T>`, and `borrow_mut` re `RefMut<T>`. Both types implement `Deref`, so w references.

The `RefCell<T>` keeps track of how many `Ref<` are currently active. Every time we call `borrow`, how many immutable borrows are active. When the count of immutable borrows goes down by borrowing rules, `RefCell<T>` lets us have many borrow at any point in time.

If we try to violate these rules, rather than gettir references, the implementation of `RefCell<T>` shows a modification of the implementation of deliberately trying to create two mutable borrov illustrate that `RefCell<T>` prevents us from doi

Filename: src/lib.rs

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_mes
        let mut two_borrow = self.sent_mes

        one_borrow.push(String::from(messa
        two_borrow.push(String::from(messa
    }
}
```

Listing 15-23: Creating two mutable references i `RefCell<T>` will panic

We create a variable `one_borrow` for the `RefMut` `borrow_mut`. Then we create another mutable b `two_borrow`. This makes two mutable reference allowed. When we run the tests for our library, t without any errors, but the test will fail:

```
---- tests::it_sends_an_over_75_percent_wa
    thread 'tests::it_sends_an_over_75_per
at
'already borrowed: BorrowMutError', src/li
note: Run with `RUST_BACKTRACE=1` for a ba
```

Notice that the code panicked with the message
. This is how `RefCell<T>` handles violations of t

Catching borrowing errors at runtime rather tha
find a mistake in your code later in the developm
your code was deployed to production. Also, you
performance penalty as a result of keeping track
than compile time. However, using `RefCell<T>`
object that can modify itself to keep track of the
using it in a context where only immutable value
`RefCell<T>` despite its trade-offs to get more fu
provide.

Having Multiple Owners of Mutable Da `RefCell<T>`

A common way to use `RefCell<T>` is in combin
lets you have multiple owners of some data, but
data. If you have an `Rc<T>` that holds a `RefCell`
have multiple owners *and* that you can mutate!

For example, recall the cons list example in Listi
allow multiple lists to share ownership of anothe
immutable values, we can't change any of the va
them. Let's add in `RefCell<T>` to gain the ability
Listing 15-24 shows that by using a `RefCell<T>`
the value stored in all the lists:

Filename: src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value)

    let b = Cons(Rc::new(RefCell::new(6)),
    let c = Cons(Rc::new(RefCell::new(10))

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

Listing 15-24: Using `Rc<RefCell<i32>>` to creat

We create a value that is an instance of `Rc<RefC` named `value` so we can access it directly later. `Cons` variant that holds `value`. We need to clon ownership of the inner `5` value rather than tran or having `a` borrow from `value`.

We wrap the list `a` in an `Rc<T>` so when we cre to `a`, which is what we did in Listing 15-18.

After we've created the lists in `a`, `b`, and `c`, we this by calling `borrow_mut` on `value`, which use we discussed in Chapter 5 (see the section "Whe dereference the `Rc<T>` to the inner `RefCell<T>` returns a `RefMut<T>` smart pointer, and we use change the inner value.

When we print `a`, `b`, and `c`, we can see that th rather than 5:

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(
c after = Cons(RefCell { value: 10 }, Cons
```

This technique is pretty neat! By using `RefCell<` `List` value. But we can use the methods on `Re` interior mutability so we can modify our data wh of the borrowing rules protect us from data race bit of speed for this flexibility in our data structu

The standard library has other types that provid , which is similar except that instead of giving re is copied in and out of the `Cell<T>` . There's also mutability that's safe to use across threads; we'l out the standard library docs for more details or types.

Reference Cycles Can Leak Me

Rust's memory safety guarantees make it difficu create memory that is never cleaned up (known memory leaks entirely is not one of Rust's guara disallowing data races at compile time is, meani Rust. We can see that Rust allows memory leaks possible to create references where items refer memory leaks because the reference count of e 0, and the values will never be dropped.

Creating a Reference Cycle

Let's look at how a reference cycle might happen the definition of the `List` enum and a `tail` me

Filename: src/main.rs

```
use std::rc::Rc;
use std::cell::RefCell;
use List::{Cons, Nil};

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<L
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
```

Listing 15-25: A cons list definition that holds a `Cons` variant is referring to

We're using another variation of the `List` defin element in the `Cons` variant is now `RefCell<Rc` having the ability to modify the `i32` value as we modify which `List` value a `Cons` variant is poin method to make it convenient for us to access t variant.

In Listing 15-26, we're adding a `main` function t 15-25. This code creates a list in `a` and a list in modifies the list in `a` to point to `b`, creating a r statements along the way to show what the refe this process.

Filename: src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(R

    println!("a initial rc count = {}", Rc
    println!("a next item = {:?}", a.tail(

    let b = Rc::new(Cons(10, RefCell::new(

    println!("a rc count after b creation
    println!("b initial rc count = {}", Rc
    println!("b next item = {:?}", b.tail(

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b)
    }

    println!("b rc count after changing a
    println!("a rc count after changing a

    // Uncomment the next line to see that
    // it will overflow the stack
    // println!("a next item = {:?}", a.ta
}
```

Listing 15-26: Creating a reference cycle of two

We create an `Rc<List>` instance holding a `List` initial list of `5, Nil`. We then create an `Rc<List` value in the variable `b` that contains the value 1

We modify `a` so it points to `b` instead of `Nil`, c the `tail` method to get a reference to the `RefC` the variable `link`. Then we use the `borrow_mut` to change the value inside from an `Rc<List>` th in `b`.

When we run this code, keeping the last `printl` we'll get this output:

```
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5
b rc count after changing a = 2
a rc count after changing a = 2
```

The reference count of the `Rc<List>` instances change the list in `a` to point to `b`. At the end of which will decrease the count in each of the `Rc<`

However, because `a` is still referencing the `Rc<` has a count of 1 rather than 0, so the memory th be dropped. The memory will just sit there with reference cycle, we've created a diagram in Figu

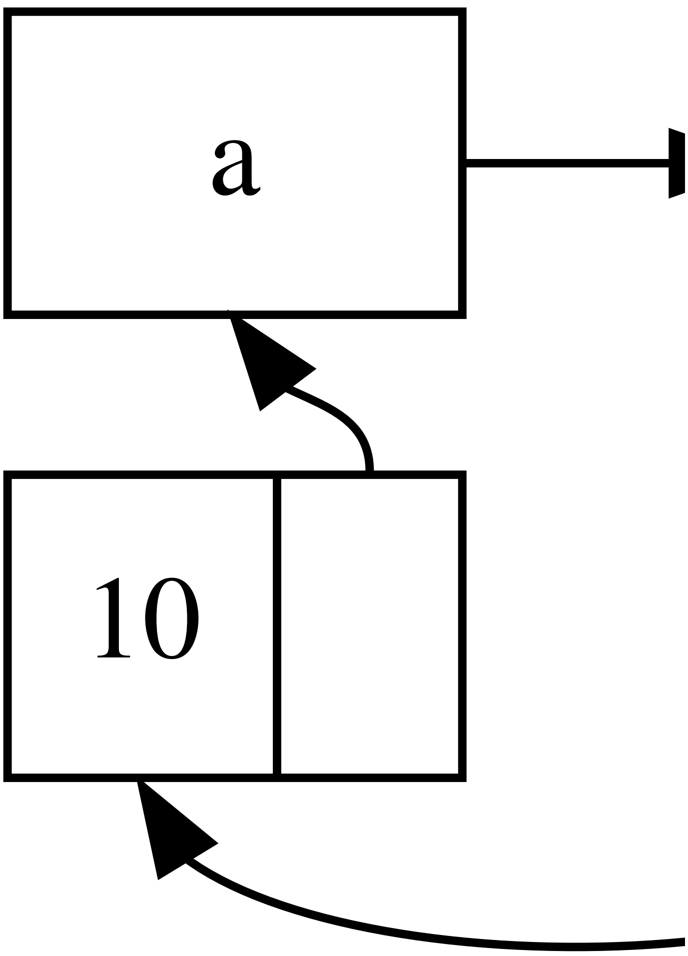

Figure 15-4: A reference cycle of lists `a` and `b` p

If you uncomment the last `println!` and run th cycle with `a` pointing to `b` pointing to `a` and sc

In this case, right after we create the reference c consequences of this cycle aren't very dire. How allocated lots of memory in a cycle and held ont would use more memory than it needed and mi to run out of available memory.

Creating reference cycles is not easily done, but `RefCell<T>` values that contain `Rc<T>` values o types with interior mutability and reference cou

create cycles; you can't rely on Rust to catch ther
be a logic bug in your program that you should u
and other software development practices to m

Another solution for avoiding reference cycles is
that some references express ownership and so
can have cycles made up of some ownership rel
relationships, and only the ownership relationsh
be dropped. In Listing 15-25, we always want `Co`
reorganizing the data structure isn't possible. Le
made up of parent nodes and child nodes to see
are an appropriate way to prevent reference cyc

Preventing Reference Cycles: Turning a

So far, we've demonstrated that calling `Rc::clo`
`Rc<T>` instance, and an `Rc<T>` instance is only c
You can also create a *weak reference* to the value
`Rc::downgrade` and passing a reference to the
, you get a smart pointer of type `Weak<T>` . Inste
the `Rc<T>` instance by 1, calling `Rc::downgrade`
`Rc<T>` type uses `weak_count` to keep track of h
similar to `strong_count` . The difference is the `w`
the `Rc<T>` instance to be cleaned up.

Strong references are how you can share owner
references don't express an ownership relations
cycle because any cycle involving some weak ref
strong reference count of values involved is 0.

Because the value that `Weak<T>` references mig
with the value that a `Weak<T>` is pointing to, you
Do this by calling the `upgrade` method on a `Wea`
`Option<Rc<T>>` . You'll get a result of `Some` if the
yet and a result of `None` if the `Rc<T>` value has
returns an `Option<T>` , Rust will ensure that the
handled, and there won't be an invalid pointer.

As an example, rather than using a list whose ite
we'll create a tree whose items know about their
items.

Creating a Tree Data Structure: a `Node` with (

To start, we'll build a tree with nodes that know
struct named `Node` that holds its own `i32` valu
`Node` values:

Filename: src/main.rs

```
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}
```

We want a `Node` to own its children, and we wa
variables so we can access each `Node` in the tre
`Vec<T>` items to be values of type `Rc<Node>` . W
are children of another node, so we have a `Ref(`
`Vec<Rc<Node>>` .

Next, we'll use our struct definition and create o
the value 3 and no children, and another instan
and `leaf` as one of its children, as shown in Lis

Filename: src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::cl
    });
}
```

Listing 15-27: Creating a `leaf` node with no chil
as one of its children

We clone the `Rc<Node>` in `leaf` and store that i
`leaf` now has two owners: `leaf` and `branch` . '

through `branch.children`, but there's no way t
reason is that `leaf` has no reference to `branch`
want `leaf` to know that `branch` is its parent. W

Adding a Reference from a Child to Its Parent

To make the child node aware of its parent, we r
`Node` struct definition. The trouble is in deciding
We know it can't contain an `Rc<T>`, because tha
`leaf.parent` pointing to `branch` and `branch.c`
would cause their `strong_count` values to neve

Thinking about the relationships another way, a
if a parent node is dropped, its child nodes shou
child should not own its parent: if we drop a chil
This is a case for weak references!

So instead of `Rc<T>`, we'll make the type of `par`
`RefCell<Weak<Node>>`. Now our `Node` struct de

Filename: src/main.rs

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

A node will be able to refer to its parent node bu
15-28, we update `main` to use this new definitio
refer to its parent, `branch`:

Filename: src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.pa

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::cl
    });

    *leaf.parent.borrow_mut() = Rc::downgr

    println!("leaf parent = {:?}", leaf.pa
}
```

Listing 15-28: A `leaf` node with a weak referen

Creating the `leaf` node looks similar to how cre 15-27 with the exception of the `parent` field: `le` create a new, empty `Weak<Node>` reference inst

At this point, when we try to get a reference to t `upgrade` method, we get a `None` value. We see `println!` statement:

```
leaf parent = None
```

When we create the `branch` node, it will also ha the `parent` field, because `branch` doesn't have one of the children of `branch`. Once we have th modify `leaf` to give it a `Weak<Node>` reference method on the `RefCell<Weak<Node>>` in the `pa` the `Rc::downgrade` function to create a `Weak<N` `Rc<Node>` in `branch`.

When we print the parent of `leaf` again, this tir `branch`: now `leaf` can access its parent! When cycle that eventually ended in a stack overflow li `Weak<Node>` references are printed as `(Weak)`:

```
leaf parent = Some(Node { value: 5, parent
children: RefCell { value: [Node { value:
(Weak) },
children: RefCell { value: [] } }] } })
```

The lack of infinite output indicates that this cod
can also tell this by looking at the values we get
`Rc::weak_count`.

Visualizing Changes to `strong_count` and `wea

Let's look at how the `strong_count` and `weak_c`
instances change by creating a new inner scope
into that scope. By doing so, we can see what ha
then dropped when it goes out of scope. The mo
15-29:

Filename: src/main.rs

```rust
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new
            children: RefCell::new(vec![Rc
        });

        *leaf.parent.borrow_mut() = Rc::do

        println!(
            "branch strong = {}, weak = {}
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf strong = {}, weak = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf parent = {:?}", leaf.pa
    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

Listing 15-29: Creating `branch` in an inner scope
reference counts

After `leaf` is created, its `Rc<Node>` has a strong
the inner scope, we create `branch` and associat
we print the counts, the `Rc<Node>` in `branch` wi
count of 1 (for `leaf.parent` pointing to `branch`

the counts in `leaf`, we'll see it will have a strong
a clone of the `Rc<Node>` of `leaf` stored in `bran`
weak count of 0.

When the inner scope ends, `branch` goes out of
`Rc<Node>` decreases to 0, so its `Node` is droppe
`leaf.parent` has no bearing on whether or not
memory leaks!

If we try to access the parent of `leaf` after the e
At the end of the program, the `Rc<Node>` in `lea`
count of 0, because the variable `leaf` is now th
again.

All of the logic that manages the counts and valu
`Weak<T>` and their implementations of the `Drop`
relationship from a child to its parent should be
of `Node`, you're able to have parent nodes point
creating a reference cycle and memory leaks.

Summary

This chapter covered how to use smart pointers
trade-offs than those Rust makes by default with
has a known size and points to data allocated on
track of the number of references to data on the
owners. The `RefCell<T>` type with its interior m
use when we need an immutable type but need
it also enforces the borrowing rules at runtime i

Also discussed were the `Deref` and `Drop` traits,
functionality of smart pointers. We explored refe
leaks and how to prevent them using `Weak<T>`.

If this chapter has piqued your interest and you
pointers, check out "The Rustonomicon" for mor

Next, we'll talk about concurrency in Rust. You'll
pointers.

Fearless Concurrency

Handling concurrent programming safely and ef
goals. *Concurrent programming*, where different
independently, and *parallel programming*, where
at the same time, are becoming increasingly imp
advantage of their multiple processors. Historica
has been difficult and error prone: Rust hopes to

Initially, the Rust team thought that ensuring me
concurrency problems were two separate challe
methods. Over time, the team discovered that th
powerful set of tools to help manage memory sa
leveraging ownership and type checking, many c
errors in Rust rather than runtime errors. There
lots of time trying to reproduce the exact circum
concurrency bug occurs, incorrect code will refu
explaining the problem. As a result, you can fix y
rather than potentially after it has been shipped
aspect of Rust *fearless concurrency*. Fearless con
is free of subtle bugs and is easy to refactor with

> Note: For simplicity's sake, we'll refer to many
> rather than being more precise by saying *con*
> were about concurrency and/or parallelism, w
> chapter, please mentally substitute *concurren*
> *concurrent*.

Many languages are dogmatic about the solutio
problems. For example, Erlang has elegant funct
concurrency but has only obscure ways to share
only a subset of possible solutions is a reasonab
because a higher-level language promises bene
gain abstractions. However, lower-level languag
solution with the best performance in any given
over the hardware. Therefore, Rust offers a vari
whatever way is appropriate for your situation a

Here are the topics we'll cover in this chapter:

- How to create threads to run multiple piec
- *Message-passing* concurrency, where chann
- *Shared-state* concurrency, where multiple t
  data
- The `Sync` and `Send` traits, which extend R

defined types as well as types provided by

Using Threads to Run Code Sir

In most current operating systems, an executed and the operating system manages multiple prc you can also have independent parts that run si these independent parts are called *threads*.

Splitting the computation in your program into r performance because the program does multipl adds complexity. Because threads can run simu guarantee about the order in which parts of you This can lead to problems, such as:

- Race conditions, where threads are access inconsistent order
- Deadlocks, where two threads are waiting resource the other thread has, preventing
- Bugs that happen only in certain situations reliably

Rust attempts to mitigate the negative effects of multithreaded context still takes careful thought different from that in programs running in a sin

Programming languages implement threads in a systems provide an API for creating new threads the operating system APIs to create threads is s operating system thread per one language threa

Many programming languages provide their own Programming language-provided threads are kr that use these green threads will execute them i operating system threads. For this reason, the g model: there are `M` green threads per `N` operat are not necessarily the same number.

Each model has its own advantages and trade-o to Rust is runtime support. *Runtime* is a confusir meanings in different contexts.

In this context, by *runtime* we mean code that is

binary. This code can be large or small dependir
assembly language will have some amount of ru
colloquially when people say a language has "no
runtime." Smaller runtimes have fewer features
smaller binaries, which make it easier to combir
in more contexts. Although many languages are
in exchange for more features, Rust needs to ha
compromise on being able to call into C to main

The green-threading M:N model requires a large
threads. As such, the Rust standard library only
threading. Because Rust is such a low-level langu
M:N threading if you would rather trade overhea
over which threads run when and lower costs of

Now that we've defined threads in Rust, let's exp
API provided by the standard library.

Creating a New Thread with `spawn`

To create a new thread, we call the `thread::spa`
talked about closures in Chapter 13) containing
thread. The example in Listing 16-1 prints some
text from a new thread:

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from th
            thread::sleep(Duration::from_m
        }
    });

    for i in 1..5 {
        println!("hi number {} from the ma
        thread::sleep(Duration::from_milli
    }
}
```

Listing 16-1: Creating a new thread to print one
something else

Note that with this function, the new thread will
ends, whether or not it has finished running. Th
a little different every time, but it will look simila

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

The calls to `thread::sleep` force a thread to sto
allowing a different thread to run. The threads w
guaranteed: it depends on how your operating s
run, the main thread printed first, even though t
thread appears first in the code. And even thoug
print until `i` is 9, it only got to 5 before the mai

If you run this code and only see output from th
overlap, try increasing the numbers in the range
operating system to switch between the threads

Waiting for All Threads to Finish Using

The code in Listing 16-1 not only stops the spaw
time due to the main thread ending, but also ca
thread will get to run at all. The reason is that th
which threads run!

We can fix the problem of the spawned thread r
completely, by saving the return value of `threa`
type of `thread::spawn` is `JoinHandle`. A `JoinH`
we call the `join` method on it, will wait for its th
to use the `JoinHandle` of the thread we create
sure the spawned thread finishes before `main` e

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from th
            thread::sleep(Duration::from_m
        }
    });

    for i in 1..5 {
        println!("hi number {} from the ma
        thread::sleep(Duration::from_milli
    }

    handle.join().unwrap();
}
```

Listing 16-2: Saving a `JoinHandle` from `thread:`
run to completion

Calling `join` on the handle blocks the thread cu
represented by the handle terminates. *Blocking*
prevented from performing work or exiting. Bec
the main thread's `for` loop, running Listing 16-2
this:

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

The two threads continue alternating, but the m
`handle.join()` and does not end until the spaw

But let's see what happens when we instead mo
loop in `main`, like this:

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from th
            thread::sleep(Duration::from_m
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the ma
        thread::sleep(Duration::from_milli
    }
}
```

The main thread will wait for the spawned threa
so the output won't be interleaved anymore, as

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

Small details, such as where `join` is called, can
run at the same time.

Using `move` Closures with Threads

The `move` closure is often used alongside `threa`
use data from one thread in another thread.

In Chapter 13, we mentioned we can use the `mo`
of a closure to force the closure to take ownersh
environment. This technique is especially useful

to transfer ownership of values from one threac

Notice in Listing 16-1 that the closure we pass tc we're not using any data from the main thread i data from the main thread in the spawned threa capture the values it needs. Listing 16-3 shows a main thread and use it in the spawned thread. H see in a moment.

Filename: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}",
    });

    handle.join().unwrap();
}
```

Listing 16-3: Attempting to use a vector created

The closure uses `v`, so it will capture `v` and ma environment. Because `thread::spawn` runs this be able to access `v` inside that new thread. But get the following error:

```
error[E0373]: closure may outlive the curr
`v`,
which is owned by the current function
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may
7 |         println!("Here's a vector: {:?
  |
  |
help: to force the closure to take ownersh
referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move ||
  |                                ^^^^^^^
```

Rust *infers* how to capture `v`, and because `prir` the closure tries to borrow `v`. However, there's

the spawned thread will run, so it doesn't know
valid.

Listing 16-4 provides a scenario that's more likel
be valid:

Filename: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}",
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

Listing 16-4: A thread with a closure that attemp
main thread that drops `v`

If we were allowed to run this code, there's a po
immediately put in the background without runı
reference to `v` inside, but the main thread imm
function we discussed in Chapter 15. Then, wheı
execute, `v` is no longer valid, so a reference to i

To fix the compiler error in Listing 16-3, we can ι

```
help: to force the closure to take ownersh
referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move ||
  |                                ^^^^^^^
```

By adding the `move` keyword before the closure
ownership of the values it's using rather than all
borrow the values. The modification to Listing 1(
and run as we intend:

Filename: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}",
    });

    handle.join().unwrap();
}
```

Listing 16-5: Using the `move` keyword to force a
values it uses

What would happen to the code in Listing 16-4 w
we use a `move` closure? Would `move` fix that cas
different error because what Listing 16-4 is tryin
reason. If we added `move` to the closure, we wo
environment, and we could no longer call `drop`
get this compiler error instead:

```
error[E0382]: use of moved value: `v`
  --> src/main.rs:10:10
   |
6  |     let handle = thread::spawn(move |
   |                                ------
here
...
10 |     drop(v); // oh no!
   |          ^ value used here after move
   |
   = note: move occurs because `v` has typ
does
   not implement the `Copy` trait
```

Rust's ownership rules have saved us again! We
16-3 because Rust was being conservative and c
which meant the main thread could theoretically
reference. By telling Rust to move ownership of
guaranteeing Rust that the main thread won't us
16-4 in the same way, we're then violating the ov
in the main thread. The `move` keyword override
borrowing; it doesn't let us violate the ownershi

With a basic understanding of threads and the t
with threads.

Using Message Passing to Trar Threads

One increasingly popular approach to ensuring : where threads or actors communicate by sendir data. Here's the idea in a slogan from the Go lar communicate by sharing memory; instead, shar

One major tool Rust has for accomplishing mes *channel*, a programming concept that Rust's star implementation of. You can imagine a channel ii channel of water, such as a stream or a river. If y or boat into a stream, it will travel downstream t

A channel in programming has two halves: a trai transmitter half is the upstream location where and the receiver half is where the rubber duck e code calls methods on the transmitter with the c part checks the receiving end for arriving messa either the transmitter or receiver half is droppe

Here, we'll work up to a program that has one th them down a channel, and another thread that v out. We'll be sending simple values between thr feature. Once you're familiar with the technique implement a chat system or a system where ma calculation and send the parts to one thread tha

First, in Listing 16-6, we'll create a channel but n won't compile yet because Rust can't tell what ty channel.

Filename: src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

Listing 16-6: Creating a channel and assigning th

We create a new channel using the `mpsc::chanr` *producer, single consumer*. In short, the way Rust channels means a channel can have multiple *ser*

only one *receiving* end that consumes those valu
together into one big river: everything sent dow
one river at the end. We'll start with a single pro
producers when we get this example working.

The `mpsc::channel` function returns a tuple, th
end and the second element is the receiving enc
traditionally used in many fields for *transmitter* a
our variables as such to indicate each end. We're
pattern that destructures the tuples; we'll discus
statements and destructuring in Chapter 18. Usi
convenient approach to extract the pieces of the

Let's move the transmitting end into a spawned
the spawned thread is communicating with the
This is like putting a rubber duck in the river ups
from one thread to another.

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

Listing 16-7: Moving `tx` to a spawned thread an

Again, we're using `thread::spawn` to create a ne
move `tx` into the closure so the spawned threa
needs to own the transmitting end of the chann
through the channel.

The transmitting end has a `send` method that ta
`send` method returns a `Result<T, E>` type, so
dropped and there's nowhere to send a value, th
In this example, we're calling `unwrap` to panic in
application, we would handle it properly: return
proper error handling.

In Listing 16-8, we'll get the value from the recei
thread. This is like retrieving the rubber duck fr
like getting a chat message.

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Listing 16-8: Receiving the value "hi" in the main

The receiving end of a channel has two useful m
using `recv`, short for *receive*, which will block th
until a value is sent down the channel. Once a va
`Result<T, E>`. When the sending end of the ch
error to signal that no more values will be comin

The `try_recv` method doesn't block, but will ins
immediately: an `Ok` value holding a message if
there aren't any messages this time. Using `try_`
work to do while waiting for messages: we could
every so often, handles a message if one is avail
for a little while until checking again.

We've used `recv` in this example for simplicity;
main thread to do other than wait for messages
appropriate.

When we run the code in Listing 16-8, we'll see t
thread:

```
Got: hi
```

Perfect!

Channels and Ownership Transference

The ownership rules play a vital role in message
safe, concurrent code. Preventing errors in conc
of thinking about ownership throughout your Ru
to show how channels and ownership work toge
use a `val` value in the spawned thread *after* we
compiling the code in Listing 16-9 to see why thi

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Listing 16-9: Attempting to use `val` after we've s

Here, we try to print `val` after we've sent it dow
this would be a bad idea: once the value has bee
could modify or drop it before we try to use the
thread's modifications could cause errors or une
or nonexistent data. However, Rust gives us an e
Listing 16-9:

```
error[E0382]: use of moved value: `val`
  --> src/main.rs:10:31
   |
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {}", val);
   |                               ^^^ val
   |
   = note: move occurs because `val` has t
does
not implement the `Copy` trait
```

Our concurrency mistake has caused a compile
ownership of its parameter, and when the value
ownership of it. This stops us from accidentally u
the ownership system checks that everything is

Sending Multiple Values and Seeing th

The code in Listing 16-8 compiled and ran, but it
separate threads were talking to each other ove
made some modifications that will prove the co
concurrently: the spawned thread will now send
second between each message.

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_s
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Listing 16-10: Sending multiple messages and pa

This time, the spawned thread has a vector of st
main thread. We iterate over them, sending eacl
each by calling the `thread::sleep` function witl

In the main thread, we're not calling the `recv` fu
we're treating `rx` as an iterator. For each value
channel is closed, iteration will end.

When running the code in Listing 16-10, you sho
1-second pause in between each line:

```
Got: hi
Got: from
Got: the
Got: thread
```

Because we don't have any code that pauses or
thread, we can tell that the main thread is waitir
thread.

Creating Multiple Producers by Clonin

Earlier we mentioned that `mpsc` was an acronyr
consumer. Let's put `mpsc` to use and expand the
multiple threads that all send values to the sam
the transmitting half of the channel, as shown ir

Filename: src/main.rs

```
// --snip--

let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(
    }
});

for received in rx {
    println!("Got: {}", received);
}

// --snip--
```

Listing 16-11: Sending multiple messages from r

This time, before we create the first spawned th end of the channel. This will give us a new sendi spawned thread. We pass the original sending e spawned thread. This gives us two threads, each receiving end of the channel.

When you run the code, your output should lool

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

You might see the values in another order; it de
makes concurrency interesting as well as difficul
`thread::sleep`, giving it various values in the di
nondeterministic and create different output ea

Now that we've looked at how channels work, le
concurrency.

Shared-State Concurrency

Message passing is a fine way of handling concu
Consider this part of the slogan from the Go lan
"communicate by sharing memory."

What would communicating by sharing memory
message-passing enthusiasts not use it and do t

In a way, channels in any programming languag
because once you transfer a value down a chan
value. Shared memory concurrency is like multi
access the same memory location at the same ti
smart pointers made multiple ownership possib
complexity because these different owners nee
ownership rules greatly assist in getting this ma
let's look at mutexes, one of the more common
memory.

Using Mutexes to Allow Access to Data

Mutex is an abbreviation for *mutual exclusion*, as
access some data at any given time. To access th
signal that it wants access by asking to acquire t
structure that is part of the mutex that keeps tra

access to the data. Therefore, the mutex is desc
the locking system.

Mutexes have a reputation for being difficult to
two rules:

- You must attempt to acquire the lock befo
- When you're done with the data that the m
  data so other threads can acquire the lock.

For a real-world metaphor for a mutex, imagine
with only one microphone. Before a panelist ca
they want to use the microphone. When they ge
long as they want to and then hand the microph
requests to speak. If a panelist forgets to hand t
finished with it, no one else is able to speak. If m
microphone goes wrong, the panel won't work a

Management of mutexes can be incredibly trick
people are enthusiastic about channels. Howeve
ownership rules, you can't get locking and unloc

The API of `Mutex<T>`

As an example of how to use a mutex, let's start
context, as shown in Listing 16-12:

Filename: src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Listing 16-12: Exploring the API of `Mutex<T>` in a

As with many types, we create a `Mutex<T>` using
access the data inside the mutex, we use the `lo`
call will block the current thread so it can't do ar

lock.

The call to `lock` would fail if another thread hol
no one would ever be able to get the lock, so we
thread panic if we're in that situation.

After we've acquired the lock, we can treat the re
as a mutable reference to the data inside. The ty
lock before using the value in `m` : `Mutex<i32>` is
lock to be able to use the `i32` value. We can't fo
access the inner `i32` otherwise.

As you might suspect, `Mutex<T>` is a smart poin
returns a smart pointer called `MutexGuard` . This
point at our inner data; the smart pointer also h
releases the lock automatically when a `MutexGu`
at the end of the inner scope in Listing 16-12. As
release the lock and blocking the mutex from be
the lock release happens automatically.

After dropping the lock, we can print the mutex
change the inner `i32` to 6.

Sharing a `Mutex<T>` Between Multiple Threac

Now, let's try to share a value between multiple
10 threads and have them each increment a cou
from 0 to 10. Note that the next few examples w
those errors to learn more about using `Mutex<T`
correctly. Listing 16-13 has our starting example

Filename: src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move ||
            let mut num = counter.lock().u

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock()
}
```

Listing 16-13: Ten threads each increment a cou

We create a `counter` variable to hold an `i32` in 16-12. Next, we create 10 threads by iterating ov `thread::spawn` and give all the threads the sam counter into the thread, acquires a lock on the and then adds 1 to the value in the mutex. Whe `num` will go out of scope and release the lock so

In the main thread, we collect all the join handle call `join` on each handle to make sure all the th thread will acquire the lock and print the result

We hinted that this example wouldn't compile. N

```
error[E0382]: capture of moved value: `cou
  --> src/main.rs:10:27
   |
9  |         let handle = thread::spawn(mo
   |                                    --
here
10 |             let mut num = counter.loc
   |                           ^^^^^^^ val
   |
   = note: move occurs because `counter` h
   which does not implement the `Copy` tra

error[E0382]: use of moved value: `counter
  --> src/main.rs:21:29
   |
9  |         let handle = thread::spawn(mo
   |                                    --
here
...
21 |     println!("Result: {}", *counter.l
   |                             ^^^^^^^ v
   |
   = note: move occurs because `counter` h
   which does not implement the `Copy` tra

error: aborting due to 2 previous errors
```

The error message states that the `counter` valu captured when we call `lock`. That description s not allowed!

Let's figure this out by simplifying the program. `for` loop, let's just make two threads without a the first `for` loop in Listing 16-13 with this code

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwra

        *num += 1;
    });
    handles.push(handle);

    let handle2 = thread::spawn(move || {
        let mut num2 = counter.lock().unwr

        *num2 += 1;
    });
    handles.push(handle2);

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock()
}
```

We make two threads and change the variable r `handle2` and `num2`. When we run the code this following:

```
error[E0382]: capture of moved value: `cou
  --> src/main.rs:16:24
   |
8  |     let handle = thread::spawn(move |
   |                                ------
here
...
16 |         let mut num2 = counter.lock()
   |                        ^^^^^^^ value
   |
   = note: move occurs because `counter` h
   which does not implement the `Copy` tra

error[E0382]: use of moved value: `counter
  --> src/main.rs:26:29
   |
8  |     let handle = thread::spawn(move |
   |                                ------
here
...
26 |     println!("Result: {}", *counter.l
   |                             ^^^^^^^ v
   |
   = note: move occurs because `counter` h
   which does not implement the `Copy` tra

error: aborting due to 2 previous errors
```

Aha! The first error message indicates that `coun` thread associated with `handle`. That move is pr when we try to call `lock` on it and store the res Rust is telling us that we can't move ownership c was hard to see earlier because our threads wer different threads in different iterations of the lo multiple-ownership method we discussed in Cha

Multiple Ownership with Multiple Threads

In Chapter 15, we gave a value multiple owners create a reference counted value. Let's do the sa wrap the `Mutex<T>` in `Rc<T>` in Listing 16-14 an ownership to the thread. Now that we've seen th using the `for` loop, and we'll keep the `move` key

Filename: src/main.rs

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move ||
            let mut num = counter.lock().u

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock()
}
```

Listing 16-14: Attempting to use `Rc<T>` to allow `Mutex<T>`

Once again, we compile and get... different erro

```
error[E0277]: the trait bound `std::rc::Rc
std::marker::Send` is not satisfied in `[c
15:10 counter:std::rc::Rc<std::sync::Mute
  --> src/main.rs:11:22
   |
11 |         let handle = thread::spawn(mo
   |                      ^^^^^^^^^^^^^
`std::rc::Rc<std::sync::Mutex<i32>>`
cannot be sent between threads safely
   |
   = help: within `[closure@src/main.rs:11
counter:std::rc::Rc<std::sync::Mutex<i32>>
`std::marker::Send` is
not implemented for `std::rc::Rc<std::sync
   = note: required because it appears wit
`[closure@src/main.rs:11:36: 15:10
counter:std::rc::Rc<std::sync::Mutex<i32>>
   = note: required by `std::thread::spawn
```

Wow, that error message is very wordy! Here ar the first inline error says

`std::rc::Rc<std::sync::Mutex<i32>>` canno
The reason for this is in the next important part distilled error message says `the trait bound ` about `Send` in the next section: it's one of the tr with threads are meant for use in concurrent sit

Unfortunately, `Rc<T>` is not safe to share across reference count, it adds to the count for each ca count when each clone is dropped. But it doesn' make sure that changes to the count can't be int could lead to wrong counts—subtle bugs that cc value being dropped before we're done with it. \ `Rc<T>` but one that makes changes to the refer

Atomic Reference Counting with `Arc<T>`

Fortunately, `Arc<T>` *is* a type like `Rc<T>` that is The *a* stands for *atomic*, meaning it's an *atomica* an additional kind of concurrency primitive that standard library documentation for `std::sync:` point, you just need to know that atomics work l share across threads.

You might then wonder why all primitive types a types aren't implemented to use `Arc<T>` by def comes with a performance penalty that you only If you're just performing operations on values w run faster if it doesn't have to enforce the guara

Let's return to our example: `Arc<T>` and `Rc<T>` program by changing the `use` line, the call to `n` Listing 16-15 will finally compile and run:

Filename: src/main.rs

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move ||
            let mut num = counter.lock().u

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock()
}
```

Listing 16-15: Using an `Arc<T>` to wrap the `Mut` across multiple threads

This code will print the following:

```
Result: 10
```

We did it! We counted from 0 to 10, which may r teach us a lot about `Mutex<T>` and thread safet structure to do more complicated operations th this strategy, you can divide a calculation into in across threads, and then use a `Mutex<T>` to hav with its part.

Similarities Between `RefCell<T>` / `Rc<T`

You might have noticed that `counter` is immuta reference to the value inside it; this means `Mute` the `Cell` family does. In the same way we used us to mutate contents inside an `Rc<T>`, we use an `Arc<T>`.

Another detail to note is that Rust can't protect y
you use `Mutex<T>` . Recall in Chapter 15 that usi
creating reference cycles, where two `Rc<T>` valu
memory leaks. Similarly, `Mutex<T>` comes with t
occur when an operation needs to lock two reso
acquired one of the locks, causing them to wait
interested in deadlocks, try creating a Rust prog
research deadlock mitigation strategies for mut
implementing them in Rust. The standard library
and `MutexGuard` offers useful information.

We'll round out this chapter by talking about the
can use them with custom types.

Extensible Concurrency with t
Traits

Interestingly, the Rust language has *very* few cor
concurrency feature we've talked about so far in
standard library, not the language. Your options
limited to the language or the standard library; y
features or use those written by others.

However, two concurrency concepts are embed
traits `Sync` and `Send` .

Allowing Transference of Ownership B

The `Send` marker trait indicates that ownership
be transferred between threads. Almost every R
exceptions, including `Rc<T>` : this cannot be `Ser`
value and tried to transfer ownership of the clon
might update the reference count at the same ti
implemented for use in single-threaded situatio
thread-safe performance penalty.

Therefore, Rust's type system and trait bounds e
send an `Rc<T>` value across threads unsafely. W
16-14, we got the error `the trait Send is not`
When we switched to `Arc<T>` , which is `Send` , th

Any type composed entirely of `Send` types is au
Almost all primitive types are `Send` , aside from
Chapter 19.

Allowing Access from Multiple Threads

The `Sync` marker trait indicates that it is safe fo
referenced from multiple threads. In other word
reference to `T`) is `Send` , meaning the reference
Similar to `Send` , primitive types are `Sync` , and t
are `Sync` are also `Sync` .

The smart pointer `Rc<T>` is also not `Sync` for th
The `RefCell<T>` type (which we talked about in
`Cell<T>` types are not `Sync` . The implementati
`RefCell<T>` does at runtime is not thread-safe.
and can be used to share access with multiple th
`Mutex<T>` Between Multiple Threads" section.

Implementing `Send` and `Sync` Manual

Because types that are made up of `Send` and `S
and `Sync` , we don't have to implement those tra
don't even have any methods to implement. The
invariants related to concurrency.

Manually implementing these traits involves imp
talk about using unsafe Rust code in Chapter 19
is that building new concurrent types not made
careful thought to uphold the safety guarantees
information about these guarantees and how to

Summary

This isn't the last you'll see of concurrency in this
use the concepts in this chapter in a more realis
examples discussed here.

As mentioned earlier, because very little of how

the language, many concurrency solutions are ir
more quickly than the standard library, so be su
state-of-the-art crates to use in multithreaded si

The Rust standard library provides channels for
types, such as `Mutex<T>` and `Arc<T>`, that are s
type system and the borrow checker ensure tha
end up with data races or invalid references. On
can rest assured that it will happily run on multi
to-track-down bugs common in other languages
longer a concept to be afraid of: go forth and ma
fearlessly!

Next, we'll talk about idiomatic ways to model pr
your Rust programs get bigger. In addition, we'll
those you might be familiar with from object-ori

Object Oriented Progra of Rust

Object-oriented programming (OOP) is a way of
from Simula in the 1960s. Those objects influen
architecture in which objects pass messages to e
oriented programming in 1967 to describe this ar
definitions describe what OOP is; some definitio
oriented, but other definitions would not. In this
characteristics that are commonly considered ol
characteristics translate to idiomatic Rust. We'll t
object-oriented design pattern in Rust and discu
implementing a solution using some of Rust's st

Characteristics of Object-Orie

There is no consensus in the programming com
language must have to be considered object ori
programming paradigms, including OOP; for exa
came from functional programming in Chapter 1
certain common characteristics, namely objects,
look at what each of those characteristics mean

Objects Contain Data and Behavior

The book *Design Patterns: Elements of Reusable O*
Gamma, Richard Helm, Ralph Johnson, and John
Professional, 1994) colloquially referred to as *Th*
object-oriented design patterns. It defines OOP

> Object-oriented programs are made up of ob
> and the procedures that operate on that data
> called *methods* or *operations*.

Using this definition, Rust is object oriented: stru
blocks provide methods on structs and enums. I
methods aren't *called* objects, they provide the s
Gang of Four's definition of objects.

Encapsulation that Hides Implementat

Another aspect commonly associated with OOP
means that the implementation details of an ob
that object. Therefore, the only way to interact w
code using the object shouldn't be able to reach
data or behavior directly. This enables the progr
object's internals without needing to change the

We discussed how to control encapsulation in C
keyword to decide which modules, types, functio
be public, and by default everything else is priva
struct `AveragedCollection` that has a field cont
struct can also have a field that contains the ave
meaning the average doesn't have to be comput
needs it. In other words, `AveragedCollection` w
us. Listing 17-1 has the definition of the `Average`

Filename: src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

Listing 17-1: An `AveragedCollection` struct that
average of the items in the collection

The struct is marked `pub` so that other code car
remain private. This is important in this case bec
whenever a value is added or removed from the
do this by implementing `add` , `remove` , and `ave`
shown in Listing 17-2:

Filename: src/lib.rs

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            },
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().
        self.average = total as f64 / self
    }
}
```

Listing 17-2: Implementations of the public meth
`AveragedCollection`

The public methods `add` , `remove` , and `average`
instance of `AveragedCollection` . When an item
method or removed using the `remove` method,
private `update_average` method that handles u

We leave the `list` and `average` fields private s
add or remove items to the `list` field directly;

become out of sync when the `list` changes. Th
in the `average` field, allowing external code to r

Because we've encapsulated the implementation
`AveragedCollection`, we can easily change asp
future. For instance, we could use a `HashSet` ins
long as the signatures of the `add`, `remove`, and
same, code using `AveragedCollection` wouldn'
public instead, this wouldn't necessarily be the c
methods for adding and removing items, so the
change if it were modifying `list` directly.

If encapsulation is a required aspect for a langu
then Rust meets that requirement. The option t
code enables encapsulation of implementation

Inheritance as a Type System and as C

Inheritance is a mechanism whereby an object c
definition, thus gaining the parent object's data
define them again.

If a language must have inheritance to be an obj
not one. There is no way to define a struct that i
method implementations. However, if you're us
programming toolbox, you can use other solutio
for reaching for inheritance in the first place.

You choose inheritance for two main reasons. O
implement particular behavior for one type, and
that implementation for a different type. You ca
method implementations instead, which you sa
default implementation of the `summarize` meth
implementing the `Summary` trait would have the
without any further code. This is similar to a par
a method and an inheriting child class also havi
We can also override the default implementatio
implement the `Summary` trait, which is similar to
implementation of a method inherited from a p

The other reason to use inheritance relates to th
to be used in the same places as the parent type
which means that you can substitute multiple o

share certain characteristics.

Polymorphism

To many people, polymorphism is synonymo
a more general concept that refers to code th
types. For inheritance, those types are genera

Rust instead uses generics to abstract over di
bounds to impose constraints on what those
sometimes called *bounded parametric polymo*

Inheritance has recently fallen out of favor as a
programming languages because it's often at ris
necessary. Subclasses shouldn't always share all
but will do so with inheritance. This can make a
introduces the possibility of calling methods on
that cause errors because the methods don't ap
languages will only allow a subclass to inherit fr
flexibility of a program's design.

For these reasons, Rust takes a different approa
inheritance. Let's look at how trait objects enabl

Using Trait Objects that Allow Types

In Chapter 8, we mentioned that one limitation
elements of only one type. We created a workar
a `SpreadsheetCell` enum that had variants to h
meant we could store different types of data in
represented a row of cells. This is a perfectly go
items are a fixed set of types that we know whe

However, sometimes we want our library user t
that are valid in a particular situation. To show h
an example graphical user interface (GUI) tool th
calling a `draw` method on each one to draw it to
for GUI tools. We'll create a library crate called

GUI library. This crate might include some types
`TextField`. In addition, `gui` users will want to c
drawn: for instance, one programmer might add
`SelectBox`.

We won't implement a fully fledged GUI library f
pieces would fit together. At the time of writing t
all the types other programmers might want to c
needs to keep track of many values of different t
method on each of these differently typed value
what will happen when we call the `draw` metho
method available for us to call.

To do this in a language with inheritance, we mig
that has a method named `draw` on it. The other
`SelectBox`, would inherit from `Component` and
could each override the `draw` method to define
framework could treat all of the types as if they w
`draw` on them. But because Rust doesn't have in
structure the `gui` library to allow users to exter

Defining a Trait for Common Behavior

To implement the behavior we want `gui` to hav
that will have one method named `draw`. Then w
object. A trait object points to an instance of a ty
specify. We create a trait object by specifying so
reference or a `Box<T>` smart pointer, and then s
`dyn` keyword. (We'll talk about the reason trait o
19 in the section "Dynamically Sized Types & Size
of a generic or concrete type. Wherever we use a
ensure at compile time that any value used in th
object's trait. Consequently, we don't need to kn
time.

We've mentioned that in Rust, we refrain from ca
distinguish them from other languages' objects.
struct fields and the behavior in `impl` blocks are
languages, the data and behavior combined into
object. However, trait objects *are* more like obje
that they combine data and behavior. But trait o
in that we can't add data to a trait object. Trait o

}

objects in other languages: their specific purpos
common behavior.

Listing 17-3 shows how to define a trait named

Filename: src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

Listing 17-3: Definition of the `Draw` trait

This syntax should look familiar from our discus
Chapter 10. Next comes some new syntax: Listir
`Screen` that holds a vector named `components`
, which is a trait object; it's a stand-in for any typ
`Draw` trait.

Filename: src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

Listing 17-4: Definition of the `Screen` struct witl
of trait objects that implement the `Draw` trait

On the `Screen` struct, we'll define a method nar
method on each of its `components`, as shown in

Filename: src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.i
            component.draw();
        }
    }
}
```

Listing 17-5: A `run` method on `Screen` that calls
component

This works differently than defining a struct that

trait bounds. A generic type parameter can only
type at a time, whereas trait objects allow for mu
trait object at runtime. For example, we could ha
generic type and a trait bound as in Listing 17-6:

Filename: src/lib.rs

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
    where T: Draw {
    pub fn run(&self) {
        for component in self.components.i
            component.draw();
        }
    }
}
```

Listing 17-6: An alternate implementation of the
using generics and trait bounds

This restricts us to a `Screen` instance that has a
or all of type `TextField`. If you'll only ever have
generics and trait bounds is preferable because
monomorphized at compile time to use the con

On the other hand, with the method using trait o
hold a `Vec` that contains a `Box<Button>` as well
how this works, and then we'll talk about the run

Implementing the Trait

Now we'll add some types that implement the `D`
type. Again, actually implementing a GUI library
the `draw` method won't have any useful implem
the implementation might look like, a `Button` st
`height`, and `label`, as shown in Listing 17-7:

Filename: src/lib.rs

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

Listing 17-7: A `Button` struct that implements th

The `width`, `height`, and `label` fields on `Buttc` components, such as a `TextField` type, that mi `placeholder` field instead. Each of the types we implement the `Draw` trait but will use different how to draw that particular type, as `Button` has which is beyond the scope of this chapter). The an additional `impl` block containing methods re clicks the button. These kinds of methods won't

If someone using our library decides to impleme `width`, `height`, and `options` fields, they imple `SelectBox` type as well, as shown in Listing 17-8

Filename: src/main.rs

```
extern crate gui;
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select
    }
}
```

Listing 17-8: Another crate using `gui` and imple `SelectBox` struct

Our library's user can now write their `main` func the `Screen` instance, they can add a `SelectBox` `Box<T>` to become a trait object. They can then instance, which will call `draw` on each of the cor implementation:

Filename: src/main.rs

```
use gui::{Screen, Button};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No")
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

Listing 17-9: Using trait objects to store values o same trait

When we wrote the library, we didn't know that type, but our `Screen` implementation was able it because `SelectBox` implements the `Draw` typ `draw` method.

This concept—of being concerned only with the than the value's concrete type—is similar to the typed languages: if it walks like a duck and quac In the implementation of `run` on `Screen` in List what the concrete type of each component is. It is an instance of a `Button` or a `SelectBox`, it ju component. By specifying `Box<dyn Draw>` as the

`components` vector, we've defined `Screen` to ne
method on.

The advantage of using trait objects and Rust's t
code using duck typing is that we never have to
particular method at runtime or worry about ge
implement a method but we call it anyway. Rust
don't implement the traits that the trait objects r

For example, Listing 17-10 shows what happens
`String` as a component:

Filename: src/main.rs

```
extern crate gui;
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(String::from("Hi")),
        ],
    };

    screen.run();
}
```

Listing 17-10: Attempting to use a type that does

We'll get this error because `String` doesn't imp

```
error[E0277]: the trait bound `std::string
satisfied
  --> src/main.rs:7:13
   |
 7 |             Box::new(String::from("Hi
   |             ^^^^^^^^^^^^^^^^^^^^^^^^
   implemented for `std::string::String`
   |
   = note: required for the cast to the ob
```

This error lets us know that either we're passing
to pass and we should pass a different type or w
`String` so that `Screen` is able to call `draw` on i

Trait Objects Perform Dynamic Dispat

Recall in the “Performance of Code Using Gener
discussion on the monomorphization process p
use trait bounds on generics: the compiler gene
functions and methods for each concrete type t
parameter. The code that results from monomo
which is when the compiler knows what methoc
opposed to *dynamic dispatch*, which is when the
which method you’re calling. In dynamic dispatc
at runtime will figure out which method to call.

When we use trait objects, Rust must use dynam
know all the types that might be used with the c
doesn’t know which method implemented on wl
Rust uses the pointers inside the trait object to k
runtime cost when this lookup happens that doe
Dynamic dispatch also prevents the compiler fro
code, which in turn prevents some optimization
in the code that we wrote in Listing 17-5 and we
it’s a trade-off to consider.

Object Safety Is Required for Trait Obj

You can only make *object-safe* traits into trait ob
the properties that make a trait object safe, but
relevant. A trait is object safe if all the methods
properties:

- The return type isn’t `Self`.
- There are no generic type parameters.

The `Self` keyword is an alias for the type we’re
on. Trait objects must be object safe because or
longer knows the concrete type that’s implemen
returns the concrete `Self` type, but a trait obje
there is no way the method can use the original
generic type parameters that are filled in with co
trait is used: the concrete types become part of
When the type is forgotten through the use of a
what types to fill in the generic type parameters

An example of a trait whose methods are not ob
`Clone` trait. The signature for the `clone` metho

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

The `String` type implements the `Clone` trait, a
on an instance of `String` we get back an instan
`clone` on an instance of `Vec`, we get back an in
`clone` needs to know what type will stand in for
type.

The compiler will indicate when you're trying to
of object safety in regard to trait objects. For exa
the `Screen` struct in Listing 17-4 to hold types th
of the `Draw` trait, like this:

```
pub struct Screen {
    pub components: Vec<Box<dyn Clone>>,
}
```

We would get this error:

```
error[E0038]: the trait `std::clone::Clone
 --> src/lib.rs:2:5
  |
2 |     pub components: Vec<Box<dyn Clone>
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ th
cannot be
made into an object
  |
  = note: the trait cannot require that `S
```

This error means you can't use this trait as a trai
interested in more details on object safety, see R

Implementing an Object-Orien

The *state pattern* is an object-oriented design pa
value has some internal state, which is represen
value's behavior changes based on the internal s
functionality: in Rust, of course, we use structs a
inheritance. Each state object is responsible for
when it should change into another state. The va
nothing about the different behavior of the state

states.

Using the state pattern means when the busine change, we won't need to change the code of th that uses the value. We'll only need to update th to change its rules or perhaps add more state ol state design pattern and how to use it in Rust.

We'll implement a blog post workflow in an incre functionality will look like this:

1. A blog post starts as an empty draft.
2. When the draft is done, a review of the pos
3. When the post is approved, it gets publish
4. Only published blog posts return content t accidentally be published.

Any other changes attempted on a post should l to approve a draft blog post before we've reque an unpublished draft.

Listing 17-11 shows this workflow in code form: we'll implement in a library crate named `blog`. haven't implemented the `blog` crate yet.

Filename: src/main.rs

```
extern crate blog;
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch tc
}
```

Listing 17-11: Code that demonstrates the desir to have

We want to allow the user to create a new draft want to allow text to be added to the blog post v

get the post’s content immediately, before appr
the post is still a draft. We’ve added `assert_eq!`
purposes. An excellent unit test for this would b
returns an empty string from the `content` meth
for this example.

Next, we want to enable a request for a review c
return an empty string while waiting for the revi
it should get published, meaning the text of the
is called.

Notice that the only type we’re interacting with f
type will use the state pattern and will hold a val
objects representing the various states a post ca
published. Changing from one state to another v
`Post` type. The states change in response to the
on the `Post` instance, but they don’t have to ma
users can’t make a mistake with the states, like p

Defining `Post` and Creating a New Inst

Let’s get started on the implementation of the lil
`Post` struct that holds some content, so we’ll sta
and an associated public `new` function to create
Listing 17-12. We’ll also make a private `State` tr
of `Box<dyn State>` inside an `Option<T>` in a pr
why the `Option<T>` is necessary in a bit.

Filename: src/lib.rs

```rust
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

Listing 17-12: Definition of a `Post` struct and a
instance, a `State` trait, and a `Draft` struct

The `State` trait defines the behavior shared by
`PendingReview`, and `Published` states will all ir
the trait doesn't have any methods, and we'll sta
because that is the state we want a post to start

When we create a new `Post`, we set its `state` f
This `Box` points to a new instance of the `Draft`
create a new instance of `Post`, it will start out a
`Post` is private, there is no way to create a `Post`
function, we set the `content` field to a new, em

Storing the Text of the Post Content

Listing 17-11 showed that we want to be able to
pass it a `&str` that is then added to the text cor
this as a method rather than exposing the `cont`
implement a method later that will control how
`add_text` method is pretty straightforward, so l
17-13 to the `impl Post` block:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn add_text(&mut self, text: &str)
        self.content.push_str(text);
    }
}
```

Listing 17-13: Implementing the `add_text` meth

The `add_text` method takes a mutable referen the `Post` instance that we're calling `add_text` o `String` in `content` and pass the `text` argume behavior doesn't depend on the state the post is pattern. The `add_text` method doesn't interact part of the behavior we want to support.

Ensuring the Content of a Draft Post Is

Even after we've called `add_text` and added so the `content` method to return an empty string draft state, as shown on line 8 of Listing 17-11. F method with the simplest thing that will fulfill th empty string slice. We'll change this later once w post's state so it can be published. So far, posts post content should always be empty. Listing 17 implementation:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        ""
    }
}
```

Listing 17-14: Adding a placeholder implementat `Post` that always returns an empty string slice

With this added `content` method, everything in intended.

Requesting a Review of the Post Chang

Next, we need to add functionality to request a r
its state from `Draft` to `PendingReview`. Listing

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take()
            self.state = Some(s.request_re
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) ->
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) ->
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) ->
        self
    }
}
```

Listing 17-15: Implementing `request_review` m

We give `Post` a public method named `request_`
reference to `self`. Then we call an internal `req`
state of `Post`, and this second `request_review`
and returns a new state.

We've added the `request_review` method to th
implement the trait will now need to implement
that rather than having `self`, `&self`, or `&mut s`
method, we have `self: Box<Self>`. This syntax
called on a `Box` holding the type. This syntax tal

invalidating the old state so the state value of th
state.

To consume the old state, the `request_review`
the state value. This is where the `Option` in the
the `take` method to take the `Some` value out of
its place, because Rust doesn't let us have unpo
move the `state` value out of `Post` rather than
`state` value to the result of this operation.

We need to set `state` to `None` temporarily rath
like `self.state = self.state.request_review`
value. This ensures `Post` can't use the old `stat`
into a new state.

The `request_review` method on `Draft` needs t
new `PendingReview` struct, which represents th
review. The `PendingReview` struct also impleme
doesn't do any transformations. Rather, it return
review on a post already in the `PendingReview`
`PendingReview` state.

Now we can start seeing the advantages of the s
method on `Post` is the same no matter its `stat`
its own rules.

We'll leave the `content` method on `Post` as is,
can now have a `Post` in the `PendingReview` sta
we want the same behavior in the `PendingRevi`
to line 11!

Adding the `approve` Method that Chan

The `approve` method will be similar to the `requ`
to the value that the current state says it should
shown in Listing 17-16:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take()
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) ->
    fn approve(self: Box<Self>) -> Box<dyn
}

struct Draft {}

impl State for Draft {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) ->
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn
        self
    }
}
```

Listing 17-16: Implementing the `approve` metho

We add the `approve` method to the `State` trait implements `State`, the `Published` state.

Similar to `request_review`, if we call the `approv`

effect because it will return `self` . When we call
returns a new, boxed instance of the `Published`
implements the `State` trait, and for both the `r`
`approve` method, it returns itself, because the p
state in those cases.

Now we need to update the `content` method o
want to return the value in the post's `content` f
empty string slice, as shown in Listing 17-17:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().conte
    }
    // --snip--
}
```

Listing 17-17: Updating the `content` method on
method on `State`

Because the goal is to keep all these rules inside
we call a `content` method on the value in `stat`
`self`) as an argument. Then we return the valu
`content` method on the `state` value.

We call the `as_ref` method on the `Option` beca
inside the `Option` rather than ownership of the
`Option<Box<dyn State>>` , when we call `as_ref`
returned. If we didn't call `as_ref` , we would get
`state` out of the borrowed `&self` of the functi

We then call the `unwrap` method, which we kno
the methods on `Post` ensure that `state` will al
those methods are done. This is one of the case
You Have More Information Than the Compiler"
that a `None` value is never possible, even thoug
understand that.

At this point, when we call `content` on the `&Bo`
effect on the `&` and the `Box` so the `content` m
type that implements the `State` trait. That mea

`State` trait definition, and that is where we'll pu
depending on which state we have, as shown in

Filename: src/lib.rs

```
trait State {
    // --snip--
    fn content<'a>(&self, post: &'a Post)
        ""
    }
}

// --snip--
struct Published {}

impl State for Published {
    // --snip--
    fn content<'a>(&self, post: &'a Post)
        &post.content
    }
}
```

Listing 17-18: Adding the `content` method to th

We add a default implementation for the `conte`
string slice. That means we don't need to impler
`PendingReview` structs. The `Published` struct w
return the value in `post.content`.

Note that we need lifetime annotations on this r
10. We're taking a reference to a `post` as an arg
part of that `post`, so the lifetime of the returne
the `post` argument.

And we're done—all of Listing 17-11 now works!
with the rules of the blog post workflow. The log
state objects rather than being scattered throug

Trade-offs of the State Pattern

We've shown that Rust is capable of implementi
to encapsulate the different kinds of behavior a
methods on `Post` know nothing about the vario
the code, we have to look in only one place to kr
post can behave: the implementation of the `Sta`

If we were to create an alternative implementati
we might instead use `match` expressions in the
`main` code that checks the state of the post and
That would mean we would have to look in seve
implications of a post being in the published sta
states we added: each of those `match` expressic

With the state pattern, the `Post` methods and t
`match` expressions, and to add a new state, we
and implement the trait methods on that one st

The implementation using the state pattern is e
functionality. To see the simplicity of maintainin
a few of these suggestions:

- Add a `reject` method that changes the pc
  to `Draft`.
- Require two calls to `approve` before the st
- Allow users to add text content only when
  have the state object responsible for what
  not responsible for modifying the `Post`.

One downside of the state pattern is that, becau
transitions between states, some of the states a
another state between `PendingReview` and `Pub`
would have to change the code in `PendingRevie`
It would be less work if `PendingReview` didn't ne
new state, but that would mean switching to an

Another downside is that we've duplicated some
duplication, we might try to make default implem
and `approve` methods on the `State` trait that r
violate object safety, because the trait doesn't kr
exactly. We want to be able to use `State` as a tr
be object safe.

Other duplication includes the similar implemer
`approve` methods on `Post`. Both methods dele
same method on the value in the `state` field of
`state` field to the result. If we had a lot of meth
pattern, we might consider defining a macro to
D for more on macros).

By implementing the state pattern exactly as it's

languages, we're not taking as full advantage of
look at some changes we can make to the `blog`
and transitions into compile time errors.

Encoding States and Behavior as Types

We'll show you how to rethink the state pattern
Rather than encapsulating the states and transit
no knowledge of them, we'll encode the states in
Rust's type checking system will prevent attemp
published posts are allowed by issuing a compil

Let's consider the first part of `main` in Listing 17

Filename: src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch
    assert_eq!("", post.content());
}
```

We still enable the creation of new posts in the c
ability to add text to the post's content. But inste
draft post that returns an empty string, we'll ma
`content` method at all. That way, if we try to get
compiler error telling us the method doesn't exis
us to accidentally display draft post content in p
even compile. Listing 17-19 shows the definition
struct, as well as methods on each:

Filename: src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str)
        self.content.push_str(text);
    }
}
```

Listing 17-19: A `Post` with a `content` method a
method

Both the `Post` and `DraftPost` structs have a pr
blog post text. The structs no longer have the `s`
encoding of the state to the types of the structs.
published post, and it has a `content` method th

We still have a `Post::new` function, but instead
returns an instance of `DraftPost`. Because `con`
functions that return `Post`, it's not possible to c

The `DraftPost` struct has an `add_text` methoc
before, but note that `DraftPost` does not have
the program ensures all posts start as draft post
content available for display. Any attempt to get
in a compiler error.

Implementing Transitions as Transformation

So how do we get a published post? We want to

to be reviewed and approved before it can be pu
state should still not display any content. Let's in
another struct, `PendingReviewPost`, defining th
`DraftPost` to return a `PendingReviewPost`, and
`PendingReviewPost` to return a `Post`, as shown

Filename: src/lib.rs

```
impl DraftPost {
    // --snip--

    pub fn request_review(self) -> Pending
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}
```

Listing 17-20: A `PendingReviewPost` that gets cr
`DraftPost` and an `approve` method that turns
published `Post`

The `request_review` and `approve` methods tak
consuming the `DraftPost` and `PendingReviewP`
into a `PendingReviewPost` and a published `Pos`
have any lingering `DraftPost` instances after w
and so forth. The `PendingReviewPost` struct do
on it, so attempting to read its content results in
Because the only way to get a published `Post` i
method defined is to call the `approve` method o
way to get a `PendingReviewPost` is to call the `r`
`DraftPost`, we've now encoded the blog post w

But we also have to make some small changes t

`approve` methods return new instances rather t
called on, so we need to add more `let post =`
returned instances. We also can't have the asser
review post's contents be empty strings, nor do
that tries to use the content of posts in those sta
`main` is shown in Listing 17-21:

Filename: src/main.rs

```
extern crate blog;
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch to
}
```

Listing 17-21: Modifications to `main` to use the r
workflow

The changes we needed to make to `main` to rea
implementation doesn't quite follow the object-o
transformations between the states are no long
`Post` implementation. However, our gain is that
because of the type system and the type checkin
ensures that certain bugs, such as display of the
be discovered before they make it to production

Try the tasks suggested for additional requireme
this section on the `blog` crate as it is after Listin
the design of this version of the code. Note that
completed already in this design.

We've seen that even though Rust is capable of i
patterns, other patterns, such as encoding state
available in Rust. These patterns have different t
familiar with object-oriented patterns, rethinkin
Rust's features can provide benefits, such as pre
Object-oriented patterns won't always be the be
features, like ownership, that object-oriented lan

Summary

No matter whether or not you think Rust is an o this chapter, you now know that you can use tra features in Rust. Dynamic dispatch can give you a bit of runtime performance. You can use this f patterns that can help your code's maintainabili ownership, that object-oriented languages don't won't always be the best way to take advantage option.

Next, we'll look at patterns, which are another o flexibility. We've looked at them briefly througho capability yet. Let's go!

Patterns and Matching

Patterns are a special syntax in Rust for matchin complex and simple. Using patterns in conjuncti constructs gives you more control over a progra some combination of the following:

- Literals
- Destructured arrays, enums, structs, or tup
- Variables
- Wildcards
- Placeholders

These components describe the shape of the da match against values to determine whether our continue running a particular piece of code.

To use a pattern, we compare it to some value. I use the value parts in our code. Recall the `matcl` patterns, such as the coin-sorting machine exam pattern, we can use the named pieces. If it does pattern won't run.

This chapter is a reference on all things related t places to use patterns, the difference between r the different kinds of pattern syntax that you mi you'll know how to use patterns to express man

All the Places Patterns Can Be

Patterns pop up in a number of places in Rust, a
without realizing it! This section discusses all the

`match` Arms

As discussed in Chapter 6, we use patterns in th
Formally, `match` expressions are defined as the
and one or more match arms that consist of a p
value matches that arm's pattern, like this:

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

One requirement for `match` expressions is that
sense that all possibilities for the value in the `ma`
for. One way to ensure you've covered every pos
for the last arm: for example, a variable name m
thus covers every remaining case.

A particular pattern `_` will match anything, but i
often used in the last match arm. The `_` patterr
ignore any value not specified, for example. We'
in the "Ignoring Values in a Pattern" section later

Conditional `if let` Expressions

In Chapter 6 we discussed how to use `if let` e
write the equivalent of a `match` that only match
have a corresponding `else` containing code to r
doesn't match.

Listing 18-1 shows that it's also possible to mix a
`else if let` expressions. Doing so gives us mo
in which we can express only one value to comp
conditions in a series of `if let`, `else if`, `else`
relate to each other.

The code in Listing 18-1 shows a series of checks
what the background color should be. For this e
hardcoded values that a real program might rec

Filename: src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = Nor
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite colo
color);
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the
        } else {
            println!("Using orange as the
        }
    } else {
        println!("Using blue as the backgr
    }
}
```

Listing 18-1: Mixing `if let`, `else if`, `else if`

If the user specifies a favorite color, that color is
Tuesday, the background color is green. If the us
we can parse it as a number successfully, the co
depending on the value of the number. If none o
background color is blue.

This conditional structure lets us support compl
values we have here, this example will print `Usi`
.

You can see that `if let` can also introduce sha
`match` arms can: the line `if let Ok(age) = ag`
variable that contains the value inside the `Ok` va
the `if age > 30` condition within that block: we
into `if let Ok(age) = age && age > 30`. The s
to 30 isn't valid until the new scope starts with th

The downside of using `if let` expressions is th
exhaustiveness, whereas with `match` expression

block and therefore missed handling some case
the possible logic bug.

`while let` Conditional Loops

Similar in construction to `if let`, the `while le`
loop to run for as long as a pattern continues to
shows a `while let` loop that uses a vector as a
vector in the opposite order in which they were

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

Listing 18-2: Using a `while let` loop to print val
returns `Some`

This example prints 3, 2, and then 1. The `pop` m
the vector and returns `Some(value)`. If the vect
`while` loop continues running the code in its bl
When `pop` returns `None`, the loop stops. We ca
element off our stack.

`for` Loops

In Chapter 3, we mentioned that the `for` loop is
in Rust code, but we haven't yet discussed the p
the pattern is the value that directly follows the
`x` is the pattern.

Listing 18-3 demonstrates how to use a pattern
apart, a tuple as part of the `for` loop.

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate()
    println!("{} is at index {}", value, i
}
```

Listing 18-3: Using a pattern in a `for` loop to de

The code in Listing 18-3 will print the following:

```
a is at index 0
b is at index 1
c is at index 2
```

We use the `enumerate` method to adapt an iter value's index in the iterator, placed into a tuple. the tuple `(0, 'a')`. When this value is matchec `index` will be `0` and `value` will be `'a'`, printin

`let` Statements

Prior to this chapter, we had only explicitly discu `if let`, but in fact, we've used patterns in othe statements. For example, consider this straightf :

```
let x = 5;
```

Throughout this book, we've used `let` like this l might not have realized it, you were using patte looks like this:

```
let PATTERN = EXPRESSION;
```

In statements like `let x = 5;` with a variable n name is just a particularly simple form of a patte against the pattern and assigns any names it fin is a pattern that means "bind what matches her name `x` is the whole pattern, this pattern effect variable `x`, whatever the value is."

To see the pattern matching aspect of `let` mor uses a pattern with `let` to destructure a tuple.

```
let (x, y, z) = (1, 2, 3);
```

Listing 18-4: Using a pattern to destructure a tup

Here, we match a tuple against a pattern. Rust c pattern `(x, y, z)` and sees that the value mat `x`, `2` to `y`, and `3` to `z`. You can think of this tu individual variable patterns inside it.

If the number of elements in the pattern doesn't the tuple, the overall type won't match and we'll Listing 18-5 shows an attempt to destructure a t variables, which won't work.

```
let (x, y) = (1, 2, 3);
```

Listing 18-5: Incorrectly constructing a pattern w number of elements in the tuple

Attempting to compile this code results in this ty

```
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
2 |     let (x, y) = (1, 2, 3);
  |         ^^^^^^ expected a tuple with 3
elements
  |
  = note: expected type `({integer}, {inte
             found type `(_, _)`
```

If we wanted to ignore one or more of the value as you'll see in the "Ignoring Values in a Pattern" have too many variables in the pattern, the solu removing variables so the number of variables e tuple.

Function Parameters

Function parameters can also be patterns. The c function named `foo` that takes one parameter look familiar.

```
fn foo(x: i32) {
    // code goes here
}
```

Listing 18-6: A function signature uses patterns i

The `x` part is a pattern! As we did with `let`, we arguments to the pattern. Listing 18-7 splits the function.

Filename: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32))
    println!("Current location: ({}, {})",
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

Listing 18-7: A function with parameters that de

This code prints `Current location: (3, 5)`. Th `&(x, y)`, so `x` is the value `3` and `y` is the valu

We can also use patterns in closure parameter li parameter lists, because closures are similar to

At this point, you've seen several ways of using p same in every place we can use them. In some p irrefutable; in other circumstances, they can be concepts next.

Refutability: Whether a Patte

Patterns come in two forms: refutable and irrefu possible value passed are *irrefutable*. An exampl `let x = 5;` because `x` matches anything and t Patterns that can fail to match for some possible would be `Some(x)` in the expression `if let So` in the `a_value` variable is `None` rather than `So` match.

Function parameters, `let` statements, and `for`
patterns, because the program cannot do anyth
match. The `if let` and `while let` expressions
because by definition they're intended to handle
conditional is in its ability to perform differently

In general, you shouldn't have to worry about th
irrefutable patterns; however, you do need to be
refutability so you can respond when you see it
you'll need to change either the pattern or the c
depending on the intended behavior of the code

Let's look at an example of what happens when
where Rust requires an irrefutable pattern and v
statement, but for the pattern we've specified `S`
might expect, this code will not compile.

```
let Some(x) = some_option_value;
```

Listing 18-8: Attempting to use a refutable patte

If `some_option_value` was a `None` value, it wou
meaning the pattern is refutable. However, the
irrefutable pattern because there is nothing vali
At compile time, Rust will complain that we've tr
an irrefutable pattern is required:

```
error[E0005]: refutable pattern in local b
 -->
  |
3 | let Some(x) = some_option_value;
  |     ^^^^^^^ pattern `None` not coverec
```

Because we didn't cover (and couldn't cover!) eve
`Some(x)`, Rust rightfully produces a compiler er

To fix the problem where we have a refutable pa
needed, we can change the code that uses the p
use `if let`. Then if the pattern doesn't match, t
curly brackets, giving it a way to continue validly
code in Listing 18-8.

```
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

Listing 18-9: Using `if let` and a block with refu

We've given the code an out! This code is perfec use an irrefutable pattern without receiving an e will always match, such as `x`, as shown in Listin

```
if let x = 5 {
    println!("{}", x);
};
```

Listing 18-10: Attempting to use an irrefutable p

Rust complains that it doesn't make sense to us

```
error[E0162]: irrefutable if-let pattern
 --> <anon>:2:8
  |
2 | if let x = 5 {
  |        ^ irrefutable pattern
```

For this reason, match arms must use refutable which should match any remaining values with a to use an irrefutable pattern in a `match` with on particularly useful and could be replaced with a

Now that you know where to use patterns and t irrefutable patterns, let's cover all the syntax we

Pattern Syntax

Throughout the book, you've seen examples of r we gather all the syntax valid in patterns and dis one.

Matching Literals

As you saw in Chapter 6, you can match pattern following code gives some examples:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This code prints `one` because the value in `x` is
your code to take an action if it gets a particular

Matching Named Variables

Named variables are irrefutable patterns that m
many times in the book. However, there is a con
variables in `match` expressions. Because `match`
declared as part of a pattern inside the `match` e
same name outside the `match` construct, as is t
18-11, we declare a variable named `x` with the
the value `10`. We then create a `match` expressi
patterns in the match arms and `println!` at th
code will print before running this code or readi

Filename: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y =
        _ => println!("Default case, x =
    }

    println!("at the end: x = {:?}, y = {
}
```

Listing 18-11: A `match` expression with an arm t
`y`

Let's walk through what happens when the `matc` first match arm doesn't match the defined value

The pattern in the second match arm introduce match any value inside a `Some` value. Because w expression, this is a new `y` variable, not the `y` v value 10. This new `y` binding will match any valu have in `x`. Therefore, this new `y` binds to the ir value is `5`, so the expression for that arm execu

If `x` had been a `None` value instead of `Some(5)` wouldn't have matched, so the value would have didn't introduce the `x` variable in the pattern of expression is still the outer `x` that hasn't been s the `match` would print `Default case, x = Non`

When the `match` expression is done, its scope e inner `y`. The last `println!` produces `at the e`

To create a `match` expression that compares th than introducing a shadowed variable, we would conditional instead. We'll talk about match guard Match Guards" section.

Multiple Patterns

In `match` expressions, you can match multiple p means *or*. For example, the following code matc arms, the first of which has an *or* option, meanir the values in that arm, that arm's code will run:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This code prints `one or two`.

Matching Ranges of Values with `...`

The ... syntax allows us to match to an inclusi
code, when a pattern matches any of the values
execute:

```
let x = 5;

match x {
    1 ... 5 => println!("one through five'
    _ => println!("something else"),
}
```

If x is 1, 2, 3, 4, or 5, the first arm will match. Th
using the | operator to express the same idea;
to specify 1 | 2 | 3 | 4 | 5 if we used | . Sp
especially if we want to match, say, any number

Ranges are only allowed with numeric values or
checks that the range isn't empty at compile tim
tell if a range is empty or not are char and num

Here is an example using ranges of char values

```
let x = 'c';

match x {
    'a' ... 'j' => println!("early ASCII l
    'k' ... 'z' => println!("late ASCII le
    _ => println!("something else"),
}
```

Rust can tell that c is within the first pattern's ra

Destructuring to Break Apart Values

We can also use patterns to destructure structs,
different parts of these values. Let's walk throug

Destructuring Structs

Listing 18-12 shows a Point struct with two fiel
using a pattern with a let statement.

Filename: src/main.rs

```rust
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

Listing 18-12: Destructuring a struct's fields into

This code creates the variables `a` and `b` that m of the `p` variable. This example shows that the r don't have to match the field names of the struc variable names to match the field names to mak variables came from which fields.

Because having variable names match the fields `let Point { x: x, y: y } = p;` contains a lot for patterns that match struct fields: you only ne and the variables created from the pattern will h shows code that behaves in the same way as the variables created in the `let` pattern are `x` and

Filename: src/main.rs

```rust
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

Listing 18-13: Destructuring struct fields using st

This code creates the variables `x` and `y` that m variable. The outcome is that the variables `x` ar struct.

We can also destructure with literal values as pa
creating variables for all the fields. Doing so allo
particular values while creating variables to dest

Listing 18-14 shows a `match` expression that se
cases: points that lie directly on the `x` axis (whic
(`x = 0`), or neither.

Filename: src/main.rs

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On
        Point { x: 0, y } => println!("On
        Point { x, y } => println!("On ne
    }
}
```

Listing 18-14: Destructuring and matching literal

The first arm will match any point that lies on th
matches if its value matches the literal `0` . The p
we can use in the code for this arm.

Similarly, the second arm matches any point on
field matches if its value is `0` and creates a vari
The third arm doesn't specify any literals, so it m
variables for both the `x` and `y` fields.

In this example, the value `p` matches the secon
so this code will print `On the y axis at 7` .

Destructuring Enums

We've destructured enums earlier in this book, f
`Option<i32>` in Listing 6-5 in Chapter 6. One de
that the pattern to destructure an enum should
stored within the enum is defined. As an examp
`Message` enum from Listing 6-2 and write a `mat`
each inner value.

Filename: src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160,

    match msg {
        Message::Quit => {
            println!("The Quit variant has
        },
        Message::Move { x, y } => {
            println!(
                "Move in the x direction {
                x,
                y
            );
        }
        Message::Write(text) => println!("
        Message::ChangeColor(r, g, b) => {
            println!(
                "Change the color to red {
                r,
                g,
                b
            )
        }
    }
}
```

Listing 18-15: Destructuring enum variants that

This code will print `Change the color to red 0` changing the value of `msg` to see the code from

For enum variants without any data, like `Messag` value any further. We can only match on the lite variables are in that pattern.

For struct-like enum variants, such as `Message:` the pattern we specify to match structs. After th brackets and then list the fields with variables sc the code for this arm. Here we use the shorthan

For tuple-like enum variants, like `Message::Writ` element and `Message::ChangeColor` that holds

pattern is similar to the pattern we specify to ma
in the pattern must match the number of eleme

Destructuring Nested Structs & Enums

Up until now, all of our examples have been ma
deep. Matching can work on nested structures t

We can refactor the example above to support b

```
enum Color {
   Rgb(i32, i32, i32),
   Hsv(i32, i32, i32)
}

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(Color),
}

fn main() {
    let msg = Message::ChangeColor(Color::

    match msg {
        Message::ChangeColor(Color::Rgb(r,
            println!(
                "Change the color to red {
                r,
                g,
                b
            )
        },
        Message::ChangeColor(Color::Hsv(h,
            println!(
                "Change the color to hue {
                h,
                s,
                v
            )
        }
        _ => ()
    }
}
```

Destructuring References

When the value we're matching to our pattern c

destructure the reference from the value, which
pattern. Doing so lets us get a variable holding t
rather than getting a variable that holds the refe
useful in closures where we have iterators that i
to use the values in the closure rather than the r

The example in Listing 18-16 iterates over refere
destructuring the reference and the struct so we
and `y` values easily.

```
let points = vec![
    Point { x: 0, y: 0 },
    Point { x: 1, y: 5 },
    Point { x: 10, y: -3 },
];

let sum_of_squares: i32 = points
    .iter()
    .map(|&Point { x, y }| x * x + y * y)
    .sum();
```

Listing 18-16: Destructuring a reference to a stru

This code gives us the variable `sum_of_squares`
result of squaring the `x` value and the `y` value,
adding the result for each `Point` in the `points`

If we had not included the `&` in `&Point { x, y`
because `iter` would then iterate over reference
than the actual values. The error would look like

```
error[E0308]: mismatched types
  -->
   |
14 |         .map(|Point { x, y }| x * x +
   |               ^^^^^^^^^^^^ expected &
   |
   = note: expected type `&Point`
              found type `Point`
```

This error indicates that Rust was expecting our
tried to match directly to a `Point` value, not a re

Destructuring Structs and Tuples

We can mix, match, and nest destructuring patte

following example shows a complicated destruc
inside a tuple and destructure all the primitive v

```
let ((feet, inches), Point {x, y}) = ((3,
```

This code lets us break complex types into their
values we're interested in separately.

Destructuring with patterns is a convenient way
value from each field in a struct, separately from

Ignoring Values in a Pattern

You've seen that it's sometimes useful to ignore
arm of a `match`, to get a catchall that doesn't ac
for all remaining possible values. There are a fe
of values in a pattern: using the `_` pattern (whic
within another pattern, using a name that starts
ignore remaining parts of a value. Let's explore
patterns.

Ignoring an Entire Value with `_`

We've used the underscore (`_`) as a wildcard pa
not bind to the value. Although the underscore
last arm in a `match` expression, we can use it in
parameters, as shown in Listing 18-17.

Filename: src/main.rs

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y pa
}

fn main() {
    foo(3, 4);
}
```

Listing 18-17: Using `_` in a function signature

This code will completely ignore the value passe
print `This code only uses the y parameter:`

In most cases when you no longer need a partic
change the signature so it doesn't include the ur
parameter can be especially useful in some case
trait when you need a certain type signature but
implementation doesn't need one of the parame
about unused function parameters, as it would i

Ignoring Parts of a Value with a Nested _

We can also use _ inside another pattern to ign
when we want to test for only part of a value bu
the corresponding code we want to run. Listing
managing a setting's value. The business require
be allowed to overwrite an existing customizatio
setting and can give the setting a value if it is cur

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an exist
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value)
```

Listing 18-18: Using an underscore within patter
don't need to use the value inside the `Some`

This code will print `Can't overwrite an existi`
`setting is Some(5)`. In the first match arm, we
values inside either `Some` variant, but we do nee
`setting_value` and `new_setting_value` are th
why we're not changing `setting_value`, and it c

In all other cases (if either `setting_value` or `ne`
expressed by the _ pattern in the second arm,
to become `setting_value`.

We can also use underscores in multiple places
particular values. Listing 18-19 shows an examp

values in a tuple of five items.

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {}, {}, {]
    },
}
```

Listing 18-19: Ignoring multiple parts of a tuple

This code will print `Some numbers: 2, 8, 32`, ar

Ignoring an Unused Variable by Starting Its N

If you create a variable but don't use it anywhere because that could be a bug. But sometimes it's use yet, such as when you're prototyping or just you can tell Rust not to warn you about the unu the variable with an underscore. In Listing 18-20 when we run this code, we should only get a wa

Filename: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Listing 18-20: Starting a variable name with an u variable warnings

Here we get a warning about not using the varia about not using the variable preceded by the un

Note that there is a subtle difference between u starts with an underscore. The syntax `_x` still bi `_` doesn't bind at all. To show a case where this provide us with an error.

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    println!("found a string");
}

println!("{:?}", s);
```

Listing 18-21: An unused variable starting with a
which might take ownership of the value

We'll receive an error because the `s` value will s
us from using `s` again. However, using the und
the value. Listing 18-22 will compile without any
into `_`.

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

println!("{:?}", s);
```

Listing 18-22: Using an underscore does not bin

This code works just fine because we never bind

Ignoring Remaining Parts of a Value with `..`

With values that have many parts, we can use th
and ignore the rest, avoiding the need to list un
`..` pattern ignores any parts of a value that we
of the pattern. In Listing 18-23, we have a `Point`
three-dimensional space. In the `match` expressi
coordinate and ignore the values in the `y` and

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}",
}
```

Listing 18-23: Ignoring all fields of a `Point` exce

We list the `x` value and then just include the `..` to list `y: _` and `z: _`, particularly when we're w fields in situations where only one or two fields

The syntax `..` will expand to as many values as how to use `..` with a tuple.

Filename: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}
        },
    }
}
```

Listing 18-24: Matching only the first and last val values

In this code, the first and last value are matched match and ignore everything in the middle.

However, using `..` must be unambiguous. If it i for matching and which should be ignored, Rust shows an example of using `..` ambiguously, so

Filename: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", s
        },
    }
}
```

Listing 18-25: An attempt to use `..` in an ambig

When we compile this example, we get this erro

```
error: `..` can only be used once per tupl
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |                      ^^
```

It's impossible for Rust to determine how many matching a value with `second` and then how ma thereafter. This code could mean that we want t then ignore `8`, `16`, and `32`; or that we want to and then ignore `16` and `32`; and so forth. The v anything special to Rust, so we get a compiler er like this is ambiguous.

Extra Conditionals with Match Guards

A *match guard* is an additional `if` condition spe arm that must also match, along with the patter chosen. Match guards are useful for expressing alone allows.

The condition can use variables created in the p where the first arm has the pattern `Some(x)` an `if x < 5`.

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less tha
    Some(x) => println!("{}", x),
    None => (),
}
```

Listing 18-26: Adding a match guard to a patterr

This example will print `less than five: 4`. Wh the first arm, it matches, because `Some(4)` matc checks whether the value in `x` is less than `5`, ar selected.

If `num` had been `Some(10)` instead, the match g false because 10 is not less than 5. Rust would tl would match because the second arm doesn't h matches any `Some` variant.

There is no way to express the `if x < 5` condit guard gives us the ability to express this logic.

In Listing 18-11, we mentioned that we could us shadowing problem. Recall that a new variable v `match` expression instead of using the variable meant we couldn't test against the value of the c how we can use a match guard to fix this proble

Filename: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Mat
        _ => println!("Default case, x = {
    }

    println!("at the end: x = {:?}, y = {:
}
```

Listing 18-27: Using a match guard to test for eq

This code will now print `Default case, x = Sor`

match arm doesn't introduce a new variable `y` t
meaning we can use the outer `y` in the match g
pattern as `Some(y)`, which would have shadowe
This creates a new variable `n` that doesn't shad
variable outside the `match`.

The match guard `if n == y` is not a pattern an
variables. This `y` *is* the outer `y` rather than a ne
a value that has the same value as the outer `y` l

You can also use the *or* operator `|` in a match g
match guard condition will apply to all the patte
precedence of combining a match guard with a |
part of this example is that the `if y` match gua
though it might look like `if y` only applies to `6`

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

Listing 18-18: Combining multiple patterns with

The match condition states that the arm only ma
`5`, or `6` *and* if `y` is `true`. When this code runs,
because `x` is `4`, but the match guard `if y` is fa
The code moves on to the second arm, which do
`no`. The reason is that the `if` condition applies
only to the last value `6`. In other words, the pre
to a pattern behaves like this:

```
(4 | 5 | 6) if y => ...
```

rather than this:

```
4 | 5 | (6 if y) => ...
```

After running the code, the precedence behavio
applied only to the final value in the list of value
arm would have matched and the program wou

@ Bindings

The *at* operator (`@`) lets us create a variable tha
testing that value to see whether it matches a pa
example where we want to test that a `Message:`
`3...7` . But we also want to bind the value to th
it in the code associated with the arm. We could
the field, but for this example we'll use a differe

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3..
        println!("Found an id in range: {}
    },
    Message::Hello { id: 10...12 } => {
        println!("Found an id in another r
    },
    Message::Hello { id } => {
        println!("Found some other id: {}"
    },
}
```

Listing 18-19: Using `@` to bind to a value in a pat

This example will print `Found an id in range:`
before the range `3...7` , we're capturing whate
also testing that the value matched the range pa

In the second arm, where we only have a range
associated with the arm doesn't have a variable
`id` field. The `id` field's value could have been 1
with that pattern doesn't know which it is. The p
from the `id` field, because we haven't saved the

In the last arm, where we've specified a variable
value available to use in the arm's code in a vari
we've used the struct field shorthand syntax. Bu
value in the `id` field in this arm, as we did with
match this pattern.

Using `@` lets us test a value and save it in a varia

Legacy patterns: `ref` and `ref mut`

In older versions of Rust, `match` would assume
matched. But sometimes, that's not what you wa

```
let robot_name = &Some(String::from("Bors'

match robot_name {
    Some(name) => println!("Found a name:
    None => (),
}

println!("robot_name is: {:?}", robot_name
```

Here, `robot_name` is a `&Option<String>`. Rust w
doesn't match up with `&Option<T>`, so you'd ha

```
let robot_name = &Some(String::from("Bors'

match robot_name {
    &Some(name) => println!("Found a name:
    None => (),
}

println!("robot_name is: {:?}", robot_name
```

Next, Rust would complain that `name` is trying to
but because it's a reference to an option, it's bor
This is where the `ref` keyword comes into play:

```
let robot_name = &Some(String::from("Bors'

match robot_name {
    &Some(ref name) => println!("Found a n
    None => (),
}

println!("robot_name is: {:?}", robot_name
```

The `ref` keyword is like the opposite of `&` in pa
be a `&String`, don't try to move it out. In other w
against a reference, but `ref` *creates* a reference
mutable references.

Anyway, today's Rust doesn't work like this. If yo
borrowed, then all of the bindings you create wi

means that the original code works as you'd exp

Because Rust is backwards compatible, we coul
they're sometimes useful in obscure situations,
part of a struct as mutable and another part as i
older Rust code, so knowing what they do is still

Summary

Rust's patterns are very useful in that they help
data. When used in `match` expressions, Rust en
possible value, or your program won't compile. I
function parameters make those constructs mo
of values into smaller parts at the same time as
simple or complex patterns to suit our needs.

Next, for the penultimate chapter of the book, w
a variety of Rust's features.

Advanced Features

By now, you've learned the most commonly use
language. Before we do one more project in Ch
the language you might run into every once in a
reference for when you encounter any unknow
learn to use in this chapter are useful in very sp
not reach for them often, we want to make sure
Rust has to offer.

In this chapter, we'll cover:

- Unsafe Rust: how to opt out of some of Ru
  for manually upholding those guarantees
- Advanced lifetimes: syntax for complex life
- Advanced traits: associated types, default t
  syntax, supertraits, and the newtype patte
- Advanced types: more about the newtype
  and dynamically sized types
- Advanced functions and closures: function

It's a panoply of Rust features with something fo

Unsafe Rust

All the code we've discussed so far has had Rust
enforced at compile time. However, Rust has a s
doesn't enforce these memory safety guarantee
like regular Rust, but gives us extra superpower

Unsafe Rust exists because, by nature, static an
compiler tries to determine whether or not code
for it to reject some valid programs rather than
Although the code might be okay, as far as Rust
you can use unsafe code to tell the compiler, "Tr
downside is that you use it at your own risk: if y
problems due to memory unsafety, such as null

Another reason Rust has an unsafe alter ego is t
hardware is inherently unsafe. If Rust didn't let y
do certain tasks. Rust needs to allow you to do l
as directly interacting with the operating system
system. Working with low-level systems progran
language. Let's explore what we can do with uns

Unsafe Superpowers

To switch to unsafe Rust, use the `unsafe` keywo
holds the unsafe code. You can take four action
superpowers, that you can't in safe Rust. Those s

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

It's important to understand that `unsafe` doesn
disable any other of Rust's safety checks: if you u
still be checked. The `unsafe` keyword only gives
that are then not checked by the compiler for m
degree of safety inside of an unsafe block.

In addition, `unsafe` does not mean the code ins
or that it will definitely have memory safety prob
programmer, you'll ensure the code inside an `u`
valid way.

People are fallible, and mistakes will happen, bu
operations to be inside blocks annotated with `u
related to memory safety must be within an `uns
small; you'll be thankful later when you investiga

To isolate unsafe code as much as possible, it's b
safe abstraction and provide a safe API, which w
we examine unsafe functions and methods. Par
implemented as safe abstractions over unsafe c
unsafe code in a safe abstraction prevents uses
the places that you or your users might want to
with `unsafe` code, because using a safe abstrac

Let's look at each of the four unsafe superpowe
abstractions that provide a safe interface to uns

Dereferencing a Raw Pointer

In Chapter 4, in the "Dangling References" sectio
ensures references are always valid. Unsafe Rus
pointers that are similar to references. As with re
immutable or mutable and are written as `*cons
asterisk isn't the dereference operator; it's part o
raw pointers, *immutable* means that the pointer
being dereferenced.

Different from references and smart pointers, ra

- Are allowed to ignore the borrowing rules l
  mutable pointers or multiple mutable poin
- Aren't guaranteed to point to valid memory
- Are allowed to be null
- Don't implement any automatic cleanup

By opting out of having Rust enforce these guara
safety in exchange for greater performance or th
language or hardware where Rust's guarantees a

Listing 19-1 shows how to create an immutable a
references.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Listing 19-1: Creating raw pointers from referen

Notice that we don't include the `unsafe` keywor pointers in safe code; we just can't dereference as you'll see in a bit.

We've created raw pointers by using `as` to cast reference into their corresponding raw pointer t directly from references guaranteed to be valid, pointers are valid, but we can't make that assum

Next, we'll create a raw pointer whose validity w shows how to create a raw pointer to an arbitra arbitrary memory is undefined: there might be d not, the compiler might optimize the code so th program might error with a segmentation fault. write code like this, but it is possible.

```
let address = 0x012345usize;
let r = address as *const i32;
```

Listing 19-2: Creating a raw pointer to an arbitra

Recall that we can create raw pointers in safe co pointers and read the data being pointed to. In l operator `*` on a raw pointer that requires an u

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Listing 19-3: Dereferencing raw pointers within a

Creating a pointer does no harm; it's only when
points at that we might end up dealing with an i

Note also that in Listing 19-1 and 19-3, we create
pointers that both pointed to the same memory
instead tried to create an immutable and a muta
not have compiled because Rust's ownership ru
the same time as any immutable references. Wi
mutable pointer and an immutable pointer to th
through the mutable pointer, potentially creatin

With all of these dangers, why would you ever u
is when interfacing with C code, as you'll see in t
Function or Method." Another case is when buil
borrow checker doesn't understand. We'll introd
an example of a safe abstraction that uses unsa

Calling an Unsafe Function or Method

The second type of operation that requires an u
functions. Unsafe functions and methods look e
methods, but they have an extra `unsafe` before
`unsafe` keyword in this context indicates the fu
uphold when we call this function, because Rust
requirements. By calling an unsafe function with
we've read this function's documentation and ta
function's contracts.

Here is an unsafe function named `dangerous` th

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

We must call the `dangerous` function within a se
`dangerous` without the `unsafe` block, we'll get a

```
error[E0133]: call to unsafe function requ
 -->
  |
4 |     dangerous();
  |     ^^^^^^^^^^^ call to unsafe functic
```

By inserting the `unsafe` block around our call tc that we've read the function's documentation, w and we've verified that we're fulfilling the contra

Bodies of unsafe functions are effectively `unsaf` operations within an unsafe function, we don't r

Creating a Safe Abstraction over Unsafe Code

Just because a function contains unsafe code dc entire function as unsafe. In fact, wrapping unsa common abstraction. As an example, let's study `split_at_mut`, that requires some unsafe code implement it. This safe method is defined on mu makes it two by splitting the slice at the index gi shows how to use `split_at_mut`.

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

Listing 19-4: Using the safe `split_at_mut` funct

We can't implement this function using only safe something like Listing 19-5, which won't compile `split_at_mut` as a function rather than a meth rather than for a generic type `T`.

```
fn split_at_mut(slice: &mut [i32], mid: us
{
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

Listing 19-5: An attempted implementation of `s`

This function first gets the total length of the slic as a parameter is within the slice by checking wh length. The assertion means that if we pass an in split the slice at, the function will panic before it

Then we return two mutable slices in a tuple: on the `mid` index and another from `mid` to the end

When we try to compile the code in Listing 19-5,

```
error[E0499]: cannot borrow `*slice` as mu
 -->
  |
6 |     (&mut slice[..mid],
  |           ----- first mutable borrow c
7 |      &mut slice[mid..])
  |           ^^^^^ second mutable borrow
8 | }
  | - first borrow ends here
```

Rust's borrow checker can't understand that we' slice; it only knows that we're borrowing from th different parts of a slice is fundamentally okay b overlapping, but Rust isn't smart enough to know but Rust doesn't, it's time to reach for unsafe co

Listing 19-6 shows how to use an `unsafe` block, unsafe functions to make the implementation o

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: us
{
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mi
         slice::from_raw_parts_mut(ptr.of
    }
}
```

Listing 19-6: Using unsafe code in the implemen

Recall from “The Slice Type” section in Chapter 4 and the length of the slice. We use the `len` met the `as_mut_ptr` method to access the raw poin have a mutable slice to `i32` values, `as_mut_ptr` `*mut i32`, which we’ve stored in the variable `pt`

We keep the assertion that the `mid` index is witl unsafe code: the `slice::from_raw_parts_mut` f length, and it creates a slice. We use this functio `ptr` and is `mid` items long. Then we call the `of` argument to get a raw pointer that starts at `mid` pointer and the remaining number of items afte

The function `slice::from_raw_parts_mut` is un and must trust that this pointer is valid. The `off` unsafe, because it must trust that the offset loca we had to put an `unsafe` block around our calls `offset` so we could call them. By looking at the that `mid` must be less than or equal to `len`, we within the `unsafe` block will be valid pointers to acceptable and appropriate use of `unsafe`.

Note that we don’t need to mark the resulting `s` we can call this function from safe Rust. We’ve c unsafe code with an implementation of the func way, because it creates only valid pointers from

In contrast, the use of `slice::from_raw_parts_`

when the slice is used. This code takes an arbitr
slice 10,000 items long.

```
use std::slice;

let address = 0x01234usize;
let r = address as *mut i32;

let slice : &[i32] = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

Listing 19-7: Creating a slice from an arbitrary m

We don't own the memory at this arbitrary locat the slice this code creates contains valid `i32` va though it's a valid slice results in undefined beh to align `address` to 4 (the alignment of `i32`), th `slice::from_raw_parts_mut` would already be always be aligned, even if they are not used (an

Using `extern` Functions to Call External Code

Sometimes, your Rust code might need to inter language. For this, Rust has a keyword, `extern` , of a *Foreign Function Interface (FFI)*. An FFI is a wa define functions and enable a different (foreign) functions.

Listing 19-8 demonstrates how to set up an inte the C standard library. Functions declared withi call from Rust code. The reason is that other lan guarantees, and Rust can't check them, so resp ensure safety.

Filename: src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 acc
    }
}
```

Listing 19-8: Declaring and calling an `extern` fu

Within the `extern "C"` block, we list the names from another language we want to call. The `"C"` *interface (ABI)* the external function uses: the ABI the assembly level. The `"C"` ABI is the most con language's ABI.

Calling Rust Functions from Other Langua

We can also use `extern` to create an interfac call Rust functions. Instead of an `extern` bloc and specify the ABI to use just before the `fn` `#[no_mangle]` annotation to tell the Rust con this function. *Mangling* is when a compiler cha function to a different name that contains mo the compilation process to consume but is les programming language compiler mangles na Rust function to be nameable by other langua compiler's name mangling.

In the following example, we make the `call_` code, after it's compiled to a shared library ar

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust functi
}
```

This usage of `extern` does not require `unsaf`

Accessing or Modifying a Mutable Stat

Until now, we've not talked about *global variable* problematic with Rust's ownership rules. If two t mutable global variable, it can cause a data race

In Rust, global variables are called *static* variable declaration and use of a static variable with a str

Filename: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!'

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

Listing 19-9: Defining and using an immutable st

Static variables are similar to constants, which w Between Variables and Constants" section in Ch are in `SCREAMING_SNAKE_CASE` by convention, ar type, which is `&'static str` in this example. St references with the `'static` lifetime, which me the lifetime; we don't need to annotate it explicit variable is safe.

Constants and immutable static variables might is that values in a static variable have a fixed adc always access the same data. Constants, on the their data whenever they're used.

Another difference between constants and stati be mutable. Accessing and modifying mutable s shows how to declare, access, and modify a mut

Filename: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Listing 19-10: Reading from or writing to a muta

As with regular variables, we specify mutability u reads or writes from `COUNTER` must be within ar and prints `COUNTER: 3` as we would expect beca multiple threads access `COUNTER` would likely re

With mutable data that is globally accessible, it's races, which is why Rust considers mutable stati possible, it's preferable to use the concurrency t pointers we discussed in Chapter 16 so the com different threads is done safely.

Implementing an Unsafe Trait

The final action that works only with `unsafe` is i unsafe when at least one of its methods has sor verify. We can declare that a trait is `unsafe` by a `trait` and marking the implementation of the t Listing 19-11.

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}
```

Listing 19-11: Defining and implementing an uns

By using `unsafe impl` , we're promising that we'
compiler can't verify.

As an example, recall the `Sync` and `Send` mark
"Extensible Concurrency with the `Sync` and `Sen`
compiler implements these traits automatically
`Send` and `Sync` types. If we implement a type t
or `Sync` , such as raw pointers, and we want to r
must use `unsafe` . Rust can't verify that our type
be safely sent across threads or accessed from r
to do those checks manually and indicate as suc

When to Use Unsafe Code

Using `unsafe` to take one of the four actions (su
or even frowned upon. But it is trickier to get `ur`
compiler can't help uphold memory safety. Whe
code, you can do so, and having the explicit `uns`
track down the source of problems if they occur

Advanced Lifetimes

In Chapter 10 in the "Validating References with
to annotate references with lifetime parameters
references relate. You saw how every reference
Rust will let you elide lifetimes. Now we'll look at
that we haven't covered yet:

- Lifetime subtyping: ensures that one lifetin
- Lifetime bounds: specifies a lifetime for a r
- Inference of trait object lifetimes: allows th lifetimes and when they need to be specifi
- The anonymous lifetime: making elision m

Ensuring One Lifetime Outlives Anothe

Lifetime subtyping specifies that one lifetime sho

explore lifetime subtyping, imagine we want to v
called Context that holds a reference to the str
that will parse this string and return success or f
the Context to do the parsing. Listing 19-12 imp
code doesn't have the required lifetime annotati

Filename: src/lib.rs

```
struct Context(&str);

struct Parser {
    context: &Context,
}

impl Parser {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-12: Defining a parser without lifetime

Compiling the code results in errors because Ru
string slice in Context and the reference to a C

For simplicity's sake, the parse function returns
function will do nothing on success and, on failu
slice that didn't parse correctly. A real implemen
information and would return a structured data
won't be discussing those details because they a
this example.

To keep this code simple, we won't write any par
that somewhere in the parsing logic we would h
error that references the part of the input that is
the code example interesting in regard to lifetim
parser is that the input is invalid after the first by
the first byte is not on a valid character boundar
example to focus on the lifetimes involved.

To get this code to compile, we need to fill in the
slice in Context and the reference to the Conte
straightforward way to do this is to use the same
in Listing 19-13. Recall from the "Lifetime Annota
Chapter 10 that each of struct Context<'a>, s
declaring a new lifetime parameter. While their r

three lifetime parameters declared in this examp

Filename: src/lib.rs

```
struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-13: Annotating all references in `Conte`
parameters

This code compiles just fine. It tells Rust that a `P`
`Context` with lifetime `'a` and that `Context` ho
as the reference to the `Context` in `Parser`. Rus
lifetime parameters were required for these refe
lifetime parameters.

Next, in Listing 19-14, we'll add a function that ta
`Parser` to parse that context, and returns what
quite work.

Filename: src/lib.rs

```
fn parse_context(context: Context) -> Resu
    Parser { context: &context }.parse()
}
```

Listing 19-14: An attempt to add a `parse_conte`
uses a `Parser`

We get two verbose errors when we try to comp
`parse_context` function:

```
error[E0597]: borrowed value does not live
  --> src/lib.rs:14:5
   |
14 |     Parser { context: &context }.pars
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ does
15 | }
   | - temporary value only lives until he
   |
note: borrowed value must be valid for the
on the function body at 13:1...
  --> src/lib.rs:13:1
   |
13 | / fn parse_context(context: Context)
14 | |     Parser { context: &context }.pa
15 | | }
   | |_^

error[E0597]: `context` does not live long
  --> src/lib.rs:14:24
   |
14 |     Parser { context: &context }.pars
   |                        ^^^^^^^ does r
15 | }
   | - borrowed value only lives until her
   |
note: borrowed value must be valid for the
on the function body at 13:1...
  --> src/lib.rs:13:1
   |
13 | / fn parse_context(context: Context)
14 | |     Parser { context: &context }.pa
15 | | }
   | |_^
```

These errors state that the `Parser` instance tha
parameter live only until the end of the `parse_c`
to live for the entire lifetime of the function.

In other words, `Parser` and `context` need to *o*
before the function starts as well as after it ends
always be valid. The `Parser` we're creating and
scope at the end of the function, because `parse`
`context`.

To figure out why these errors occur, let's look a
specifically the references in the signature of the

```
    fn parse(&self) -> Result<(), &str> {
```

Remember the elision rules? If we annotate the
eliding, the signature would be as follows:

```
fn parse<'a>(&'a self) -> Result<(), &
```

That is, the error part of the return value of `par`
lifetime of the `Parser` instance (that of `&self` i
makes sense: the returned string slice reference
instance held by the `Parser`, and the definition
the lifetime of the reference to `Context` and the
`Context` holds should be the same.

The problem is that the `parse_context` functio
`parse`, so the lifetime of the return value of `pa`
the `Parser` as well. But the `Parser` instance cre
won't live past the end of the function (it's temp
scope at the end of the function (`parse_context`

Rust thinks we're trying to return a reference to
end of the function, because we annotated all th
parameter. The annotations told Rust the lifetim
holds is the same as that of the lifetime of the re
holds.

The `parse_context` function can't see that withi
returned will outlive `Context` and `Parser` and t
returns refers to the string slice, not to `Context`

By knowing what the implementation of `parse`
the return value of `parse` is tied to the `Parser`
`Parser` instance's `Context`, which is referencin
lifetime of the string slice that `parse_context` n
tell Rust that the string slice in `Context` and the
have different lifetimes and that the return valu
lifetime of the string slice in `Context`.

First, we'll try giving `Parser` and `Context` differ
Listing 19-15. We'll use `'s` and `'c` as lifetime p
lifetime goes with the string slice in `Context` an
`Context` in `Parser`. Note that this solution wor
a start. We'll look at why this fix isn't sufficient w

Filename: src/lib.rs

```
struct Context<'s>(&'s str);

struct Parser<'c, 's> {
    context: &'c Context<'s>,
}

impl<'c, 's> Parser<'c, 's> {
    fn parse(&self) -> Result<(), &'s str>
        Err(&self.context.0[1..])
    }
}

fn parse_context(context: Context) -> Resu
    Parser { context: &context }.parse()
}
```

Listing 19-15: Specifying different lifetime param slice and to `Context`

We've annotated the lifetimes of the references annotated them in Listing 19-13. But this time w depending on whether the reference goes with t We've also added an annotation to the string sli to indicate that it goes with the lifetime of the st

When we try to compile now, we get the followir

```
error[E0491]: in type `&'c Context<'s>`, r
than the data it references
 --> src/lib.rs:4:5
  |
4 |     context: &'c Context<'s>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^
  |
note: the pointer is valid for the lifetin
3:1
 --> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
  | |_^
note: but the referenced data is only vali
on the struct at 3:1
 --> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
  | |_^
```

Rust doesn't know of any relationship between referenced data in `Context` with lifetime `'s` ne that it lives longer than the reference with lifetin the reference to `Context` might not be valid.

Now we get to the point of this section: the Rust that one lifetime parameter lives at least as long where we declare lifetime parameters, we can d declare a lifetime `'b` that lives at least as long a syntax `'b: 'a`.

In our definition of `Parser`, to say that `'s` (the guaranteed to live at least as long as `'c` (the life we change the lifetime declarations to look like t

Filename: src/lib.rs

```
struct Parser<'c, 's: 'c> {
    context: &'c Context<'s>,
}
```

Now the reference to `Context` in the `Parser` ar the `Context` have different lifetimes; we've ensu slice is longer than the reference to the `Context`

That was a very long-winded example, but as we chapter, Rust's advanced features are very speci we described in this example, but in such situati something and give it the necessary lifetime.

Lifetime Bounds on References to Gen

In the "Trait Bounds" section in Chapter 10, we c generic types. We can also add lifetime paramet these are called *lifetime bounds*. Lifetime bounds generic types won't outlive the data they're refer

As an example, consider a type that is a wrappe `RefCell<T>` type from the "`RefCell<T>` and the in Chapter 15: its `borrow` and `borrow_mut` meth `RefMut`, respectively. These types are wrappers borrowing rules at runtime. The definition of the without lifetime bounds for now.

Filename: src/lib.rs

```
struct Ref<'a, T>(&'a T);
```

Listing 19-16: Defining a struct to wrap a referer
bounds

Without explicitly constraining the lifetime `'a` ir
Rust will error because it doesn't know how long

```
error[E0309]: the parameter type `T` may r
 --> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |                   ^^^^^^
  |
  = help: consider adding an explicit life
note: ...so that the reference type `&'a T
points at
 --> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |                   ^^^^^^
```

Because `T` can be any type, `T` could be a refere
references, each of which could have their own l
as long as `'a`.

Fortunately, the error provides helpful advice or
this case:

```
consider adding an explicit lifetime bounc
type
`&'a T` does not outlive the data it point
```

Listing 19-17 shows how to apply this advice by
we declare the generic type `T`.

```
struct Ref<'a, T: 'a>(&'a T);
```

Listing 19-17: Adding lifetime bounds on `T` to sp
least as long as `'a`

This code now compiles because the `T: 'a` syn
but if it contains any references, the references

We could solve this problem in a different way, a
StaticRef struct in Listing 19-18, by adding the
means if T contains any references, they must l

```
struct StaticRef<T: 'static>(&'static T);
```

Listing 19-18: Adding a 'static lifetime bound
have only 'static references or no references

Because 'static means the reference must liv
type that contains no references meets the crite
the entire program (because there are no refere
concerned about references living long enough,
type that has no references and a type that has
the same for determining whether or not a refe
it refers to.

Inference of Trait Object Lifetimes

In Chapter 17 in the "Using Trait Objects that All
section, we discussed trait objects, consisting of
us to use dynamic dispatch. We haven't yet discu
implementing the trait in the trait object has a li
19-19 where we have a trait Red and a struct Ba
reference (and thus has a lifetime parameter) an
want to use an instance of Ball as the trait obj

Filename: src/main.rs

```
trait Red { }

struct Ball<'a> {
    diameter: &'a i32,
}

impl<'a> Red for Ball<'a> { }

fn main() {
    let num = 5;

    let obj = Box::new(Ball { diameter: &n
}
```

Listing 19-19: Using a type that has a lifetime pa

This code compiles without any errors, even tho
the lifetimes involved in `obj`. This code works b
with lifetimes and trait objects:

- The default lifetime of a trait object is `'sta
- With `&'a Trait` or `&'a mut Trait`, the de
- With a single `T: 'a` clause, the default life
- With multiple clauses like `T: 'a`, there is r
  explicit.

When we must be explicit, we can add a lifetime
`Box<dyn Red>` using the syntax `Box<dyn Red +
depending on whether the reference lives for th
other bounds, the syntax adding a lifetime boun
`Red` trait that has references inside the type mu
the trait object bounds as those references.

The anonymous lifetime

Let's say that we have a struct that's a wrapper a

```
struct StrWrap<'a>(&'a str);
```

We can write a function that returns one of thes

```
fn foo<'a>(string: &'a str) -> StrWrap<'a>
    StrWrap(string)
}
```

But that's a lot of `'a`s! To cut down on some of
lifetime, `'_`, like this:

```
fn foo(string: &str) -> StrWrap<'_> {
    StrWrap(string)
}
```

The `'_` says "use the elidied lifetime here." This
`StrWrap` contains a reference, but we don't nee
make sense of it.

It works in `impl` headers too; for example:

```
// verbose
impl<'a> fmt::Debug for StrWrap<'a> {

// elided
impl fmt::Debug for StrWrap<'_> {
```

Next, let's look at some other advanced features

Advanced Traits

We first covered traits in the "Traits: Defining Sh
but as with lifetimes, we didn't discuss the more
more about Rust, we can get into the nitty-gritty

Specifying Placeholder Types in Trait D Types

Associated types connect a type placeholder with
definitions can use these placeholder types in th
trait will specify the concrete type to be used in
implementation. That way, we can define a trait
needing to know exactly what those types are u

We've described most of the advanced features
needed. Associated types are somewhere in the
than features explained in the rest of the book b
other features discussed in this chapter.

One example of a trait with an associated type is
library provides. The associated type is named `I`
values the type implementing the `Iterator` trai
Trait and the `next` Method" section of Chapter
of the `Iterator` trait is as shown in Listing 19-2

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Ite
}
```

Listing 19-20: The definition of the `Iterator` tra

The type `Item` is a placeholder type, and the `ne` will return values of type `Option<Self::Item>` . will specify the concrete type for `Item` , and the containing a value of that concrete type.

Associated types might seem like a similar conce us to define a function without specifying what t associated types?

Let's examine the difference between the two co Chapter 13 that implements the `Iterator` trait 13-21, we specified that the `Item` type was `u32`

Filename: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Ite
        // --snip--
```

This syntax seems comparable to that of generi `Iterator` trait with generics, as shown in Listin

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Listing 19-21: A hypothetical definition of the `It`

The difference is that when using generics, as in types in each implementation; because we can a `Iterator<String> for Counter` or any other ty implementations of `Iterator` for `Counter` . In c parameter, it can be implemented for a type mu types of the generic type parameters each time. `Counter` , we would have to provide type annota implementation of `Iterator` we want to use.

With associated types, we don't need to annotat trait on a type multiple times. In Listing 19-20 wi types, we can only choose what the type of `Iter` be one `impl Iterator for Counter` . We don't iterator of `u32` values everywhere that we call `

Default Generic Type Parameters and

When we use generic type parameters, we can s generic type. This eliminates the need for imple concrete type if the default type works. The synt generic type is `<PlaceholderType=ConcreteType

A great example of a situation where this techni overloading. *Operator overloading* is customizing `+`) in particular situations.

Rust doesn't allow you to create your own opera But you can overload the operations and corres implementing the traits associated with the ope overload the `+` operator to add two `Point` inst implementing the `Add` trait on a `Point` struct:

Filename: src/main.rs

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Poir
               Point { x: 3, y: 3 });
}
```

Listing 19-22: Implementing the `Add` trait to ove instances

The `add` method adds the `x` values of two `Poi` `Point` instances to create a new `Point`. The `Ac` `Output` that determines the type returned from

The default generic type in this code is within th

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output
}
```

This code should look generally familiar: a trait w type. The new part is `RHS=Self`: this syntax is ca generic type parameter (short for "right hand sid parameter in the `add` method. If we don't speci implement the `Add` trait, the type of `RHS` will de we're implementing `Add` on.

When we implemented `Add` for `Point`, we used wanted to add two `Point` instances. Let's look a `Add` trait where we want to customize the `RHS`

We have two structs, `Millimeters` and `Meters`, want to add values in millimeters to values in me `Add` do the conversion correctly. We can implem `Meters` as the `RHS`, as shown in Listing 19-23.

Filename: src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millime
        Millimeters(self.0 + (other.0 * 10
    }
}
```

Listing 19-23: Implementing the `Add` trait on `Mi` `Meters`

To add `Millimeters` and `Meters`, we specify `i

the `RHS` type parameter instead of using the de

You'll use default type parameters in two main w

- To extend a type without breaking existing
- To allow customization in specific cases mo

The standard library's `Add` trait is an example o add two like types, but the `Add` trait provides th Using a default type parameter in the `Add` trait specify the extra parameter most of the time. In boilerplate isn't needed, making it easier to use

The first purpose is similar to the second but in parameter to an existing trait, you can give it a c functionality of the trait without breaking the ex

Fully Qualified Syntax for Disambiguat Same Name

Nothing in Rust prevents a trait from having a m another trait's method, nor does Rust prevent yo one type. It's also possible to implement a meth name as methods from traits.

When calling methods with the same name, you want to use. Consider the code in Listing 19-24 w and `Wizard`, that both have a method called `fl a type `Human` that already has a method named method does something different.

Filename: src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain spe
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*"
    }
}
```

Listing 19-24: Two traits are defined to have a `f` the `Human` type, and a `fly` method is implemer

When we call `fly` on an instance of `Human`, the method that is directly implemented on the type

Filename: src/main.rs

```
fn main() {
    let person = Human;
    person.fly();
}
```

Listing 19-25: Calling `fly` on an instance of `Hum`

Running this code will print `*waving arms furi` `fly` method implemented on `Human` directly.

To call the `fly` methods from either the `Pilot` use more explicit syntax to specify which `fly` m demonstrates this syntax.

Filename: src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Listing 19-26: Specifying which trait's `fly` meth

Specifying the trait name before the method na
implementation of `fly` we want to call. We cou
which is equivalent to the `person.fly()` that w
longer to write if we don't need to disambiguate

Running this code prints the following:

```
This is your captain speaking.
Up!
*waving arms furiously*
```

Because the `fly` method takes a `self` parame
implement one *trait*, Rust could figure out which
based on the type of `self`.

However, associated functions that are part of t
When two types in the same scope implement t
type you mean unless you use *fully qualified synt*
Listing 19-27 has the associated function `baby_`
for the struct `Dog`, and the associated function

Filename: src/main.rs

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}",
}
```

Listing 19-27: A trait with an associated function
function of the same name that also implement

This code is for an animal shelter that wants to r
implemented in the `baby_name` associated func
type also implements the trait `Animal`, which de
animals have. Baby dogs are called puppies, and
implementation of the `Animal` trait on `Dog` in t
with the `Animal` trait.

In `main`, we call the `Dog::baby_name` function, v
defined on `Dog` directly. This code prints the fol

```
A baby dog is called a Spot
```

This output isn't what we wanted. We want to ca
of the `Animal` trait that we implemented on `Do`
`A baby dog is called a puppy`. The technique
used in Listing 19-26 doesn't help here; if we cha
19-28, we'll get a compilation error.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}",
}
```

Listing 19-28: Attempting to call the baby_name
Rust doesn't know which implementation to use

Because Animal::baby_name is an associated fu
thus doesn't have a self parameter, Rust can't
Animal::baby_name we want. We'll get this com

```
error[E0283]: type annotations required: c
  --> src/main.rs:20:43
   |
20 |     println!("A baby dog is called a
   |
   |
   = note: required by `Animal::baby_name`
```

To disambiguate and tell Rust that we want to u
Dog, we need to use fully qualified syntax. Listir
fully qualified syntax.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}",
}
```

Listing 19-29: Using fully qualified syntax to spec
baby_name function from the Animal trait as im

We're providing Rust with a type annotation witl
we want to call the baby_name method from the
Dog by saying that we want to treat the Dog typ
This code will now print what we want:

```
A baby dog is called a puppy
```

In general, fully qualified syntax is defined as fol

```
<Type as Trait>::function(receiver_if_meth
```

For associated functions, there would not be a
of other arguments. You could use fully qualified
functions or methods. However, you're allowed

Rust can figure out from other information in th
more verbose syntax in cases where there are m
same name and Rust needs help to identify whic

Using Supertraits to Require One Trait Another Trait

Sometimes, you might need one trait to use anc
you need to rely on the dependent traits also be
on is a *supertrait* of the trait you're implementing

For example, let's say we want to make an `Outl`
`outline_print` method that will print a value fr
`Point` struct that implements `Display` to resul
`outline_print` on a `Point` instance that has `1`
the following:

```
**********
*        *
* (1, 3) *
*        *
**********
```

In the implementation of `outline_print`, we wa
functionality. Therefore, we need to specify that
for types that also implement `Display` and prov
`OutlinePrint` needs. We can do that in the trai
`OutlinePrint: Display`. This technique is simil
Listing 19-30 shows an implementation of the `O`

Filename: src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4)
        println!("*{}*", " ".repeat(len +
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len +
        println!("{}", "*".repeat(len + 4)
    }
}
```

Listing 19-30: Implementing the `OutlinePrint` t
from `Display`

Because we've specified that `OutlinePrint` req
the `to_string` function that is automatically im
implements `Display`. If we tried to use `to_str`
specifying the `Display` trait after the trait name
method named `to_string` was found for the ty

Let's see what happens when we try to impleme
doesn't implement `Display`, such as the `Point`

Filename: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

We get an error saying that `Display` is required

```
error[E0277]: the trait bound `Point: std:
  --> src/main.rs:20:6
   |
20 | impl OutlinePrint for Point {}
   |      ^^^^^^^^^^^^ `Point` cannot be f
formatter;
try using `:?` instead if you are using a
   |
   = help: the trait `std::fmt::Display` i
```

To fix this, we implement `Display` on `Point` ar
`OutlinePrint` requires, like so:

Filename: src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter)
        write!(f, "({}, {})", self.x, sel
    }
}
```

Then implementing the `OutlinePrint` trait on
we can call `outline_print` on a `Point` instanc
asterisks.

Using the Newtype Pattern to Implem
Types

In Chapter 10 in the "Implementing a Trait on a
orphan rule that states we're allowed to implem
the trait or the type are local to our crate. It's po
using the *newtype pattern*, which involves creatir
covered tuple structs in the "Using Tuple Structs
Different Types" section of Chapter 5.) The tuple
thin wrapper around the type we want to impler
type is local to our crate, and we can implement
term that originates from the Haskell programm
performance penalty for using this pattern, and
time.

As an example, let's say we want to implement
orphan rule prevents us from doing directly bec
`Vec<T>` type are defined outside our crate. We
an instance of `Vec<T>`; then we can implement
`Vec<T>` value, as shown in Listing 19-31.

Filename: src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter)
        write!(f, "[{}]", self.0.join(", '
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hel
    println!("w = {}", w);
}
```

Listing 19-31: Creating a `Wrapper` type around `\

The implementation of `Display` uses `self.0` t
`Wrapper` is a tuple struct and `Vec<T>` is the iten
use the functionality of the `Display` type on `Wr

The downside of using this technique is that `Wra
the methods of the value it's holding. We would
`Vec<T>` directly on `Wrapper` such that the meth
allow us to treat `Wrapper` exactly like a `Vec<T>` .
every method the inner type has, implementing
15 in the "Treating Smart Pointers like Regular R
section) on the `Wrapper` to return the inner type
the `Wrapper` type to have all the methods of the
the `Wrapper` type's behavior—we would have to
want manually.

Now you know how the newtype pattern is used
pattern even when traits are not involved. Let's
advanced ways to interact with Rust's type syste

Advanced Types

The Rust type system has some features that we
haven't yet discussed. We'll start by discussing n
why newtypes are useful as types. Then we'll mo
similar to newtypes but with slightly different se
and dynamically sized types.

> Note: The next section assumes you've read t
> Pattern to Implement External Traits on Exter

Using the Newtype Pattern for Type Sa

The newtype pattern is useful for tasks beyond t statically enforcing that values are never confus You saw an example of using newtypes to indica the `Millimeters` and `Meters` structs wrapped a function with a parameter of type `Millimeter` that accidentally tried to call that function with a

Another use of the newtype pattern is in abstrac details of a type: the new type can expose a pub the private inner type if we used the new type di functionality, for example.

Newtypes can also hide internal implementation `People` type to wrap a `HashMap<i32, String>` with their name. Code using `People` would only provide, such as a method to add a name string wouldn't need to know that we assign an `i32` ID pattern is a lightweight way to achieve encapsul which we discussed in the "Encapsulation that H of Chapter 17.

Creating Type Synonyms with Type Ali

Along with the newtype pattern, Rust provides t an existing type another name. For this we use t can create the alias `Kilometers` to `i32` like so:

```
type Kilometers = i32;
```

Now, the alias `Kilometers` is a *synonym* for `i32` `Meters` types we created in Listing 19-23, `Kilom` Values that have the type `Kilometers` will be tre

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

Because `Kilometers` and `i32` are the same typ
and we can pass `Kilometers` values to function
However, using this method, we don't get the ty
the newtype pattern discussed earlier.

The main use case for type synonyms is to redu
have a lengthy type like this:

```
Box<dyn Fn() + Send + 'static>
```

Writing this lengthy type in function signatures a
code can be tiresome and error prone. Imagine
in Listing 19-32.

```
let f: Box<dyn Fn() + Send + 'static> = Bo

fn takes_long_type(f: Box<dyn Fn() + Send
    // --snip--
}

fn returns_long_type() -> Box<dyn Fn() + S
    // --snip--
}
```

Listing 19-32: Using a long type in many places

A type alias makes this code more manageable l
19-33, we've introduced an alias named `Thunk` f
all uses of the type with the shorter alias `Thunk`

```
type Thunk = Box<dyn Fn() + Send + 'static

let f: Thunk = Box::new(|| println!("hi"))

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

Listing 19-33: Introducing a type alias `Thunk` to r

This code is much easier to read and write! Choo alias can help communicate your intent as well (evaluated at a later time, so it's an appropriate n

Type aliases are also commonly used with the `R` repetition. Consider the `std::io` module in the return a `Result<T, E>` to handle situations whe has a `std::io::Error` struct that represents all functions in `std::io` will be returning `Result<T` `std::io::Error`, such as these functions in the

```
use std::io::Error;
use std::fmt;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Res
    fn flush(&mut self) -> Result<(), Erro

    fn write_all(&mut self, buf: &[u8]) ->
    fn write_fmt(&mut self, fmt: fmt::Argu
}
```

The `Result<..., Error>` is repeated a lot. As su declaration:

```
type Result<T> = Result<T, std::io::Error>
```

Because this declaration is in the `std::io` modu `std::io::Result<T>`—that is, a `Result<T, E>` `std::io::Error`. The `Write` trait function signa

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Res
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) ->
    fn write_fmt(&mut self, fmt: Arguments
}
```

The type alias helps in two ways: it makes code e
consistent interface across all of `std::io`. Beca
`Result<T, E>`, which means we can use any me
with it, as well as special syntax like the `?` opera

The Never Type that Never Returns

Rust has a special type named `!` that's known in
because it has no values. We prefer to call it the
place of the return type when a function will nev

```
fn bar() -> ! {
    // --snip--
}
```

This code is read as "the function `bar` returns n
called *diverging functions*. We can't create values
possibly return.

But what use is a type you can never create valu
2-5; we've reproduced part of it here in Listing 1

```
let guess: u32 = match guess.trim().parse(
    Ok(num) => num,
    Err(_) => continue,
};
```

Listing 19-34: A `match` with an arm that ends in

At the time, we skipped over some details in this
Control Flow Operator" section, we discussed th
same type. So, for example, the following code c

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
}
```

The type of `guess` in this code would have to be requires that `guess` have only one type. So wha we allowed to return a `u32` from one arm and h `continue` in Listing 19-34?

As you might have guessed, `continue` has a `!` the type of `guess`, it looks at both match arms, the latter with a `!` value. Because `!` can never type of `guess` is `u32`.

The formal way of describing this behavior is tha coerced into any other type. We're allowed to er because `continue` doesn't return a value; instea the loop, so in the `Err` case, we never assign a v

The never type is useful with the `panic!` macro function that we call on `Option<T>` values to pr definition:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Optior
        }
    }
}
```

In this code, the same thing happens as in the m `val` has the type `T` and `panic!` has the type `!` expression is `T`. This code works because `pani` the program. In the `None` case, we won't be retu code is valid.

One final expression that has the type `!` is a `lo`

```
print!("forever ");

loop {
    print!("and ever ");
}
```

Here, the loop never ends, so `!` is the value of t
be true if we included a `break`, because the loo
`break`.

Dynamically Sized Types and the `Size(`

Due to Rust's need to know certain details, such
value of a particular type, there is a corner of its
the concept of *dynamically sized types*. Sometime
these types let us write code using values whose

Let's dig into the details of a dynamically sized ty
using throughout the book. That's right, not `&st`
can't know how long the string is until runtime, r
type `str`, nor can we take an argument of type
which does not work:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

Rust needs to know how much memory to alloca
and all values of a type must use the same amou
write this code, these two `str` values would nee
space. But they have different lengths: `s1` need
15. This is why it's not possible to create a variab

So what do we do? In this case, you already know
`s1` and `s2` a `&str` rather than a `str`. Recall th
Chapter 4, we said the slice data structure stores
of the slice.

So although a `&T` is a single value that stores th
located, a `&str` is *two* values: the address of the
know the size of a `&str` value at compile time: i
is, we always know the size of a `&str`, no matte
general, this is the way in which dynamically size
an extra bit of metadata that stores the size of tl
rule of dynamically sized types is that we must a
types behind a pointer of some kind.

We can combine `str` with all kinds of pointers:
In fact, you've seen this before but with a differe
trait is a dynamically sized type we can refer to b

Chapter 17 in the "Using Trait Objects that Allow
section, we mentioned that to use traits as trait
pointer, such as `&dyn Trait` or `Box<dyn Trait`

To work with DSTs, Rust has a particular trait cal
whether or not a type's size is known at compile
implemented for everything whose size is know
implicitly adds a bound on `Sized` to every gene
definition like this:

```
fn generic<T>(t: T) {
    // --snip--
}
```

is actually treated as though we had written this

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

By default, generic functions will work only on ty
compile time. However, you can use the followir
restriction:

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

A trait bound on `?Sized` is the opposite of a tra
this as " `T` may or may not be `Sized` ." This synta
other traits.

Also note that we switched the type of the `t` pa
type might not be `Sized` , we need to use it behi
we've chosen a reference.

Next, we'll talk about functions and closures!

Advanced Functions and Closu

Finally, we'll explore some advanced features re
include function pointers and returning closures

Function Pointers

We've talked about how to pass closures to func
functions to functions! This technique is useful v
you've already defined rather than defining a ne
pointers will allow you to use functions as argun
coerce to the type `fn` (with a lowercase f), not t
trait. The `fn` type is called a *function pointer*. Th
parameter is a function pointer is similar to that
19-35.

Filename: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

Listing 19-35: Using the `fn` type to accept a fun

This code prints `The answer is: 12`. We specify
is an `fn` that takes one parameter of type `i32`
`f` in the body of `do_twice`. In `main`, we can pa
first argument to `do_twice`.

Unlike closures, `fn` is a type rather than a trait,
type directly rather than declaring a generic type
as a trait bound.

Function pointers implement all three of the clo
so you can always pass a function pointer as an
a closure. It's best to write functions using a gen
so your functions can accept either functions or

An example of where you would want to only ac
interfacing with external code that doesn't have
functions as arguments, but C doesn't have clos

As an example of where you could use either a c
function, let's look at a use of `map`. To use the m
numbers into a vector of strings, we could use a

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of
    .iter()
    .map(|i| i.to_string())
    .collect();
```

Or we could name a function as the argument to

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of
    .iter()
    .map(ToString::to_string)
    .collect();
```

Note that we must use the fully qualified syntax
"Advanced Traits" section because there are mu
`to_string`. Here, we're using the `to_string` fu
which the standard library has implemented for

Some people prefer this style, and some people
compiling to the same code, so use whichever st

Returning Closures

Closures are represented by traits, which means
most cases where you might want to return a tra
type that implements the trait as the return valu
that with closures because they don't have a cor
not allowed to use the function pointer `fn` as a

The following code tries to return a closure direc

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

The compiler error is as follows:

```
error[E0277]: the trait bound `std::ops::F
std::marker::Sized` is not satisfied
 -->
  |
1 | fn returns_closure() -> Fn(i32) -> i32
  |                         ^^^^^^^^^^^^^^
'static`
  does not have a constant size known at c
  |
  = help: the trait `std::marker::Sized` i
  `std::ops::Fn(i32) -> i32 + 'static`
  = note: the return type of a function mu
```

The error references the `Sized` trait again! Rust
need to store the closure. We saw a solution to t
trait object:

```
fn returns_closure() -> Box<dyn Fn(i32) ->
    Box::new(|x| x + 1)
}
```

This code will compile just fine. For more about
Objects That Allow for Values of Different Types"

Summary

Whew! Now you have some features of Rust in y
but you'll know they're available in very particula
several complex topics so that when you encour
suggestions or in other peoples' code, you'll be a
syntax. Use this chapter as a reference to guide

Next, we'll put everything we've discussed throu
one more project!

Final Project: Building Web Server

It's been a long journey, but we've reached the e
build one more project together to demonstrate
the final chapters, as well as recap some earlier

For our final project, we'll make a web server tha
20-1 in a web browser.

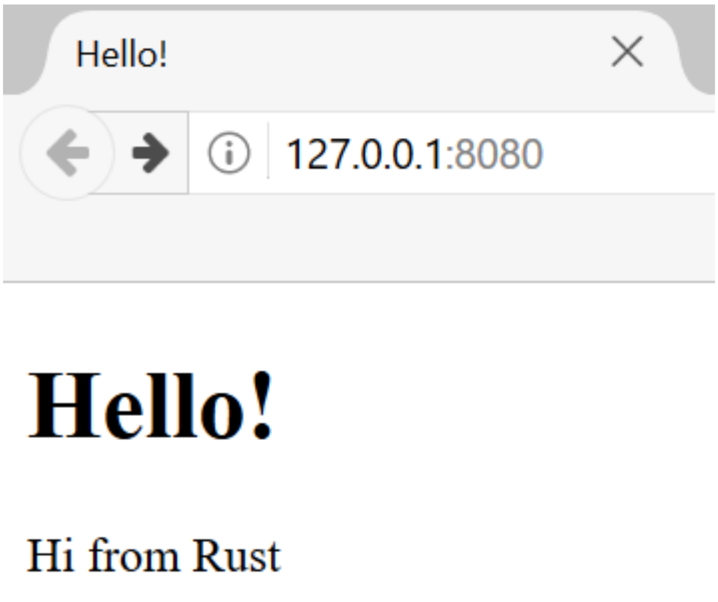

Figure 20-1: Our final shared project

Here is the plan to build the web server:

1. Learn a bit about TCP and HTTP.
2. Listen for TCP connections on a socket.
3. Parse a small number of HTTP requests.
4. Create a proper HTTP response.
5. Improve the throughput of our server with

But before we get started, we should mention o
be the best way to build a web server with Rust.
are available on *https://crates.io/* that provide m
pool implementations than we'll build.

However, our intention in this chapter is to help
Because Rust is a systems programming langua
abstraction we want to work with and can go to
practical in other languages. We'll write the basi
manually so you can learn the general ideas and
might use in the future.

Building a Single-Threaded We

We'll start by getting a single-threaded web serv

at a quick overview of the protocols involved in l
these protocols are beyond the scope of this bo
the information you need.

The two main protocols involved in web servers
(HTTP) and the *Transmission Control Protocol (TCF*
protocols, meaning a *client* initiates requests an
provides a response to the client. The contents c
defined by the protocols.

TCP is the lower-level protocol that describes th
one server to another but doesn't specify what t
of TCP by defining the contents of the requests a
to use HTTP with other protocols, but in the vast
data over TCP. We'll work with the raw bytes of T
responses.

Listening to the TCP Connection

Our web server needs to listen to a TCP connect
on. The standard library offers a `std::net` mod
new project in the usual fashion:

```
$ cargo new hello
     Created binary (application) `hello`
$ cd hello
```

Now enter the code in Listing 20-1 in *src/main.rs*
address `127.0.0.1:7878` for incoming TCP stre
stream, it will print `Connection established!` .

Filename: src/main.rs

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!"
    }
}
```

Listing 20-1: Listening for incoming streams and
a stream

Using `TcpListener` , we can listen for TCP conne
`127.0.0.1:7878` . In the address, the section be
representing your computer (this is the same or
represent the authors' computer specifically), ar
port for two reasons: HTTP is normally accepted
on a telephone.

The `bind` function in this scenario works like th
new `TcpListener` instance. The reason the func
networking, connecting to a port to listen to is k

The `bind` function returns a `Result<T, E>` , wh
For example, connecting to port 80 requires adr
(nonadministrators can listen only on ports high
connect to port 80 without being an administrat
example, binding wouldn't work if we ran two in
two programs listening to the same port. Becau
learning purposes, we won't worry about handli
use `unwrap` to stop the program if errors happe

The `incoming` method on `TcpListener` returns
of streams (more specifically, streams of type `T`
an open connection between the client and the
the full request and response process in which a
server generates a response, and the server clos
`TcpStream` will read from itself to see what the
our response to the stream. Overall, this `for` lo
turn and produce a series of streams for us to h

For now, our handling of the stream consists of
program if the stream has any errors; if there ar
message. We'll add more functionality for the su
reason we might receive errors from the `incom`
the server is that we're not actually iterating ove
over *connection attempts*. The connection might
reasons, many of them operating system specifi
systems have a limit to the number of simultane
support; new connection attempts beyond that
some of the open connections are closed.

Let's try running this code! Invoke `cargo run` in

127.0.0.1:7878 in a web browser. The browser sh
“Connection reset,” because the server isn’t curr
when you look at your terminal, you should see
when the browser connected to the server!

```
     Running `target/debug/hello`
Connection established!
Connection established!
Connection established!
```

Sometimes, you’ll see multiple messages printec
might be that the browser is making a request f
other resources, like the *favicon.ico* icon that ap

It could also be that the browser is trying to con
because the server isn’t responding with any dat
and is dropped at the end of the loop, the conne
implementation. Browsers sometimes deal with
because the problem might be temporary. The i
successfully gotten a handle to a TCP connectior

Remember to stop the program by pressing ctrl
particular version of the code. Then restart `carg`
code changes to make sure you’re running the r

Reading the Request

Let’s implement the functionality to read the rec
the concerns of first getting a connection and th
connection, we’ll start a new function for proces
`handle_connection` function, we’ll read data fro
can see the data being sent from the browser. C
20-2.

Filename: src/main.rs

```
use std::io::prelude::*;
use std::net::TcpStream;
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStrean
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    println!("Request: {}", String::from_u
}
```

Listing 20-2: Reading from the `TcpStream` and p

We bring `std::io::prelude` into scope to get a from and write to the stream. In the `for` loop ir printing a message that says we made a connec `handle_connection` function and pass the `stre`

In the `handle_connection` function, we've made reason is that the `TcpStream` instance keeps tra internally. It might read more data than we aske time we ask for data. It therefore needs to be `mu` change; usually, we think of "reading" as not nee need the `mut` keyword.

Next, we need to actually read from the stream. declare a `buffer` on the stack to hold the data t 512 bytes in size, which is big enough to hold the sufficient for our purposes in this chapter. If we arbitrary size, buffer management would need t simple for now. We pass the buffer to `stream.r` `TcpStream` and put them in the buffer.

Second, we convert the bytes in the buffer to a s `String::from_utf8_lossy` function takes a `&[u` The "lossy" part of the name indicates the behav

invalid UTF-8 sequence: it will replace the invalic
`U+FFFD REPLACEMENT CHARACTER`. You might see
characters in the buffer that aren't filled by requ

Let's try this code! Start the program and make a
Note that we'll still get an error page in the brow
terminal will now look similar to this:

```
$ cargo run
   Compiling hello v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/hello`
Request: GET / HTTP/1.1
Host: 127.0.0.1:7878
User-Agent: Mozilla/5.0 (Windows NT 10.0;
Firefox/52.0
Accept: text/html,application/xhtml+xml,ap
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
������������������������������������
```

Depending on your browser, you might get sligh
printing the request data, we can see why we ge
browser request by looking at the path after `Re`
connections are all requesting /, we know the br
because it's not getting a response from our pro

Let's break down this request data to understan
program.

A Closer Look at an HTTP Request

HTTP is a text-based protocol, and a request tak

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

The first line is the *request line* that holds inform
requesting. The first part of the request line indi
`GET` or `POST`, which describes how the client is
`GET` request.

The next part of the request line is /, which indic

(URI) the client is requesting: a URI is almost, but *Resource Locator (URL)*. The difference between U purposes in this chapter, but the HTTP spec use: mentally substitute URL for URI here.

The last part is the HTTP version the client uses, *CRLF sequence*. (CRLF stands for *carriage return* a typewriter days!) The CRLF sequence can also be carriage return and `\n` is a line feed. The CRLF s from the rest of the request data. Note that whe line start rather than `\r\n`.

Looking at the request line data we received fro that `GET` is the method, */* is the request URI, and

After the request line, the remaining lines startir `GET` requests have no body.

Try making a request from a different browser o as *127.0.0.1:7878/test*, to see how the request da

Now that we know what the browser is asking fo

Writing a Response

Now we'll implement sending data in response t the following format:

```
HTTP-Version Status-Code Reason-Phrase CRL
headers CRLF
message-body
```

The first line is a *status line* that contains the HT numeric status code that summarizes the result that provides a text description of the status coc headers, another CRLF sequence, and the body

Here is an example response that uses HTTP ver OK reason phrase, no headers, and no body:

```
HTTP/1.1 200 OK\r\n\r\n
```

The status code 200 is the standard success res HTTP response. Let's write this to the stream as

From the `handle_connection` function, remove
request data and replace it with the code in Listi

Filename: src/main.rs

```
fn handle_connection(mut stream: TcpStream
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    let response = "HTTP/1.1 200 OK\r\n\r\

    stream.write(response.as_bytes()).unwr
    stream.flush().unwrap();
}
```

Listing 20-3: Writing a tiny successful HTTP respo

The first new line defines the `response` variable
data. Then we call `as_bytes` on our `response` t
`write` method on `stream` takes a `&[u8]` and s
connection.

Because the `write` operation could fail, we use
Again, in a real application you would add error
and prevent the program from continuing until a
connection; `TcpStream` contains an internal buf
operating system.

With these changes, let's run our code and make
any data to the terminal, so we won't see any ou
Cargo. When you load *127.0.0.1:7878* in a web br
instead of an error. You've just hand-coded an H

Returning Real HTML

Let's implement the functionality for returning m
file, *hello.html*, in the root of your project directo
input any HTML you want; Listing 20-4 shows on

Filename: hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

Listing 20-4: A sample HTML file to return in a re

This is a minimal HTML5 document with a headi
the server when a request is received, we'll mod
Listing 20-5 to read the HTML file, add it to the r

Filename: src/main.rs

```
use std::fs;
// --snip--

fn handle_connection(mut stream: TcpStrean
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let contents = fs::read_to_string("hel

    let response = format!("HTTP/1.1 200 (

    stream.write(response.as_bytes()).unwr
    stream.flush().unwrap();
}
```

Listing 20-5: Sending the contents of *hello.html a*

We've added a line at the top to bring the stand
code for opening a file and reading the contents
Chapter 12 when we read the contents of a file f

Next, we use `format!` to add the file's contents

Run this code with `cargo run` and load *127.0.0.*
see your HTML rendered!

Currently, we're ignoring the request data in `bu`
contents of the HTML file unconditionally. That r

127.0.0.1:7878/something-else in your browser, y
response. Our server is very limited and is not w
customize our responses depending on the requ
for a well-formed request to */*.

Validating the Request and Selectively

Right now, our web server will return the HTML
requested. Let's add functionality to check that t
returning the HTML file and return an error if th
this we need to modify `handle_connection` , as
checks the content of the request received agair
like and adds `if` and `else` blocks to treat requ

Filename: src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStrean
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    if buffer.starts_with(get) {
        let contents = fs::read_to_string(

        let response = format!("HTTP/1.1 2

        stream.write(response.as_bytes()).
        stream.flush().unwrap();
    } else {
        // some other request
    }
}
```

Listing 20-6: Matching the request and handling
requests

First, we hardcode the data corresponding to th
Because we're reading raw bytes into the buffer
by adding the `b""` byte string syntax at the star
whether `buffer` starts with the bytes in `get` . If
well-formed request to */*, which is the success ca
returns the contents of our HTML file.

If `buffer` does *not* start with the bytes in `get`, i
request. We'll add code to the `else` block in a m
requests.

Run this code now and request *127.0.0.1:7878*; y
If you make any other request, such as *127.0.0.1*
connection error like those you saw when runni
20-2.

Now let's add the code in Listing 20-7 to the `els`
status code 404, which signals that the content f
also return some HTML for a page to render in t
the end user.

Filename: src/main.rs

```
// --snip--

} else {
    let status_line = "HTTP/1.1 404 NOT FO
    let contents = fs::read_to_string("404

    let response = format!("{}{}", status_

    stream.write(response.as_bytes()).unwr
    stream.flush().unwrap();
}
```

Listing 20-7: Responding with status code 404 ar
than / was requested

Here, our response has a status line with status
`NOT FOUND`. We're still not returning headers, ar
HTML in the file *404.html*. You'll need to create a
error page; again feel free to use any HTML you
Listing 20-8.

Filename: 404.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're ask
  </body>
</html>
```

Listing 20-8: Sample content for the page to sen

With these changes, run your server again. Requ the contents of *hello.html*, and any other reques return the error HTML from *404.html*.

A Touch of Refactoring

At the moment the `if` and `else` blocks have a files and writing the contents of the files to the s status line and the filename. Let's make the cod differences into separate `if` and `else` lines th line and the filename to variables; we can then u the code to read the file and write the response. after replacing the large `if` and `else` blocks.

Filename: src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStream
    // --snip--

    let (status_line, filename) = if buffe
        ("HTTP/1.1 200 OK\r\n\r\n", "hello
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n",
    };

    let contents = fs::read_to_string(file

    let response = format!("{}{}", status_

    stream.write(response.as_bytes()).unwr
    stream.flush().unwrap();
}
```

Listing 20-9: Refactoring the `if` and `else` block between the two cases

Now the `if` and `else` blocks only return the ap and filename in a tuple; we then use destructuri `status_line` and `filename` using a pattern in t Chapter 18.

The previously duplicated code is now outside tl `status_line` and `filename` variables. This mak between the two cases, and it means we have o want to change how the file reading and respon code in Listing 20-9 will be the same as that in Li

Awesome! We now have a simple web server in that responds to one request with a page of con requests with a 404 response.

Currently, our server runs in a single thread, me a time. Let's examine how that can be a problem Then we'll fix it so our server can handle multipl

Turning Our Single-Threaded S Multithreaded Server

Right now, the server will process each request i
second connection until the first is finished proc
and more requests, this serial execution would b
receives a request that takes a long time to proc
wait until the long request is finished, even if the
quickly. We'll need to fix this, but first, we'll look

Simulating a Slow Request in the Curre

We'll look at how a slow-processing request can
current server implementation. Listing 20-10 im
with a simulated slow response that will cause t
before responding.

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;
// --snip--

fn handle_connection(mut stream: TcpStream
    // --snip--

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n'

    let (status_line, filename) = if buffe
        ("HTTP/1.1 200 OK\r\n\r\n", "hellc
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(
        ("HTTP/1.1 200 OK\r\n\r\n", "hellc
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n",
    };

    // --snip--
}
```

Listing 20-10: Simulating a slow request by recog
seconds

This code is a bit messy, but it's good enough for
second request `sleep` , whose data our server r
after the `if` block to check for the request to */s*
the server will sleep for 5 seconds before render

You can see how primitive our server is: real libr
multiple requests in a much less verbose way!

Start the server using `cargo run`. Then open tw
http://127.0.0.1:7878/ and the other for *http://127*
URI a few times, as before, you'll see it respond
then load */*, you'll see that */* waits until `sleep` ha
loading.

There are multiple ways we could change how o
more requests back up behind a slow request; t
pool.

Improving Throughput with a Thread P

A *thread pool* is a group of spawned threads tha
task. When the program receives a new task, it a
to the task, and that thread will process the task
are available to handle any other tasks that com
processing. When the first thread is done proce
of idle threads, ready to handle a new task. A th
connections concurrently, increasing the throug

We'll limit the number of threads in the pool to a
Denial of Service (DoS) attacks; if we had our pro
request as it came in, someone making 10 millio
havoc by using up all our server's resources and
to a halt.

Rather than spawning unlimited threads, we'll ha
in the pool. As requests come in, they'll be sent t
will maintain a queue of incoming requests. Eac
off a request from this queue, handle the reque
another request. With this design, we can proce
is the number of threads. If each thread is respo
subsequent requests can still back up in the que
of long-running requests we can handle before r

This technique is just one of many ways to impr
Other options you might explore are the fork/joi
async I/O model. If you're interested in this topic
solutions and try to implement them in Rust; wit
these options are possible.

Before we begin implementing a thread pool, le should look like. When you're trying to design cc can help guide your design. Write the API of the want to call it; then implement the functionality implementing the functionality and then designi

Similar to how we used test-driven developmen compiler-driven development here. We'll write tl want, and then we'll look at errors from the com change next to get the code to work.

Code Structure If We Could Spawn a Thread f

First, let's explore how our code might look if it c connection. As mentioned earlier, this isn't our f potentially spawning an unlimited number of th 20-11 shows the changes to make to `main` to sp stream within the `for` loop.

Filename: src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
```

Listing 20-11: Spawning a new thread for each st

As you learned in Chapter 16, `thread::spawn` w the code in the closure in the new thread. If you browser, then / in two more browser tabs, you'll don't have to wait for */sleep* to finish. But as we r overwhelm the system because you'd be makin

Creating a Similar Interface for a Finite Numl

We want our thread pool to work in a similar, fa to a thread pool doesn't require large changes t

20-12 shows the hypothetical interface for a `Thr`
instead of `thread::spawn` .

Filename: src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```

Listing 20-12: Our ideal `ThreadPool` interface

We use `ThreadPool::new` to create a new threa
threads, in this case four. Then, in the `for` loop,
as `thread::spawn` in that it takes a closure the p
need to implement `pool.execute` so it takes th
the pool to run. This code won't yet compile, but
in how to fix it.

Building the `ThreadPool` Struct Using Compil

Make the changes in Listing 20-12 to *src/main.rs*,
errors from `cargo check` to drive our developm

```
$ cargo check
   Compiling hello v0.1.0 (file:///project
error[E0433]: failed to resolve. Use of ur
`ThreadPool`
  --> src\main.rs:10:16
   |
10 |     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^^^^^^ Use of
   `ThreadPool`

error: aborting due to previous error
```

Great! This error tells us we need a `ThreadPool`
now. Our `ThreadPool` implementation will be ir
web server is doing. So, let's switch the `hello` c

crate to hold our `ThreadPool` implementation. A
could also use the separate thread pool library f
thread pool, not just for serving web requests.

Create a *src/lib.rs* that contains the following, wh
`ThreadPool` struct that we can have for now:

Filename: src/lib.rs

```
pub struct ThreadPool;
```

Then create a new directory, *src/bin*, and move t
into *src/bin/main.rs*. Doing so will make the libra
directory; we can still run the binary in *src/bin/m*
the *main.rs* file, edit it to bring the library crate in
adding the following code to the top of *src/bin/m*

Filename: src/bin/main.rs

```
extern crate hello;
use hello::ThreadPool;
```

This code still won't work, but let's check it again
address:

```
$ cargo check
   Compiling hello v0.1.0 (file:///project
error[E0599]: no function or associated it
`hello::ThreadPool` in the current scope
 --> src/bin/main.rs:13:16
   |
13 |     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^^^^^^ functi
in
   `hello::ThreadPool`
```

This error indicates that next we need to create
for `ThreadPool`. We also know that `new` needs t
accept `4` as an argument and should return a `T`
the simplest `new` function that will have those c

Filename: src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool
        ThreadPool
    }
}
```

We chose `usize` as the type of the `size` param negative number of threads doesn't make any s the number of elements in a collection of threac for, as discussed in the "Integer Types" section o

Let's check the code again:

```
$ cargo check
   Compiling hello v0.1.0 (file:///project
warning: unused variable: `size`
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadP
  |                ^^^^
  |
  = note: #[warn(unused_variables)] on by
  = note: to avoid this warning, consider

error[E0599]: no method named `execute` fc
in the current scope
  --> src/bin/main.rs:18:14
   |
18 |         pool.execute(|| {
   |              ^^^^^^^
```

Now we get a warning and an error. Ignoring the occurs because we don't have an `execute` meth "Creating a Similar Interface for a Finite Number our thread pool should have an interface similar implement the `execute` function so it takes the idle thread in the pool to run.

We'll define the `execute` method on `ThreadPoo` Recall from the "Storing Closures Using Generic section in Chapter 13 that we can take closures traits: `Fn`, `FnMut`, and `FnOnce`. We need to deci We know we'll end up doing something similar t implementation, so we can look at what bounds on its parameter. The documentation shows us

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

The `F` type parameter is the one we're concern
related to the return value, and we're not conce
`spawn` uses `FnOnce` as the trait bound on `F`. Th
because we'll eventually pass the argument we g
further confident that `FnOnce` is the trait we wa
running a request will only execute that request
the `Once` in `FnOnce`.

The `F` type parameter also has the trait bound
`'static`, which are useful in our situation: we r
from one thread to another and `'static` becau
thread will take to execute. Let's create an `exec
take a generic parameter of type `F` with these b

Filename: src/lib.rs

```
impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {

    }
}
```

We still use the `()` after `FnOnce` because this `F
no parameters and doesn't return a value. Just li
type can be omitted from the signature, but eve
need the parentheses.

Again, this is the simplest implementation of the
but we're trying only to make our code compile.

```
$ cargo check
   Compiling hello v0.1.0 (file:///project
warning: unused variable: `size`
 --> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadP
  |                ^^^^
  |
  = note: #[warn(unused_variables)] on by
  = note: to avoid this warning, consider

warning: unused variable: `f`
 --> src/lib.rs:8:30
  |
8 |     pub fn execute<F>(&self, f: F)
  |                              ^
  |
  = note: to avoid this warning, consider
```

We're receiving only warnings now, which mean `cargo run` and make a request in the browser, that we saw at the beginning of the chapter. Our closure passed to `execute` yet!

> Note: A saying you might hear about languag Haskell and Rust, is "if the code compiles, it w universally true. Our project compiles, but it c building a real, complete project, this would b tests to check that the code compiles *and* has

---

Validating the Number of Threads in `new`

We'll continue to get warnings because we aren' to `new` and `execute`. Let's implement the bodie we want. To start, let's think about `new`. Earlier v `size` parameter, because a pool with a negative However, a pool with zero threads also makes n `usize`. We'll add code to check that `size` is gre `ThreadPool` instance and have the program par `assert!` macro, as shown in Listing 20-13.

Filename: src/lib.rs

```
impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads
    ///
    /// # Panics
    ///
    /// The `new` function will panic if t
    pub fn new(size: usize) -> ThreadPool
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}
```

Listing 20-13: Implementing `ThreadPool::new` t

We've added some documentation for our `Thre` that we followed good documentation practices situations in which our function can panic, as dis `cargo doc --open` and clicking the `ThreadPool` docs for `new` look like!

Instead of adding the `assert!` macro as we've c a `Result` like we did with `Config::new` in the I/ decided in this case that trying to create a threa an unrecoverable error. If you're feeling ambitio the following signature to compare both versior

```
pub fn new(size: usize) -> Result<ThreadPo
```

Creating Space to Store the Threads

Now that we have a way to know we have a valic pool, we can create those threads and store the returning it. But how do we "store" a thread? Let `thread::spawn` signature:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T + Send + 'static,
        T: Send + 'static
```

The `spawn` function returns a `JoinHandle<T>` , w
returns. Let's try using `JoinHandle` too and see
closures we're passing to the thread pool will ha
anything, so `T` will be the unit type `()` .

The code in Listing 20-14 will compile but doesn
changed the definition of `ThreadPool` to hold a
instances, initialized the vector with a capacity o
run some code to create the threads, and returr
them.

Filename: src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool
        assert!(size > 0);

        let mut threads = Vec::with_capaci

        for _ in 0..size {
            // create some threads and st
        }

        ThreadPool {
            threads
        }
    }

    // --snip--
}
```

Listing 20-14: Creating a vector for `ThreadPool`

We've brought `std::thread` into scope in the lil
`thread::JoinHandle` as the type of the items in

Once a valid size is received, our `ThreadPool` cr
`size` items. We haven't used the `with_capacit`
performs the same task as `Vec::new` but with a
space in the vector. Because we know we need t
doing this allocation up front is slightly more eff

resizes itself as elements are inserted.

When you run `cargo check` again, you'll get a fe
succeed.

A `Worker` Struct Responsible for Sending Cod

We left a comment in the `for` loop in Listing 20
Here, we'll look at how we actually create threac
`thread::spawn` as a way to create threads, and
code the thread should run as soon as the threa
want to create the threads and have them *wait* f
standard library's implementation of threads do
have to implement it manually.

We'll implement this behavior by introducing a r
`ThreadPool` and the threads that will manage tl
structure `Worker`, which is a common term in p
people working in the kitchen at a restaurant: th
from customers, and then they're responsible fc

Instead of storing a vector of `JoinHandle<()>` ir
instances of the `Worker` struct. Each `Worker` wi
instance. Then we'll implement a method on `Wo`
to run and send it to the already running thread
worker an `id` so we can distinguish between th
logging or debugging.

Let's make the following changes to what happe
We'll implement the code that sends the closure
set up in this way:

1. Define a `Worker` struct that holds an `id` a
2. Change `ThreadPool` to hold a vector of `Wo`
3. Define a `Worker::new` function that takes
   instance that holds the `id` and a thread sp
4. In `ThreadPool::new`, use the `for` loop cou
   `Worker` with that `id`, and store the worke

If you're up for a challenge, try implementing the
looking at the code in Listing 20-15.

Ready? Here is Listing 20-15 with one way to ma

Filename: src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool
        assert!(size > 0);

        let mut workers = Vec::with_capaci

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers
        }
    }
    // --snip--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-15: Modifying `ThreadPool` to hold `Wo` threads directly

We've changed the name of the field on `Thread` because it's now holding `Worker` instances inste use the counter in the `for` loop as an argumen new `Worker` in the vector named `workers`.

External code (like our server in *src/bin/main.rs*) implementation details regarding using a `Worke` make the `Worker` struct and its `new` function pr the `id` we give it and stores a `JoinHandle<()>` a new thread using an empty closure.

This code will compile and will store the numbe an argument to `ThreadPool::new`. But we're *stil* get in `execute`. Let's look at how to do that nex

Sending Requests to Threads via Channels

Now we'll tackle the problem that the closures g nothing. Currently, we get the closure we want t But we need to give `thread::spawn` a closure to during the creation of the `ThreadPool`.

We want the `Worker` structs that we just create held in the `ThreadPool` and send that code to it

In Chapter 16, you learned about *channels*—a si two threads—that would be perfect for this use as the queue of jobs, and `execute` will send a jo `Worker` instances, which will send the job to its

1. The `ThreadPool` will create a channel and channel.
2. Each `Worker` will hold on to the receiving s
3. We'll create a new `Job` struct that will hold the channel.
4. The `execute` method will send the job it w side of the channel.
5. In its thread, the `Worker` will loop over its execute the closures of any jobs it receives

Let's start by creating a channel in `ThreadPool:` the `ThreadPool` instance, as shown in Listing 20 anything for now but will be the type of item we

Filename: src/lib.rs

```rust
// --snip--
use std::sync::mpsc;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool
        assert!(size > 0);

        let (sender, receiver) = mpsc::cha

        let mut workers = Vec::with_capaci

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}
```

Listing 20-16: Modifying `ThreadPool` to store th `Job` instances

In `ThreadPool::new`, we create our new channe end. This will successfully compile, still with warn

Let's try passing a receiving end of the channel i creates the channel. We know we want to use th workers spawn, so we'll reference the `receiver` in Listing 20-17 won't quite compile yet.

Filename: src/lib.rs

```
impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool
        assert!(size > 0);

        let (sender, receiver) = mpsc::cha

        let mut workers = Vec::with_capaci

        for id in 0..size {
            workers.push(Worker::new(id, r
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Rece
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-17: Passing the receiving end of the ch

We've made some small and straightforward cha the channel into `Worker::new` , and then we use

When we try to check this code, we get this erro

```
$ cargo check
   Compiling hello v0.1.0 (file:///project
error[E0382]: use of moved value: `receive
  --> src/lib.rs:27:42
   |
27 |             workers.push(Worker::new(
   |
   previous iteration of loop
   |
   = note: move occurs because `receiver`
   `std::sync::mpsc::Receiver<Job>`, which
trait
```

The code is trying to pass `receiver` to multiple you'll recall from Chapter 16: the channel implen multiple *producer*, single *consumer*. This means v of the channel to fix this code. Even if we could, want to use; instead, we want to distribute the jo single `receiver` among all the workers.

Additionally, taking a job off the channel queue i the threads need a safe way to share and modify race conditions (as covered in Chapter 16).

Recall the thread-safe smart pointers discussed across multiple threads and allow the threads to `Arc<Mutex<T>>`. The `Arc` type will let multiple w will ensure that only one worker gets a job from shows the changes we need to make.

Filename: src/lib.rs

```
use std::sync::Arc;
use std::sync::Mutex;
// --snip--

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool
        assert!(size > 0);

        let (sender, receiver) = mpsc::cha

        let receiver = Arc::new(Mutex::new

        let mut workers = Vec::with_capaci

        for id in 0..size {
            workers.push(Worker::new(id, A
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    // --snip--
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<
{
        // --snip--
    }
}
```

Listing 20-18: Sharing the receiving end of the ch and `Mutex`

In `ThreadPool::new`, we put the receiving end o For each new worker, we clone the `Arc` to bum can share ownership of the receiving end.

With these changes, the code compiles! We're ge

Implementing the `execute` Method

Let's finally implement the `execute` method on from a struct to a type alias for a trait object tha

`execute` receives. As discussed in the “Creating
section of Chapter 19, type aliases allow us to m
20-19.

Filename: src/lib.rs

```
// --snip--

type Job = Box<dyn FnOnce() + Send + 'sta

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

Listing 20-19: Creating a `Job` type alias for a `Bo`
sending the job down the channel

After creating a new `Job` instance using the clos
job down the sending end of the channel. We're
that sending fails. This might happen if, for exam
executing, meaning the receiving end has stopp
moment, we can't stop our threads from execut
long as the pool exists. The reason we use `unwr`
won't happen, but the compiler doesn't know th

But we're not quite done yet! In the worker, our
`thread::spawn` still only *references* the receiving
the closure to loop forever, asking the receiving
running the job when it gets one. Let's make the
`Worker::new`.

Filename: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<
{
        let thread = thread::spawn(move ||
            loop {
                let job = receiver.lock().

                println!("Worker {} got a

                (*job)();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-20: Receiving and executing the jobs i

Here, we first call `lock` on the `receiver` to acq `unwrap` to panic on any errors. Acquiring a lock state, which can happen if some other thread pa than releasing the lock. In this situation, calling the correct action to take. Feel free to change th message that is meaningful to you.

If we get the lock on the mutex, we call `recv` to final `unwrap` moves past any errors here as well holding the sending side of the channel has shu method returns `Err` if the receiving side shuts

The call to `recv` blocks, so if there is no job yet, becomes available. The `Mutex<T>` ensures that trying to request a job.

Theoretically, this code should compile. Unfortu yet, and we get this error:

```
error[E0161]: cannot move a value of type
std::marker::Send: the size of std::ops::F
cannot be
statically determined
  --> src/lib.rs:63:17
   |
63 |                 (*job)();
   |                 ^^^^^^
```

This error is fairly cryptic because the problem is
closure that is stored in a `Box<T>` (which is wha
needs to move itself *out* of the `Box<T>` because
when we call it. In general, Rust doesn't allow us
because Rust doesn't know how big the value in
Chapter 15 that we used `Box<T>` precisely beca
size that we wanted to store in a `Box<T>` to get

As you saw in Listing 17-15, we can write metho
`self: Box<dyn Self>`, which allows the metho
stored in a `Box<T>`. That's exactly what we want
won't let us: the part of Rust that implements be
implemented using `self: Box<dyn Self>`. So R
could use `self: Box<dyn Self>` in this situatio
move the closure out of the `Box<T>`.

Rust is still a work in progress with places where
in the future, the code in Listing 20-20 should w
working to fix this and other issues! After you've
you to join in.

But for now, let's work around this problem usir
explicitly that in this case we can take ownershi
`self: Box<dyn Self>`; then, once we have own
This involves defining a new trait `FnBox` with th
`self: Box<dyn Self>` in its signature, defining
`FnOnce()`, changing our type alias to use the ne
the `call_box` method. These changes are show

Filename: src/lib.rs

```
trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<dyn FnBox + Send + 'static>

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<
{
        let thread = thread::spawn(move ||
            loop {
                let job = receiver.lock().

                println!("Worker {} got a

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-21: Adding a new trait `FnBox` to work
`Box<dyn FnOnce()>`

First, we create a new trait named `FnBox`. This t
which is similar to the `call` methods on the oth
`self: Box<dyn Self>` to take ownership of `sel`
`Box<T>`.

Next, we implement the `FnBox` trait for any type
trait. Effectively, this means that any `FnOnce()`
method. The implementation of `call_box` uses
of the `Box<T>` and call the closure.

We now need our `Job` type alias to be a `Box` of
trait `FnBox`. This will allow us to use `call_box` i

instead of invoking the closure directly. Impleme
`FnOnce()` closure means we don't have to chan
we're sending down the channel. Now Rust is ab
do is fine.

This trick is very sneaky and complicated. Don't
someday, it will be completely unnecessary.

With the implementation of this trick, our threac
`cargo run` and make some requests:

```
$ cargo run
   Compiling hello v0.1.0 (file:///project
warning: field is never used: `workers`
 --> src/lib.rs:7:5
  |
7 |     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `id`
  --> src/lib.rs:61:5
   |
61 |     id: usize,
   |     ^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by defaul

warning: field is never used: `thread`
  --> src/lib.rs:62:5
   |
62 |     thread: thread::JoinHandle<()>,
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by defaul

    Finished dev [unoptimized + debuginfo]
     Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

Success! We now have a thread pool that execut
There are never more than four threads created
if the server receives a lot of requests. If we mak
be able to serve other requests by having anoth

Note that if you open */sleep* in multiple browser
load 5 seconds apart from each other, because s
instances of the same request sequentially for c
caused by our web server.

After learning about the `while let` loop in Cha
we didn't write the worker thread code as shown

Filename: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<
{
        let thread = thread::spawn(move ||
            while let Ok(job) = receiver.l
                println!("Worker {} got a

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-22: An alternative implementation of

This code compiles and runs but doesn't result i
slow request will still cause other requests to wa
somewhat subtle: the `Mutex` struct has no publi
ownership of the lock is based on the lifetime of
`LockResult<MutexGuard<T>>` that the `lock` me
borrow checker can then enforce the rule that a
be accessed unless we hold the lock. But this im
lock being held longer than intended if we don't
the `MutexGuard<T>`. Because the values in the `w
the duration of the block, the lock remains held

`job.call_box()` , meaning other workers cannc

By using `loop` instead and acquiring the lock ar outside it, the `MutexGuard` returned from the `l` the `let job` statement ends. This ensures that `recv` , but it is released before the call to `job.ca` to be serviced concurrently.

Graceful Shutdown and Clean

The code in Listing 20-21 is responding to reque of a thread pool, as we intended. We get some v `thread` fields that we're not using in a direct wa up anything. When we use the less elegant ctrl-c other threads are stopped immediately as well, a request.

Now we'll implement the `Drop` trait to call `join` they can finish the requests they're working on b way to tell the threads they should stop acceptir see this code in action, we'll modify our server tc gracefully shutting down its thread pool.

Implementing the `Drop` Trait on `Threa`

Let's start with implementing `Drop` on our threa our threads should all join to make sure they fin first attempt at a `Drop` implementation; this coc

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker

            worker.thread.join().unwrap();
        }
    }
}
```

Listing 20-23: Joining each thread when the thre

First, we loop through each of the thread pool `w
because `self` is a mutable reference, and we al
For each worker, we print a message saying that
down, and then we call `join` on that worker's th
`unwrap` to make Rust panic and go into an ungr

Here is the error we get when we compile this co

```
error[E0507]: cannot move out of borrowed
  --> src/lib.rs:65:13
   |
65 |             worker.thread.join().unwr
   |             ^^^^^^ cannot move out of
```

The error tells us we can't call `join` because we
`worker` and `join` takes ownership of its argum
move the thread out of the `Worker` instance tha
consume the thread. We did this in Listing 17-15
`Option<thread::JoinHandle<()>>` instead, we ca
`Option` to move the value out of the `Some` varia
place. In other words, a `Worker` that is running
and when we want to clean up a `Worker`, we'll re
`Worker` doesn't have a thread to run.

So we know we want to update the definition of

Filename: src/lib.rs

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>
}
```

Now let's lean on the compiler to find the other
this code, we get two errors:

```
error[E0599]: no method named `join` found
`std::option::Option<std::thread::JoinHand
  --> src/lib.rs:65:27
   |
65 |             worker.thread.join().unwr
   |                           ^^^^

error[E0308]: mismatched types
  --> src/lib.rs:89:13
   |
89 |             thread,
   |             ^^^^^^
   |             |
   |             expected enum `std::optic
   `std::thread::JoinHandle`
   |             help: try using a variant
`Some(thread)`
   |
   = note: expected type
`std::option::Option<std::thread::JoinHand
              found type `std::thread::Joi
```

Let's address the second error, which points to t we need to wrap the `thread` value in `Some` whe following changes to fix this error:

Filename: src/lib.rs

```rust
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<
{
        // --snip--

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

The first error is in our `Drop` implementation. W to call `take` on the `Option` value to move `thre` changes will do so:

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker

            if let Some(thread) = worker.t
                thread.join().unwrap();
            }
        }
    }
}
```

As discussed in Chapter 17, the `take` method o and leaves `None` in its place. We're using `if le` thread; then we call `join` on the thread. If a wo know that worker has already had its thread cle case.

Signaling to the Threads to Stop Listen

With all the changes we've made, our code comp bad news is this code doesn't function the way w the closures run by the threads of the `Worker` in `join`, but that won't shut down the threads bec jobs. If we try to drop our `ThreadPool` with our main thread will block forever waiting for the fir

To fix this problem, we'll modify the threads so t signal that they should stop listening and exit th instances, our channel will send one of these tw

Filename: src/lib.rs

```
enum Message {
    NewJob(Job),
    Terminate,
}
```

This `Message` enum will either be a `NewJob` vari should run, or it will be a `Terminate` variant tha and stop.

We need to adjust the channel to use values of t as shown in Listing 20-24.

Filename: src/lib.rs

```rust
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

// --snip--

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(j
    }
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<
        Worker {

        let thread = thread::spawn(move ||
            loop {
                let message = receiver.loc

                match message {
                    Message::NewJob(job) =
                        println!("Worker {

                        job.call_box();
                    },
                    Message::Terminate =>
                        println!("Worker {

                        break;
                    },
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

Listing 20-24: Sending and receiving `Message` va
receives `Message::Terminate`

To incorporate the `Message` enum, we need to c
the definition of `ThreadPool` and the signature
method of `ThreadPool` needs to send jobs wrap
Then, in `Worker::new` where a `Message` is recei
processed if the `NewJob` variant is received, and
if the `Terminate` variant is received.

With these changes, the code will compile and c
as it did after Listing 20-21. But we'll get a warnin
messages of the `Terminate` variety. Let's fix this
implementation to look like Listing 20-25.

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate messag

        for _ in &mut self.workers {
            self.sender.send(Message::Term
        }

        println!("Shutting down all worker

        for worker in &mut self.workers {
            println!("Shutting down worker

            if let Some(thread) = worker.t
                thread.join().unwrap();
            }
        }
    }
}
```

Listing 20-25: Sending `Message::Terminate` to t
each worker thread

We're now iterating over the workers twice: once
each worker and once to call `join` on each wor
message and `join` immediately in the same loo
worker in the current iteration would be the one
channel.

To better understand why we need two separate

workers. If we used a single loop to iterate throu
a terminate message would be sent down the ch
worker's thread. If that first worker was busy pro
the second worker would pick up the terminate
down. We would be left waiting on the first work
because the second thread picked up the termin

To prevent this scenario, we first put all of our `T`
in one loop; then we join on all the threads in an
receiving requests on the channel once it gets a
sure that if we send the same number of termin
each worker will receive a terminate message be

To see this code in action, let's modify `main` to a
gracefully shutting down the server, as shown in

Filename: src/bin/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

Listing 20-26: Shut down the server after serving

You wouldn't want a real-world web server to sh
requests. This code just demonstrates that the g
working order.

The `take` method is defined in the `Iterator` tr
two items at most. The `ThreadPool` will go out o
`drop` implementation will run.

Start the server with `cargo run`, and make thre
error, and in your terminal you should see outpu

```
$ cargo run
   Compiling hello v0.1.0 (file:///project
    Finished dev [unoptimized + debuginfo]
     Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
Shutting down all workers.
Shutting down worker 0
Worker 1 was told to terminate.
Worker 2 was told to terminate.
Worker 0 was told to terminate.
Worker 3 was told to terminate.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

You might see a different ordering of workers ar this code works from the messages: workers 0 a then on the third request, the server stopped ac `ThreadPool` goes out of scope at the end of `ma` and the pool tells all workers to terminate. The w they see the terminate message, and then the th each worker thread.

Notice one interesting aspect of this particular e terminate messages down the channel, and befo messages, we tried to join worker 0. Worker 0 ha message, so the main thread blocked waiting fo each of the workers received the termination me main thread waited for the rest of the workers to received the termination message and were abl

Congrats! We've now completed our project; we thread pool to respond asynchronously. We're a of the server, which cleans up all the threads in t

Here's the full code for reference:

Filename: src/bin/main.rs

```
extern crate hello;
use hello::ThreadPool;

use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::fs;
use std::thread;
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n'

    let (status_line, filename) = if buffe
        ("HTTP/1.1 200 OK\r\n\r\n", "hellc
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(
        ("HTTP/1.1 200 OK\r\n\r\n", "hellc
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n",
    };

     let contents = fs::read_to_string(fil

     let response = format!("{}{}", status

     stream.write(response.as_bytes()).unw
     stream.flush().unwrap();
}
```

Filename: src/lib.rs

```
use std::thread;
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;

enum Message {
    NewJob(Job),
    Terminate,
}

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<dyn FnBox + Send + 'static>

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads
    ///
    /// # Panics
    ///
    /// The `new` function will panic if 
    pub fn new(size: usize) -> ThreadPool
        assert!(size > 0);

        let (sender, receiver) = mpsc::cha

        let receiver = Arc::new(Mutex::new

        let mut workers = Vec::with_capaci

        for id in 0..size {
            workers.push(Worker::new(id, 
        }

        ThreadPool {
            workers,
            sender,
```

```
        }
    }

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(j
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate messag

        for _ in &mut self.workers {
            self.sender.send(Message::Term
        }

        println!("Shutting down all worker

        for worker in &mut self.workers {
            println!("Shutting down worker

            if let Some(thread) = worker.t
                thread.join().unwrap();
            }
        }
    }
}

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<
        Worker {

        let thread = thread::spawn(move ||
            loop {
                let message = receiver.loc

                match message {
                    Message::NewJob(job) =
                        println!("Worker {

                        job.call_box();
                    },
```

```
                    Message::Terminate =>
                        println!("Worker {

                        break;
                    },
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

We could do more here! If you want to continue
some ideas:

- Add more documentation to `ThreadPool` a
- Add tests of the library's functionality.
- Change calls to `unwrap` to more robust err
- Use `ThreadPool` to perform some task oth
- Find a thread pool crate on *https://crates.io*
  using the crate instead. Then compare its A
  we implemented.

Summary

Well done! You've made it to the end of the bool
on this tour of Rust. You're now ready to implem
with other peoples' projects. Keep in mind that t
other Rustaceans who would love to help you w
your Rust journey.

Appendix

The following sections contain reference materia
journey.

Appendix A: Keywords

The following list contains keywords that are res
Rust language. As such, they cannot be used as
including names of functions, variables, parame
constants, macros, static values, attributes, type

Keywords Currently in Use

The following keywords currently have the funct

- `as` - perform primitive casting, disambigua item, or rename items in `use` and `extern`
- `break` - exit a loop immediately
- `const` - define constant items or constant
- `continue` - continue to the next loop itera
- `crate` - link an external crate or a macro v which the macro is defined
- `else` - fallback for `if` and `if let` contro
- `enum` - define an enumeration
- `extern` - link an external crate, function, o
- `false` - Boolean false literal
- `fn` - define a function or the function poin
- `for` - loop over items from an iterator, im ranked lifetime
- `if` - branch based on the result of a condi
- `impl` - implement inherent or trait functio
- `in` - part of `for` loop syntax
- `let` - bind a variable
- `loop` - loop unconditionally
- `match` - match a value to patterns
- `mod` - define a module
- `move` - make a closure take ownership of a
- `mut` - denote mutability in references, raw
- `pub` - denote public visibility in struct field
- `ref` - bind by reference
- `return` - return from function
- `Self` - a type alias for the type implement
- `self` - method subject or current module
- `static` - global variable or lifetime lasting
- `struct` - define a structure
- `super` - parent module of the current mod
- `trait` - define a trait

* `true` - Boolean true literal
* `type` - define a type alias or associated typ
* `unsafe` - denote unsafe code, functions, tr
* `use` - import symbols into scope
* `where` - denote clauses that constrain a ty
* `while` - loop conditionally based on the re

Keywords Reserved for Future Use

The following keywords do not have any functio
potential future use.

* `abstract`
* `alignof`
* `become`
* `box`
* `do`
* `final`
* `macro`
* `offsetof`
* `override`
* `priv`
* `proc`
* `pure`
* `sizeof`
* `typeof`
* `unsized`
* `virtual`
* `yield`

Raw identifiers

Raw identifiers let you use keywords where they
prefixing them with `r#`.

For example, `match` is a keyword. If you try to co

```
fn match(needle: &str, haystack: &str) ->
    haystack.contains(needle)
}
```

You'll get this error:

```
error: expected identifier, found keyword
 --> src/main.rs:4:4
  |
4 | fn match(needle: &str, haystack: &str)
  |    ^^^^^ expected identifier, found ke
```

You can write this with a raw identifier:

```
fn r#match(needle: &str, haystack: &str) -
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

Note the `r#` prefix on both the function name a

Motivation

This feature is useful for a few reasons, but the p
situations. For example, `try` is not a keyword ir
edition. So if you have a library that is written in
call it in Rust 2018, you'll need to use the raw ide

Appendix B: Operators and Sy

This appendix contains a glossary of Rust's synta
symbols that appear by themselves or in the cor
macros, attributes, comments, tuples, and brack

Operators

Table B-1 contains the operators in Rust, an exa
appear in context, a short explanation, and whe
an operator is overloadable, the relevant trait to
listed.

Table B-1: Operators

Operator	Example	Expl
`!`	`ident!(...)`, `ident!{...}`, `ident![...]`	Macro expa
`!`	`!expr`	Bitwise or l complemer
`!=`	`var != expr`	Nonequalit
`%`	`expr % expr`	Arithmetic
`%=`	`var %= expr`	Arithmetic assignment
`&`	`&expr`, `&mut expr`	Borrow
`&`	`&type`, `&mut type`, `&'a type`, `&'a mut type`	Borrowed p
`&`	`expr & expr`	Bitwise ANI
`&=`	`var &= expr`	Bitwise ANI assignment
`&&`	`expr && expr`	Logical ANI
`*`	`expr * expr`	Arithmetic
`*=`	`var *= expr`	Arithmetic and assignr
`*`	`*expr`	Dereferenc
`*`	`*const type`, `*mut type`	Raw pointe
`+`	`trait + trait`, `'a + trait`	Compound constraint
`+`	`expr + expr`	Arithmetic
`+=`	`var += expr`	Arithmetic assignment
`,`	`expr, expr`	Argument a separator
`-`	`- expr`	Arithmetic
`-`	`expr - expr`	Arithmetic

Operator	Example	Expl
`-=`	`var -= expr`	Arithmetic s and assignm
`->`	`fn(...) -> type`, `|...| -> type`	Function ar return type
`.`	`expr.ident`	Member ac
`..`	`..`, `expr..`, `..expr`, `expr..expr`	Right-exclu literal
`..=`	`..=expr`, `expr..=expr`	Right-inclus literal
`..`	`..expr`	Struct litera syntax
`..`	`variant(x, ..)`, `struct_type { x, .. }`	"And the re binding
`...`	`expr...expr`	In a patterr range patte
`/`	`expr / expr`	Arithmetic
`/=`	`var /= expr`	Arithmetic assignment
`:`	`pat: type`, `ident: type`	Constraints
`:`	`ident: expr`	Struct field
`:`	`'a: loop {...}`	Loop label
`;`	`expr;`	Statement terminator
`;`	`[...; len]`	Part of fixe syntax
`<<`	`expr << expr`	Left-shift
`<<=`	`var <<= expr`	Left-shift ar
`<`	`expr < expr`	Less than c
`<=`	`expr <= expr`	Less than o comparisor
`=`	`var = expr`, `ident = type`	Assignmen

Operator	Example	Expl
`==`	`expr == expr`	Equality co
`=>`	`pat => expr`	Part of mat
`>`	`expr > expr`	Greater tha
`>=`	`expr >= expr`	Greater tha comparisor
`>>`	`expr >> expr`	Right-shift
`>>=`	`var >>= expr`	Right-shift a assignment
`@`	`ident @ pat`	Pattern bin
`^`	`expr ^ expr`	Bitwise exc
`^=`	`var ^= expr`	Bitwise exc assignment
`|`	`pat | pat`	Pattern alte
`|`	`expr | expr`	Bitwise OR
`|=`	`var |= expr`	Bitwise OR assignment
`||`	`expr || expr`	Logical OR
`?`	`expr?`	Error propa

Non-operator Symbols

The following list contains all non-letters that do
don't behave like a function or method call.

Table B-2 shows symbols that appear on their o
locations.

Table B-2: Stand-Alone Syntax

Symbol	
`'ident`	Named life
`...u8`, `...i32`, `...f64`, `...usize`, etc.	Numeric li
`"..."`	String liter

Symbol	
`r"..."`, `r#"..."#`, `r##"..."##`, etc.	Raw string processed
`b"..."`	Byte string of a string
`br"..."`, `br#"..."#`, `br##"..."##`, etc.	Raw byte s and byte s
`'...'`	Character
`b'...'`	ASCII byte
`|...| expr`	Closure
`!`	Always em functions
`_`	"Ignored" integer lite

Table B-3 shows symbols that appear in the con hierarchy to an item.

Table B-3: Path-Related Syntax

Symbol	
`ident::ident`	Namespac
`::path`	Path relat explicitly a
`self::path`	Path relat explicitly r
`super::path`	Path relat module
`type::ident`, `<type as trait>::ident`	Associated
`<type>::...`	Associated directly na `<[T]>::.`
`trait::method(...)`	Disambigu trait that d
`type::method(...)`	Disambigu type for w

Symbol	
`<type as trait>::method(...)`	Disambigu trait and t

Table B-4 shows symbols that appear in the con

Table B-4: Generics

Symbol	
`path<...>`	Specifies parame `Vec<u8>`)
`path::<...>`, `method::<...>`	Specifies parame method in an exp turbofish (e.g., `"4`
`fn ident<...> ...`	Define generic fu
`struct ident<...> ...`	Define generic st
`enum ident<...> ...`	Define generic er
`impl<...> ...`	Define generic im
`for<...> type`	Higher-ranked lif
`type<ident=type>`	A generic type wh have specific assi

Table B-5 shows symbols that appear in the con parameters with trait bounds.

Table B-5: Trait Bound Constraints

Symbol	
`T: U`	Generic parameter `T` (implement `U`
`T: 'a`	Generic type `T` must c type cannot transitively lifetimes shorter than
`T : 'static`	Generic type `T` contain than `'static` ones
`'b: 'a`	Generic lifetime `'b` mu
`T: ?Sized`	Allow generic type para type

Symbol	
'a + trait, trait + trait	Compound type constr

Table B-6 shows symbols that appear in the con specifying attributes on an item.

Table B-6: Macros and Attributes

Symbol	Ex
#[meta]	Oute
#![meta]	Inne
$ident	Macr
$ident:kind	Macr
$(…)…	Macr

Table B-7 shows symbols that create comments

Table B-7: Comments

Symbol	Exp
//	Line comr
//!	Inner line
///	Outer line
/*...*/	Block com
/*!...*/	Inner bloc
/**...*/	Outer blo

Table B-8 shows symbols that appear in the con

Table B-8: Tuples

Symbol	
()	Empty tuple
(expr)	Parenthesize
(expr,)	Single-eleme
(type,)	Single-eleme
(expr, ...)	Tuple expres

Symbol	
`(type, ...)`	Tuple type
`expr(expr, ...)`	Function call tuple `struc`
`ident!(...)`, `ident!{...}`, `ident![...]`	Macro invoc
`expr.0`, `expr.1`, etc.	Tuple indexi

Table B-9 shows the contexts in which curly brac

Table B-9: Curly Brackets

Context	Ex
`{...}`	Bloc
`Type {...}`	`str`

Table B-10 shows the contexts in which square b

Table B-10: Square Brackets

Context	
`[...]`	Array literal
`[expr; len]`	Array literal co
`[type; len]`	Array type con
`expr[expr]`	Collection inde `IndexMut`)
`expr[..]`, `expr[a..]`, `expr[..b]`, `expr[a..b]`	Collection inde slicing, using `R` `RangeFull` as

Appendix C: Derivable Traits

In various places in the book, we've discussed th apply to a struct or enum definition. The `derive` implement a trait with its own default implemer with the `derive` syntax.

In this appendix, we provide a reference of all th
you can use with `derive`. Each section covers:

- What operators and methods deriving this
- What the implementation of the trait provi
- What implementing the trait signifies abou
- The conditions in which you're allowed or n
- Examples of operations that require the tra

If you want different behavior than that provide
standard library documentation for each trait fo
implement them.

The rest of the traits defined in the standard libr
types using `derive`. These traits don't have sen
you to implement them in the way that makes s
accomplish.

An example of a trait that can't be derived is `Dis`
end users. You should always consider the appr
end user. What parts of the type should an end
would they find relevant? What format of the da
The Rust compiler doesn't have this insight, so it
behavior for you.

The list of derivable traits provided in this appen
can implement `derive` for their own traits, mak
`derive` with truly open-ended. Implementing `d`
macro, which is covered in Appendix D.

`Debug` for Programmer Output

The `Debug` trait enables debug formatting in for
adding `:?` within `{}` placeholders.

The `Debug` trait allows you to print instances of
you and other programmers using your type can
point in a program's execution.

The `Debug` trait is required, for example, in use
prints the values of instances given as argument
programmers can see why the two instances we

`PartialEq` and `Eq` for Equality Compa

The `PartialEq` trait allows you to compare inst
and enables use of the `==` and `!=` operators.

Deriving `PartialEq` implements the `eq` metho
structs, two instances are equal only if *all* fields
equal if any fields are not equal. When derived o
itself and not equal to the other variants.

The `PartialEq` trait is required, for example, wi
which needs to be able to compare two instance

The `Eq` trait has no methods. Its purpose is to s
annotated type, the value is equal to itself. The
that also implement `PartialEq`, although not a
can implement `Eq`. One example of this is floati
implementation of floating point numbers state
a-number (`NaN`) value are not equal to each oth

An example of when `Eq` is required is for keys i
`HashMap<K, V>` can tell whether two keys are th

`PartialOrd` and `Ord` for Ordering Com

The `PartialOrd` trait allows you to compare ins
A type that implements `PartialOrd` can be use
operators. You can only apply the `PartialOrd` t
`PartialEq`.

Deriving `PartialOrd` implements the `partial_`
`Option<Ordering>` that will be `None` when the
ordering. An example of a value that doesn't pro
values of that type can be compared, is the not-
Calling `partial_cmp` with any floating point nur
will return `None`.

When derived on structs, `PartialOrd` compares
value in each field in the order in which the field
When derived on enums, variants of the enum c
are considered less than the variants listed later

The `PartialOrd` trait is required, for example, f

`rand` crate that generates a random value in th
high value.

The `Ord` trait allows you to know that for any tw
valid ordering will exist. The `Ord` trait implemer
`Ordering` rather than an `Option<Ordering>` be
possible. You can only apply the `Ord` trait to typ
and `Eq` (and `Eq` requires `PartialEq`). When de
behaves the same way as the derived implemen
`PartialOrd`.

An example of when `Ord` is required is when st
structure that stores data based on the sort ord

`Clone` and `Copy` for Duplicating Value

The `Clone` trait allows you to explicitly create a
duplication process might involve running arbitr
the "Ways Variables and Data Interact: Clone" se
information on `Clone`.

Deriving `Clone` implements the `clone` method
whole type, calls `clone` on each of the parts of t
values in the type must also implement `Clone` t

An example of when `Clone` is required is when
The slice doesn't own the type instances it conta
`to_vec` will need to own its instances, so `to_ve`
type stored in the slice must implement `Clone`.

The `Copy` trait allows you to duplicate a value by
stack; no arbitrary code is necessary. See the "St
Chapter 4 for more information on `Copy`.

The `Copy` trait doesn't define any methods to pr
overloading those methods and violating the ass
being run. That way, all programmers can assum
fast.

You can derive `Copy` on any type whose parts a
apply the `Copy` trait to types that also implemer
implements `Copy` has a trivial implementation o
as `Copy`.

The `Copy` trait is rarely required; types that imp
available, meaning you don't have to call `clone`

Everything possible with `Copy` you can also acc
might be slower or have to use `clone` in places.

`Hash` for Mapping a Value to a Value o

The `Hash` trait allows you to take an instance of
instance to a value of fixed size using a hash fun
`hash` method. The derived implementation of t
of calling `hash` on each of the parts of the type,
implement `Hash` to derive `Hash`.

An example of when `Hash` is required is in stori
data efficiently.

`Default` for Default Values

The `Default` trait allows you to create a default
implements the `default` function. The derived
function calls the `default` function on each par
values in the type must also implement `Default`

The `Default::default` function is commonly us
update syntax discussed in the "Creating Instanc
Update Syntax" section in Chapter 5. You can cu
then set and use a default value for the rest of t
`..Default::default()`.

The `Default` trait is required when you use the
`Option<T>` instances, for example. If the `Optio`
`unwrap_or_default` will return the result of `De`
in the `Option<T>`.

Appendix D: Macros

We've used macros like `println!` throughout t
what a macro is and how it works. This appendi

- What macros are and how they differ from
- How to define a declarative macro to do m
- How to define a procedural macro to creat

We’re covering the details of macros in an apper
Rust. Macros have changed and, in the near futu
the rest of the language and standard library sir
likely to become out-of-date than the rest of the
guarantees, the code shown here will continue t
may be additional capabilities or easier ways to
the time of this publication. Bear that in mind w
from this appendix.

The Difference Between Macros and Fu

Fundamentally, macros are a way of writing cod
known as *metaprogramming*. In Appendix C, we
generates an implementation of various traits fc
and `vec!` macros throughout the book. All of th
code than the code you’ve written manually.

Metaprogramming is useful for reducing the am
maintain, which is also one of the roles of functi
additional powers that functions don’t have.

A function signature must declare the number a
has. Macros, on the other hand, can take a varia
call `println!("hello")` with one argument or
two arguments. Also, macros are expanded befc
meaning of the code, so a macro can, for examp
A function can’t, because it gets called at runtim
at compile time.

The downside to implementing a macro instead
are more complex than function definitions beca
writes Rust code. Due to this indirection, macro
to read, understand, and maintain than functior

Another difference between macros and functio
namespaced within modules like function defini
name clashes when using external crates, you h
the scope of your project at the same time as yc
using the `#[macro_use]` annotation. The followi

macros defined in the `serde` crate into the scop

```
#[macro_use]
extern crate serde;
```

If `extern crate` was able to bring macros into s annotation, you would be prevented from using macros with the same name. In practice, this co more crates you use, the more likely it is.

There is one last important difference between define or bring macros into scope *before* you cal define functions anywhere and call them anywh

Declarative Macros with `macro_rules!` Metaprogramming

The most widely used form of macros in Rust ar sometimes referred to as *macros by example*, `ma` *macros*. At their core, declarative macros allow y Rust `match` expression. As discussed in Chapter structures that take an expression, compare the patterns, and then run the code associated with compare a value to patterns that have code ass the value is the literal Rust source code passed t compared with the structure of that source code pattern is the code that replaces the code passe during compilation.

To define a macro, you use the `macro_rules!` c `macro_rules!` by looking at how the `vec!` mac we can use the `vec!` macro to create a new vec example, the following macro creates a new vec

```
let v: Vec<u32> = vec![1, 2, 3];
```

We could also use the `vec!` macro to make a ve string slices. We wouldn't be able to use a functi wouldn't know the number or type of values up

Let's look at a slightly simplified definition of the

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Listing D-1: A simplified version of the `vec!` ma

> Note: The actual definition of the `vec!` macr
> code to preallocate the correct amount of me
> optimization that we don't include here to ma

The `#[macro_export]` annotation indicates that whenever the crate in which we're defining the r annotation, even if someone depending on this annotation, the macro wouldn't be brought into

We then start the macro definition with `macro_r` we're defining *without* the exclamation mark. Th by curly brackets denoting the body of the macr

The structure in the `vec!` body is similar to the Here we have one arm with the pattern `( $( $x` the block of code associated with this pattern. If block of code will be emitted. Given that this is t only one valid way to match; any other will be ar have more than one arm.

Valid pattern syntax in macro definitions is diffe in Chapter 18 because macro patterns are matc rather than values. Let's walk through what the p mean; for the full macro pattern syntax, see the

First, a set of parentheses encompasses the who `$`) followed by a set of parentheses, which capt within the parentheses for use in the replaceme which matches any Rust expression and gives th

The comma following `$()` indicates that a litera
optionally appear after the code that matches th
following the comma specifies that the pattern m
precedes the `*`.

When we call this macro with `vec![1, 2, 3];`,
with the three expressions `1`, `2`, and `3`.

Now let's look at the pattern in the body of the c
`temp_vec.push()` code within the `$()*` part is
`$()` in the pattern, zero or more times dependi
matches. The `$x` is replaced with each expressi
with `vec![1, 2, 3];`, the code generated that r
following:

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

We've defined a macro that can take any numbe
generate code to create a vector containing the

Given that most Rust programmers will *use* mac
discuss `macro_rules!` any further. To learn mor
the online documentation or other resources, su
Macros".

Procedural Macros for Custom `derive`

The second form of macros is called *procedural m*
functions (which are a type of procedure). Proce
as an input, operate on that code, and produce s
than matching against patterns and replacing th
macros do. At the time of this writing, you can o
your traits to be implemented on a type by spec
annotation.

We'll create a crate named `hello_macro` that de
one associated function named `hello_macro`. R
implement the `HelloMacro` trait for each of thei
macro so users can annotate their type with `#[c`

implementation of the `hello_macro` function. T
`Hello, Macro! My name is TypeName!` where
which this trait has been defined. In other words
another programmer to write code like Listing D

Filename: src/main.rs

```
extern crate hello_macro;
#[macro_use]
extern crate hello_macro_derive;

use hello_macro::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Listing D-2: The code a user of our crate will be a
procedural macro

This code will print `Hello, Macro! My name is`
step is to make a new library crate, like this:

```
$ cargo new hello_macro --lib
```

Next, we'll define the `HelloMacro` trait and its as

Filename: src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

We have a trait and its function. At this point, ou
to achieve the desired functionality, like so:

```
extern crate hello_macro;

use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

However, they would need to write the impleme wanted to use with `hello_macro`; we want to sp work.

Additionally, we can't yet provide a default imple function that will print the name of the type the have reflection capabilities, so it can't look up th macro to generate code at compile time.

The next step is to define the procedural macro. macros need to be in their own crate. Eventually convention for structuring crates and macro cra `foo`, a custom derive procedural macro crate is crate called `hello_macro_derive` inside our `he

```
$ cargo new hello_macro_derive --lib
```

Our two crates are tightly related, so we create t directory of our `hello_macro` crate. If we chang we'll have to change the implementation of the `hello_macro_derive` as well. The two crates wil and programmers using these crates will need t bring them both into scope. We could instead ha `hello_macro_derive` as a dependency and reex the way we've structured the project makes it pc `hello_macro` even if they don't want the `derive`

We need to declare the `hello_macro_derive` cr also need functionality from the `syn` and `quote` we need to add them as dependencies. Add the `hello_macro_derive`:

Filename: hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "0.14.4"
quote = "0.6.3"
```

To start defining the procedural macro, place th
src/lib.rs file for the `hello_macro_derive` crate.
until we add a definition for the `impl_hello_mac`

Filename: hello_macro_derive/src/lib.rs

```
extern crate proc_macro;
extern crate syn;
#[macro_use]
extern crate quote;

use proc_macro::TokenStream;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStre
    // Construct a represntation of Rust c
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

Listing D-3: Code that most procedural macro cr
Rust code

Notice the way we've split the functions in D-3; t
procedural macro crate you see or create, becau
macro more convenient. What you choose to do
`impl_hello_macro` function is called will be diffe
macro's purpose.

We've introduced three new crates: `proc_macro`
crate comes with Rust, so we didn't need to add
Cargo.toml. The `proc_macro` crate allows us to c
containing that Rust code. The `syn` crate parses
structure that we can perform operations on. Th

structures and turns them back into Rust code. T
parse any sort of Rust code we might want to ha
code is no simple task.

The `hello_macro_derive` function will get callec
`#[derive(HelloMacro)]` on a type. The reason i
`hello_macro_derive` function here with `proc_m`
`HelloMacro`, which matches our trait name; tha
macros follow.

This function first converts the `input` from a `To`
we can then interpret and perform operations o
The `parse` function in `syn` takes a `TokenStrea`
representing the parsed Rust code. The followin
the `DeriveInput` struct we get from parsing the

```
DeriveInput {
    // --snip--

    ident: Ident(
        "Pancakes"
    ),
    body: Struct(
        Unit
    )
}
```

The fields of this struct show that the Rust code
`ident` (identifier, meaning the name) of `Pancak`
struct for describing all sorts of Rust code; check
`DeriveInput` for more information.

At this point, we haven't defined the `impl_hello`
build the new Rust code we want to include. But
also a `TokenStream` which is added to the code
they compile their crate, they'll get extra functio

You might have noticed that we're calling `unwra`
`syn::parse` function fails here. Panicking on err
code because `proc_macro_derive` functions mu
`Result` to conform to the procedural macro AP
example by using `unwrap`; in production code, y
messages about what went wrong by using `pan`

Now that we have the code to turn the annotate

a `DeriveInput` instance, let's generate the code
trait on the annotated type:

Filename: hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My
(#name));
            }
        }
    };
    gen.into()
}
```

We get an `Ident` struct instance containing the
type using `ast.ident`. The code in Listing D-2 s
`Ident("Pancakes")`.

The `quote!` macro lets us write the Rust code t
result of its execution is not what is expected by
converted to a `TokenStream` by calling the `into`
intermediate representation and returns a value

This macro also provides some very cool templa
and `quote!` will replace it with the value in the
do some repetition similar to the way regular m
crate's docs for a thorough introduction.

We want our procedural macro to generate an i
trait for the type the user annotated, which we c
implementation has one function, `hello_macro`
functionality we want to provide: printing `Hello`
name of the annotated type.

The `stringify!` macro used here is built into R
`1 + 2`, and at compile time turns the expressio
. This is different than `format!` or `println!`, w
turn the result into a `String`. There is a possibil
expression to print literally, so we use `stringif`
allocation by converting `#name` to a string literal

At this point, `cargo build` should complete suc

hello_macro_derive . Let's hook up these crate
procedural macro in action! Create a new binary
using cargo new pancakes . We need to add he
as dependencies in the pancakes crate's *Cargo.*
of hello_macro and hello_macro_derive to *ht*
dependencies; if not, you can specify them as p

```
[dependencies]
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_ma
```

Put the code from Listing D-2 into *src/main.rs*, ar
Hello, Macro! My name is Pancakes! The imp
from the procedural macro was included withou
implement it; the #[derive(HelloMacro)] adde

The Future of Macros

In the future, Rust will expand declarative and p
better declarative macro system with the macro
procedural macros for more powerful tasks thar
under development at the time of this publicatio
documentation for the latest information.

Appendix E: Translations of th

For resources in languages other than English. M
Translations label to help or let us know about a

- Português (BR)
- Português (PT)
- Tiếng việt
- 简体中文, alternate
- Українська
- Español
- Italiano
- Русский
- 한국어
- 日本語
- Français

- Polski
- עברית
- Cebuano
- Tagalog

Appendix F - How Rust "Nightly Rust"

This appendix is about how Rust is made and hc developer.

Stability Without Stagnation

As a language, Rust cares a *lot* about the stabilit rock-solid foundation you can build on, and if th would be impossible. At the same time, if we car may not find out important flaws until after thei change things.

Our solution to this problem is what we call "sta guiding principle is this: you should never have t stable Rust. Each upgrade should be painless, bu fewer bugs, and faster compile times.

Choo, Choo! Release Channels and Ridi

Rust development operates on a *train schedule*. the `master` branch of the Rust repository. Relea model, which has been used by Cisco IOS and ot three *release channels* for Rust:

- Nightly
- Beta
- Stable

Most Rust developers primarily use the stable ch experimental new features may use nightly or b

Here's an example of how the development and that the Rust team is working on the release of R

December of 2015, but it will provide us with rea
is added to Rust: a new commit lands on the `ma
nightly version of Rust is produced. Every day is
created by our release infrastructure automatica
look like this, once a night:

```
nightly: * - - * - - *
```

Every six weeks, it's time to prepare a new releas
repository branches off from the `master` branch
releases:

```
nightly: * - - * - - *
                     |
beta:                *
```

Most Rust users do not use beta releases activel
system to help Rust discover possible regression
nightly release every night:

```
nightly: * - - * - - * - - * - - *
                     |
beta:                *
```

Let's say a regression is found. Good thing we ha
before the regression snuck into a stable release
nightly is fixed, and then the fix is backported to
of beta is produced:

```
nightly: * - - * - - * - - * - - * - - *
                     |
beta:                * - - - - - - - - *
```

Six weeks after the first beta was created, it's tim
branch is produced from the `beta` branch:

```
nightly: * - - * - - * - - * - - * - - * -
                     |
beta:                * - - - - - - - - *
                                       |
stable:                                *
```

Hooray! Rust 1.5 is done! However, we've forgot
have gone by, we also need a new beta of the *ne*
`stable` branches off of `beta`, the next version

again:

```
nightly: * - - * - - * - - * - - * - - * -
                     |
beta:                * - - - - - - - - - *
                                         |
stable:                                  *
```

This is called the "train model" because every six
but still has to take a journey through the beta c
release.

Rust releases every six weeks, like clockwork. If y
release, you can know the date of the next one:
having releases scheduled every six weeks is tha
feature happens to miss a particular release, the
happening in a short time! This helps reduce pre
features in close to the release deadline.

Thanks to this process, you can always check ou
yourself that it's easy to upgrade to: if a beta rel
can report it to the team and get it fixed before
Breakage in a beta release is relatively rare, but
bugs do exist.

Unstable Features

There's one more catch with this release model:
technique called "feature flags" to determine wh
release. If a new feature is under active develop
therefore, in nightly, but behind a *feature flag*. If
work-in-progress feature, you can, but you must
annotate your source code with the appropriate

If you're using a beta or stable release of Rust, y
the key that allows us to get practical use with n
stable forever. Those who wish to opt into the b
who want a rock-solid experience can stick with
break. Stability without stagnation.

This book only contains information about stabl
still changing, and surely they'll be different betw
when they get enabled in stable builds. You can
features online.

Rustup and the Role of Rust Nightly

Rustup makes it easy to change between differe
global or per-project basis. By default, you'll hav
nightly, for example:

```
$ rustup install nightly
```

You can see all of the *toolchains* (releases of Rus
have installed with `rustup` as well. Here's an ex
Windows computer:

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

As you can see, the stable toolchain is the defau
the time. You might want to use stable most of t
project, because you care about a cutting-edge f
`rustup override` in that project's directory to s
`rustup` should use when you're in that director

```
$ cd ~/projects/needs-nightly
$ rustup override set nightly
```

Now, every time you call `rustc` or `cargo` inside
will make sure that you are using nightly Rust, ra
This comes in handy when you have a lot of Rus

The RFC Process and Teams

So how do you learn about these new features?
Request For Comments (RFC) process. If you'd like
write up a proposal, called an RFC.

Anyone can write RFCs to improve Rust, and the
discussed by the Rust team, which is comprised
list of the teams on Rust's website, which includ
language design, compiler implementation, infra
The appropriate team reads the proposal and th
of their own, and eventually, there's consensus t

If the feature is accepted, an issue is opened on

can implement it. The person who implements i
proposed the feature in the first place! When th
the `master` branch behind a feature gate, as we
section.

After some time, once Rust developers who use
out the new feature, team members will discuss
nightly, and decide if it should make it into stabl
move forward, the feature gate is removed, and
It rides the trains into a new stable release of Ru

G - Other useful tools

In this appendix, we'll talk about some additiona
project, and are useful when developing Rust co

Automatic formatting with `ru

`rustfmt` is a tool that can re-format your code a
projects use `rustfmt` to prevent arguments abo
Rust: just do what the tool does!

`rustfmt` is not at 1.0 yet, but a preview is availa
Please give it a try and let us know how it goes!

To install `rustfmt`:

```
$ rustup component add rustfmt-preview
```

This will give you both `rustfmt` and `cargo-fmt`
`rustc` and `cargo`. To take any Cargo project ar

```
$ cargo fmt
```

IDE integration with the Rust

To help IDE integration, the Rust project distribu
as in http://langserver.org/. This can be used by
plugin for Visual Studio: Code.

The `rls` is not at 1.0 yet, but a preview is availa
Please give it a try and let us know how it goes!

To install the `rls`:

```
$ rustup component add rls-preview
```

Then, install the language server support in your