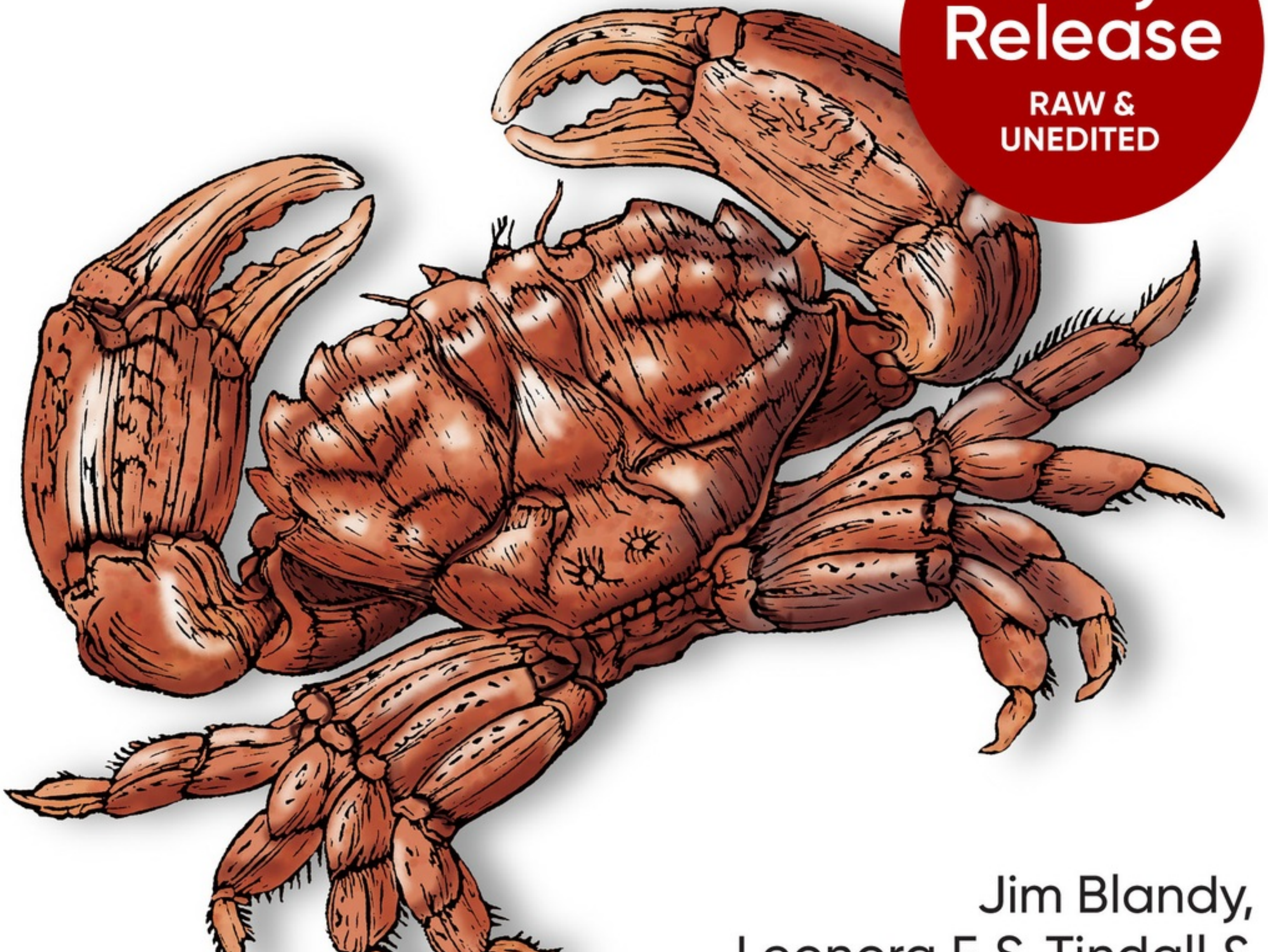
O'REILLY®

3rd Edition

Programming Rust

Fast, Safe Systems Development

Early Release
RAW & UNEDITED



Jim Blandy,
Leonora F. S. Tindall &
Jason Orendorff

Programming Rust

THIRD EDITION

Fast, Safe Systems Development

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jim Blandy, Jason Orendorff, and Leonora F. S. Tindall

O'REILLY®

Programming Rust

by Jim Blandy, Jason Orendorff, and Leonora F. S. Tindall

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Developmental Editor: Jeff Bleiel

Production Editor: Clare Laylock

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

March 2026: Third Edition

Revision History for the Early Release

- 2025-03-13: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098176235* for release details.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17617-4

[LSI]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. A Tour of Rust

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

Rust presents the authors of a book like this one with a challenge: what gives the language its character is not some specific, amazing feature that we can show off on the first page, but rather, the way all its parts are designed to work together smoothly in service of the goals we laid out in the last chapter: safe, performant systems programming. Each part of the language is best justified in the context of all the rest.

So rather than tackle one language feature at a time, we've prepared a tour of a few small but complete programs, each of which introduces some more features of the language, in context:

- As a warm-up, we have a program that does a simple calculation on its command-line arguments, with unit tests. This shows Rust's core types and introduces *traits*.

- Next, we build a web server. We'll use a third-party library to handle the details of HTTP and introduce string handling, closures, and error handling.

- Our last program plots a beautiful fractal, distributing the computation across multiple threads for speed. This includes an example of a generic function, illustrates how to handle something like a buffer of pixels, and shows off Rust's support for concurrency.

Rust's promise to prevent undefined behavior with minimal impact on performance influences the design of every part of the system, from the standard data structures like vectors and strings to the way Rust programs use third-party libraries. The details of how this is managed are covered throughout the book. But for now, we want to show you that Rust is a capable and pleasant language to use.

First, of course, you need to install Rust on your computer.

rustup and Cargo

The best way to install Rust is to use `rustup`. Go to *https://rustup.rs* and follow the instructions there.

You can, alternatively, go to the Rust website to get pre-built packages for Linux, macOS, and Windows. Rust is also included in some operating system distributions. We prefer `rustup` because it's a tool for managing Rust installations, like RVM for Ruby or NVM for Node. For example, when a new version of Rust is released, you'll be able to upgrade with zero clicks by typing `rustup update`.

In any case, once you've completed the installation, you should have three new commands available at your command line:

```
$ cargo --version
cargo 1.85.0 (d73d2caf9 2024-12-31)
$ rustc --version
rustc 1.85.0 (4d91de4e4 2025-02-17)
$ rustdoc --version
rustdoc 1.85.0 (4d91de4e4 2025-02-17)
```

Here, the `$` is the command prompt; on Windows, this would be `PS C:\>` or something similar. In this transcript we run the three commands we installed, asking each to report which version it is. Taking each command in turn:

- `cargo` is Rust's compilation manager, package manager, and general-purpose tool. You can use Cargo to start a new project, build and run your program, and manage any external libraries your code depends on.
- `rustc` is the Rust compiler. Usually we let Cargo invoke the compiler for us, but sometimes it's useful to run it directly.
- `rustdoc` is the Rust documentation tool. If you write documentation in comments of the appropriate form in your program's source code, `rustdoc` can build nicely formatted HTML from them. Like `rustc`, we usually let Cargo run `rustdoc` for us.

As a convenience, Cargo can create a new Rust package for us, with some standard metadata arranged appropriately:

```
$ cargo new hello
    Creating binary (application) `hello` package
```

This command creates a new package directory named *hello*, ready to build a command-line executable.

Looking inside the package's top-level directory:

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
drwx------. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
```

```
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src
```

We can see that Cargo has created a file *Cargo.toml* to hold metadata for the package. At the moment this file doesn't contain much:

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

[dependencies]
```

If our program ever acquires dependencies on other libraries, we can record them in this file, and Cargo will take care of downloading, building, and updating those libraries for us. We'll cover the *Cargo.toml* file in detail in [Link to Come].

Cargo has set up our package for use with the `git` version control system, creating a *.git* metadata subdirectory and a *.gitignore* file. You can tell Cargo to skip this step by passing `--vcs none` to `cargo new` on the command line.

The *src* subdirectory contains the actual Rust code:

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

It seems that Cargo has begun writing the program on our behalf. The *main.rs* file contains the text:

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

In Rust, you don’t even need to write your own “Hello, World!” program. And this is the extent of the boilerplate for a new Rust program: two files, totaling nine lines.

We can invoke the `cargo run` command from any directory in the package to build and run our program:

```
$ cargo run
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in
0.28s
     Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

Here, Cargo has invoked the Rust compiler, `rustc`, and then run the executable it produced. Cargo places the executable in the *target* subdirectory at the top of the package:

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb    198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb     68 Sep 22 21:37 incremental
$ ../target/debug/hello
Hello, world!
```

When we’re through, Cargo can clean up the generated files for us:

```
$ cargo clean
     Removed 21 files, 7.7MiB total
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

Rust Functions

Rust’s syntax is deliberately unoriginal. If you are familiar with C, C++, Java, or JavaScript, you can probably find your way through the general structure of a Rust program. Here is a function that computes the greatest common divisor of two integers, using Euclid’s algorithm. You can add this code to the end of *src/main.rs*:

```rust
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}
```

The `fn` keyword (pronounced “fun”) introduces a function. Here, we’re defining a function named `gcd`, which takes two parameters `n` and `m`, each of which is of type `u64`, an unsigned 64-bit integer. The `->` token precedes the return type: our function returns a `u64` value. Four-space indentation is standard Rust style.

Rust’s machine integer type names reflect their size and signedness: `i32` is a signed 32-bit integer; `u8` is an unsigned 8-bit integer (used for “byte” values), and so on. The `isize` and `usize` types hold pointer-sized signed and unsigned integers, 32 bits long on 32-bit platforms, and 64 bits long on 64-bit platforms. Rust also has two floating-point types, `f32` and `f64`, which are the IEEE single- and double-precision floating-point types, like `float` and `double` in C and C++.

By default, once a variable is initialized, its value can’t be changed, but placing the `mut` keyword (pronounced “mute,” short for *mutable*) before the parameters `n` and `m` allows our function body to assign to them. In practice, most variables don't get assigned to; the mut keyword on those that do can be a helpful hint when reading code.

The function's body starts with a call to the assert! macro, verifying that neither argument is zero. The ! character marks this as a macro invocation, not a function call. Like the assert macro in C and C++, Rust's assert! checks that its argument is true, and if it is not, terminates the program with a helpful message including the source location of the failing check; this kind of abrupt termination is called a *panic*. Unlike C and C++, in which assertions can be skipped, Rust always checks assertions regardless of how the program was compiled. There is also a debug_assert! macro, whose assertions are skipped when the program is compiled for speed.

The heart of our function is a while loop containing an if statement and an assignment. Unlike C and C++, Rust does not require parentheses around the conditional expressions, but it does require curly braces around the statements they control.

A let statement declares a local variable, like t in our function. We don't need to write out t's type, as long as Rust can infer it from how the variable is used. In our function, the only type that works for t is u64, matching m and n. Rust only infers types within function bodies: you must write out the types of function parameters and return values, as we did before. If we wanted to spell out t's type, we could write:

```
let t: u64 = m;
```

Rust has a return statement, but the gcd function doesn't need one. If a function body ends with an expression that is *not* followed by a semicolon, that's the function's return value. In fact, any block surrounded by curly braces can function as an expression. For example, this is an expression that prints a message and then yields x.cos() as its value:

```
{
    println!("evaluating cos x");
    x.cos()
}
```

It’s typical in Rust to use this form to establish the function’s value when control “falls off the end” of the function, and use `return` statements only for explicit early returns from the midst of a function.

Writing and Running Unit Tests

Rust has simple support for testing built into the language. To test our `gcd` function, we can add this code at the end of *src/main.rs*:

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);
    assert_eq!(gcd(2 * 3 * 54321, 5 * 7 * 54321), 54321);
}
```

Here we define a function named `test_gcd`, which calls `gcd` and checks that it returns correct values. The `#[test]` atop the definition marks `test_gcd` as a test function, to be skipped in normal compilations, but included and called automatically if we run our program with the `cargo test` command. We can have test functions scattered throughout our source tree, placed next to the code they exercise, and `cargo test` will automatically gather them up and run them all.

The `#[test]` marker is an example of an *attribute*. Attributes are an open-ended system for marking functions and other declarations with extra information, like attributes in C++ and C#, or annotations in Java. They’re used to control compiler warnings and code style checks, include code conditionally (like `#ifdef` in C and C++), tell Rust how to interact with code written in other languages, and so on. We’ll see more examples of attributes as we go.

With our `gcd` and `test_gcd` definitions added to the *hello* package we created at the beginning of the chapter, and our current directory somewhere within the package's subtree, we can run the tests as follows:

```
$ cargo test
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished `test` profile [unoptimized + debuginfo] target(s)
in 0.35s
     Running unittests src/main.rs
(.../hello/target/debug/deps/hello-2375...)

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
filtered out;
    finished in 0.00s
```

Handling Command-Line Arguments

In order for our program to take a series of numbers as command-line arguments and print their greatest common divisor, we can replace the `main` function in *src/main.rs* with the following:

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg).expect("error parsing
argument"));
    }

    if numbers.len() == 0 {
        eprintln!("Usage: gcd NUMBER ...");
        std::process::exit(1);
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    println!("The greatest common divisor of {numbers:?} is
{d}");
}
```

This is a large block of code, so let's take it piece by piece:

```
use std::str::FromStr;
use std::env;
```

The first `use` declaration brings the standard library *trait* `FromStr` into scope. A trait is a collection of methods that types can implement. Any type that implements the `FromStr` trait has a `from_str` method that tries to parse a value of that type from a string. The `u64` type implements `FromStr`, and we'll call `u64::from_str` to parse our command-line arguments. Although we never use the name `FromStr` elsewhere in the program, a trait must be in scope in order to use its methods. We'll cover traits in detail in [Link to Come].

The second `use` declaration brings in the `std::env` module, which provides several useful functions and types for interacting with the execution environment, including the `args` function, which gives us access to the program's command-line arguments.

Moving on to the program's `main` function:

```
fn main() {
```

Our `main` function doesn't return a value, so we can simply omit the `->` and return type that would normally follow the parameter list.

```
let mut numbers = Vec::new();
```

We declare a mutable local variable `numbers` and initialize it to an empty vector. `Vec` is Rust's growable vector type, analogous to C++'s `std::vector`, a Python list, or a JavaScript array. Even though vectors are designed to be grown and shrunk dynamically, we must still mark the variable `mut` for Rust to let us push numbers onto the end of it.

The type of `numbers` is `Vec<u64>`, a vector of `u64` values, but as before, we don't need to write that out. Rust will infer it for us, in part because what we push onto the vector are `u64` values, but also because we pass the vector's elements to `gcd`, which accepts only `u64` values.

```
for arg in env::args().skip(1) {
```

Here we use a `for` loop to process our command-line arguments, setting the variable `arg` to each argument in turn and evaluating the loop body.

The `std::env` module's `args` function returns an *iterator*, a value that produces each argument on demand, and indicates when we're done. Iterators are ubiquitous in Rust; the standard library includes other iterators that produce the elements of a vector, the lines of a file, messages received on a communications channel, and almost anything else that makes sense to loop over. Rust's iterators are very efficient: the compiler is usually able to translate them into the same code as a handwritten loop. We'll show how this works and give examples in [Link to Come].

Beyond their use with `for` loops, iterators include a broad selection of methods you can use directly. For example, the first value produced by the iterator returned by `args` is always the name of the program being run. We want to skip that, so we call the iterator's `skip` method to produce a new iterator that omits that first value.

```
numbers.push(u64::from_str(&arg).expect("error parsing
argument"));
```

Here we call u64::from_str to attempt to parse our command-line argument arg as an unsigned 64-bit integer. Rather than a method we're invoking on some u64 value we have at hand, u64::from_str is a function associated with the u64 type, akin to a static method in C++ or Java. The from_str function doesn't return a u64 directly, but rather a Result value that indicates whether the parse succeeded or failed. A Result value is one of two variants:

- A value written Ok(v), indicating that the parse succeeded and v is the value produced
- A value written Err(e), indicating that the parse failed and e is an error value explaining why

Functions that do anything that might fail, such as doing input or output or otherwise interacting with the operating system, can return Result types whose Ok variants carry successful results—the count of bytes transferred, the file opened, and so on—and whose Err variants carry an error code indicating what went wrong. Unlike most modern languages, Rust does not have exceptions: all errors are handled using either Result or panic, as outlined in [Link to Come].

We use Result's expect method to check the success of our parse. If the result is an Err(e), expect prints a message that includes a description of e and exits the program immediately. However, if the result is Ok(v), expect simply returns v itself, which we are finally able to push onto the end of our vector of numbers.

```
if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}
```

There's no greatest common divisor of an empty set of numbers, so we check that our vector has at least one element and exit the program with an error if it doesn’t. We use the `eprintln!` macro to write our error message to the standard error output stream.

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

This loop uses `d` as its running value, updating it to stay the greatest common divisor of all the numbers we’ve processed so far. As before, we must mark `d` as mutable so that we can assign to it in the loop.

The `for` loop has two surprising bits to it. First, we wrote `for m in &numbers[1..]`; what is the `&` operator for? Second, we wrote `gcd(d, *m)`; what is the `*` in `*m` for? These two details are complementary to each other.

Up to this point, our code has operated only on simple values like integers that fit in fixed-size blocks of memory. But now we’re about to iterate over a vector, which could be of any size whatsoever—possibly very large. Rust is cautious when handling such values: it wants to leave the programmer in control over memory consumption, making it clear how long each value lives, while still ensuring memory is freed promptly when no longer needed.

So when we iterate, we want to tell Rust that *ownership* of the vector should remain with `numbers`; we are merely *borrowing* its elements for the loop. The `&` operator in `&numbers[1..]` borrows a *reference* to the vector’s elements from the second onward. The `for` loop iterates over the referenced elements, letting `m` borrow each element in succession. The `*` operator in `*m` *dereferences* `m`, yielding the value it refers to; this is the next `u64` we want to pass to `gcd`. Finally, since `numbers` owns the vector, Rust automatically frees it when `numbers` goes out of scope at the end of `main`.

Rust’s rules for ownership and references are key to Rust’s memory management and safe concurrency; we discuss them in detail in Chapters 3 and 4. You’ll need to be comfortable with those rules to be comfortable in Rust, but for this introductory tour, all you need to know is that `&x` borrows a reference to `x`, and that `*r` is the value that the reference `r` refers to.

Continuing our walk through the program:

```
println!("The greatest common divisor of {numbers:?} is {d}");
```

Having iterated over the elements of `numbers`, the program prints the results to the standard output stream. The `println!` macro takes a template string, replaces the parts in braces `{...}` with the values of `numbers` and `d`, and writes the result to the standard output stream.

Unlike C and C++, where `main` returns zero to indicate success, or a nonzero exit status if something went wrong, Rust assumes that if `main` returns at all, the program finished successfully. Only by explicitly calling functions like `expect` or `std::process::exit` can we cause the program to terminate with an error status code.

The `cargo run` command allows us to pass arguments to our program, so we can try out our command-line handling:

```
$ cargo run 42 56
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in
0.22s
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished `dev` profile [unoptimized + debuginfo] target(s) in
0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 799459
28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
$ cargo run 83
    Finished `dev` profile [unoptimized + debuginfo] target(s) in
0.02s
```

```
     Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in
0.02s
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
```

We've used a few features from Rust's standard library in this section. If you're curious about what else is available, we strongly encourage you to try out Rust's online documentation. It has a live search feature that makes exploration easy and even includes links to the source code. The `rustup` command automatically installs a copy on your computer when you install Rust itself. You can view the standard library documentation on the Rust website, or in your browser with the command:

```
$ rustup doc --std
```

Serving Pages to the Web

One of Rust's strengths is the collection of freely available library packages published on the website crates.io. The `cargo` command makes it easy for your code to use a crates.io package: it will download the right version of the package, build it, and update it as requested. A Rust package, whether a library or an executable, is called a *crate*; Cargo and crates.io both derive their names from this term.

To show how this works, we'll put together a simple web server using the `actix-web` web framework crate, the `serde` serialization crate, and various other crates on which they depend. As shown in Figure 1-1, our website will prompt the user for two numbers and compute their greatest common divisor.

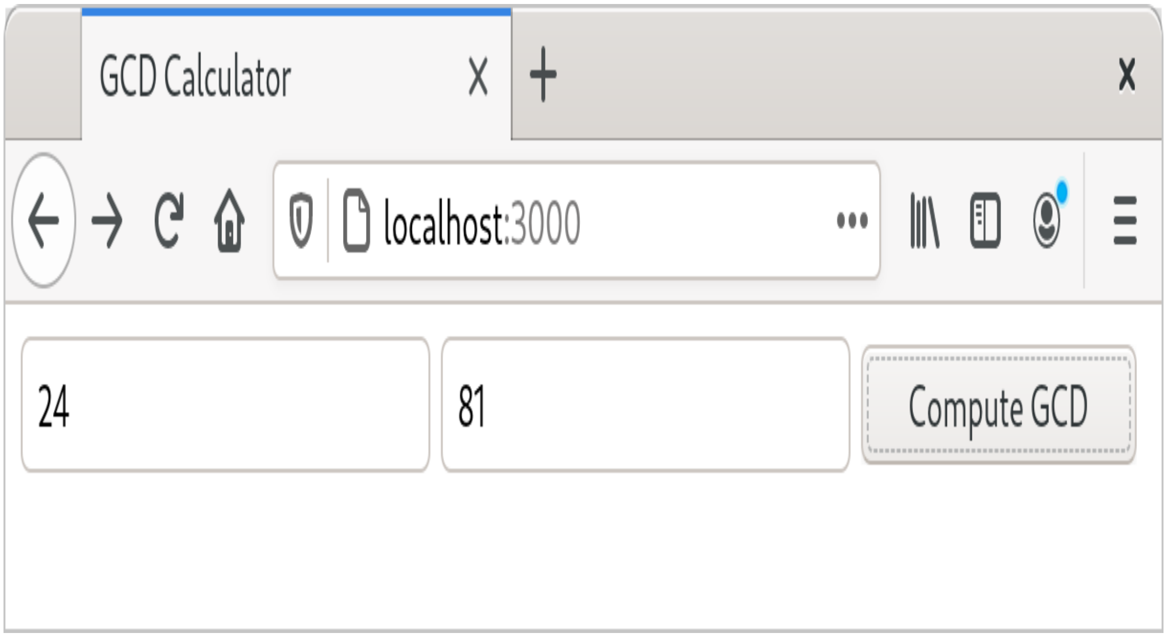


Figure 1-1. Web page offering to compute GCD

First, we'll have Cargo create a new package for us, named `actix-gcd`:

```
$ cargo new actix-gcd
     Creating binary (application) `actix-gcd` package
$ cd actix-gcd
```

Then, we'll tell Cargo which packages we want to use:

```
$ cargo add actix-web
     Updating crates.io index
       Adding actix-web v4.9.0 to dependencies
     ...
$ cargo add serde --features derive
     Updating crates.io index
       Adding serde v1.0.217 to dependencies
     ...
```

These commands add entries to our new project's *Cargo.toml* file, which now looks like this:

```
[package]
```

```toml
name = "actix-gcd"
version = "0.1.0"
edition = "2024"

[dependencies]
actix-web = "4.9.0"
serde = { version = "1.0.217", features = ["derive"] }
```

Instead of using `cargo add`, you could simply open *Cargo.toml* in your editor and add the last two lines yourself. From now on, we'll show *Cargo.toml* snippets instead of `cargo add` commands, as that is a bit easier to copy and paste and includes the version numbers for each package we're using.

Each line in the `[dependencies]` section of *Cargo.toml* gives the name of a crate on crates.io, and the version of that crate we would like to use. New versions of these crates are published frequently. By default, whenever we build this project from scratch on a new machine or run `cargo update`, Cargo will download the latest compatible version of these packages, perhaps Actix 4.10.1 and Serde 1.0.220. We'll discuss version management in more detail in [Link to Come].

Crates can have optional features: parts of the interface or implementation that not all users need, but that nonetheless make sense to include in that crate. The `serde` crate offers a wonderfully terse way to handle data from web forms, but according to `serde`'s documentation, it is only available if we select the crate's `derive` feature, so we've requested it in our *Cargo.toml* file as shown.

Note that we need only name those crates we'll use directly; `cargo` takes care of bringing in whatever other crates those need in turn.

For our first iteration, we'll keep the web server simple: it will serve only the page that prompts the user for numbers to compute with. In *actix-gcd/src/main.rs*, we'll place the following text:

```rust
use actix_web::{web, App, HttpResponse, HttpServer};
```

```
#[actix_web::main]
async fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000")
        .expect("error binding server to address")
        .run()
        .await
        .expect("error running server");
}

async fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                <input type="text" name="n"/>
                <input type="text" name="m"/>
                <button type="submit">Compute GCD</button>
                </form>
            "#
        )
}
```

We start with a `use` declaration to make some of the `actix-web` crate's definitions easier to get at. When we write `use actix_web::{...}`, each of the names listed inside the curly brackets becomes directly usable in our code; instead of having to spell out the full name `actix_web::HttpResponse` each time we use it, we can simply refer to it as `HttpResponse`. (We'll get to the `serde` crate in a bit.)

This time our `main` function is an `async fn`, marked with the attribute `#[actix_web::main]`. Actix is written using *asynchronous code* to support serving thousands of connections at a time without spawning thousands of system threads. Rust's `async` and `await` features are in the same general family as `async/await` in C# and JavaScript, or the lightweight processes in Erlang. They're a way to deal with tasks that don't need the CPU all the time—perhaps because, as in our case, they will spend a lot of time waiting for the network. A single CPU core can keep many concurrent `async` tasks running responsively without the overhead of system threads. We'll explain in [Link to Come].

The body of our `main` function is simple: it calls `HttpServer::new` to create a server that responds to requests for a single path, `"/"`; prints a message reminding us how to connect to it; and then sets it listening on TCP port 3000 on the local machine.

The argument we pass to `HttpServer::new` is the Rust *closure* expression `|| { App::new() ... }`. A closure is a value that can be called as if it were a function. This closure takes no arguments, but if it did, their names would appear between the `||` vertical bars. The `{ ... }` is the body of the closure. When we start our server, Actix starts a pool of threads to handle incoming requests. Each thread calls our closure to get a fresh copy of the `App` value that tells it how to route and handle requests.

The closure calls `App::new` to create a new, empty `App` and then calls its `route` method to add a single route for the path `"/"`. The handler provided for that route, `web::get().to(get_index)`, treats HTTP `GET` requests by calling the function `get_index`. The `route` method returns the same `App` it was invoked on, now enhanced with the new route. Since there's no semicolon at the end of the closure's body, the `App` is the closure's return value, ready for the `HttpServer` thread to use.

The `get_index` function builds an `HttpResponse` value representing the response to an HTTP `GET /` request. `HttpResponse::Ok()` represents an HTTP `200 OK` status, indicating that the request succeeded. We call its `content_type` and `body` methods to fill in the details of the response; each call returns the `HttpResponse` it was applied to, with the modifications made. Finally, the return value from `body` serves as the return value of `get_index`.

Since the response text contains a lot of double quotes, we write it using the Rust “raw string” syntax: the letter `r`, zero or more hash marks (that is, the `#` character), a double quote, and then the contents of the string, terminated by another double quote followed by the same number of hash marks. Any character may occur within a raw string without being escaped, including double quotes; in fact, no escape sequences like `\"` are recognized. We can always ensure the string ends where we intend by using more hash marks around the quotes than ever appear in the text.

Having written *main.rs*, we can use the `cargo run` command to do everything needed to set it running: fetching the needed crates, compiling them, building our own program, linking everything together, and starting it up:

```
$ cargo run
    Updating crates.io index
 Downloading crates ...
  Downloaded serde v1.0.100
  Downloaded actix-web v1.0.8
  Downloaded serde_derive v1.0.100
...
   Compiling serde_json v1.0.40
   Compiling actix-router v0.1.5
   Compiling actix-http v0.2.10
   Compiling awc v0.2.7
   Compiling actix-web v1.0.8
   Compiling gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in
1m 24s
     Running `/home/jimb/rust/actix-gcd/target/debug/actix-gcd`
Serving on http://localhost:3000...
```

At this point, we can visit the given URL in our browser and see the page shown earlier in Figure 1-1.

Unfortunately, clicking Compute GCD doesn’t do anything, other than navigate our browser to a blank page. Let’s fix that next, by adding another route to our `App` to handle the `POST` request from our form.

It's finally time to use the serde crate we listed in our *Cargo.toml* file: it provides a handy tool that will help us process the form data. First, we'll need to add the following use directive to the top of *src/main.rs*:

```
use serde::Deserialize;
```

Rust programmers typically gather all their use declarations together toward the top of the file, but this isn't strictly necessary: Rust allows declarations to occur in any order, as long as they appear at the appropriate level of nesting.

Next, let's define a Rust structure type that represents the values we expect from our form:

```
#[derive(Deserialize)]
struct GcdParameters {
    n: u64,
    m: u64,
}
```

This defines a new type named GcdParameters that has two fields, n and m, each of which is a u64—the argument type our gcd function expects.

The annotation above the struct definition is an attribute, like the #[test] attribute we used earlier to mark test functions. Placing a #[derive(Deserialize)] attribute above a type definition tells the serde crate to examine the type when the program is compiled and automatically generate code to parse a value of this type from data in the format that HTML forms use for POST requests. In fact, that attribute is sufficient to let you parse a GcdParameters value from almost any sort of structured data: JSON, YAML, TOML, or any one of a number of other textual and binary formats. The serde crate also provides a Serialize attribute that generates code to do the reverse, taking Rust values and writing them out in a structured format.

The comma after `m: u64` is optional, since it’s the last field. Rust consistently permits an extra trailing comma everywhere commas are used, and it is the standard style to include it whenever a list spans multiple lines. This includes array elements and function arguments. Using the same syntax on every line means less fiddling with punctuation when things change, and it makes for cleaner diffs.

With this definition in place, we can write our handler function quite easily:

```
async fn post_gcd(form: web::Form<GcdParameters>) -> HttpResponse
{
    if form.n == 0 || form.m == 0 {
        return HttpResponse::BadRequest()
            .content_type("text/html")
            .body("Computing the GCD with zero is boring.");
    }

    let response = format!(
        "The greatest common divisor of the numbers {} and {} \
            is <b>{}</b>\n",
        form.n,
        form.m,
        gcd(form.n, form.m),
    );

    HttpResponse::Ok()
        .content_type("text/html")
        .body(response)
}
```

For a function to serve as an Actix request handler, its arguments must all have types Actix knows how to extract from an HTTP request. Our `post_gcd` function takes one argument, `form`, whose type is `web::Form<GcdParameters>`. Actix knows how to extract a value of any type `web::Form<T>` from an HTTP request if, and only if, `T` can be deserialized from HTML form `POST` data. Since we’ve placed the `#[derive(Deserialize)]` attribute on our `GcdParameters` type definition, Actix can deserialize it from form data, so request handlers can expect a `web::Form<GcdParameters>` value as a parameter. These relationships between types and functions are all worked out at compile time; if you write a handler function with an argument type that Actix doesn't know how to handle, the Rust compiler lets you know of your mistake immediately.

Looking inside `post_gcd`, the function first returns an HTTP `400 BAD REQUEST` error if either parameter is zero, since our `gcd` function will panic if they are. Then, it constructs a response to the request using the `format!` macro. The `format!` macro is just like the `println!` macro, except that instead of writing the text to the standard output, it returns it as a string. In this example, since the values we want to print are not simple variable names, we must use empty braces `{}` to mark the place where we want to insert them, then pass the values as extra arguments. Once it has obtained the text of the response, `post_gcd` wraps it up in an HTTP `200 OK` response, sets its content type, and returns it to be delivered to the sender.

We also have to register `post_gcd` as the handler for the form. We'll replace our `main` function with this version:

```rust
#[actix_web::main]
async fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
            .route("/gcd", web::post().to(post_gcd))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000")
        .expect("error binding server to address")
        .run()
        .await
        .expect("error running server");
}
```

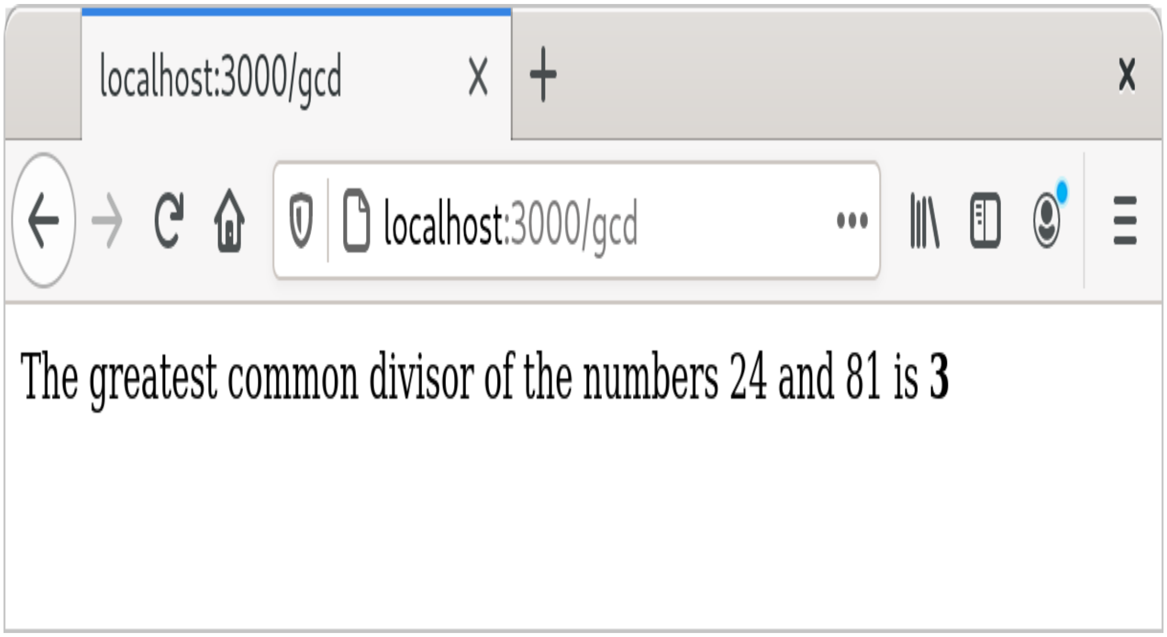
The only change here is that we've added another call to `route`, establishing `web::post().to(post_gcd)` as the handler for the path

"/gcd".

The last remaining piece is the gcd function we wrote earlier, in "Rust Functions". Add that code to the *actix-gcd/src/main.rs* file. With that in place, you can interrupt any servers you might have left running and rebuild and restart the program:

```
$ cargo run
   Compiling actix-gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in
0.0 secs
     Running `target/debug/actix-gcd`
Serving on http://localhost:3000...
```

This time, by visiting *http://localhost:3000*, entering some numbers, and clicking the Compute GCD button, you should actually see some results (Figure 1-2).



The greatest common divisor of the numbers 24 and 81 is **3**

Figure 1-2. Web page showing results of computing GCD

Concurrency

One of Rust’s great strengths is its support for concurrent programming. The same rules that ensure Rust programs are free of memory errors also ensure threads can share memory only in ways that avoid data races. For example:

- If you use a mutex to coordinate threads making changes to a shared data structure, Rust ensures that you can’t access the data except when you’re holding the lock, and releases the lock automatically when you’re done. In C and C++, the relationship between a mutex and the data it protects is left to the comments.

- If you want to share read-only data among several threads, Rust ensures that you cannot modify the data accidentally. In C and C++, the type system can help with this, but it’s easy to get it wrong.

- If you transfer ownership of a data structure from one thread to another, Rust makes sure you have indeed relinquished all access to it. In C and C++, it’s up to you to check that nothing on the sending thread will ever touch the data again. If you don’t get it right, the effects can depend on what happens to be in the processor’s cache and how many writes to memory you’ve done recently. Not that we’re bitter.

In this section, we’ll walk you through the process of writing your second multi-threaded program.

You’ve already written your first: the Actix web framework you used to implement the Greatest Common Divisor server uses a pool of threads to run request handler functions. If the server receives simultaneous requests, it may run the `get_index` and `post_gcd` functions in several threads at once. That may come as a bit of a shock, since we certainly didn’t have concurrency in mind when we wrote those functions. But Rust guarantees this is safe to do, no matter how elaborate your server gets: if your program compiles, it is free of data races. All Rust functions are thread-safe.

This section's program plots the Mandelbrot set, a fractal produced by iterating a simple function on complex numbers. Plotting the Mandelbrot set is often called an *embarrassingly parallel* algorithm, because the pattern of communication between the threads is so simple; we'll cover more complex patterns in [Link to Come], but this task demonstrates some of the essentials.

To get started, we'll create a fresh Rust project:

```
$ cargo new mandelbrot
    Creating binary (application) `mandelbrot` package
$ cd mandelbrot
```

All the code will go in *mandelbrot/src/main.rs*, and we'll add some dependencies to *mandelbrot/Cargo.toml*.

Before we get into the concurrent Mandelbrot implementation, we need to describe the computation we're going to perform.

What the Mandelbrot Set Actually Is

When reading code, it's helpful to have a concrete idea of what it's trying to do, so let's take a short excursion into some pure mathematics. We'll start with a simple case and then add complicating details until we arrive at the calculation at the heart of the Mandelbrot set.

Here's an infinite loop, written using Rust's dedicated syntax for that, a `loop` statement:

```
fn square_loop(mut x: f64) {
    loop {
        x = x * x;
    }
}
```

In real life, Rust can see that `x` is never used for anything and so might not bother computing its value. But for the time being, assume the code runs as written. What happens to the value of x? Squaring any number smaller than 1 makes it smaller, so it approaches zero; squaring 1 yields 1; squaring a number larger than 1 makes it larger, so it approaches infinity; and squaring a negative number makes it positive, after which it behaves like one of the prior cases (Figure 1-3).

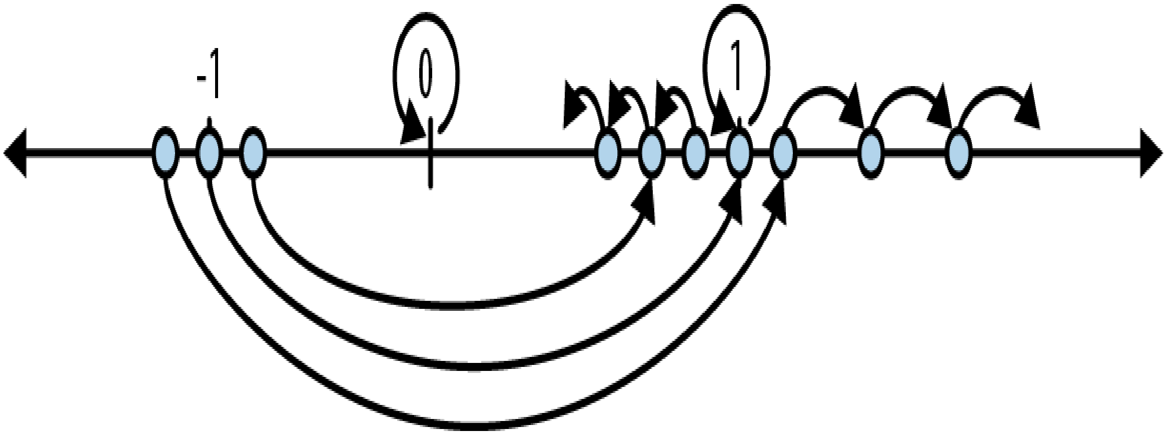


Figure 1-3. Effects of repeatedly squaring a number

So depending on the value you pass to square_loop, x stays at either zero or one, approaches zero, or approaches infinity.

Now consider a slightly different loop:

```
fn square_add_loop(c: f64) {
    let mut x = 0.;
    loop {
        x = x * x + c;
    }
}
```

This time, x starts at zero, and we tweak its progress in each iteration by adding in c after squaring it. This makes it harder to see how x fares, but some experimentation shows that if c is greater than 0.25 or less than –2.0, then x eventually becomes infinitely large; otherwise, it stays somewhere in the neighborhood of zero.

The next wrinkle: instead of using f64 values, consider the same loop using complex numbers. The num crate on crates.io provides a complex number type we can use, so we must add a line for `num` to the `[dependencies]` section in our program's *Cargo.toml* file. While we're here, we can also add the `image` crate, which we'll use later.

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2024"

[dependencies]
num = "0.4"
image = "0.25"
```

Now we can write the penultimate version of our loop:

```
use num::Complex;

fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0 };
    loop {
        z = z * z + c;
    }
}
```

It's traditional to use `z` for complex numbers, so we've renamed our looping variable. The expression `Complex { re: 0.0, im: 0.0 }` is the way we write complex zero using the `num` crate's `Complex` type. `Complex` is a Rust structure type (or *struct*), defined like this:

```
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

The preceding code defines a struct named `Complex`, with two fields, `re` and `im`. `Complex` is a *generic* structure: you can read the `<T>` after the type name as “for any type `T`.” For example, `Complex<f64>` is a complex number whose `re` and `im` fields are `f64` values, `Complex<f32>` would use 32-bit floats, and so on. Given this definition, an expression like `Complex { re: 0.24, im: 0.3 }` produces a `Complex` value with its `re` field initialized to 0.24, and its `im` field initialized to 0.3.

The `num` crate arranges for `*`, `+`, and other arithmetic operators to work on `Complex` values, so the rest of the function works just like the prior version, except that it operates on points on the complex plane, not just points along the real number line. We’ll explain how you can make Rust’s operators work with your own types in [Link to Come].

Finally, we’ve reached the destination of our pure math excursion. The Mandelbrot set is defined as the set of complex numbers `c` for which `z` does not fly out to infinity. Our original simple squaring loop was predictable enough: any number greater than 1 or less than –1 flies away. Throwing a `+ c` into each iteration makes the behavior a little harder to anticipate: as we said earlier, values of `c` greater than 0.25 or less than –2 cause `z` to fly away. But expanding the game to complex numbers produces truly bizarre and beautiful patterns, which are what we want to plot.

Since a complex number `c` has both real and imaginary components `c.re` and `c.im`, we’ll treat these as the `x` and `y` coordinates of a point on the Cartesian plane, and color the point black if `c` is in the Mandelbrot set, or a lighter color otherwise. So for each pixel in our image, we must run the preceding loop on the corresponding point on the complex plane, see whether it escapes to infinity or orbits around the origin forever, and color it accordingly.

The infinite loop takes a while to run, but there are two tricks for the impatient. First, if we give up on running the loop forever and just try some limited number of iterations, it turns out that we still get a decent approximation of the set. How many iterations we need depends on how precisely we want to plot the boundary. Second, it's been shown that, if z ever once leaves the circle of radius 2 centered at the origin, it will definitely fly infinitely far away from the origin eventually. So here's the final version of our loop, and the heart of our program:

```
use num::Complex;

/// Try to determine if `c` is in the Mandelbrot set, using at
most `limit`
/// iterations to decide.
///
/// If `c` is not a member, return `Some(i)`, where `i` is the
number of
/// iterations it took for `c` to leave the circle of radius 2
centered on the
/// origin. If `c` seems to be a member (more precisely, if we
reached the
/// iteration limit without being able to prove that `c` is not a
member),
/// return `None`.
fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
        z = z * z + c;
    }

    None
}
```

This function takes the complex number c that we want to test for membership in the Mandelbrot set and a limit on the number of iterations to try before giving up and declaring c to probably be a member.

The function's return value is an Option<usize>. Rust's standard library defines the Option type as follows:

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option` is an *enumerated type*, often called an *enum*, because its definition enumerates several variants that a value of this type could be: for any type `T`, a value of type `Option<T>` is either `Some(v)`, where `v` is a value of type `T`, or `None`, indicating no `T` value is available. Like the `Complex` type we discussed earlier, `Option` is a generic type: you can use `Option<T>` to represent an optional value of any type `T` you like.

In our case, `escape_time` returns an `Option<usize>` to indicate whether `c` is in the Mandelbrot set—and if it's not, how long we had to iterate to find that out. If `c` is not in the set, `escape_time` returns `Some(i)`, where `i` is the number of the iteration at which `z` left the circle of radius 2. Otherwise, `c` is apparently in the set, and `escape_time` returns `None`.

```
for i in 0..limit {
```

The earlier examples showed `for` loops iterating over command-line arguments and vector elements; this `for` loop simply iterates over the range of integers starting with `0` and up to (but not including) `limit`.

The `z.norm_sqr()` method call returns the square of `z`'s distance from the origin. To decide whether `z` has left the circle of radius 2, instead of computing a square root, we just compare the squared distance with 4.0, which is faster.

You may have noticed that we use `///` to mark the comment lines above the function definition; the comments above the members of the `Complex` structure start with `///` as well. These are *documentation comments*; the `rustdoc` utility knows how to parse them, together with the code they describe, and produce online documentation. The documentation for Rust's standard library is written in this form. We describe documentation comments in detail in [Link to Come].

The rest of the program is concerned with deciding which portion of the set to plot at what resolution and distributing the work across several threads to speed up the calculation.

Parsing Pair Command-Line Arguments

The program takes several command-line arguments controlling the resolution of the image we'll write and the portion of the Mandelbrot set the image shows. Since these command-line arguments all follow a common form, here's a function to parse them:

```
use std::str::FromStr;

/// Parse the string `s` as a coordinate pair, like `"400x600"`
or `"1.0,0.5"`.
///
/// Specifically, `s` should have the form <left><sep><right>,
where <sep> is
/// the character given by the `separator` argument, and <left>
and <right> are
/// both strings that can be parsed by `T::from_str`. `separator`
must be an
/// ASCII character.
///
/// If `s` has the proper form, return `Some<(x, y)>`. If it
doesn't parse
/// correctly, return `None`.
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T,
T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index
+ 1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None,
            }
        }
    }
}

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("",         ','), None);
    assert_eq!(parse_pair::<i32>("10,",      ','), None);
    assert_eq!(parse_pair::<i32>(",10",      ','), None);
    assert_eq!(parse_pair::<i32>("10,20",    ','), Some((10,
20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ','), None);
    assert_eq!(parse_pair::<f64>("0.5x",     'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5,
1.5)));
}
```

The definition of parse_pair is a *generic function*:

```
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T,
T)> {
```

You can read the clause <T: FromStr> aloud as, “For any type T that implements the FromStr trait...” This effectively lets us define an entire family of functions at once: parse_pair::<i32> is a function that parses pairs of i32 values, parse_pair::<f64> parses pairs of floating-point values, and so on. This is very much like a function template in C++. A Rust programmer would call T a *type parameter* of parse_pair. When you use a generic function, Rust will often be able to infer type parameters for you, and you won’t need to write them out as we did in the test code.

Our return type is Option<(T, T)>: either None or a value Some((v1, v2)), where (v1, v2) is a tuple of two values, both of type T. The parse_pair function doesn’t use an explicit return statement, so its return value is the value of the last (and the only) expression in its body:

```
match s.find(separator) {
    None => None,
    Some(index) => {
        ...
    }
}
```

The `String` type's `find` method searches the string for a character that matches `separator`. If `find` returns `None`, meaning that the separator character doesn't occur in the string, the entire `match` expression evaluates to `None`, indicating that the parse failed. Otherwise, we take `index` to be the separator's position in the string.

```
match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
    (Ok(l), Ok(r)) => Some((l, r)),
    _ => None,
}
```

This begins to show off the power of the `match` expression. The argument to the match is this tuple expression:

```
(T::from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

The expressions `&s[..index]` and `&s[index + 1..]` are slices of the string, preceding and following the separator. The type parameter `T`'s associated `from_str` function takes each of these and tries to parse them as a value of type `T`, producing a tuple of results. This is what we match against:

```
(Ok(l), Ok(r)) => Some((l, r)),
```

This pattern matches only if both `Results` are `Ok` variants, indicating that both parses succeeded. If so, `Some((l, r))` is the value of the match expression and hence the return value of the function.

```
_ => None
```

The wildcard pattern _ matches anything and ignores its value. If we reach this point, then `parse_pair` has failed, so we evaluate to `None`, again providing the return value of the function.

Now that we have `parse_pair`, it's easy to write a function to parse a pair of floating-point coordinates and return them as a `Complex<f64>` value:

```
/// Parse a pair of floating-point numbers separated by a comma
as a complex
/// number.
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None,
    }
}

#[test]
fn test_parse_complex() {
    assert_eq!(
        parse_complex("1.25,-0.0625"),
        Some(Complex { re: 1.25, im: -0.0625 }),
    );
    assert_eq!(parse_complex(",-0.0625"), None);
}
```

The `parse_complex` function calls `parse_pair`, builds a `Complex` value if the coordinates were parsed successfully, and passes failures along to its caller.

If you were reading closely, you may have noticed that we used a shorthand notation to build the `Complex` value. It's common to initialize a struct's fields with variables of the same name, so rather than forcing you to write `Complex { re: re, im: im }`, Rust lets you simply write `Complex { re, im }`. This is modeled on similar notations in JavaScript and Haskell.

Mapping from Pixels to Complex Numbers

The program needs to work in two related coordinate spaces: each pixel in the output image corresponds to a point on the complex plane. The relationship between these two spaces depends on which portion of the

Mandelbrot set we’re going to plot, and the resolution of the image requested, as determined by command-line arguments. The following function converts from *image space* to *complex number space*:

```
/// Given the row and column of a pixel in the output image,
return the
/// corresponding point on the complex plane.
///
/// `bounds` is a pair giving the width and height of the image
in pixels.
/// `pixel` is a (column, row) pair indicating a particular pixel
in that image.
/// The `upper_left` and `lower_right` parameters are points on
the complex
/// plane designating the area our image covers.
fn pixel_to_point(
    bounds: (usize, usize),
    pixel: (usize, usize),
    upper_left: Complex<f64>,
    lower_right: Complex<f64>,
) -> Complex<f64> {
    let (width, height) = (
        lower_right.re - upper_left.re,
        upper_left.im - lower_right.im
    );
    Complex {
        re: upper_left.re + pixel.0 as f64 * width  / bounds.0 as
f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as
f64,
        // Why subtraction here? pixel.1 increases as we go down,
        // but the imaginary component increases as we go up.
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(
        pixel_to_point(
            (100, 200),
            (25, 175),
            Complex { re: -1.0, im: 1.0 },
            Complex { re: 1.0, im: -1.0 },
        ),
        Complex { re: -0.5, im: -0.75 },
    );
}
```

Figure 1-4 illustrates the calculation pixel_to_point performs.

The code of pixel_to_point is simply calculation, so we won’t explain it in detail. However, there are a few things to point out. Expressions with this form refer to tuple fields:

```
pixel.0
```

This refers to the first field of the tuple pixel.

```
pixel.0 as f64
```

This is Rust’s syntax for a type conversion: this converts pixel.0 to an f64 value. Unlike C and C++, Rust generally refuses to convert between numeric types implicitly; you must write out the conversions you need. This can be tedious, but being explicit about which conversions occur and when is surprisingly helpful. Implicit integer conversions seem innocent enough, but historically they have been a frequent source of bugs and security holes in real-world C and C++ code.

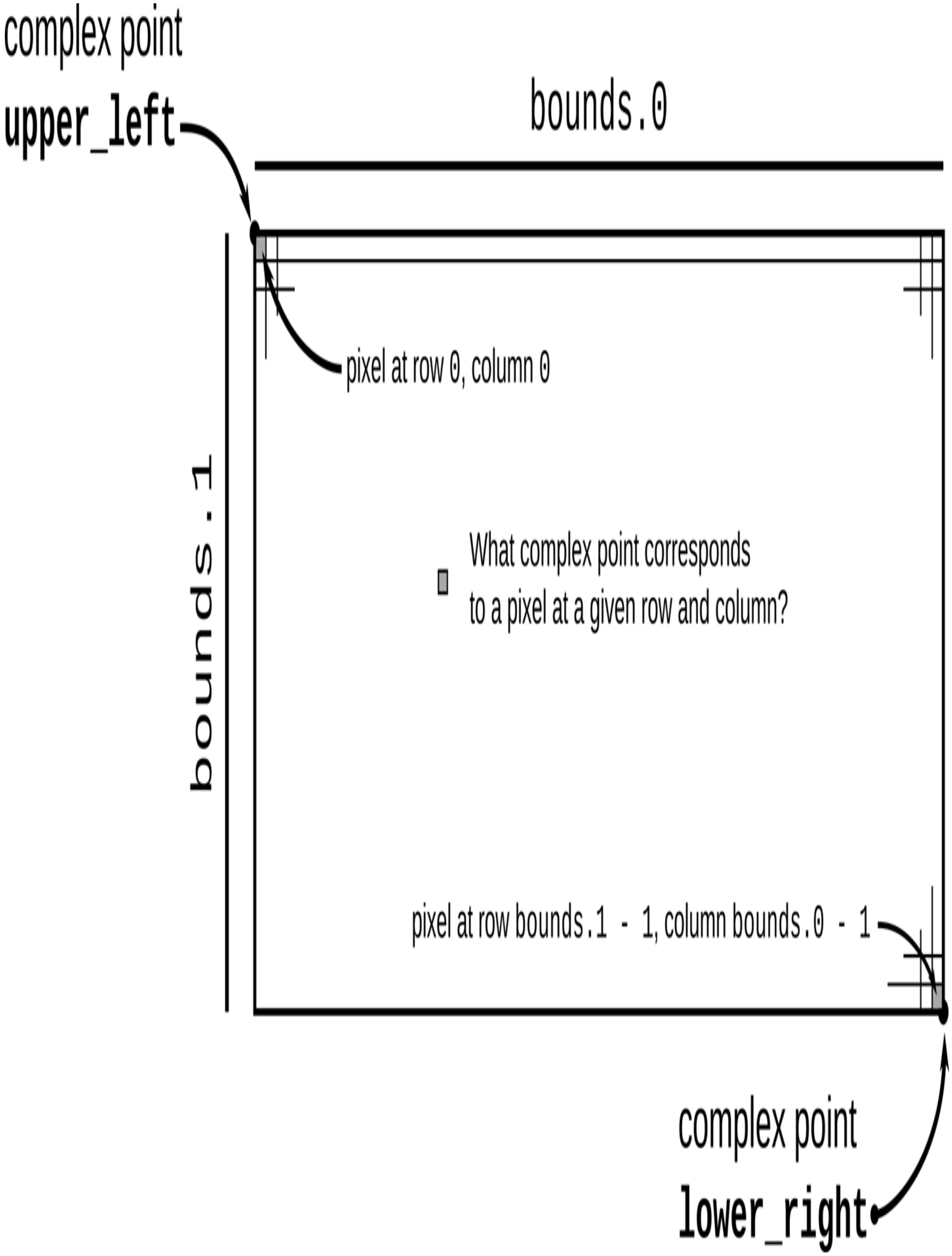

complex point
upper_left
bounds.0
bounds.1
pixel at row 0, column 0
What complex point corresponds
to a pixel at a given row and column?
pixel at row bounds.1 - 1, column bounds.0 - 1
complex point
lower_right

Figure 1-4. The relationship between the complex plane and the image’s pixels

Plotting the Set

To plot the Mandelbrot set, for every pixel in the image, we simply apply `escape_time` to the corresponding point on the complex plane, and color the pixel depending on the result:

```
/// Render a rectangle of the Mandelbrot set into a buffer of
pixels.
///
/// The `bounds` argument gives the width and height of the
buffer `pixels`,
/// which holds one grayscale pixel per byte. The `upper_left`
and `lower_right`
/// arguments specify points on the complex plane corresponding
to the upper-
/// left and lower-right corners of the pixel buffer.
fn render(
    pixels: &mut [u8],
    bounds: (usize, usize),
    upper_left: Complex<f64>,
    lower_right: Complex<f64>,
) {
    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0..bounds.1 {
        for column in 0..bounds.0 {
            let point =
                pixel_to_point(bounds, (column, row), upper_left,
lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8,
                };
        }
    }
}
```

The type of the first argument, `pixels`, is `&mut [u8]`. In English, that’s a mutable reference, `&mut`, to a slice of unsigned 8-bit integers, `[u8]`. Think of a *slice* as an array or vector; `pixels` will refer to a slab of contiguous memory, typically millions of `u8` bytes. We need a *mutable reference* in order to write to the buffer, since Rust references are read-only by default.

```
pixels[row * bounds.0 + column] =
    match escape_time(point, 255) {
        None => 0,
        Some(count) => 255 - count as u8,
    };
```

If `escape_time` says that `point` belongs to the set, `render` colors the corresponding pixel black (`0`). Otherwise, `render` assigns darker colors to the numbers that took longer to escape the circle.

Writing Image Files

The `image` crate provides functions for reading and writing a wide variety of image formats, along with some basic image manipulation functions. In particular, it includes an encoder for the PNG image file format, which this program uses to save the final results of the calculation:

```
use image::{ExtendedColorType, ImageEncoder, ImageError};
use image::codecs::png::PngEncoder;
use std::fs::File;

/// Write the buffer `pixels`, whose dimensions are given by
`bounds`, to the
/// file named `filename`.
fn write_image(
    filename: &str,
    pixels: &[u8],
    bounds: (usize, usize),
) -> Result<(), ImageError> {
    let output = File::create(filename)?;

    let encoder = PngEncoder::new(output);
    encoder.write_image(
        pixels,
        bounds.0 as u32,
        bounds.1 as u32,
        ExtendedColorType::L8,
    )?;

    Ok(())
}
```

The operation of this function is pretty straightforward: it opens a file and tries to write the image to it. The `write_image` method is provided by the `image::ImageEncoder` trait, which we import on the first line of the example. We pass the encoder the actual pixel data from `pixels`, and its width and height from `bounds`, and then a final argument that says how to interpret the bytes in `pixels`: the value `ExtendedColorType::L8` indicates that each byte is an eight-bit grayscale value.

That's all simple enough. What's interesting about this function is how it copes when something goes wrong. If we encounter an error, we need to report that back to our caller. As we've mentioned before, fallible functions in Rust should return a `Result` value, which is either `Ok(s)` on success, where `s` is the successful value, or `Err(e)` on failure, where `e` is an error code. So what are `write_image`'s success and error types?

When all goes well, our `write_image` function has no useful value to return; it wrote everything interesting to the file. So its success type is the *unit* type `()`, so called because it has only one value, also written `()`. The unit type is akin to `void` in C and C++.

When an error occurs, it's because either `File::create` wasn't able to create the file or `encoder.encode` wasn't able to write the image to it; the I/O operation returned an error code. The return type of `File::create` is `Result<std::fs::File, std::io::Error>`, while that of `encoder.encode` is `Result<(), std::io::Error>`, so both share the same error type, `std::io::Error`. It makes sense for our `write_image` function to do the same. In either case, failure should result in an immediate return, passing along the `std::io::Error` value describing what went wrong.

So to properly handle `File::create`'s result, we need to `match` on its return value, like this:

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(e) => {
        return Err(e);
    }
};
```

On success, let `output` be the `File` carried in the `Ok` value. On failure, pass along the error to our own caller.

This kind of `match` statement is such a common pattern in Rust that the language provides the `?` operator as shorthand for the whole thing. So, rather than writing out this logic explicitly every time we attempt something that could fail, you can use the following equivalent and much more legible statement:

```
let output = File::create(filename)?;
```

If `File::create` fails, the `?` operator returns from `write_image`, passing along the error. Otherwise, `output` holds the successfully opened `File`.

> **NOTE**
>
> It’s a common beginner’s mistake to attempt to use `?` in the `main` function. However, since `main` itself doesn’t return a value, this won’t work; instead, you need to use a `match` statement, or one of the shorthand methods like `unwrap` and `expect`. There’s also the option of simply changing `main` to return a `Result`, which we’ll cover later.

A Concurrent Mandelbrot Program

All the pieces are in place, and we can show you the `main` function, where we can put concurrency to work for us. First, a nonconcurrent version for simplicity:

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 5 {
        let program = &args[0];
        eprintln!("Usage: {program} FILE PIXELS LEFT,TOP
RIGHT,BOTTOM");
        eprintln!("Example: {program} mandel.png 1000x750
-1.20,0.35 -1,0.20");
        std::process::exit(1);
    }

    let bounds: (usize, usize) = parse_pair(&args[2], 'x')
        .expect("error parsing image dimensions");
    let upper_left = parse_complex(&args[3])
        .expect("error parsing upper left corner point");
    let lower_right = parse_complex(&args[4])
        .expect("error parsing lower right corner point");

    let mut pixels = vec![0; bounds.0 * bounds.1];

    render(&mut pixels, bounds, upper_left, lower_right);

    write_image(&args[1], &pixels, bounds)
        .expect("error writing PNG file");
}
```

After collecting the command-line arguments into a vector of `String`s, we parse each one and then begin calculations.

```
let mut pixels = vec![0; bounds.0 * bounds.1];
```

A macro call `vec![v; n]` creates a vector `n` elements long whose elements are initialized to `v`, so the preceding code creates a vector of zeros whose length is `bounds.0 * bounds.1`, where `bounds` is the image resolution parsed from the command line. We'll use this vector as a rectangular array of one-byte grayscale pixel values, as shown in Figure 1-5.

The next line of interest is this:

```
render(&mut pixels, bounds, upper_left, lower_right);
```

This calls the `render` function to actually compute the image. The expression `&mut pixels` borrows a mutable reference to our pixel buffer, allowing `render` to fill it with computed grayscale values, even while `pixels` remains the vector’s owner. The remaining arguments pass the image’s dimensions and the rectangle of the complex plane we’ve chosen to plot.

```
write_image(&args[1], &pixels, bounds)
    .expect("error writing PNG file");
```

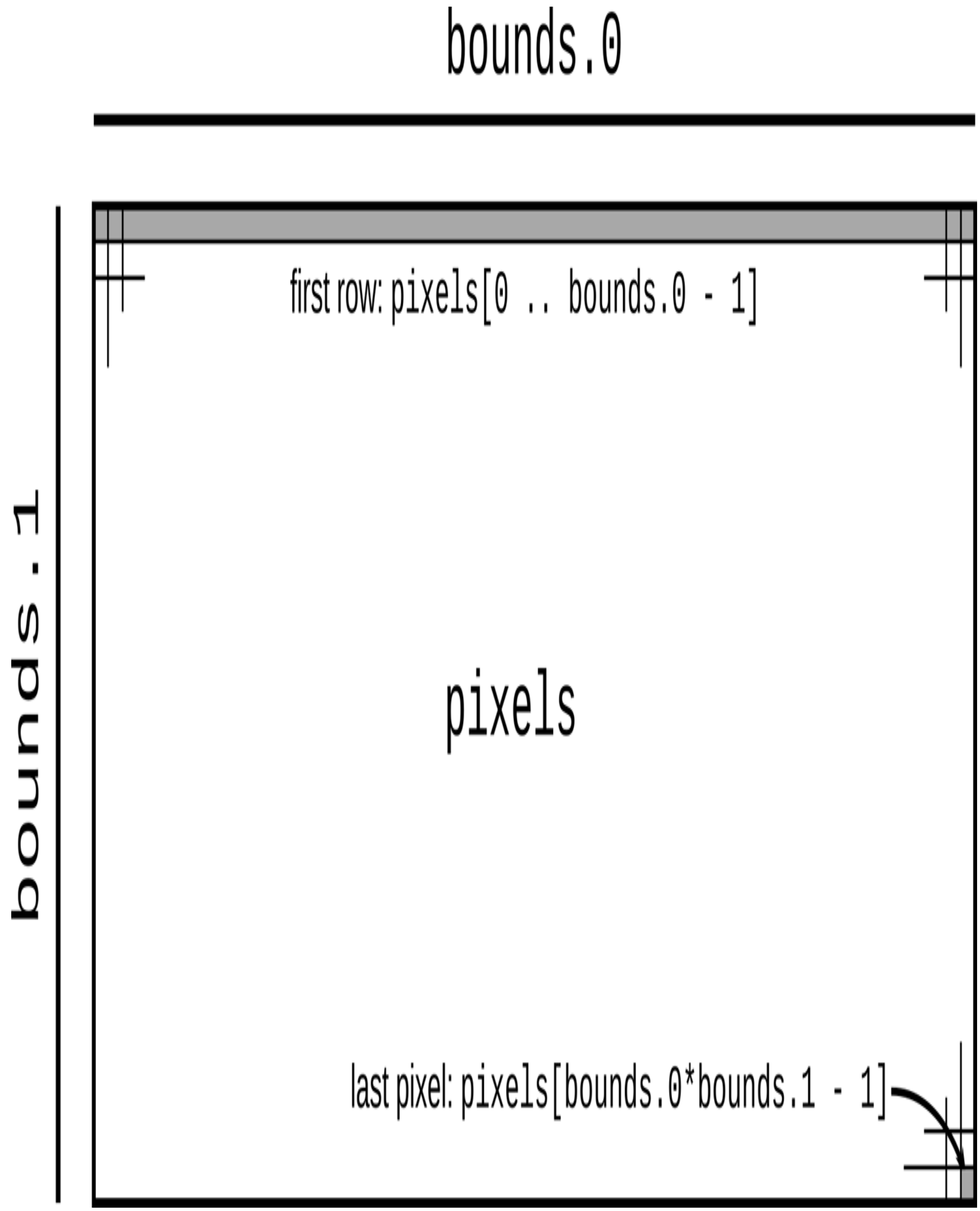

bounds.0
first row: pixels[0 .. bounds.0 - 1]
bounds.1
pixels
last pixel: pixels[bounds.0*bounds.1 - 1]

Figure 1-5. Using a vector as a rectangular array of pixels

Finally, we write the pixel buffer out to disk as a PNG file. In this case, we pass a shared (nonmutable) reference to the buffer, since `write_image` should have no need to modify the buffer's contents.

At this point, we can build and run the program in release mode, which enables many powerful compiler optimizations, and after several seconds, it will write a beautiful image to the file *mandel.png*:

```
$ cargo build --release
    Updating crates.io index
   ...
   Compiling image v0.25.5
   Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished `release` profile [optimized] target(s) in 25.36s
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35
-1,0.20
real    0m4.678s
user    0m4.661s
sys     0m0.008s
```

On Windows, instead run this command in PowerShell: `measure-command { .\target\release\mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20 }`.

This command should create a file called *mandel.png*, which you can view with your system's image viewing program or in a web browser. If all has gone well, it should look like Figure 1-6.

Figure 1-6. Results from the Mandelbrot program

In the previous transcript, we used the Unix `time` program to analyze the running time of the program: it took about five seconds total to run the Mandelbrot computation on each pixel of the image. But almost all modern machines have multiple processor cores, and this program used only one. If we could distribute the work across all the computing resources the machine has to offer, we should be able to complete the image much more quickly.

To this end, we'll divide the image into sections, one per processor, and let each processor color the pixels assigned to it. For simplicity, we'll break it into horizontal bands, as shown in Figure 1-7. When all processors have finished, we can write out the pixels to disk.

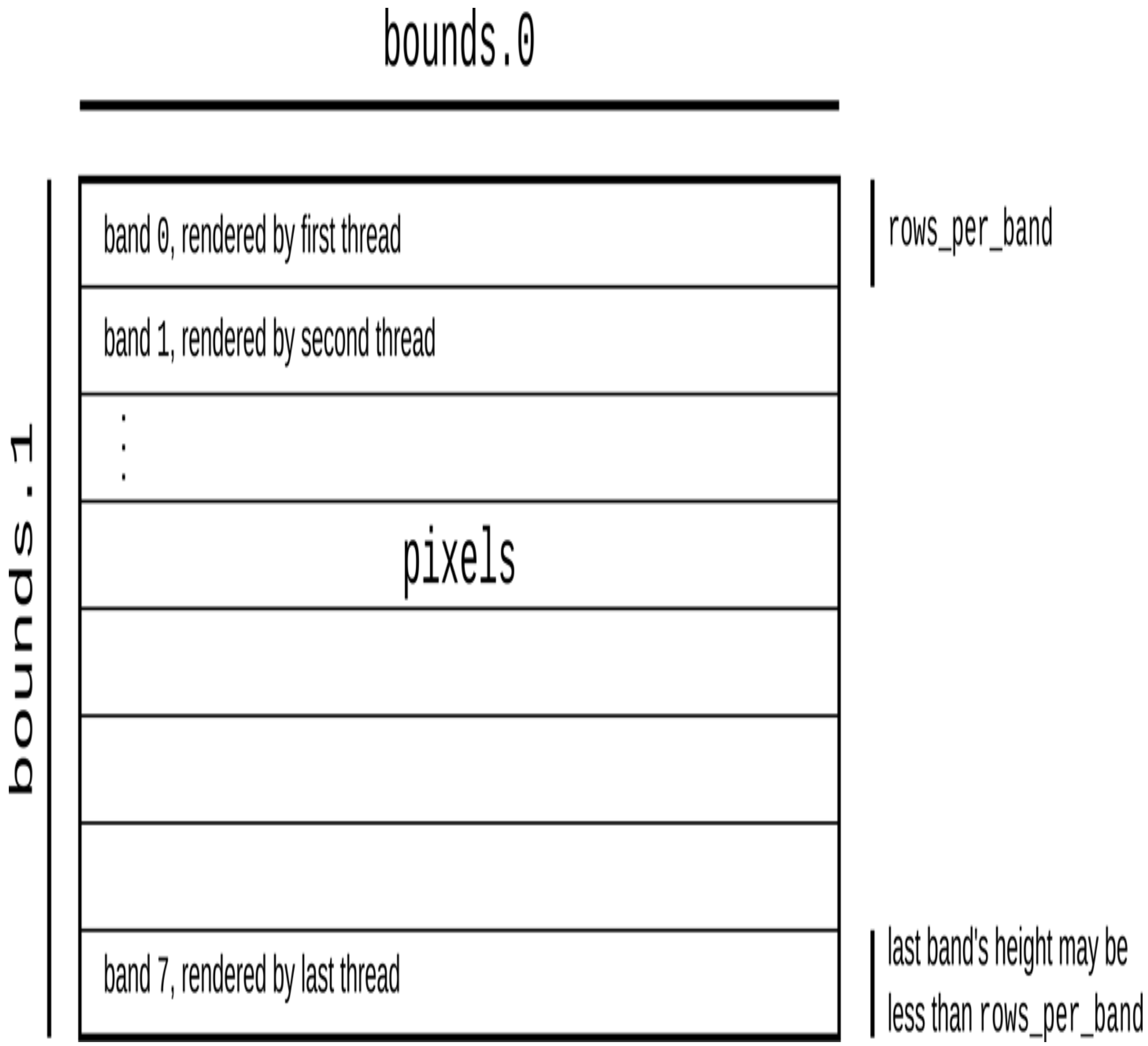

bounds.0
band 0, rendered by first thread
band 1, rendered by second thread
⋮
pixels
band 7, rendered by last thread
bounds.1
rows_per_band
last band's height may be
less than rows_per_band

Figure 1-7. Dividing the pixel buffer into bands for parallel rendering

Rust offers a *scoped thread* facility that does exactly what we need here. To use it, we need to take out the single line calling `render` and replace it with the following:

```
let threads = std::thread::available_parallelism()
    .expect("error querying CPU count")
    .get();
let rows_per_band = bounds.1.div_ceil(threads);

let bands = pixels.chunks_mut(rows_per_band * bounds.0);
std::thread::scope(|spawner| {
    for (i, band) in bands.enumerate() {
        let top = rows_per_band * i;
        let height = band.len() / bounds.0;
        let band_bounds = (bounds.0, height);
        let band_upper_left =
            pixel_to_point(bounds, (0, top), upper_left,
lower_right);
        let band_lower_right =
            pixel_to_point(bounds, (bounds.0, top + height),
                           upper_left, lower_right);

        spawner.spawn(move || {
            render(band, band_bounds, band_upper_left,
band_lower_right);
        });
    }
});
```

Breaking this down in the usual way:

```
let threads = std::thread::available_parallelism()
    .expect("error querying CPU count")
    .get();
```

We start by asking the system how many threads we should create, using `std::thread::available_parallelism()`. This function returns a `Result<NonZero<usize>>`: either an error, or an `Ok` value containing a `usize` that is guaranteed to be nonzero. We use `.expect()` to dispense with the error case and the `get` method of the `NonZero` type to convert it to a plain `usize`.

```
let rows_per_band = bounds.1.div_ceil(threads);
```

Next we compute how many rows of pixels each band should have. We divide the number of rows by the number of threads, rounding upward so that the bands will cover the entire image even if the height isn't a multiple of `threads`. Rust's numeric types provide dozens of methods, including `div_ceil` for integer division rounding up. In another language, we might write `(bounds.1 + threads - 1) / threads`, but calling `div_ceil` is shorter, more expressive, and just as fast. (The method is also correct in corner cases where the addition would overflow, although that's unlikely here!)

```
let bands = pixels.chunks_mut(rows_per_band * bounds.0);
```

Here we divide the pixel buffer into bands. The buffer's `chunks_mut` method returns an iterator producing mutable, nonoverlapping slices of the buffer, each of which encloses `rows_per_band * bounds.0` pixels—in other words, `rows_per_band` complete rows of pixels. The last slice that `chunks_mut` produces may contain fewer rows, but each row will contain the same number of pixels.

Now we can put some threads to work:

```
std::thread::scope(|spawner| {
    ...
});
```

The argument `|spawner| { ... }` is a Rust closure that expects a single argument, `spawner`. Note that, unlike functions declared with `fn`, we don't need to declare the types of a closure's arguments; Rust will infer them, along with its return type. In this case, `std::thread::scope` calls the closure, passing as the `spawner` argument a value the closure can use to create new threads. The `std::thread::scope` function waits for all such threads to finish execution before returning itself. This behavior allows Rust to be sure that such threads will not access their portions of `pixels` after it has gone out of scope, and allows us to be sure that when `std::thread::scope` returns, the computation of the image is complete.

```
for (i, band) in bands.enumerate() {
```

Here we iterate over the pixel buffer's bands. Each band is a mutable slice of the image, providing exclusive access to one band, ensuring that only one thread can write to it at a time. We explain how this works in detail in Chapter 4. The `enumerate` adapter produces tuples pairing each vector element with its index.

```
let top = rows_per_band * i;
let height = band.len() / bounds.0;
let band_bounds = (bounds.0, height);
let band_upper_left =
    pixel_to_point(bounds, (0, top), upper_left, lower_right);
let band_lower_right =
    pixel_to_point(bounds, (bounds.0, top + height),
                   upper_left, lower_right);
```

Given the index and the actual size of the band (recall that the last one might be shorter than the others), we can produce a bounding box of the sort `render` requires, but one that refers only to this band of the buffer, not the entire image. Similarly, we repurpose the renderer's `pixel_to_point` function to find where the band's upper-left and lower-right corners fall on the complex plane.

```
spawner.spawn(move || {
    render(band, band_bounds, band_upper_left, band_lower_right);
});
```

Finally, we create a thread, running the closure `move || { ... }`. The `move` keyword at the front indicates that this closure takes ownership of the variables it uses; in particular, only the closure may use the mutable slice `band`.

As we mentioned earlier, the `std::thread::scope` call ensures that all threads have completed before it returns, meaning that it is safe to save the image to a file, which is our next action.

With all that in place, we can build and run the multithreaded program:

```
$ cargo build --release
   Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished `release` profile [optimized] target(s) in #.## secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35
-1,0.20
real    0m1.436s
user    0m4.922s
sys     0m0.011s
```

Here, we’ve used `time` again to see how long the program took to run; note that even though we still spent almost five seconds of processor time, the elapsed real time was only about 1.5 seconds. You can verify that a portion of that time is spent writing the image file by commenting out the code that does so and measuring again. On the laptop where this code was tested, the concurrent version reduces the Mandelbrot calculation time proper by a factor of almost four. We’ll show how to substantially improve on this in [Link to Come].

As before, this program will have created a file called *mandel.png*. With this faster version, you can more easily explore the Mandelbrot set by changing the command-line arguments to your liking.

Safety Is Invisible

In the end, the parallel program we ended up with is not substantially different from what we might write in any other language: we apportion pieces of the pixel buffer out among the processors, let each one work on its piece separately, and when they’ve all finished, present the result. So what is so special about Rust’s concurrency support?

What we haven’t shown here is all the Rust programs we *cannot* write. The code we looked at in this chapter partitions the buffer among the threads correctly, but there are many small variations on that code that do not (and thus introduce data races); not one of those variations will pass the Rust compiler’s static checks. A C or C++ compiler will cheerfully help you explore the vast space of programs with subtle data races; Rust tells you, up front, when something could go wrong.

In Chapters 3 and 4, we'll describe Rust's rules for memory safety. [Link to Come] explains how these rules also ensure proper concurrency hygiene. But for those to make sense, it's essential to get a grounding in Rust's fundamental types, which we'll cover in the next chapter.

Chapter 2. Fundamental Types

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

> *The first principles of the universe are atoms and empty space; everything else is merely thought to exist.*
>
> —Diogenes Laertes (describing the philosophy of Democritus)

This chapter covers Rust’s fundamental types for representing values: integer and floating-point numbers, strings and characters, vectors and arrays, and so on. These source-level types have concrete machine-level counterparts with predictable costs and performance.

Compared to a dynamically typed language like JavaScript or Python, Rust requires more planning from you up front. You must spell out the types of function arguments and return values, struct fields, and a few other constructs. For the rest, Rust’s *type inference* will generally figure out the types for you. For example, you could spell out every type in a function, like this:

```
fn grade_cutoffs(tier_size: u32) -> Vec<u32> {
    let mut cutoffs: Vec<u32> = Vec::<u32>::new();
    for tier in 1u32..5u32 {
        cutoffs.push(100u32 - tier * tier_size);
    }
    cutoffs
}
```

But this is cluttered and repetitive. Given the function’s return type, it’s obvious that `cutoffs` must be a `Vec<u32>`, a vector of 32-bit signed integers; no other type would work. And from that it follows that each element of the vector must be a `u32`, and therefore all the numbers being multiplied and subtracted must be `u32` as well. This is exactly the sort of reasoning Rust’s type inference applies, allowing you to instead write:

```
fn grade_cutoffs(tier_size: u32) -> Vec<u32> {
    let mut cutoffs = Vec::new();
    for tier in 1..5 {
        cutoffs.push(100 - tier * tier_size);
    }
    cutoffs
}
```

These two definitions are exactly equivalent, and Rust will generate the same machine code either way. Type inference gives back much of the legibility of dynamically typed languages, while still catching type errors at compile time.

The rest of this chapter covers Rust’s types from the bottom up. Table 2-1 offers a sample of the sorts of types you’ll see in Rust. It shows Rust’s primitive types, some very common types from the standard library, and some examples of user-defined types.

Table 2-1. Examples of types in Rust

Type	Description	Values

`i8`, `i16`, `i32`, `i64`, `i128`
`u8`, `u16`, `u32`, `u64`, `u128`
	Signed and unsigned integers, of given bit width	`42`, `-5i8`, `0x400u16`, `0o100i16`, `20_922_789_888_000u64`, `b'*'` (`u8` byte literal)

`isize`, `usize`	Signed and unsigned integers, the same size as an address on the machine (32 or 64 bits)	`137`, `-0b0101_0010_isize`, `0xffff_fc00_usize`
`f32`, `f64`	IEEE floating-point numbers, single and double precision	`1.0`, `3.14159f32`, `6.0221e23`
`bool`	Boolean	`true`, `false`
`(char, u8, i32)`	Tuple: mixed types allowed	`('%', 0x7f, -1)`
`()`	“Unit” (empty tuple)	`()`

`struct S {`
`  x: f32,`
`  y: f32,`
`}`
	Named-field struct	`S { x: 120.0, y: 209.0 }`

`struct T(i32, char);`	Tuple-like struct	`T(120, 'X')`
`struct E;`	Unit-like struct; has no fields	`E`

Type	Description	Values

`enum Attend {`
`  OnTime,`
`  Late(u32)`
`}`
	Enumeration, algebraic data type	`Attend::Late(5)`,
		`Attend::OnTime`

`Box<Attend>`	Box: owning pointer to value in heap	`Box::new(Late(15))`
`&i32`, `&mut i32`	Shared and mutable references: non-owning pointers that must not outlive their referent	`&s.y`, `&mut v`
`String`	UTF-8 string, dynamically sized	`"ラーメン: ramen".to_string()`
`&str`	Reference to `str`: non-owning pointer to UTF-8 text	`"そば: soba"`, `&s[0..12]`
`char`	Unicode character, 32 bits wide	`'*'`, `'\n'`, `'字'`, `'\x7f'`, `'\u{CA0}'`
`[f64; 4]`, `[u8; 256]`	Array, fixed length; elements all of same type	`[1.0, 0.0, 0.0, 1.0]`, `[b' '; 256]`
`Vec<f64>`	Vector, varying length; elements all of same type	`vec![0.367, 2.718, 7.389]`
`&[u8]`, `&mut [u8]`	Reference to slice: reference to a portion of an array or vector, comprising pointer and length	`&v[10..20]`, `&mut a[..]`

Type	Description	Values
`Option<&str>`	Optional value: either `None` (absent) or `Some(v)` (present, with value `v`)	`Some("Dr.")`, `None`
`Result<u64, Error>`	Result of operation that may fail: either a success value `Ok(v)`, or an error `Err(e)`	`Ok(4096)`, `Err(Error::last_os_error())`
`&dyn Error`, `&mut dyn Read`	Trait object reference: refers to any value that implements a given set of methods	`&err as &dyn Error`, `&mut file as &mut dyn Read`
`fn(&str) -> bool`	Function pointer	`str::is_empty`
(Closure types have no written form)	Closure	`|a, b| { a*a + b*b }`

Most of these types are covered in this chapter, except for the following:

- We give `struct` types their own chapter, [Link to Come].
- We give enumerated types their own chapter, [Link to Come].
- We describe trait objects in [Link to Come].
- We describe the essentials of `String` and `&str` here, but provide more detail in [Link to Come].
- We cover function and closure types in [Link to Come].

Numeric Types

The footing of Rust’s type system is a collection of fixed-width numeric types, chosen to match the types that almost all modern processors implement directly in hardware.

Fixed-width numeric types can overflow or lose precision, but they are adequate for most applications and can be thousands of times faster than representations like arbitrary-precision integers and exact rationals. If you need those sorts of numeric representations, they are available in the `num` crate.

The names of Rust’s numeric types follow a regular pattern, spelling out their width in bits, and the representation they use (Table 2-2).

Table 2-2. Rust numeric types

Size (bits)	Unsigned integer	Signed integer	Floating-point
8	`u8`	`i8`	
16	`u16`	`i16`	
32	`u32`	`i32`	`f32`
64	`u64`	`i64`	`f64`
128	`u128`	`i128`	
Machine word	`usize`	`isize`	

Here, a *machine word* is a value the size of an address on the machine the code runs on, 32 or 64 bits.

Integer Types

Rust’s unsigned integer types use their full range to represent positive values and zero (Table 2-3).

Table 2-3. Rust unsigned integer types

Type	Range
`u8`	0 to 2^{8}–1 (0 to 255)
`u16`	0 to 2^{16}–1 (0 to 65,535)
`u32`	0 to 2^{32}–1 (0 to 4,294,967,295)
`u64`	0 to 2^{64}–1 (0 to 18,446,744,073,709,551,615, or 18 quintillion)
`u128`	0 to 2^{128}–1 (0 to around 3.4×10^{38})
`usize`	0 to either 2^{32}–1 or 2^{64}–1

Rust’s signed integer types use the two’s complement representation, using the same bit patterns as the corresponding unsigned type to cover a range of positive and negative values (Table 2-4).

Table 2-4. Rust signed integer types

Type	Range
`i8`	-2^{7} to $2^{7}-1$ (−128 to 127)
`i16`	-2^{15} to $2^{15}-1$ (−32,768 to 32,767)
`i32`	-2^{31} to $2^{31}-1$ (−2,147,483,648 to 2,147,483,647)
`i64`	-2^{63} to $2^{63}-1$ (−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
`i128`	-2^{127} to $2^{127}-1$ (roughly -1.7×10^{38} to $+1.7\times10^{38}$)
`isize`	Either -2^{31} to $2^{31}-1$, or -2^{63} to $2^{63}-1$

Rust uses the `u8` type for byte values. For example, reading data from a binary file or socket yields a stream of `u8` values.

C/C++ programmers should note that there is no single “character” type used for both text and binary data in Rust. As we’ll see when we discuss strings and characters later in this chapter, Rust’s `char` type is a Unicode character type, not an integer type, and is quite different from C’s `char`.

The `usize` and `isize` types are analogous to `size_t` and `ptrdiff_t` in C and C++. Their precision matches the size of the address space on the target machine: they are 32 bits long on 32-bit architectures, and 64 bits long on 64-bit architectures. Rust requires array indices to be `usize` values. Values representing the sizes of arrays or vectors or counts of the number of elements in some data structure also generally have the `usize` type.

Integer literals in Rust can take a suffix indicating their type: `42u8` is a `u8` value, and `1729isize` is an `isize`. If an integer literal lacks a type suffix, Rust puts off determining its type until it finds the value being used in a way that pins it down: stored in a variable of a particular type, passed to a function that expects a particular type, compared with another value of a particular type, or something like that. If the code provides no such clues, Rust defaults to i32.

The prefixes 0x, 0o, and 0b designate hexadecimal, octal, and binary literals.

To make long numbers more legible, you can insert underscores among the digits. For example, you can write the largest u32 value as 4_294_967_295. The exact placement of the underscores is not significant, so you can break hexadecimal or binary numbers into groups of four digits rather than three, as in 0xffff_ffff, or set off the type suffix from the digits, as in 127_u8.

You can convert from one integer type to another using the as operator:

```
assert_eq!(10_i8 as u16, 10_u16);
assert_eq!(0xcafedad_u64 as u8, 0xad_u8);  // value too big for
u8, truncated
```

The standard library provides some operations as methods on integers. For example:

```
assert_eq!(2_u16.pow(4), 16);               // exponentiation
assert_eq!((-4_i32).abs(), 4);              // absolute value
assert_eq!(0b101101_u8.count_ones(), 4);    // population count
assert_eq!(12_i32.max(37), 37);             // maximum of two
values
```

There are dozens of these. You can find them in the online documentation under “Primitive Type i32” and friends.

In real code, you usually won’t need to write out the type suffixes as we’ve done here, because the context will determine the type. When it doesn’t, however, the error messages can be surprising. For example, the following doesn't compile:

```
println!("{}", (-4).abs());
```

Rust complains:

```
error: can't call method `abs` on ambiguous numeric type
`{integer}`
```

This can be a little bewildering: all the signed integer types have an `abs` method, so what's the problem? For technical reasons, Rust wants to know exactly which integer type a value has before it will call the type's methods. The default of `i32` applies only if the type is still ambiguous after all method calls have been resolved, so that's too late to help here. The solution is to spell out which type you intend, either with a suffix or by using a specific type's function:

```
println!("{}", (-4_i32).abs());
println!("{}", i32::abs(-4));
```

Note that method calls have a higher precedence than unary prefix operators, so be careful when applying methods to negated values. Without the parentheses around `-4_i32` in the first statement, `-4_i32.abs()` would apply the `abs` method to the positive value `4`, producing positive `4`, and then negate that, producing `-4`.

Handling Integer Overflow

When an integer arithmetic operation overflows, Rust panics, in a debug build. In a release build, the operation *wraps around*: it produces the value equivalent to the mathematically correct result modulo the range of the value. (In neither case is overflow undefined behavior, as it is for signed integers in C and C++.)

For example, the following code panics in a debug build:

```
let mut i = 1;
loop {
    i *= 10;  // panic: attempt to multiply with overflow
              // (but only in debug builds!)
}
```

In a release build, this multiplication wraps to a negative number, and the loop runs indefinitely.

When this default behavior isn't what you need, the integer types provide methods that let you spell out exactly what you want. For example, the following panics in any build:

```
let mut i: i32 = 1;
loop {
    // panic: multiplication overflowed (in any build)
    i = i.checked_mul(10).expect("multiplication overflowed");
}
```

These integer arithmetic methods fall in four general categories:

- *Checked* operations return an `Option` of the result: `Some(v)` if the mathematically correct result can be represented as a value of that type, or `None` if it cannot. For example:

  ```
  // The sum of 10 and 20 can be represented as a u8.
  assert_eq!(10_u8.checked_add(20), Some(30));

  // Unfortunately, the sum of 100 and 200 cannot.
  assert_eq!(100_u8.checked_add(200), None);
  ```

```
// Do the addition; panic if it overflows.
let sum = x.checked_add(y).unwrap();

// Oddly, signed division can overflow too, in one
particular case.
// A signed n-bit type can represent -2ⁿ⁻¹, but not 2ⁿ⁻¹.
assert_eq!((-128_i8).checked_div(-1), None);
```

- *Wrapping* operations return the value equivalent to the mathematically correct result modulo the range of the value:

```
// The first product can be represented as a u16;
// the second cannot, so we get 250000 modulo 2¹⁶.
assert_eq!(100_u16.wrapping_mul(200), 20000);
assert_eq!(500_u16.wrapping_mul(500), 53392);

// Operations on signed types may wrap to negative
values.
assert_eq!(500_i16.wrapping_mul(500), -12144);
```

  As explained, this is how the ordinary arithmetic operators behave in release builds. The advantage of these methods is that they behave the same way in all builds.

- *Saturating* operations return the representable value that is closest to the mathematically correct result. In other words, the result is “clamped” to the maximum and minimum values the type can represent:

```
assert_eq!(32760_i16.saturating_add(10), 32767);
assert_eq!((-32760_i16).saturating_sub(10), -32768);
```

- *Overflowing* operations return a tuple `(result, overflowed)`, where `result` is what the wrapping version of the function would return, and `overflowed` is a `bool` indicating whether an overflow occurred:

  ```
  assert_eq!(255_u8.overflowing_sub(2), (253, false));
  assert_eq!(255_u8.overflowing_add(2), (1, true));
  ```

The bit-shifting operations `shl` (shift left, <<) and `shr` (shift right, >>) deviate from the pattern. For these operations it is the shift distance, not the result, that can be out of range (again causing undefined behavior in C++). So the checking, wrapping, or overflowing for these methods applies to the shift distance argument.

```
assert_eq!(10_u64.wrapping_shl(65), 20);
assert_eq!(10_u64.checked_shr(65), None);
```

Floating-Point Types

Rust provides the usual single- and double-precision floating-point types (Table 2-5). Rust's `f32` and `f64` correspond to the `float` and `double` types in C/C++ and Java, or `float32` and `float64` in Go. Both types follow the IEEE 754 floating point standard. They include positive and negative infinities, distinct positive and negative zero values, and a *not-a-number* value.

Table 2-5. Floating-point types

Type	Precision	Range
`f32`	IEEE single precision (at least 6 decimal digits)	Roughly ±3.4 × 10^{38}
`f64`	IEEE double precision (at least 15 decimal digits)	Roughly ±1.8 × 10^{308}

Floating-point literals have the general form diagrammed in Figure 2-1.

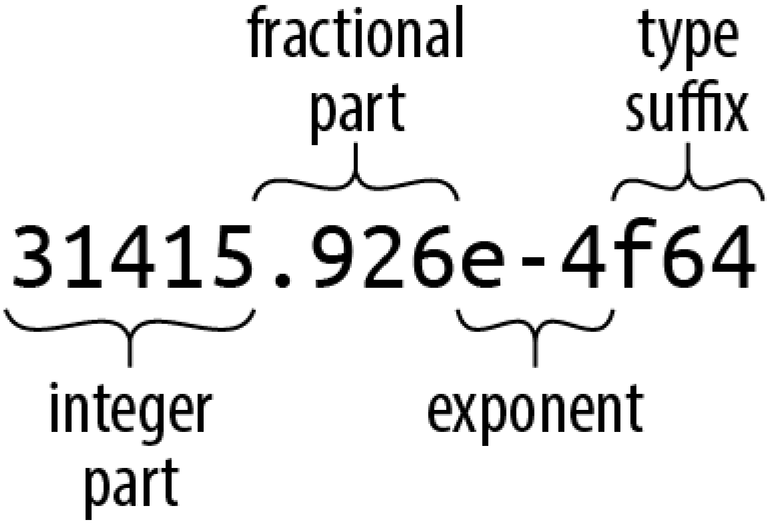


Figure 2-1. A floating-point literal

Every part of a floating-point number after the integer part is optional, but at least one of the fractional part, exponent, or type suffix must be present, to distinguish it from an integer literal. Rust never assigns a floating-point type to a integer literal, or vice versa.

The fractional part may consist of a lone decimal point, so `5.` is a valid floating-point constant.

If a floating-point literal lacks a type suffix, Rust checks the context to see how the value is used, much as it does for integer literals. The default is `f64`.

The types `f32` and `f64` have associated constants for the IEEE-required special values like `INFINITY`, `NEG_INFINITY` (negative infinity), `NAN` (the not-a-number value), and `MIN` and `MAX` (the largest and smallest finite values):

```
assert!((-1. / f32::INFINITY).is_sign_negative());
assert_eq!(-f32::MIN, f32::MAX);
```

The `f32` and `f64` types provide a full complement of methods for mathematical calculations; for example, `2f64.sqrt()` is the double-precision square root of two. Some examples:

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.);  // exactly 5.0, per
IEEE
assert_eq!((-1.01f64).floor(), -2.0);
```

These methods are documented under “Primitive Type `f32`” and “Primitive Type `f64`”. The separate modules `std::f32::consts` and `std::f64::consts` provide various commonly used mathematical constants like `E`, `PI`, and the square root of two.

As with integers, you usually won’t need to write out type suffixes on floating-point literals in real code, but when you do, putting a type on either the literal or the function will suffice:

```
println!("{}", (2.0_f64).sqrt());
println!("{}", f64::sqrt(2.0));
```

Unlike C and C++, Rust performs almost no numeric conversions implicitly. If a function expects an `f64` argument, it’s an error to pass an `i32` value as the argument. In fact, Rust won’t even implicitly convert an `i16` value to an `i32` value, even though every `i16` value is also an `i32` value. But you can always write out *explicit* conversions using the `as` operator: `i as f64`, or `x as i32`.

The lack of implicit conversions sometimes makes a Rust expression more verbose than the analogous C or C++ code would be. However, implicit integer conversions have a well-established record of causing bugs and security holes, especially when the integers in question represent the size of something in memory, and an unanticipated overflow occurs. In our experience, the act of writing out numeric conversions in Rust has alerted us to problems we would otherwise have missed.

We explain exactly how conversions behave in “Type Casts”.

The bool Type

Rust’s Boolean type, `bool`, has the usual two values for such types, `true` and `false`. Comparison operators like == and < produce `bool` results: the value of `2 < 5` is `true`.

Many languages are lenient about using values of other types in contexts that require a Boolean value: C and C++ implicitly convert characters, integers, floating-point numbers, and pointers to Boolean values, so they can be used directly as the condition in an `if` or `while` statement. Python and JavaScript permit any value at all in an `if` condition, with language-specific rules to determine which values are treated as `true` there and which are `false`. Rust, however, is very strict: control structures like `if` and `while` require their conditions to be `bool` expressions, as do the short-circuiting logical operators `&&` and `||`. You must write `if x != 0 { ... }`, not simply `if x { ... }`.

Rust’s `as` operator can convert `bool` values to integer types:

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

However, `as` won’t convert in the other direction, from numeric types to `bool`. Instead, you must write out an explicit comparison like `x != 0`.

Although a `bool` needs only a single bit to represent it, Rust uses an entire byte for a `bool` value in memory, so you can create a pointer to it.

Tuples

A *tuple* is a pair, or triple, quadruple, quintuple, etc. (hence, *n-tuple*, or *tuple*), of values of assorted types. You can write a tuple as a sequence of values, separated by commas and surrounded by parentheses. For example, `(1984_i32, false)` is a tuple whose first field is an integer, and whose second is a boolean; its type is `(i32, bool)`. Given a tuple value `t`, you can access its fields as `t.0`, `t.1`, and so on.

Tuples are very different from arrays. For one thing, each field of a tuple can have a different type, whereas an array's elements must all be the same type. Further, while accessing elements by a variable index, `arr[i]`, is practically the defining feature of arrays, there is no way to do this with a tuple. You can access a field by its numeric "name", `t.1`, but you can't write `t.i` or `t[i]` to get the `i`th field.

Rust code often uses tuple types to return multiple values from a function. For example, the `split_at` method on string slices, which divides a string into two halves and returns them both, is declared like this:

```
fn split_at(&self, mid: usize) -> (&str, &str);
```

The return type `(&str, &str)` is a tuple of two string slices. You can use pattern-matching syntax to assign each field of the return value to a different variable:

```
let text = "I see the eigenvalue in thine eye";
let (head, tail) = text.split_at(21);
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

This is more legible than the equivalent:

```
let text = "I see the eigenvalue in thine eye";
let temp = text.split_at(21);
let head = temp.0;
let tail = temp.1;
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

You'll also see tuples used as a sort of minimal-drama struct type. For example, in the Mandelbrot program in Chapter 1, we needed to pass the width and height of the image to the functions that plot it and write it to disk. We could declare a struct with `width` and `height` members, but that's pretty heavy notation for something so obvious, so we just used a tuple:

```
/// Write the buffer `pixels`, whose dimensions are given by
`bounds`, to the
/// file named `filename`.
fn write_image(
    filename: &str,
    pixels: &[u8],
    bounds: (usize, usize),
) -> Result<(), std::io::Error> {
    ...
}
```

The type of the `bounds` parameter is `(usize, usize)`, a tuple of two `usize` values. Admittedly, we could just as well write out separate `width` and `height` parameters, and the machine code would be about the same either way. It's a matter of clarity. We think of the size as one value, not two, and using a tuple lets us write what we mean.

The other commonly used tuple type is the zero-tuple `()`. This is traditionally called the *unit type* because it has only one value, also written `()`. Rust uses the unit type where there's no meaningful value to carry, but context requires some sort of type nonetheless.

For example, a function that returns no value has a return type of `()`. The standard library's `std::thread::sleep` function has no meaningful value to return; it just pauses the program for a while. The declaration for `std::thread::sleep` reads:

```
fn sleep(duration: Duration);
```

The signature omits the function's return type altogether, which is shorthand for returning the unit type:

```
fn sleep(duration: Duration) -> ();
```

Similarly, the `write_image` example we mentioned before has a return type of `Result<(), std::io::Error>`, meaning that the function returns a `std::io::Error` value if something goes wrong, but returns no value on success.

Pointer Types

Rust has several types that represent memory addresses.

This is unusual in modern programming languages for two reasons. First, in many languages, values do not nest. In Java, for example, if `class Rectangle` contains a field `Vector2D upperLeft;`, then `upperLeft` is a reference to another separately created `Vector2D` object. Objects never physically contain other objects in Java. The language doesn't need pointer types because every Java object type is implicitly a pointer type.

Rust is different. The language is designed to help keep allocations to a minimum, so values nest by default. The value `((0, 0), (1440, 900))` is stored as four adjacent integers. If you store it in a local variable, you've got a local variable four integers wide. Nothing is allocated in the heap. This is great for memory efficiency, but as a consequence, whenever a Rust program needs one value to point to another, it must say so by using a pointer type explicitly.

The second reason Rust needs several pointer types is that there’s no garbage collector to manage memory for us. It turns out memory still needs to be managed. Rust programs must decide when to allocate memory, how much to allocate, and when it’s safe to deallocate it. In Rust, these memory management policies are written out in the pointer types themselves. Here are some Rust pointer types and the policies they represent. Don’t worry if this is a bit bewildering at first; we explain all of these types fully in later chapters.

- `&T` and `&mut T` - We showed a few examples of references already in Chapter 1. References refer to “borrowed” values without taking on any responsibility for cleanup. Therefore references have no memory management overhead at all.

- `Box<T>` - This is a pointer type that points to a value of type `T` stored in the heap. Memory is allocated when the program calls `Box::new(value)`. When the box is dropped, the memory is freed. Boxes are not needed every day, but they’re the most basic way to override the default nesting of values, which is occasionally useful.

- `std::rc::Rc<T>` and `std::sync::Arc<T>` - These are Rust’s support for reference counting. Unlike `Box<T>` pointers, multiple `Rc<T>` pointers can point to the same heap-allocated `T`. This is useful when, say, multiple parts of a program all need access to a shared component. Rust tracks how many `Rc`s point to each shared value. The memory for the `T` is freed when the last `Rc` pointing to it is dropped. `Arc<T>` is the same, but it uses an atomic reference count, so `Arc`s can be shared across multiple threads.

- `*const T` and `*mut T` - These are Rust’s raw pointer types, the equivalent of C pointers. They can be dereferenced only in `unsafe` blocks, code that explicitly opts out of Rust’s safety guarantees. Raw pointers can be necessary when calling into a C library, or behind the scenes when implementing a new kind of data structure from scratch. We won’t discuss them again until [Link to Come].

All of these types point to a value of type T; they differ only in memory management strategies and corresponding rules about mutability and thread safety.

Java doesn’t need all this variety: the garbage collector provides a convenient, one-size-fits-all strategy for managing memory. C++ programmers, however, will recognize these as *smart pointer* types, similar to those in the C++ standard library: `Box<T>` is like C++’s `std::unique_ptr<T>`, and `Arc<T>` is like C++’s `std::shared_ptr<T>`.

Except for raw pointers, Rust pointer types are non-nullable. Use `Option` for optional pointers, with `None` representing `null`. There’s no extra memory cost; an `Option<&T>` is the same size as a `&T`.

Arrays, Vectors, and Slices

Rust has three types for representing a sequence of values in memory:

- The type `[T; N]` represents a fixed-size array of `N` values, each of type `T`. An array’s size is a constant determined at compile time and is part of the type; you can’t grow or shrink an array.

- The type `Vec<T>`, called a *vector of* `T`*s*, is a dynamically allocated, growable sequence of values of type `T`. A vector’s elements live in the heap, so you can resize vectors at will: push new elements onto them, append other vectors to them, delete elements, and so on.

- The types `&[T]` and `&mut [T]`, called a *shared slice of* `T`*s* and *mutable slice of* `T`*s*, are references to a series of elements that are a part of some other value, like an array or vector. You can think of a slice as a pointer to its first element, together with a count of the number of elements you can access starting at that point. A mutable slice `&mut [T]` lets you read and modify elements, but can't be shared; a shared slice `&[T]` lets you share access among several readers, but doesn't let you modify elements.

Given a value `v` of any of these three types, the expression `v.len()` gives the number of elements in `v`, and `v[i]` refers to the `i`th element of `v`. The first element is `v[0]`, and the last element is `v[v.len() - 1]`. Rust checks that `i` always falls within this range; if it doesn't, the expression panics. The length of `v` may be zero, in which case any attempt to index it will panic. `i` must be a `usize` value; you can't use any other integer type as an index.

Arrays

There are several ways to write array values. The simplest is to write a series of values within square brackets:

```
let lazy_caterer: [u32; 6] = [1, 2, 4, 7, 11, 16];
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];

assert_eq!(lazy_caterer[3], 7);
assert_eq!(taxonomy.len(), 3);
```

For the common case of a long array filled with some value, you can write `[`*V*`; `*N*`]`, where *V* is the value each element should have, and *N* is the length. For example, `[true; 10000]` is an array of 10,000 `bool` elements, all set to `true`:

```
let mut sieve = [true; 10000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 10000 {
            sieve[j] = false;
            j += i;
        }
    }
}

assert!(sieve[211]);
assert!(!sieve[9876]);
```

You'll see this syntax used for fixed-size buffers: `[0u8; 1024]` is a one-kilobyte buffer, filled with zeros. Rust has no notation for an uninitialized array. (In general, Rust ensures that code can never access any sort of uninitialized value.)

An array's length is part of its type and fixed at compile time. If `n` is a variable, you can't write `[true; n]` to get an array of `n` elements. When you need an array whose length varies at run time (and you usually do), use a vector instead.

The useful methods you'd like to see on arrays—iterating over elements, searching, sorting, filling, filtering, and so on—are all provided as methods on slices, not arrays. But Rust implicitly converts a reference to an array to a slice when searching for methods, so you can call any slice method on an array directly:

```
let mut chaos = [3, 5, 4, 1, 2];
chaos.sort();
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```

Here, the `sort` method is actually defined on slices, but since it takes its operand by reference, Rust implicitly produces a `&mut [i32]` slice referring to the entire array and passes that to `sort` to operate on. In fact, the `len` method we mentioned earlier is a slice method as well. We cover slices in more detail in "Slices".

Vectors

A vector Vec<T> is a resizable array of elements of type T, allocated in the heap.

There are several ways to create vectors. The simplest is to use the vec! macro, which gives us a syntax for vectors that looks very much like an array literal:

```
let mut primes = vec![2, 3, 5, 7];
assert_eq!(primes.iter().product::<i32>(), 210);
```

But of course, this is a vector, not an array, so we can add elements to it dynamically:

```
primes.push(11);
primes.push(13);
assert_eq!(primes.iter().product::<i32>(), 30030);
```

You can also build a vector by repeating a given value a certain number of times, again using a syntax that imitates array literals:

```
fn new_pixel_buffer(rows: usize, cols: usize) -> Vec<u8> {
    vec![0; rows * cols]
}
```

The vec! macro is equivalent to calling Vec::new to create a new, empty vector and then pushing the elements onto it:

```
let mut pal = Vec::new();
pal.push("step");
pal.push("on");
pal.push("no");
pal.push("pets");
assert_eq!(pal, vec!["step", "on", "no", "pets"]);
```

Another possibility is to build a vector from the values produced by an iterator:

```
let v: Vec<i32> = (0..5).collect();
assert_eq!(v, [0, 1, 2, 3, 4]);
```

You’ll often need to supply the type when using `collect` (as we’ve done here), because it can build many different sorts of collections, not just vectors. By specifying the type of `v`, we’ve made it unambiguous which sort of collection we want.

As with arrays, you can use slice methods on vectors:

```
// A palindrome!
let mut palindrome = vec!["a man", "a plan", "a canal",
"panama"];
palindrome.reverse();
// Reasonable yet disappointing:
assert_eq!(palindrome, vec!["panama", "a canal", "a plan", "a
man"]);
```

Here, the `reverse` method is actually defined on slices, but the call implicitly borrows a `&mut [&str]` slice from the vector and invokes `reverse` on that.

`Vec` is an essential type to Rust—it’s used almost anywhere one needs a list of dynamic size—so there are many other methods that construct new vectors or extend existing ones. We’ll cover them in [Link to Come].

A `Vec<T>` consists of three values: a pointer to the heap-allocated buffer for the elements, which is created and owned by the `Vec<T>`; the number of elements that buffer has the capacity to store; and the number it actually contains now (in other words, its length). When the buffer has reached its capacity, adding another element to the vector entails allocating a larger buffer, copying the present contents into it, updating the vector’s pointer and capacity to describe the new buffer, and finally freeing the old one.

If you know the number of elements a vector will need in advance, instead of `Vec::new` you can call `Vec::with_capacity` to create a vector with a buffer large enough to hold them all, right from the start; then, you can add the elements to the vector one at a time without causing any reallocation. The `vec!` macro uses a trick like this, since it knows how many elements the final vector will have. Note that this only establishes the vector's initial size; if you exceed your estimate, the vector simply enlarges its storage as usual.

Many library functions look for the opportunity to use `Vec::with_capacity` instead of `Vec::new`. For example, in the `collect` example, the iterator `0..5` knows in advance that it will yield five values, and the `collect` function takes advantage of this to pre-allocate the vector it returns with the correct capacity. We'll see how this works in [Link to Come].

Just as a vector's `len` method returns the number of elements it contains now, its `capacity` method returns the number of elements it could hold without reallocation:

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);

v.push(3);
assert_eq!(v.len(), 3);
// Typically prints "capacity is now 4":
println!("capacity is now {}", v.capacity());
```

The capacity printed at the end isn't guaranteed to be exactly 4, but it will be at least 3, since the vector is holding three values.

You can insert and remove elements wherever you like in a vector, although these operations shift all the elements after the affected position forward or backward, so they may be slow if the vector is long:

```
let mut v = vec![10, 20, 30, 40, 50];

// Make the element at index 3 be 35.
v.insert(3, 35);
assert_eq!(v, [10, 20, 30, 35, 40, 50]);

// Remove the element at index 1.
v.remove(1);
assert_eq!(v, [10, 30, 35, 40, 50]);
```

You can use the `pop` method to remove the last element and return it. More precisely, popping a value from a `Vec<T>` returns an `Option<T>`: `None` if the vector was already empty, or `Some(v)` if its last element had been `v`:

```
let mut v = vec!["Snow Puff", "Glass Gem"];
assert_eq!(v.pop(), Some("Glass Gem"));
assert_eq!(v.pop(), Some("Snow Puff"));
assert_eq!(v.pop(), None);
```

You can use a `for` loop to iterate over a vector:

```
// Get our command-line arguments as a vector of Strings.
let languages: Vec<String> = std::env::args().skip(1).collect();
for l in languages {
    println!(
        "{l}: {}",
        if l.len() % 2 == 0 {
            "functional"
        } else {
            "imperative"
        },
    );
}
```

Running this program with a list of programming languages is illuminating:

```
$ cargo run Lisp Scheme C C++ Fortran
   Compiling proglangs v0.1.0 (/home/jimb/rust/proglangs)
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
```

```
     Running `target/debug/proglangs Lisp Scheme C C++ Fortran`
Lisp: functional
Scheme: functional
C: imperative
C++: imperative
Fortran: imperative
$
```

Finally, a satisfying definition for the term *functional language*.

Despite its fundamental role, `Vec` is an ordinary type defined in Rust, not built into the language. We'll cover the techniques needed to implement such types in [Link to Come].

Slices

A slice, written `[T]` without specifying the length, is a region of an array or vector. Since a slice can be any length, slices can't be stored directly in variables or passed as function arguments. Slices are always passed by reference.

A reference to a slice is a *fat pointer*: a two-word value comprising a pointer to the slice's first element, and the number of elements in the slice.

Suppose you run the following code:

```
let v: Vec<f64> = vec![0.0,  0.707,  1.0,  0.707];
let a: [f64; 4] =     [0.0, -0.707, -1.0, -0.707];

let sv: &[f64] = &v;
let sa: &[f64] = &a;
```

In the last two lines, Rust automatically converts the `&Vec<f64>` reference and the `&[f64; 4]` reference to slice references that point directly to the data.

By the end, memory looks like Figure 2-2.

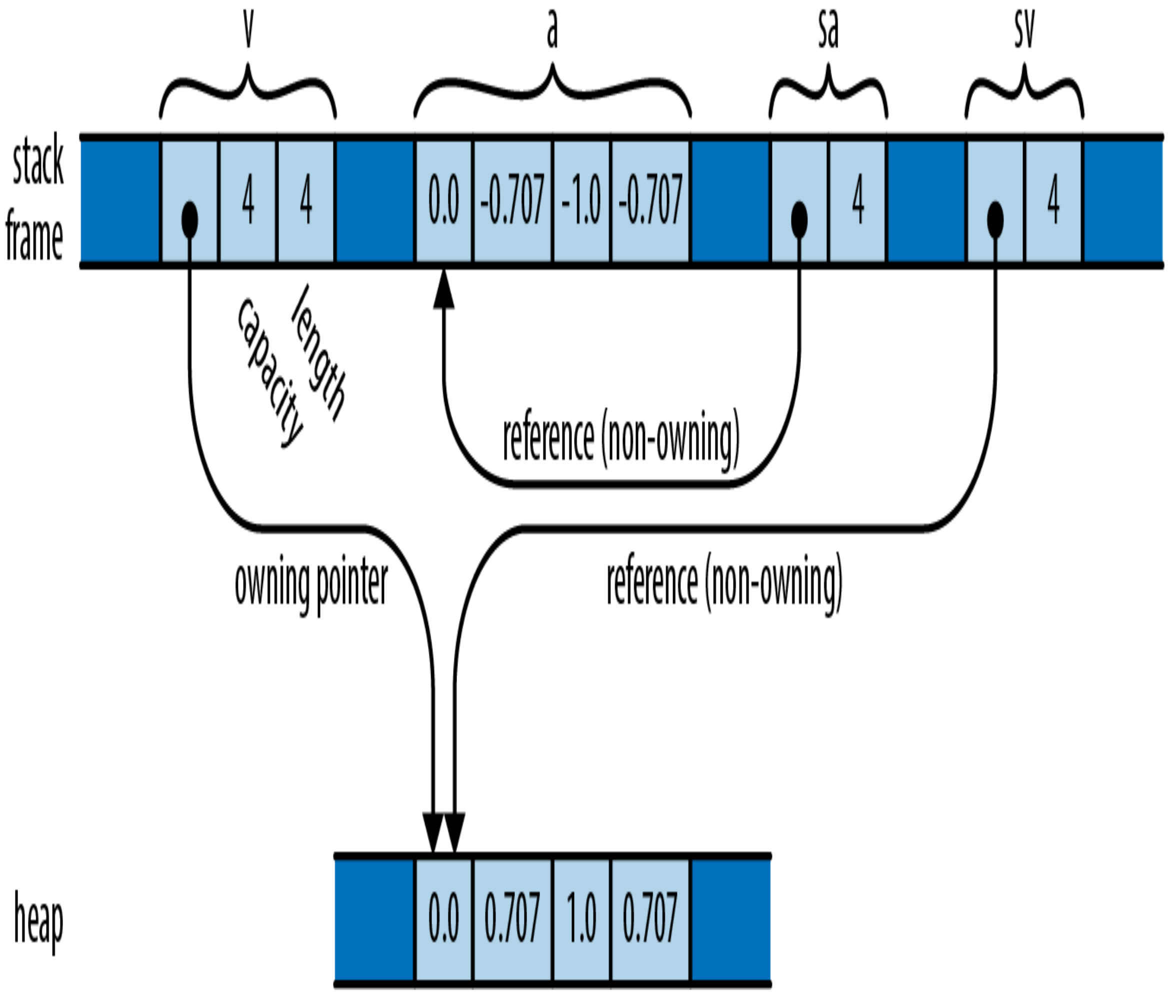


Figure 2-2. A vector `v` *and an array* `a` *in memory, with slices* `sa` *and* `sv` *referring to each*

Whereas an ordinary reference is a non-owning pointer to a single value, a reference to a slice is a non-owning pointer to a contiguous range of values in memory. This makes slice references a good choice when you want to write a function that operates on either an array or a vector. For example, here's a function that prints a slice of numbers, one per line:

```
fn print(n: &[f64]) {
    for elt in n {
        println!("{elt}");
    }
}

print(&a);  // works on arrays
print(&v);  // works on vectors
```

You can get a reference to a slice of an array or vector, or a slice of an existing slice, by indexing it with a range:

```
print(&v[0..2]);    // print the first two elements of v
print(&a[2..]);     // print elements of a starting with a[2]
print(&sv[1..3]);   // print v[1] and v[2]
```

As with ordinary array accesses, Rust checks that the indices are valid. Trying to borrow a slice that extends past the end of the data results in a panic.

If our `print` function took its argument as a `&Vec<f64>`, if would be unnecessarily limited to full vectors only; `&v[0..2]` and so on would not be valid arguments. Where possible, functions should accept slice references rather than vector or array references, for generality. This is also why the `sort` and `reverse` methods are defined on the slice type `[T]` rather than on `Vec<T>`.

Since slices almost always appear behind references, we often just refer to types like `&[T]` or `&str` as “slices,” using the shorter name for the more common concept.

String and Character Types

Programmers familiar with C++ will recall that there are two string types in the language. String literals have the pointer type `const char *`. The standard library also offers a class, `std::string`, for dynamically creating strings at run time.

Rust has a similar design. In this section, we’ll show the syntax for string and character literals and then introduce the string and character types. For much more about strings and text processing, see [Link to Come].

String and Character Literals

String literals are enclosed in double quotes.

```
let quote = "Editing is a rewording activity.";  // Alan Perlis
```

Character literals are enclosed in single quotes, like `'8'` or `'!'`. They have the type `char`, which can represent any single Unicode character: `'錆'` is a `char` literal representing the Japanese kanji for *sabi* (rust).

A string may span multiple lines:

```
println!("In the room the women come and go,
    Singing of Mount Abora");
```

The newline character in that string literal is included in the string and therefore in the output. So are the spaces at the beginning of the second line. (Windows-style line endings do not affect the string's content. A line break in a string is always treated as a single newline character, `'\n'`.)

If one line of a string ends with a backslash, then the newline character and the leading whitespace on the next line are dropped:

```
println!("It was a bright, cold day in April, and \
    there were four of us—\
    more or less.");
```

This prints a single line of text. The string contains a single space between "and" and "there" because there is a space before the backslash in the program, and no space between the em dash and "more."

As in many other languages, *escape sequences* stand for special characters (Table 2-6).

Table 2-6. Character escape sequences

Character	**Escape sequence**
Single quote, `'`	`\'`
Backslash, `\`	`\\`
Newline	`\n`
Carriage return	`\r`
Tab	`\t`
Any ASCII character (0 to 127)	`\x` exactly two hex digits
Any character	`\u{` up to six hex digits `}`

You can write any character as `\u{HHHHHH}`, where `HHHHHH` is the code point as a hexadecimal number of up to 6 digits, with underscores allowed for grouping as usual. For example, the character literal `'\u{CA0}'` represents the character "ಠ", a Kannada letter used in the Unicode Look of Disapproval, "ಠ_ಠ". The same literal could also be simply written as `'ಠ'`.

In a few cases, the need to double every backslash in a string is a nuisance. (The classic examples are regular expressions and Windows paths.) For these cases, Rust offers *raw strings*. A raw string is tagged with the lowercase letter `r`. All backslashes and whitespace characters inside a raw string are included verbatim in the string. No escape sequences are recognized:

```
let default_win_install_path = r"C:\Program Files\Gorillas";

let pattern = Regex::new(r"\d+(\.\d+)*");
```

You can’t include a double-quote character in a raw string simply by putting a backslash in front of it—remember, we said *no* escape sequences are recognized. However, there is a cure for that too. The start and end of a raw string can be marked with pound signs:

```
println!(r###"
    This raw string started with 'r###"'.
    Therefore it does not end until we reach a quote mark ('"')
    followed immediately by three pound signs ('###'):
"###);
```

You can add as few or as many pound signs as needed to make it clear where the raw string ends.

Characters and Strings in Memory

Rust’s character type `char` represents a single Unicode character by its code point. Each `char` value takes up 4 bytes of memory, the same as a `u32`.

Rust strings are sequences of Unicode characters, but they are not stored in memory as arrays of `char`s. Instead, they are stored using UTF-8, a variable-width encoding. Each ASCII character in a string is stored in one byte. Other characters take up multiple bytes.

Figure 2-3 shows the `String` and `&str` values created by the following code:

```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let disapproodles = "ಠ_ಠ";
```

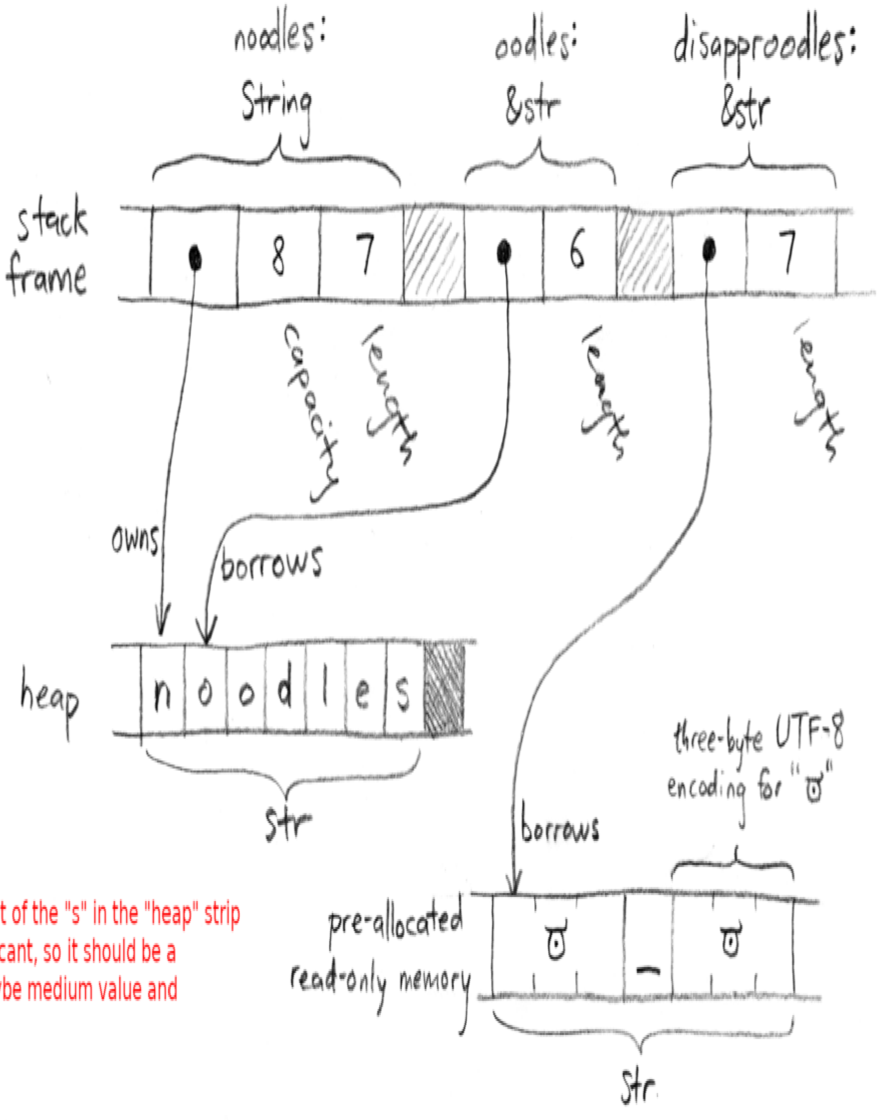
A `String` has a resizable buffer holding UTF-8 text. The buffer is allocated in the heap, so it can be resized as needed. In the figure, `noodles` is a `String` that owns an eight-byte buffer, of which seven are in use. You can think of a `String` as a `Vec<u8>` that is guaranteed to hold well-formed UTF-8; in fact, this is how `String` is implemented.

A `&str` (pronounced “stir” or “string slice”) is a reference to a run of UTF-8 text owned by someone else: it “borrows” the text. In the example, `oodles` is a `&str` referring to the last six bytes of the text belonging to `noodles`, so it represents the text `"oodles"`. Like other slice references, a `&str` is a fat pointer, containing both the address of the actual data and its length.

You can think of a `&str` as being nothing more than a `&[u8]` that is guaranteed to hold well-formed UTF-8. Likewise, a `char` is essentially a `u32` that is guaranteed to hold a valid Unicode scalar value. They are types with *invariants*: rules about their values that are enforced by the implementation, that can’t be broken except by abusing unsafe code, and that programs can therefore rely on.

Note to the illustrator:

The "fringe" of each strip and the two shaded boxes in "stack frame" should be a "don't care" color, perhaps pale and grayish. They represent parts of the computer's memory we don't care about right now.



This box to the right of the "s" in the "heap" strip is empty but significant, so it should be a different color, maybe medium value and saturation.

Figure 2-3. `String`, `&str`, and `str`

A string literal is a `&str` that refers to preallocated text, typically stored in read-only memory along with the program's machine code. In the preceding example, `"ಠ_ಠ"` is a string literal, pointing to seven bytes that are loaded into memory when the program begins execution and that last until it exits.

A `String` or `&str`’s `.len()` method returns its length. The length is measured in bytes, not characters:

```
assert_eq!("ಠ_ಠ".len(), 7);
assert_eq!("ಠ_ಠ".chars().count(), 3);
```

It is impossible to modify a `&str`:

```
let mut s = "hello";
s[0] = 'c';    // error: `&str` cannot be modified, and other
reasons
s.push('\n');  // error: no method named `push` found for
reference `&str`
```

To create new strings at run time, use `String`.

String

`&str` is very much like `&[T]`: a fat pointer to some data. `String` is analogous to `Vec<T>`, as described in Table 2-7.

Table 2-7. `Vec<T>` and `String` comparison

	Vec<T>	**String**
Automatically frees buffers	Yes	Yes
Growable	Yes	Yes
`::new()` and `::with_capacity()` type-associated functions	Yes	Yes
`.reserve()` and `.capacity()` methods	Yes	Yes
`.push()` and `.pop()` methods	Yes	Yes
Range syntax `&v[start..stop]`	Yes, returns `&[T]`	Yes, returns `&str`
Automatic conversion	`&Vec<T>` to `&[T]`	`&String` to `&str`
Inherits methods	From `&[T]`	From `&str`

Like a `Vec`, each `String` has its own heap-allocated buffer that isn't shared with any other `String`. When a `String` variable goes out of scope, the buffer is automatically freed, unless the `String` was moved.

There are several ways to create `String`s:

- The `.to_string()` method converts a `&str` to a `String`. This copies the string:

```
let error_message = "too many pets".to_string();
```

- The `format!()` macro works just like `println!()`, except that it returns a new `String` instead of writing text to stdout, and it doesn't automatically add a newline at the end:

```
let place = "Portland";
assert_eq!(format!("Hello, {place}!"), "Hello, Portland!".to_string());
```

The first argument to `format!()` or `println!()` is called a *format string*. The bits in curly braces are called *format specifiers*, and in the output, each format specifier is replaced with some value. Values can be passed as additional arguments to the macro or, for single identifiers only, included in the format specifier itself:

```
assert_eq!(
    format!("from {} to {}", start, end),
    format!("from {start} to {end}"),
);
```

A format specifier of `{}` or `{`*`name`*`}` renders a value in `Display` format, for end users; `{:?}` or `{`*`name`*`:?}` renders it in `Debug` format, for debugging. An example of the difference is that if `s` is a string, `println!("{s}")` prints the content of the string verbatim, whereas `println!("{s:?}")` adds double quotes around it and renders any special characters in `s` as escape sequences. Not all types support `Display` format, but almost everything supports `Debug`. You'll need `{:?}` to print a tuple, slice, `Vec`, or `Option`.

To include an actual curly brace in the output, double it: `println!("{{")`.

That should be enough to get on with. Rust format strings offer many more features for fine-tuning the output. We cover them in [Link to Come].

- To concatenate a list of strings, use `.concat()` or `.join(sep)`:

  ```
  let bits = vec!["veni", "vidi", "vici"];
  assert_eq!(bits.concat(), "venividivici");
  assert_eq!(bits.join(", "), "veni, vidi, vici");
  ```

  These methods are defined for slices of type `[&str]` and `[String]`, and are therefore available on arrays and vectors as well.

The choice sometimes arises of which type to use: `&str` or `String`. Chapter 4 addresses this question in detail. For now it will suffice to point out that a `&str` can refer to any slice of any string, so as with slices vs. vectors, `&str` is usually preferable to `String` for function arguments.

Using Strings

Strings support the == and `!=` operators. Two strings are equal if they contain the same characters in the same order (regardless of whether they point to the same location in memory):

```
assert!("ONE".to_lowercase() == "one");
```

Strings also support the comparison operators <, <=, >, and >=, as well as many useful methods and functions that you can find in the online documentation under “Primitive Type `str`” or the “`std::str`” module (or just flip to [Link to Come]). Here are a few examples:

```
assert!("peanut".contains("nut"));
assert_eq!("ಠ_ಠ".replace("ಠ", "■"), "■_■");
assert_eq!("    clean\n".trim(), "clean");

for word in "veni, vidi, vici".split(", ") {
    assert!(word.starts_with("v"));
}
```

Characters have some methods of their own, for classification and conversions. They are documented under “Primitive Type `char`” (and in [Link to Come]).

```
assert_eq!('β'.is_alphabetic(), true);
assert_eq!(std::char::from_u32(0x1f642), Some('†▼);
assert_eq!('†⌶len_utf8(), 4);
assert_eq!('1'.to_digit(10), Some(1));
assert_eq!(std::char::from_digit(2, 10), Some('2'));
```

Other String-Like Types

Rust guarantees that strings are valid UTF-8. Sometimes a program really needs to be able to deal with strings that are *not* valid Unicode. This usually happens when a Rust program has to interoperate with some other system that doesn’t enforce any such rules. For example, in most operating systems it’s easy to create a file with a filename that isn’t valid Unicode. What should happen when a Rust program comes across this sort of filename?

Rust’s solution is to offer a few string-like types for these situations:

- For ordinary text, use `String` and `&str`.
- When working with filenames, use `PathBuf` and `&Path` instead. You can import them from `std::path`.
- When working with environment variable names and command-line arguments, if you need to handle non-UTF-8 values, use `OsString` and `&OsStr` from `std::ffi`. But this is rarely

necessary. Most programs just use `String`, as we did when handling command line arguments in Chapter 1.

- When working with binary data that isn’t UTF-8 encoded at all, use `Vec<u8>` and `&[u8]`.
- When interoperating with C libraries that use null-terminated strings, use `CString` and `&CStr` from `std::ffi`.

For those last two situations, Rust has special syntax:

```
let elf_magic = b"\x7fELF";    // a byte string (not UTF-8)
assert_eq!(elf_magic, &[127u8, b'E', b'L', b'F']);

let branch = c"main";   // a C string (null-terminated)
assert_eq!(branch,
CStr::from_bytes_with_nul(b"main\0").unwrap());
```

A string literal prefixed with `b` is a *byte string*. For example, `b"\x7fELF"` is a byte string; its type is `&[u8; 4]`, a reference to an immutable array of bytes. Since a byte string represents bytes, not characters, it does not have to be valid UTF-8. (This particular example is the “magic number” that appears as the first four bytes of practically every Linux executable, including the ones you’ve been building with `cargo build`, if you’ve been working through the examples in this book on Linux.)

Similarly, a *byte literal* is a character-like literal prefixed with `b`. It has the type `u8`. For example, since the ASCII code for `A` is 65, the literals `b'A'` and `65u8` are exactly equivalent. Byte strings and byte literals can’t contain arbitrary Unicode characters; they must make do with ASCII and `\xHH` escape sequences.

The string `c"main"` is a *C string*. It is like the regular string `"main"` except it is *null-terminated*; that is, it’s stored with an extra 0 byte to indicate the end of the string, as required by many C APIs. Its type is `&std::ffi::CStr`.

You won’t use these features every day, but we will show example code using a byte string in [Link to Come] and a C string in [Link to Come].

Type Aliases

The `type` keyword can be used like `typedef` in C++ to declare a new name for an existing type:

```
type Bytes = Vec<u8>;
```

The type `Bytes` that we’re declaring here is shorthand for this particular kind of `Vec`:

```
fn encode(image: &Bitmap) -> Bytes {  // returns a Vec<u8>
    ...
}
```

Beyond the Basics

Types are a central part of Rust. We’ll continue talking about types and introducing new ones throughout the book. In particular, Rust’s user-defined types give the language much of its flavor, because that’s where methods are defined. There are three kinds of user-defined types, and we’ll cover them in three successive chapters: structs in [Link to Come], enums in [Link to Come], and traits in [Link to Come].

Functions and closures have their own types, covered in [Link to Come]. And the types that make up the standard library are covered throughout the book. For example, [Link to Come] presents the standard collection types.

All of that will have to wait, though. Before we move on, it’s time to tackle the concepts that are at the heart of Rust’s safety rules.

Chapter 3. Ownership and Moves

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

When it comes to managing memory, there are two characteristics we’d like from our programing languages:

- We’d like memory to be freed promptly, at a time of our choosing. This gives us control over the program’s memory consumption.
- We never want to use a pointer to an object after it’s been freed. This would be undefined behavior, leading to crashes and security holes.

But these seem to be mutually exclusive: freeing a value while pointers exist to it necessarily leaves those pointers dangling. Almost all major programming languages fall into one of two camps, depending on which of the two qualities they give up on:

- The “Safety First” camp uses garbage collection to manage memory, automatically freeing objects when all reachable pointers to them are gone. This eliminates dangling pointers by simply keeping the objects around until there are no pointers to them left to dangle. Almost all modern languages fall in this camp, from Python, JavaScript, and Ruby to Java, C#, and Haskell.

  But relying on garbage collection means relinquishing control over exactly when objects get freed to the collector. In general, garbage collectors are surprising beasts, and understanding why memory wasn’t freed when you expected can be a challenge.

- The “Control First” camp leaves you in charge of freeing memory. Your program’s memory consumption is entirely in your hands, but avoiding dangling pointers also becomes entirely your concern. C and C++ are the only mainstream languages in this camp.

  This is great if you never make mistakes, but evidence suggests that eventually you will. Pointer misuse has been a common culprit in reported security problems for as long as that data has been collected.

Rust aims to be both safe and performant, so neither of these compromises is acceptable. But if reconciliation were easy, someone would have done it long before now. Something fundamental had to change.

Rust breaks the deadlock in a surprising way: by imposing some rules restricting how your programs can use pointers, the net effect of which is to bring just enough order to the chaos to allow Rust’s compile-time checks to verify that your program is free of memory safety errors. At compile time, Rust records where and when values are created and used, tracking the ownership of each value as it moves through the program to prevent the use of uninitialized values and free them exactly once, and only when they are no longer needed, avoiding double-free bugs. The compiler also analyzes references to those values to prevent the use of dangling pointers. At run time, your pointers are simple addresses in memory, just as they would be in C and C++. The difference is that your code has been proven to use them safely.

This chapter and the next are devoted to explaining exactly what these restrictions are and why they work.

These same rules also form the basis of Rust's support for safe concurrent programming. Using Rust's carefully designed threading primitives, the rules that ensure your code uses memory correctly also serve to prove that it is free of data races. A bug in a Rust program cannot cause one thread to corrupt another's data, introducing hard-to-reproduce failures in unrelated parts of the system. The nondeterministic behavior inherent in multithreaded code is isolated to those features designed to handle it—mutexes, message channels, atomic values, and so on—rather than appearing in ordinary memory references. Multithreaded code in C and C++ has earned its ugly reputation, but Rust rehabilitates it quite nicely.

Rust's radical wager, the claim on which it stakes its success and that forms the root of the language, is that even with these restrictions in place, you'll find the language more than flexible enough for almost every task and that the benefits—the elimination of broad classes of memory management and concurrency bugs—will justify the adaptations you'll need to make to your style. The authors of this book are bullish on Rust exactly because of our extensive experience with C and C++. For us, Rust's deal is a no-brainer.

Rust's rules are probably unlike what you've seen in other programming languages. Learning how to work with them and turn them to your advantage is, in our opinion, the central challenge of learning Rust. In this chapter, we'll first provide insight into the logic and intent behind Rust's rules by showing how the same underlying issues play out in other languages. Then, we'll explain Rust's rules in detail, looking at what ownership means at a conceptual and mechanical level, how changes in ownership are tracked in various scenarios, and types that bend or break some of these rules in order to provide more flexibility.

Ownership

If you’ve read much C or C++ code, you’ve probably come across a comment saying that an instance of some class *owns* some other object that it points to. This generally means that the owning object gets to decide when to free the owned object: when the owner is destroyed, it destroys its possessions along with it.

For example, suppose you write the following C++ code:

```
std::string s = "frayed knot";
```

The string `s` is usually represented in memory as shown in Figure 3-1.

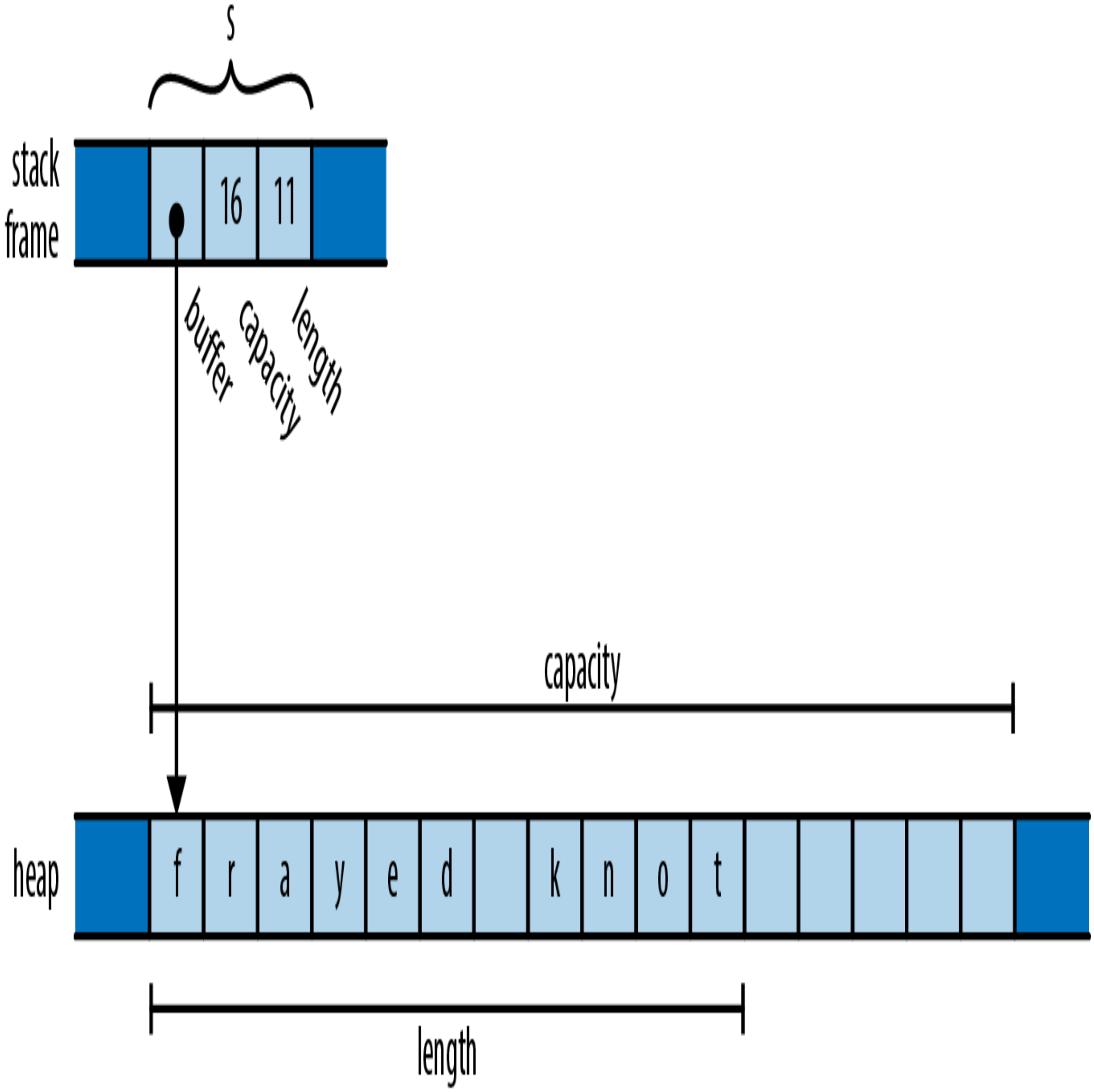


Figure 3-1. A C++ `std::string` value on the stack, pointing to its heap-allocated buffer

Here, the actual `std::string` object itself is always exactly three words long, comprising a pointer to a heap-allocated buffer, the buffer's overall capacity (that is, how large the text can grow before the string must allocate a larger buffer to hold it), and the length of the text it holds now. These are fields private to the `std::string` class, not accessible to the string's users.

A `std::string` owns its buffer: when the program destroys the string, the string's destructor frees the buffer. In the past, some C++ libraries shared a single buffer among several `std::string` values, using a reference count to decide when the buffer should be freed. Newer versions of the C++ specification effectively preclude that representation; all modern C++ libraries use the approach shown here.

In these situations it's generally understood that although it's fine for other code to create temporary pointers to the owned memory, it is that code's responsibility to make sure its pointers are gone before the owner decides to destroy the owned object. You can create a pointer to a character living in a `std::string`'s buffer, but when the string is destroyed, your pointer becomes invalid, and it's up to you to make sure you don't use it anymore. The owner determines the lifetime of the owned, and everyone else must respect its decisions.

We've used `std::string` here as an example of what ownership looks like in C++: it's just a convention that the standard library generally follows, and although the language encourages you to follow similar practices, how you design your own types is ultimately up to you.

In Rust, however, the concept of ownership is built into the language itself and enforced by compile-time checks. Every value has a single owner that determines its lifetime. When the owner is freed—*dropped*, in Rust terminology—the owned value is dropped too. These rules are meant to make it easy for you to find any given value's lifetime simply by inspecting the code, giving you the control over its lifetime that a systems language should provide.

A variable owns its value. When control leaves the block in which the variable is declared, the variable is dropped, so its value is dropped along with it. For example:

```
fn print_padovan() {
    let mut padovan = vec![1,1,1];  // allocated here
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
```

```
        println!("P(1..10) = {padovan:?}");
    }                                           // dropped here
```

The type of the variable `padovan` is `Vec<i32>`, a vector of 32-bit integers. In memory, the final value of `padovan` will look something like Figure 3-2.

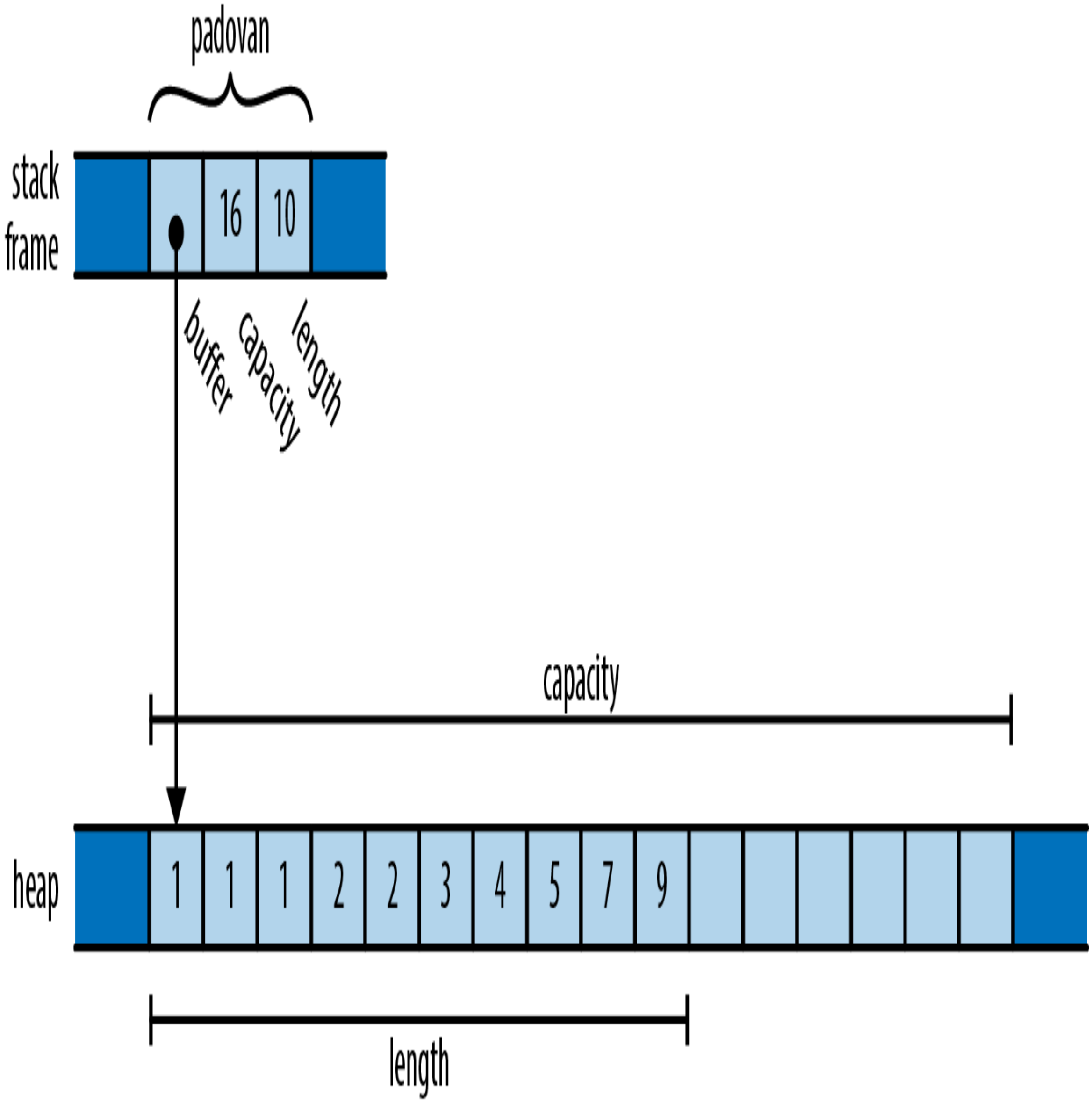


Figure 3-2. A `Vec<i32>` on the stack, pointing to its buffer in the heap

This is very similar to the C++ `std::string` we showed earlier, except that the elements in the buffer are 32-bit values, not characters. Note that the words holding padovan's pointer, capacity, and length live directly in the stack frame of the print_padovan function; only the vector's buffer is allocated in the heap.

As with the string s earlier, the vector owns the buffer holding its elements. When the variable padovan goes out of scope at the end of the function, the program drops the vector. And since the vector owns its buffer, the buffer goes with it.

Rust's Box type serves as another example of ownership. A Box<T> is a pointer to a value of type T stored in the heap. Calling Box::new(v) allocates some heap space, moves the value v into it, and returns a Box pointing to the heap space. Since a Box owns the space it points to, when the Box is dropped, it frees the space too.

For example, you can allocate a tuple in the heap like so:

```
{
    let point = Box::new((0.625, 0.5));  // point allocated here
    let label = format!("{point:?}");     // label allocated here
    assert_eq!(label, "(0.625, 0.5)");
}                                         // both dropped here
```

When the program calls Box::new, it allocates space for a tuple of two f64 values in the heap, moves its argument (0.625, 0.5) into that space, and returns a pointer to it. By the time control reaches the call to assert_eq!, the stack frame looks like Figure 3-3.

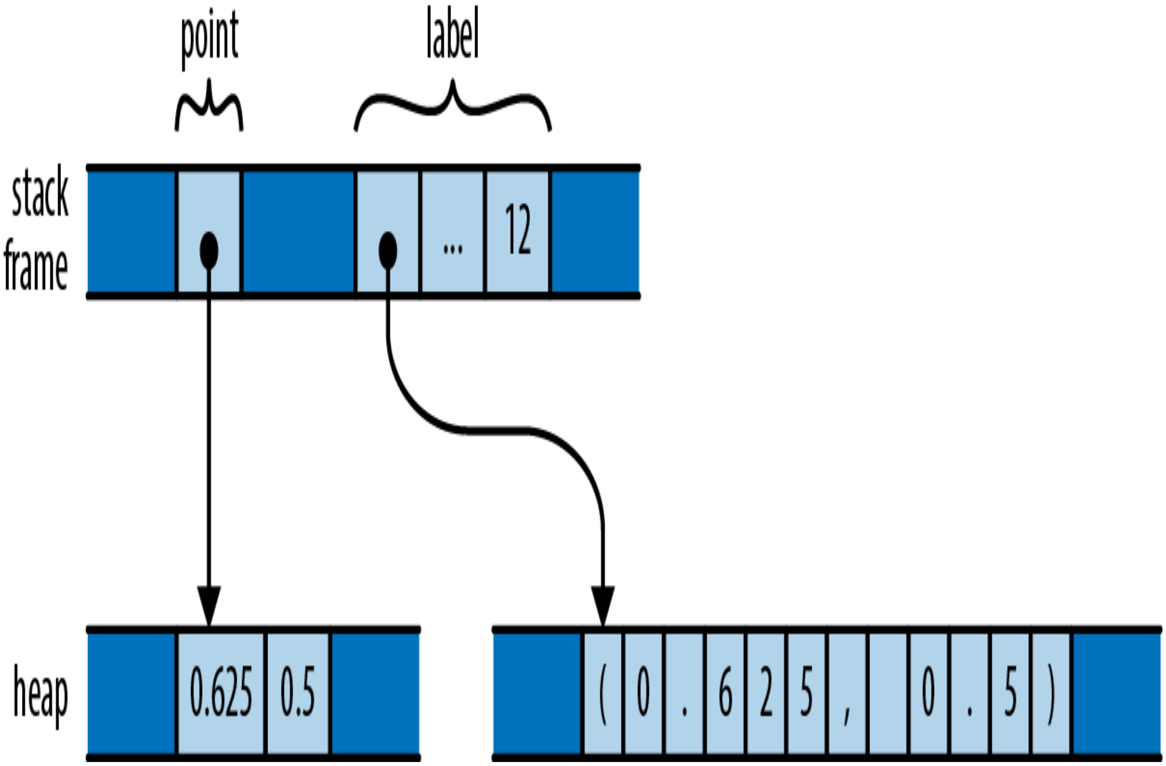


Figure 3-3. Two local variables, each owning memory in the heap

The stack frame itself holds the variables `point` and `label`, each of which refers to a heap allocation that it owns. When they are dropped, the allocations they own are freed along with them.

Just as variables own their values, structs and tuples own their fields, and arrays and vectors own their elements:

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{}, born {}", composer.name, composer.birth);
}
```

Here, `composers` is a `Vec<Person>`, a vector of structs, each of which holds a string and a number. In memory, the final value of `composers` looks like Figure 3-4.

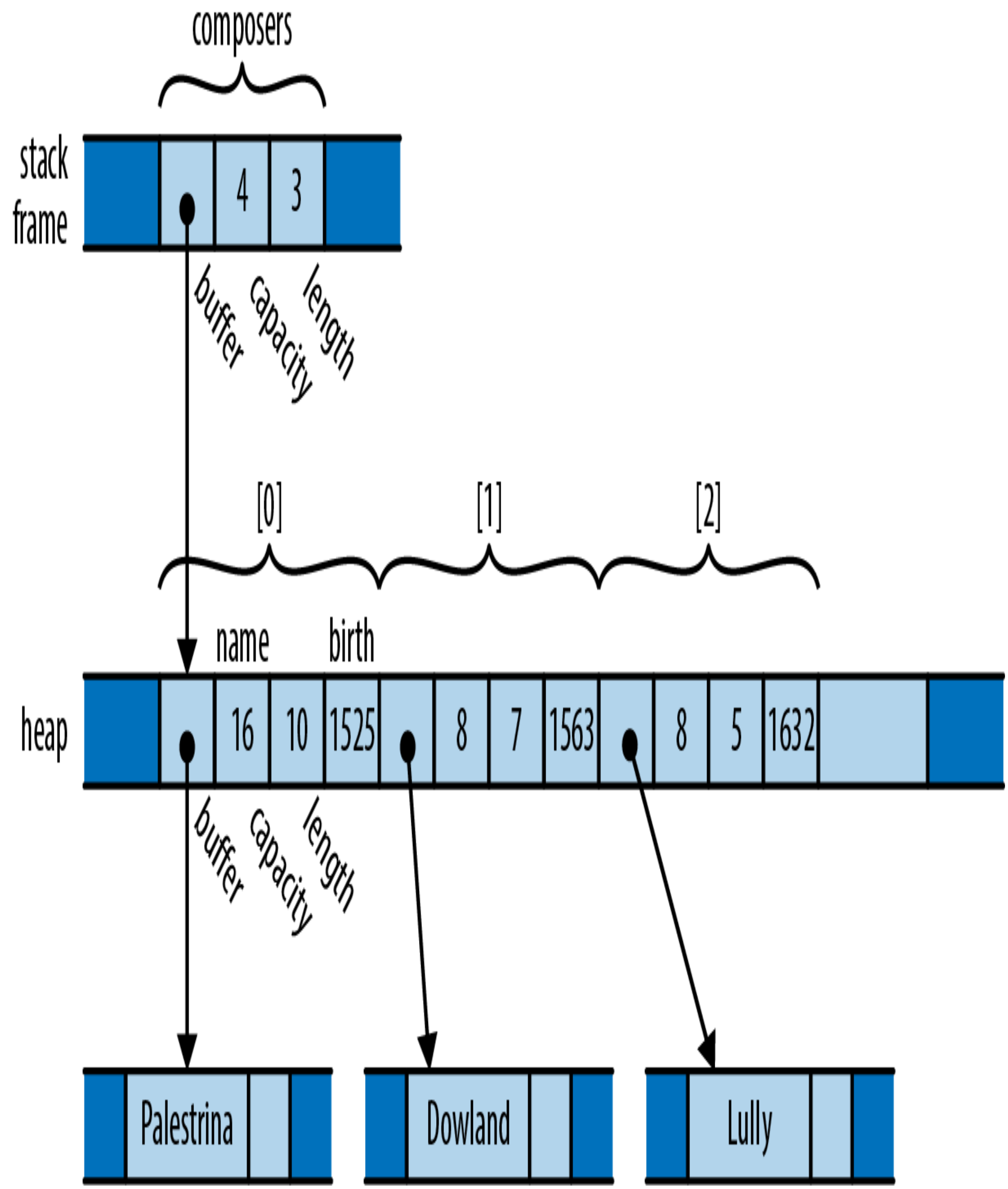


Figure 3-4. A more complex tree of ownership

There are many ownership relationships here, but each one is pretty straightforward: `composers` owns a vector; the vector owns its elements, each of which is a `Person` structure; each structure owns its fields; and the string field owns its text. When control leaves the scope in which `composers` is declared, the program drops its value and takes the entire arrangement with it. If there were other sorts of collections in the picture—a `HashMap`, perhaps, or a `BTreeSet`—the story would be the same.

At this point, take a step back and consider the consequences of the ownership relations we've presented so far. Every value has a single owner, making it easy to decide when to drop it. But a single value may own many other values: for example, the vector `composers` owns all of its elements. And those values may own other values in turn: each element of `composers` owns a string, which owns its text.

It follows that the owners and their owned values form *trees*: your owner is your parent, and the values you own are your children. And at the ultimate root of each tree is a variable; when that variable goes out of scope, the entire tree goes with it. We can see such an ownership tree in the diagram for `composers`: it's not a "tree" in the sense of a search tree data structure, or an HTML document made from DOM elements. Rather, we have a tree built from a mixture of types, with Rust's single-owner rule forbidding any rejoining of structure that could make the arrangement more complex than a tree. Every value in a Rust program is a member of some tree, rooted in some variable.

Rust programs don't usually explicitly drop values at all, in the way C and C++ programs would use `free` and `delete`. The way to drop a value in Rust is to remove it from the ownership tree somehow: by leaving the scope of a variable, or deleting an element from a vector, or something of that sort. At that point, Rust ensures the value is properly dropped, along with everything it owns.

In a certain sense, Rust is less powerful than other languages: every other practical programming language lets you build arbitrary graphs of objects that point to each other in whatever way you see fit. But it is exactly because Rust is less powerful that the analyses the language can carry out on your programs can be more powerful. Rust's safety guarantees are possible exactly because the relationships it may encounter in your code are more tractable. This is part of Rust's "radical wager" we mentioned earlier: in practice, Rust claims, there is usually more than enough flexibility in how one goes about solving a problem to ensure that at least a few perfectly fine solutions fall within the restrictions the language imposes.

That said, concept of ownership as we've explained it so far is too rigid to be useful for even relatively simple programs. Rust's design addresses this inflexibility in a few ways:

- You can move values from one owner to another. This allows you to build, rearrange, and tear down the tree.
- Very simple types like integers, floating-point numbers, and characters are excused from the ownership rules. These are called `Copy` types.
- The standard library provides the reference-counted pointer types `Rc` and `Arc`, which allow values to have multiple owners, under some restrictions.
- You can "borrow a reference" to a value; references are non-owning pointers, with limited lifetimes.

Each of these strategies contributes flexibility to the ownership model, while still upholding Rust's promises. We'll explain each one in turn, with references covered in the next chapter.

Moves

In Rust, for most types, operations like assigning a value to a variable, passing it to a function, or returning it from a function don't copy the value: they *move* it. The source relinquishes ownership of the value to the destination and becomes uninitialized; the destination now controls the value’s lifetime. Rust programs build up and tear down complex structures one value at a time, one move at a time.

You may be surprised that Rust would change the meaning of such fundamental operations; surely assignment is something that should be pretty well nailed down at this point in history. However, if you look closely at how different languages have chosen to handle assignment, you’ll see that there’s actually significant variation from one school to another. The comparison also makes the meaning and consequences of Rust’s choice easier to see.

Consider the following Python code:

```python
s = ['udon', 'ramen', 'soba']
t = s
u = s
```

Each Python object carries a reference count, tracking the number of values that are currently referring to it. So after the assignment to `s`, the state of the program looks like Figure 3-5 (note that some fields are left out).

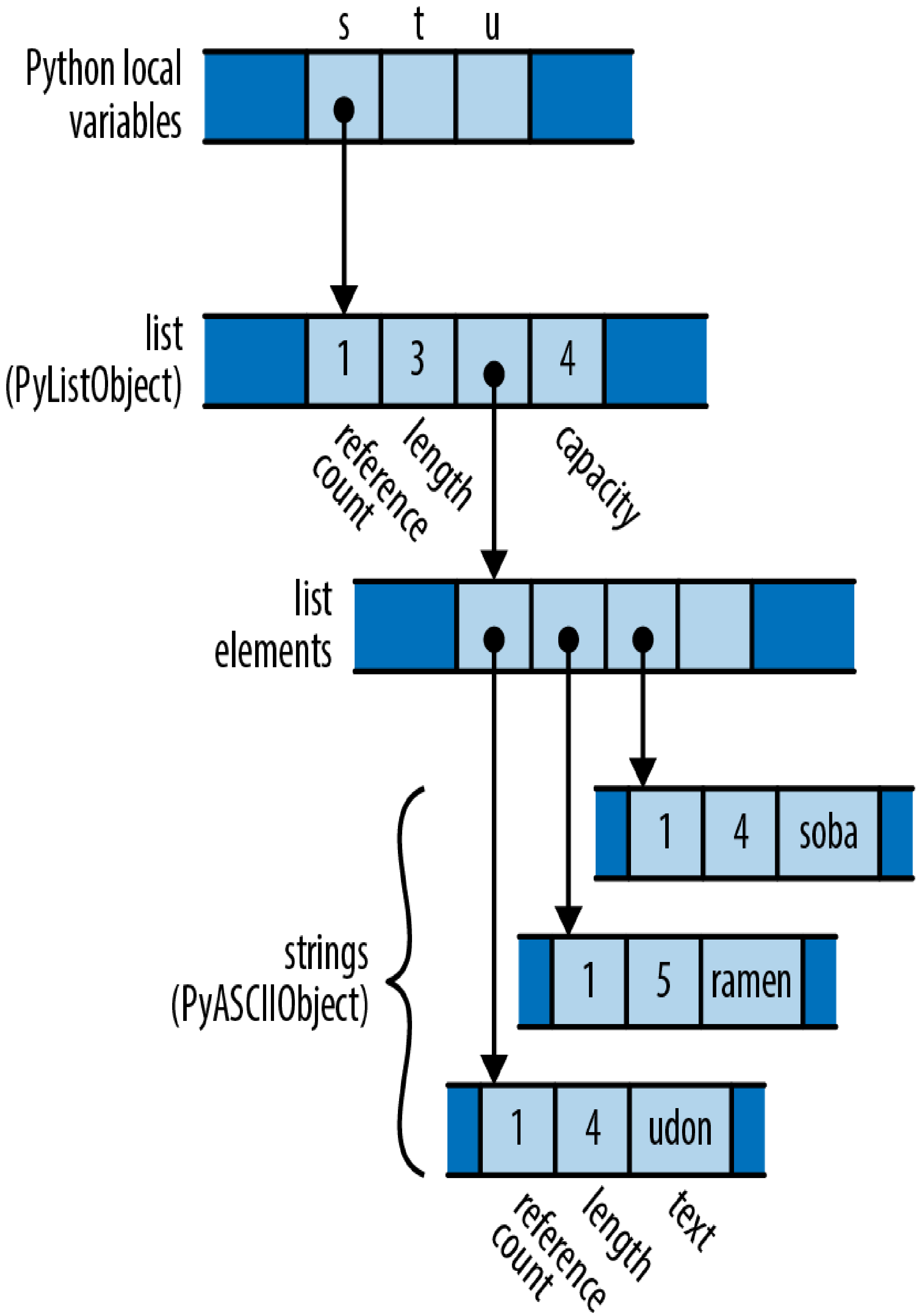

s
t
u
Python local variables
list (PyListObject)
1
3
4
reference count
length
capacity
list elements
strings (PyASCIIObject)
1
4
soba
1
5
ramen
1
4
udon
reference count
length
text

Figure 3-5. How Python represents a list of strings in memory

Since only `s` is pointing to the list, the list's reference count is 1; and since the list is the only object pointing to the strings, each of their reference counts is also 1.

What happens when the program executes the assignments to `t` and `u`? Python implements assignment simply by making the destination point to the same object as the source, and incrementing the object's reference count. So the final state of the program is something like Figure 3-6.

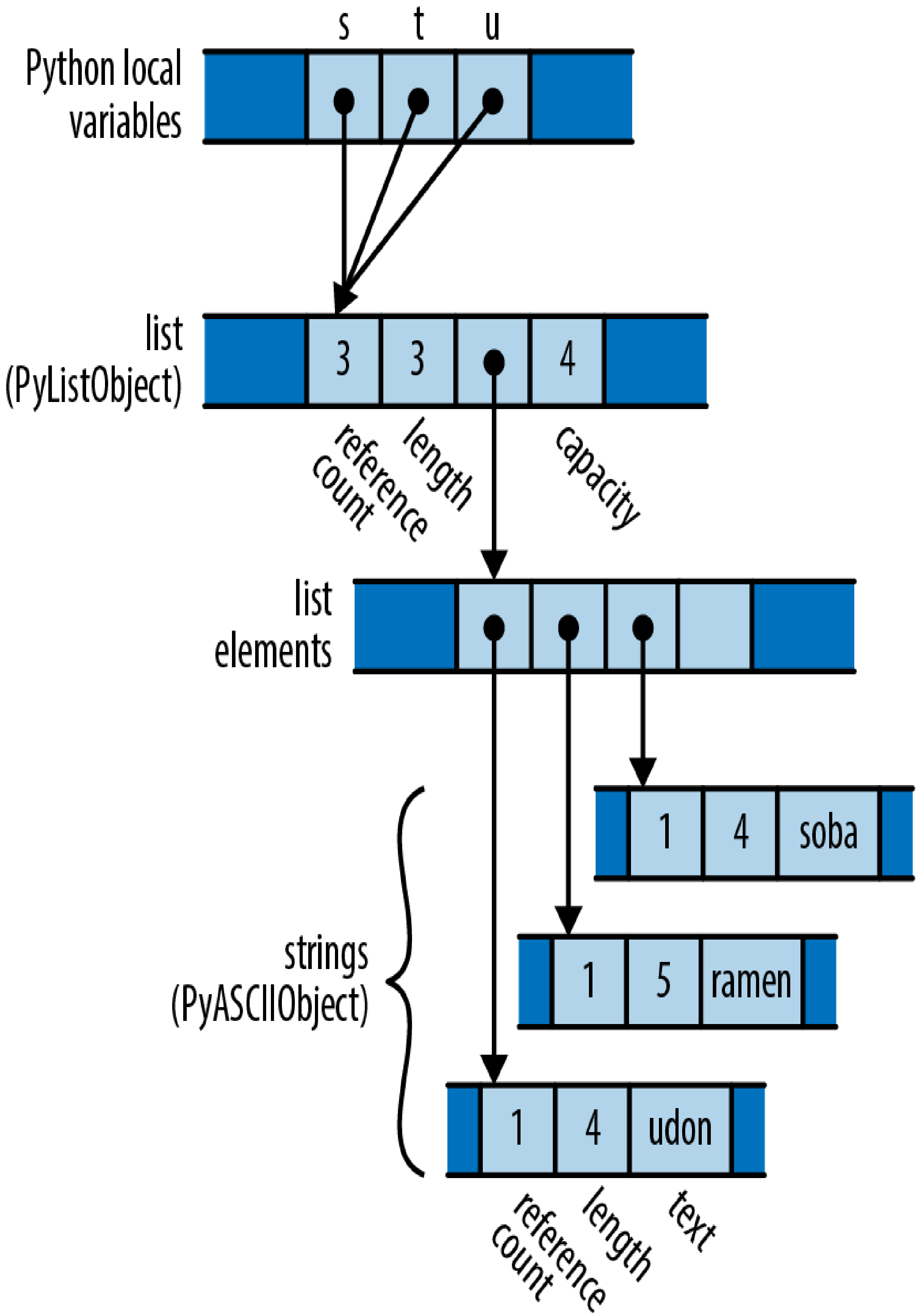

s
t
u
Python local variables
list (PyListObject)
3
3
4
reference count
length
capacity
list elements
1
4
soba
1
5
ramen
1
4
udon
strings (PyASCIIObject)
reference count
length
text

Figure 3-6. The result of assigning `s` to both `t` and `u` in Python

Python has copied the pointer from `s` into `t` and `u` and updated the list's reference count to 3. Assignment in Python is cheap, but because it creates a new reference to the object, we must maintain reference counts to know when we can free the value.

Now consider the analogous C++ code:

```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```

The original value of `s` looks like Figure 3-7 in memory.

What happens when the program assigns `s` to `t` and `u`? Assigning a `std::vector` produces a copy of the vector in C++; `std::string` behaves similarly. So by the time the program reaches the end of this code, it has actually allocated three vectors and nine strings (Figure 3-8).

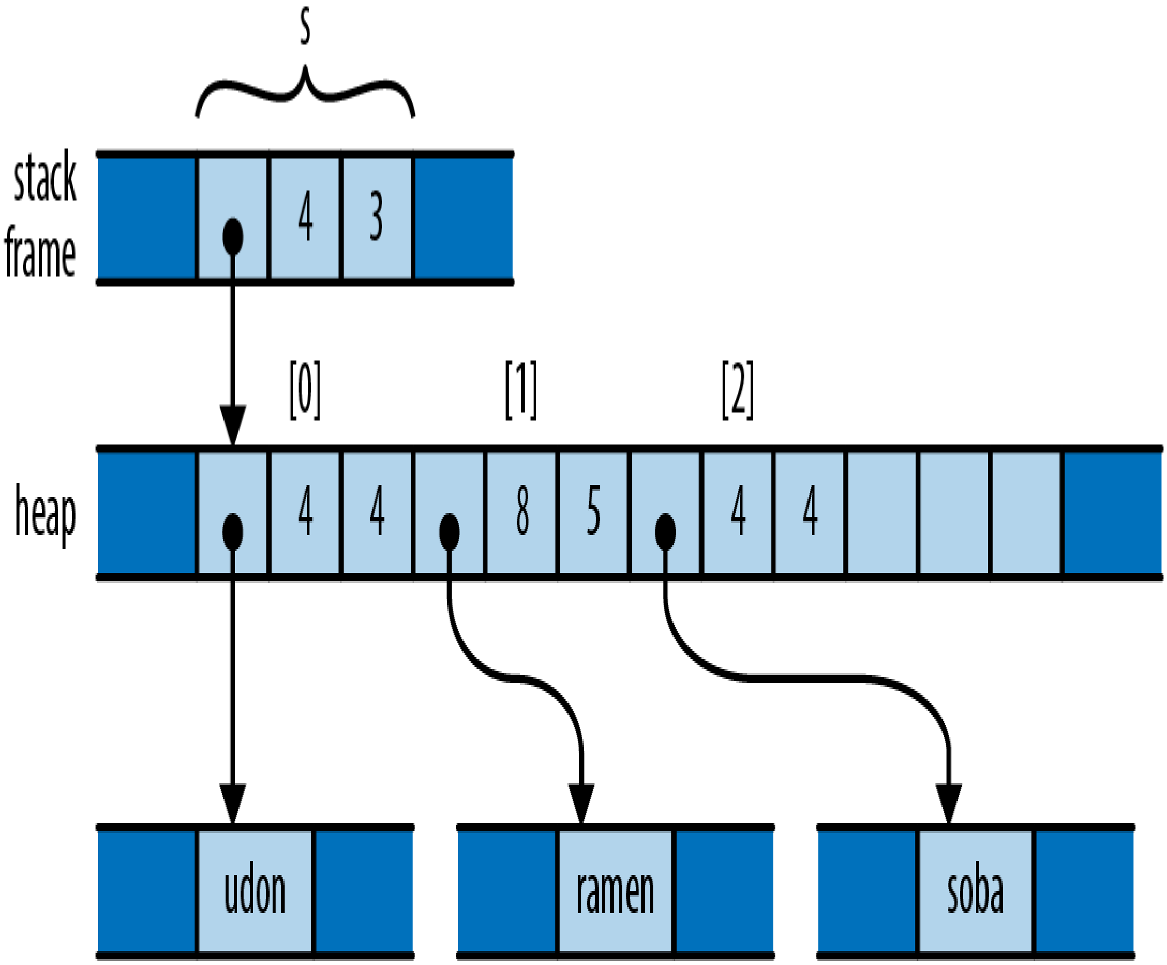


Figure 3-7. How C++ represents a vector of strings in memory

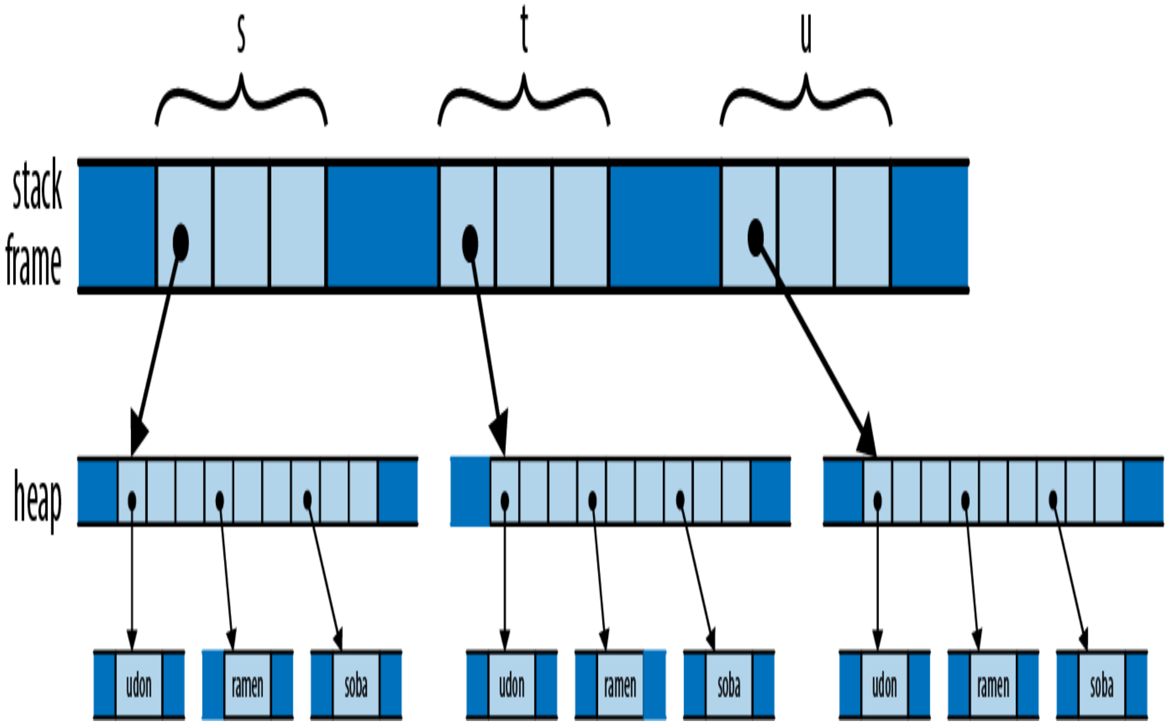


Figure 3-8. The result of assigning `s` *to both* `t` *and* `u` *in C++*

Depending on the values involved, assignment in C++ can consume unbounded amounts of memory and processor time. The advantage, however, is that it’s easy for the program to decide when to free all this memory: when the variables go out of scope, everything allocated here gets cleaned up automatically.

In a sense, C++ and Python have chosen opposite trade-offs: Python makes assignment cheap, at the expense of requiring reference counting (and in the general case, garbage collection). C++ keeps the ownership of all the memory clear, at the expense of making assignment carry out a deep copy of the object. C++ programmers are often less than enthusiastic about this choice: deep copies can be expensive, and there are usually more practical alternatives.

So what would the analogous program do in Rust? Here’s the code:

```
let s = vec!["udon".to_string(), "ramen".to_string(),
"soba".to_string()];
```

```
let t = s;
let u = s;
```

Like C and C++, Rust puts plain string literals like `"udon"` in read-only memory, so for a clearer comparison with the C++ and Python examples, we call `to_string` here to get heap-allocated `String` values.

After carrying out the initialization of `s`, since Rust and C++ use similar representations for vectors and strings, the situation looks just as it did in C++ (Figure 3-9).

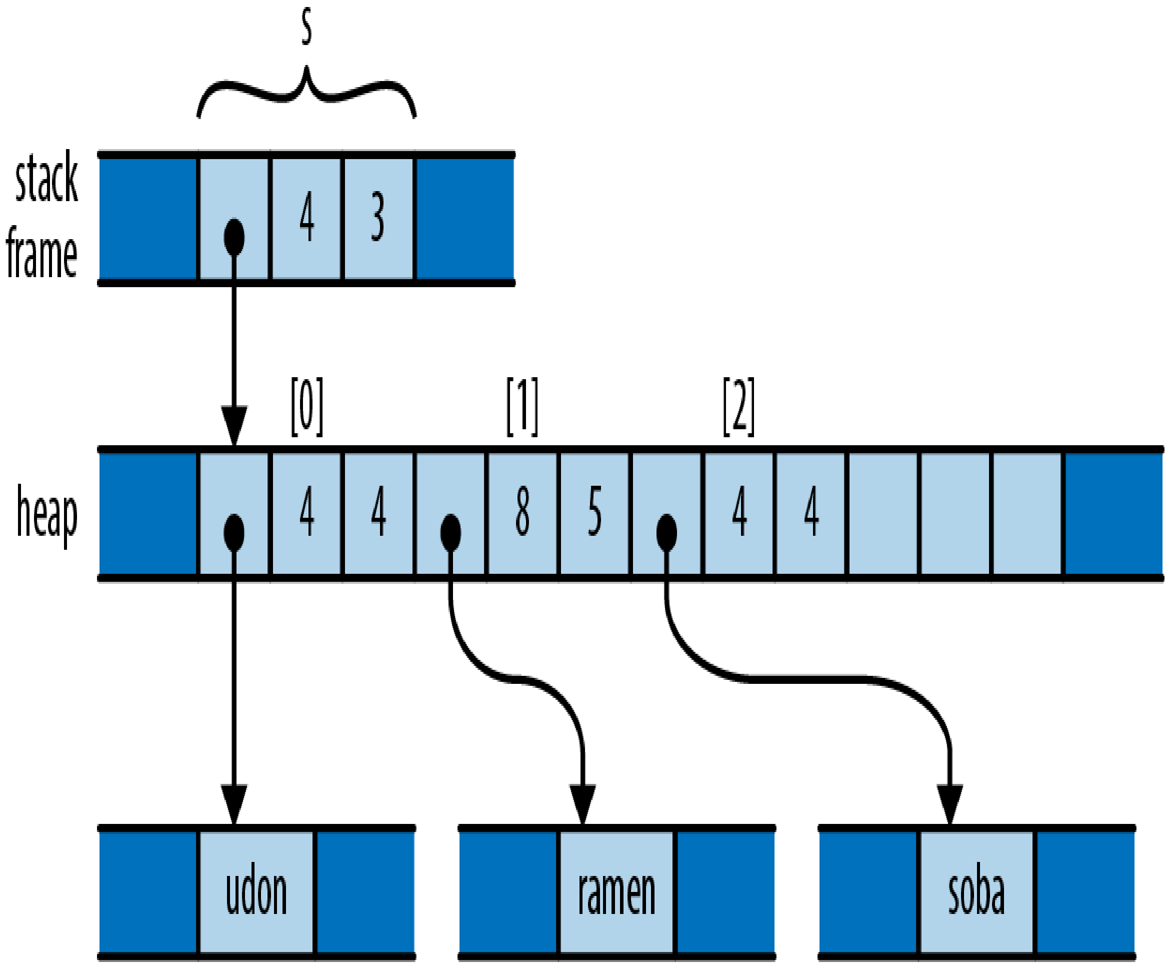


Figure 3-9. How Rust represents a vector of strings in memory

But recall that, in Rust, assignments of most types *move* the value from the source to the destination, leaving the source uninitialized. So after initializing `t`, the program's memory looks like Figure 3-10.

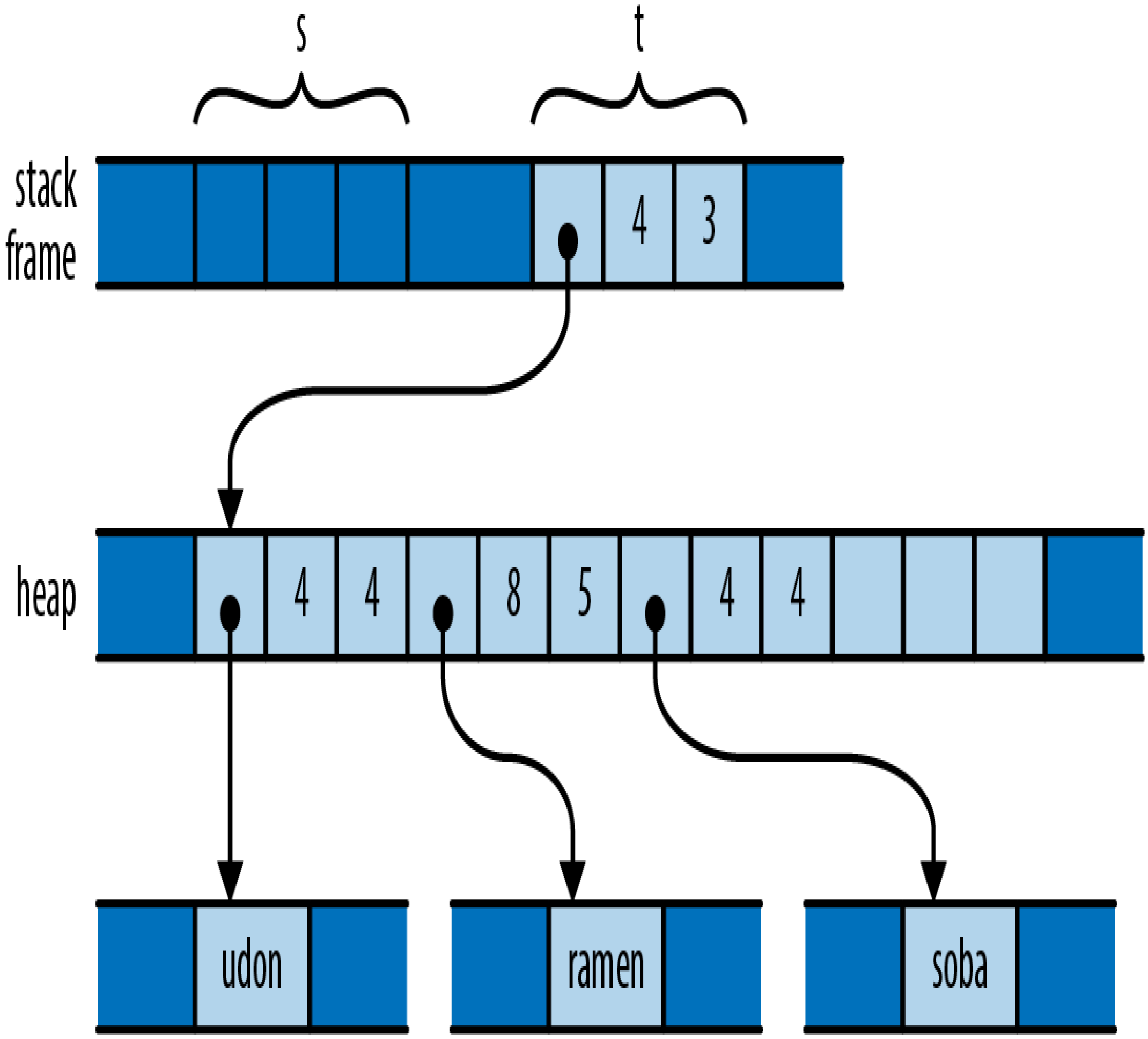


Figure 3-10. The result of assigning `s` to `t` in Rust

What has happened here? The initialization `let t = s;` moved the vector's three header fields from `s` to `t`; now `t` owns the vector. The vector's elements stayed just where they were, and nothing happened to the strings either. Every value still has a single owner, although one has changed hands. There were no reference counts to be adjusted. And the compiler now considers `s` uninitialized.

So what happens when we reach the initialization `let u = s;`? This would assign the uninitialized value `s` to `u`. Rust prudently prohibits using uninitialized values, so the compiler rejects this code with the following error:

```
error: use of moved value: `s`
  |
7 |     let s = vec!["udon".to_string(), "ramen".to_string(),
"soba".to_string()];
  |         - move occurs because `s` has type `Vec<String>`,
  |           which does not implement the `Copy` trait
8 |     let t = s;
  |             - value moved here
9 |     let u = s;
  |             ^ value used here after move
```

Consider the consequences of Rust's use of a move here. Like Python, the assignment is cheap: the program simply moves the three-word header of the vector from one spot to another. But like C++, ownership is always clear: the program doesn't need reference counting or garbage collection to know when to free the vector elements and string contents.

The price you pay is that you must explicitly ask for copies when you want them. If you want to end up in the same state as the C++ program, with each variable holding an independent copy of the structure, you must call the vector's `clone` method, which performs a deep copy of the vector and its elements:

```rust
let s = vec!["udon".to_string(), "ramen".to_string(),
"soba".to_string()];
let t = s.clone();
let u = s.clone();
```

You could also re-create Python's behavior by using Rust's reference-counted pointer types; we'll discuss those shortly in "Rc and Arc: Shared Ownership".

More Operations That Move

In the examples thus far, we've shown initializations, providing values for variables as they come into scope in a `let` statement. Assigning to a variable is slightly different, in that if you move a value into a variable that was already initialized, Rust drops the variable's prior value. For example:

```
let mut s = "Govinda".to_string();
s = "Siddhartha".to_string();  // value "Govinda" dropped here
```

In this code, when the program assigns the string `"Siddhartha"` to `s`, its prior value `"Govinda"` gets dropped first. But consider the following:

```
let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string();  // nothing is dropped here
```

This time, `t` has taken ownership of the original string from `s`, so that by the time we assign to `s`, it is uninitialized. In this scenario, no string is dropped.

We've used initializations and assignments in the examples here because they're simple, but Rust applies move semantics to almost any use of a value. Passing arguments to functions moves ownership to the function's parameters; returning a value from a function moves ownership to the caller. Building a tuple moves the values into the tuple. And so on.

You may now have better insight into what's really going on in the examples we offered in the previous section. For example, when we were constructing our vector of composers, we wrote:

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
```

This code shows several places at which moves occur, beyond initialization and assignment:

Returning values from a function

> The call `Vec::new()` constructs a new vector and returns, not a pointer to the vector, but the vector itself: its ownership moves from `Vec::new` to the variable `composers`. Similarly, the `to_string` call returns a fresh `String` instance.

Constructing new values

> The `name` field of the new `Person` structure is initialized with the return value of `to_string`. The structure takes ownership of the string.

Passing values to a function

> The entire `Person` structure, not a pointer to it, is passed to the vector's `push` method, which moves it onto the end of the vector. The vector takes ownership of the `Person` and thus becomes the indirect owner of the name `String` as well.

Moving values around like this may sound inefficient, but there are two things to keep in mind. First, the moves always apply to the value proper, not the heap storage they own. For vectors and strings, the *value proper* is the three-word header alone; the potentially large element arrays and text buffers sit where they are in the heap. Second, the Rust compiler's code generation is good at "seeing through" all these moves; in practice, the machine code often stores the value directly where it belongs.

Moves and Control Flow

The previous examples all have very simple control flow; how do moves interact with more complicated code? The general principle is that, if it's possible for a variable to have had its value moved away and it hasn't definitely been given a new value since, it's considered uninitialized. For example, if a variable still has a value after evaluating an `if` expression's condition, then we can use it in both branches:

```
let x = vec![10, 20, 30];
if c {
    f(x);  // ... ok to move from x here
} else {
    g(x);  // ... and ok to also move from x here
}
h(x);  // bad: x is uninitialized here if either path uses it
```

For similar reasons, moving from a variable in a loop is forbidden:

```
let x = vec![10, 20, 30];
while f() {
    g(x);  // bad: x would be moved in first iteration,
           // uninitialized in second
}
```

That is, unless we've definitely given it a new value by the next iteration:

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);           // move from x
    x = h();        // give x a fresh value
}
e(x);
```

Moves and Indexed Content

We've mentioned that a move leaves its source uninitialized, as the destination takes ownership of the value. But not every kind of value owner is prepared to become uninitialized. For example, consider the following code:

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101..106 {
    v.push(i.to_string());
}

// Pull out random elements from the vector.
```

```
let third = v[2];  // error: Cannot move out of index of Vec
let fifth = v[4];  // here too
```

For this to work, Rust would somehow need to remember that the third and fifth elements of the vector have become uninitialized, and track that information until the vector is dropped. In the most general case, vectors would need to carry around extra information with them to indicate which elements are live and which have become uninitialized. That is clearly not the right behavior for a systems programming language; a vector should be nothing but a vector. In fact, Rust rejects the preceding code with the following error:

```
error: cannot move out of index of `Vec<String>`
   |
14 |     let third = v[2];
   |                 ^^^^
   |                 |
   |                 move occurs because value has type `String`,
   |                 which does not implement the `Copy` trait
   |                 help: consider borrowing here: `&v[2]`
```

It also makes a similar complaint about the move to `fifth`. In the error message, Rust suggests using a reference, in case you want to access the element without moving it. This is often what you want. But what if you really do want to move an element out of a vector? You need to find a method that does so in a way that respects the limitations of the type. Here are three possibilities:

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101..106 {
    v.push(i.to_string());
}

// 1. Pop a value off the end of the vector:
let fifth = v.pop().expect("vector empty!");
assert_eq!(fifth, "105");
```

```
// 2. Move a value out of a given index in the vector,
// and move the last element into its spot:
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 3. Swap in another value for the one we're taking out:
let third = std::mem::replace(&mut v[2],
"substitute".to_string());
assert_eq!(third, "103");

// Let's see what's left of our vector.
assert_eq!(v, vec!["101", "104", "substitute"]);
```

Each one of these methods moves an element out of the vector, but does so in a way that leaves the vector in a state that is fully populated, if perhaps smaller.

Collection types like `Vec` also generally offer methods to consume all their elements in a loop:

```
let v = vec![
    "liberté".to_string(),
    "égalité".to_string(),
    "fraternité".to_string(),
];

for mut s in v {
    s.push('!');
    println!("{s}");
}
```

When we pass the vector to the loop directly, as in `for ... in v`, this *moves* the vector out of `v`, leaving `v` uninitialized. The `for` loop's internal machinery takes ownership of the vector and dissects it into its elements. At each iteration, the loop moves another element to the variable `s`. Since `s` now owns the string, we're able to modify it in the loop body before printing it. And since the vector itself is no longer visible to the code, nothing can observe it mid-loop in some partially emptied state.

If you do find yourself needing to move a value out of an owner that the compiler can't track, you might consider changing the owner's type to something that can dynamically track whether it has a value or not. For example, here's a variant on the earlier example:

```
struct Person { name: Option<String>, birth: i32 }

let mut composers = Vec::new();
composers.push(Person {
    name: Some("Palestrina".to_string()),
    birth: 1525,
});
```

You can't do this:

```
let first_name = composers[0].name;
```

That will just elicit the same "cannot move out of index" error shown earlier. But because you've changed the type of the `name` field from `String` to `Option<String>`, that means that `None` is a legitimate value for the field to hold, so this works:

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

The `replace` call moves out the value of `composers[0].name`, leaving `None` in its place, and passes ownership of the original value to its caller. In fact, using `Option` this way is common enough that the type provides a `take` method for this very purpose. You could write the preceding manipulation more legibly as follows:

```
let first_name = composers[0].name.take();
```

This call to `take` has the same effect as the earlier call to `replace`.

Copy Types: The Exception to Moves

The examples we’ve shown so far of values being moved involve vectors, strings, and other types that could potentially use a lot of memory and be expensive to copy. Moves keep ownership of such types clear and assignment cheap. But for simpler types like integers or characters, this sort of careful handling really isn’t necessary.

Compare what happens in memory when we assign a `String` with what happens when we assign an `i32` value:

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1: i32 = 36;
let num2 = num1;
```

After running this code, memory looks like Figure 3-11.

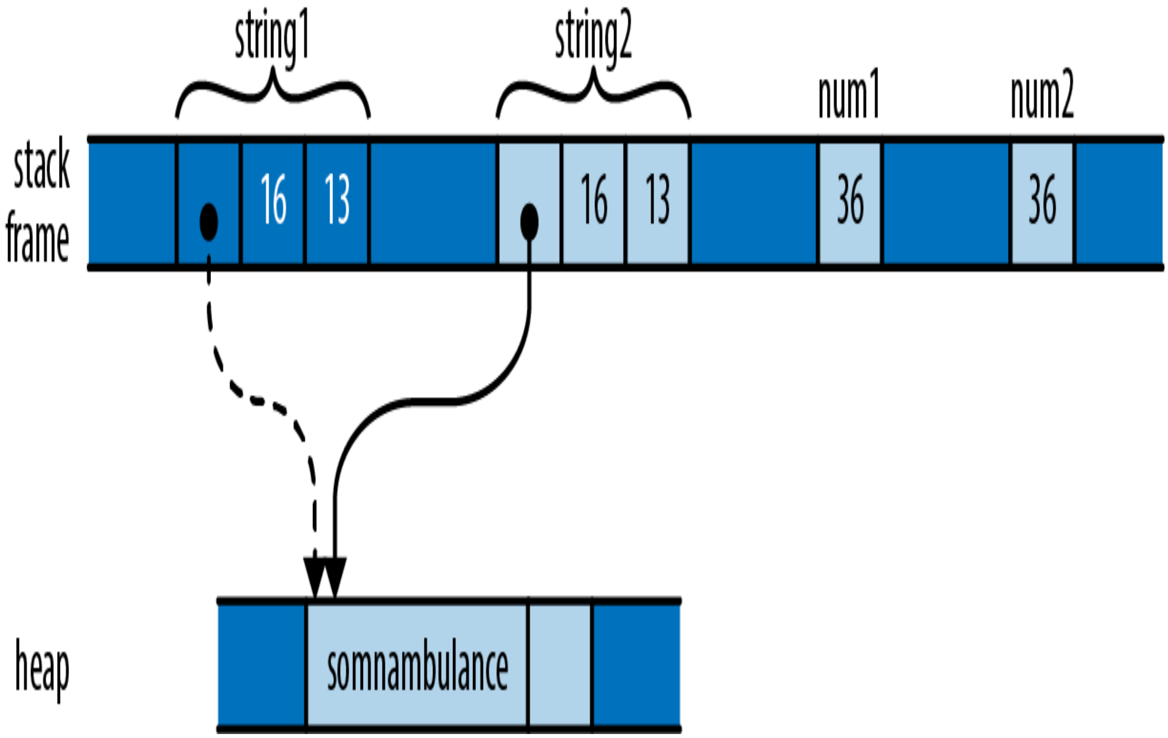


Figure 3-11. Assigning a `String` moves the value, whereas assigning an `i32` copies it

As with the vectors earlier, assignment *moves* `string1` to `string2` so that we don't end up with two strings responsible for freeing the same buffer. However, the situation with `num1` and `num2` is different. An `i32` is simply a pattern of bits in memory; it doesn't own any heap resources or really depend on anything other than the bytes it comprises. By the time we've moved its bits to `num2`, we've made a completely independent copy of `num1`.

Moving a value leaves the source of the move uninitialized. But whereas it serves an essential purpose to treat `string1` as valueless, treating `num1` that way is pointless; no harm could result from continuing to use it. The advantages of a move don't apply here, and it's inconvenient.

Earlier we were careful to say that *most* types are moved; now we've come to the exceptions, the types Rust designates as *`Copy` types*. Assigning a value of a `Copy` type copies the value, rather than moving it. The source of the assignment remains initialized and usable, with the same value it had before. Passing `Copy` types to functions and constructors behaves similarly.

The standard `Copy` types include all the machine integer and floating-point numeric types, the `char` and `bool` types, and a few others. A tuple or fixed-size array of `Copy` types is itself a `Copy` type.

Only types for which a simple bit-for-bit copy suffices can be `Copy`. As we've already explained, `String` is not a `Copy` type, because it owns a heap-allocated buffer. For similar reasons, `Box<T>` is not `Copy`; it owns its heap-allocated referent. The `File` type, representing an operating system file handle, is not `Copy`; duplicating such a value would entail asking the operating system for another file handle. Similarly, the `MutexGuard` type, representing a locked mutex, isn't `Copy`: this type isn't meaningful to copy at all, as only one thread may hold a mutex at a time.

As a rule of thumb, any type that needs to do something special when a value is dropped cannot be `Copy`: a `Vec` needs to free its elements, a `File` needs to close its file handle, a `MutexGuard` needs to unlock its mutex, and so on. Bit-for-bit duplication of such types would leave it unclear which value was now responsible for the original’s resources.

What about types you define yourself? By default, `struct` and `enum` types are not `Copy`:

```
struct Label { number: u32 }

fn print(l: Label) { println!("STAMP: {}", l.number); }

let l = Label { number: 3 };
print(l);
println!("My label number is: {}", l.number);
```

This won’t compile; Rust complains:

```
error: borrow of moved value: `l`
   |
10 |     let l = Label { number: 3 };
   |         - move occurs because `l` has type `main::Label`,
   |           which does not implement the `Copy` trait
11 |     print(l);
   |           - value moved here
12 |     println!("My label number is: {}", l.number);
   |                                        ^^^^^^^^
   |                  value borrowed here after move
```

Since `Label` is not `Copy`, passing it to `print` moved ownership of the value to the `print` function, which then dropped it before returning. But this is silly; a `Label` is nothing but a `u32` with pretensions. There’s no reason passing `l` to `print` should move the value.

But user-defined types being non-`Copy` is only the default. If all the fields of your struct are themselves `Copy`, then you can make the type `Copy` as well by placing the attribute `#[derive(Copy, Clone)]` above the definition, like so:

```
#[derive(Copy, Clone)]
```

```
struct Label { number: u32 }
```

With this change, the preceding code compiles without complaint. However, if we try this on a type whose fields are not all `Copy`, it doesn’t work. Suppose we compile the following code:

```
#[derive(Copy, Clone)]
struct StringLabel { name: String }
```

It elicits this error:

```
error: the trait `Copy` cannot be implemented for this type
  |
7 | #[derive(Copy, Clone)]
  |          ^^^^
8 | struct StringLabel { name: String }
  |                      ------------ this field does not
implement `Copy`
```

Why aren’t user-defined types automatically `Copy`, assuming they’re eligible? Whether a type is `Copy` or not has a big effect on how code is allowed to use it: `Copy` types are more flexible, since assignment and related operations don’t leave the original uninitialized. But for a type’s implementer, the opposite is true: `Copy` types are very limited in which types they can contain, whereas non-`Copy` types can use heap allocation and own other sorts of resources. So making a type `Copy` represents a serious commitment on the part of the implementer: if it’s necessary to change it to non-`Copy` later, much of the code that uses it will probably need to be adapted.

While C++ lets you overload assignment operators and define specialized copy and move constructors, Rust doesn’t permit this sort of customization. In Rust, every move is a byte-for-byte, shallow copy that leaves the source uninitialized. Copies are the same, except that the source remains initialized. This does mean that C++ classes can provide convenient interfaces that Rust types cannot, where ordinary-looking code implicitly adjusts reference counts, puts off expensive copies for later, or uses other sophisticated implementation tricks.

But the effect of this flexibility on C++ as a language is to make basic operations like assignment, passing parameters, and returning values from functions less predictable. For example, earlier in this chapter we showed how assigning one variable to another in C++ can require arbitrary amounts of memory and processor time. One of Rust's principles is that costs should be apparent to the programmer. Basic operations must remain simple. Potentially expensive operations should be explicit, like the calls to `clone` in the earlier example that make deep copies of vectors and the strings they contain.

In this section, we've talked about `Copy` and `Clone` in vague terms as characteristics a type might have. They are actually examples of *traits*, Rust's open-ended facility for categorizing types based on what you can do with them. We describe traits in general in [Link to Come], and `Copy` and `Clone` in particular in [Link to Come].

Rc and Arc: Shared Ownership

Although most values have unique owners in typical Rust code, in some cases it's difficult to find every value a single owner that has the lifetime you need; you'd like the value to simply live until everyone's done using it. For these cases, Rust provides the reference-counted pointer types `Rc` and `Arc`. As you would expect from Rust, these are entirely safe to use: you cannot forget to adjust the reference count, create other pointers to the referent that Rust doesn't notice, or stumble over any of the other sorts of problems that accompany reference-counted pointer types in C++.

The `Rc` and `Arc` types are very similar; the only difference between them is that an `Arc` is safe to share between threads directly—the name `Arc` is short for *atomic reference count*—whereas a plain `Rc` uses faster non-thread-safe code to update its reference count. If you don't need to share the pointers between threads, there's no reason to pay the performance penalty of an Arc, so you should use Rc; Rust will prevent you from accidentally passing one across a thread boundary. The two types are otherwise equivalent, so for the rest of this section, we’ll only talk about Rc.

Earlier we showed how Python uses reference counts to manage its values’ lifetimes. You can use Rc to get a similar effect in Rust. Consider the following code:

```
use std::rc::Rc;

// Rust can infer all these types; written out for clarity
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

For any type T, an Rc<T> value is a pointer to a heap-allocated T that has had a reference count affixed to it. Cloning an Rc<T> value does not copy the T; instead, it simply creates another pointer to it and increments the reference count. So the preceding code produces the situation illustrated in Figure 3-12 in memory.

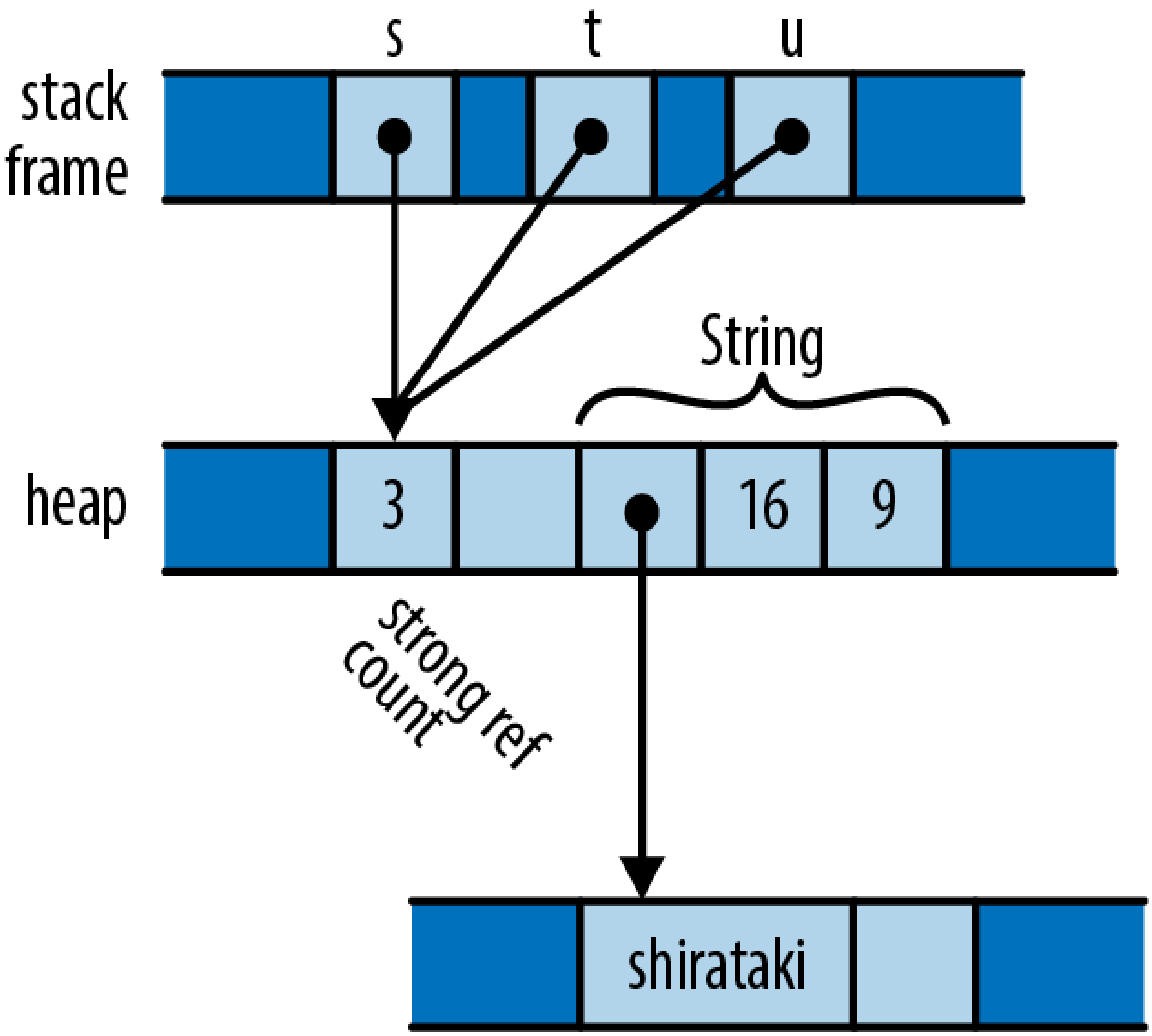


Figure 3-12. A reference-counted string with three references

Each of the three `Rc<String>` pointers is referring to the same block of memory, which holds a reference count and space for the `String`. The usual ownership rules apply to the `Rc` pointers themselves, and when the last extant `Rc` is dropped, Rust drops the `String` as well.

You can use any of `String`'s usual methods directly on an `Rc<String>`:

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{u} are quite chewy, almost bouncy, but lack flavor");
```

A value owned by an Rc pointer is immutable. Suppose you try to add some text to the end of the string:

```
s.push_str(" noodles");
```

Rust will decline:

```
error: cannot borrow data in an `Rc` as mutable
   |
13 |     s.push_str(" noodles");
   |     ^ cannot borrow as mutable
   |
```

Rust's memory and thread-safety guarantees depend on ensuring that no value is ever simultaneously shared and mutable. Rust assumes the referent of an Rc pointer might in general be shared, so it must not be mutable. We explain why this restriction is important in Chapter 4.

One well-known problem with using reference counts to manage memory is that, if there are ever two reference-counted values that point to each other, each will hold the other's reference count above zero, so the values will never be freed (Figure 3-13).

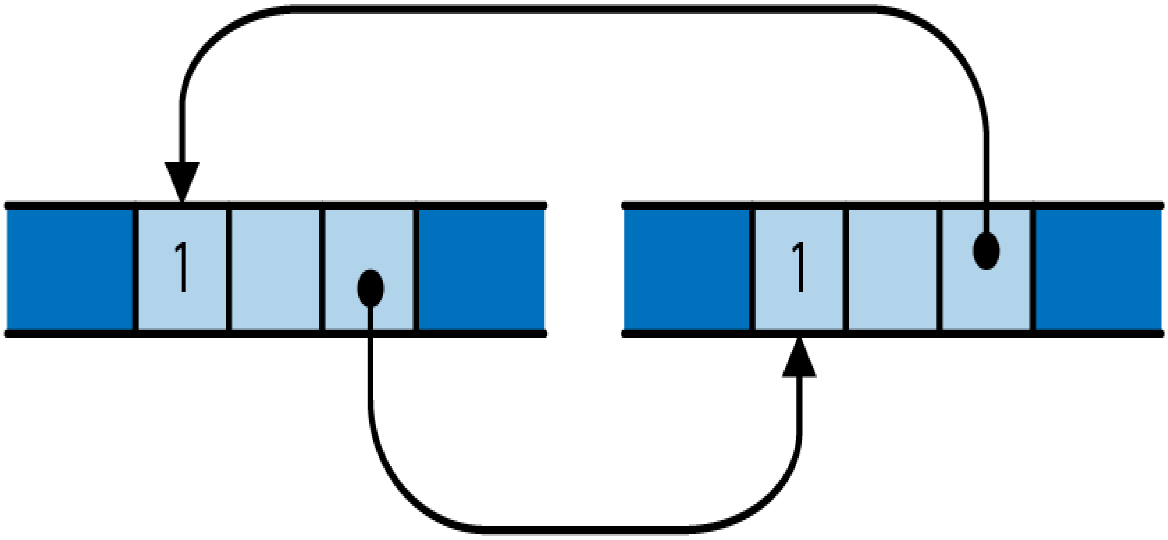


Figure 3-13. A reference-counting loop; these objects will not be freed

It is possible to leak values in Rust this way, but such situations are rare. You cannot create a cycle without, at some point, making an older value point to a newer value. This obviously requires the older value to be mutable. Since `Rc` pointers hold their referents immutable, it's not normally possible to create a cycle. However, Rust does provide ways to create mutable portions of otherwise immutable values; this is called *interior mutability*, and we cover it in [Link to Come]. If you combine those techniques with `Rc` pointers, you can create a cycle and leak memory.

You can sometimes avoid creating cycles of `Rc` pointers by using *weak pointers*, `std::rc::Weak`, for some of the links instead. However, we won't cover those in this book; see the standard library's documentation for details.

Moves and reference-counted pointers are two ways to relax the rigidity of the ownership tree. In the next chapter, we'll look at a third way: borrowing references to values. Once you have become comfortable with both ownership and borrowing, you will have climbed the steepest part of Rust's learning curve, and you'll be ready to take advantage of Rust's unique strengths.

Chapter 4. References

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

Libraries cannot provide new inabilities.

—Mark Miller

All the pointer types we discussed in the last chapter—the simple `Box<T>` heap pointer, and the pointers internal to `String` and `Vec` values—are owning pointers: dropping a `Box<T>` drops the `T` as well. Rust also has non-owning pointer types called *references*, which have no effect on their referents’ lifetimes.

In fact, it’s rather the opposite: references must never outlive their referents. You must make it apparent in your code that no reference can possibly outlive the value it points to. To emphasize this, Rust refers to creating a reference to some value as *borrowing* the value: what you have borrowed, you must eventually return to its owner.

If you felt a moment of skepticism when reading the phrase “You must make it apparent in your code,” you’re in excellent company. The references themselves are nothing special—under the hood, they’re just addresses. But the rules that keep them safe are novel to Rust; outside of research languages, you won't have seen anything like them before. And although these rules are the part of Rust that requires the most effort to master, the breadth of classic, absolutely everyday bugs they prevent is surprising, and their effect on multithreaded programming is liberating. This is Rust's radical wager, again.

In this chapter, we'll walk through how references work in Rust; show how references, functions, and user-defined types all incorporate lifetime information to ensure that they're used safely; and illustrate some common categories of bugs that these efforts prevent, at compile time and without run-time performance penalties.

References to Values

As an example, let's suppose we're going to build a table of murderous Renaissance artists and the works they're known for. Rust's standard library includes a hash table type, so we can define our type like this:

```
use std::collections::HashMap;

type Table = HashMap<String, Vec<String>>;
```

In other words, this is a hash table that maps `String` values to `Vec<String>` values, taking the name of an artist to a list of the names of their works. You can iterate over the entries of a `HashMap` with a `for` loop, so we can write a function to print out a `Table`:

```
fn show(table: Table) {
    for (artist, works) in table {
        println!("works by {artist}:");
        for work in works {
            println!("  {work}");
        }
    }
}
```

Constructing and printing the table is straightforward:

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                 vec!["many madrigals".to_string(),
                      "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                 vec!["The Musicians".to_string(),
                      "The Calling of St. Matthew".to_string()]);
    table.insert("Cellini".to_string(),
                 vec!["Perseus with the head of
Medusa".to_string(),
                      "a salt cellar".to_string()]);

    show(table);
}
```

And it all works fine:

```
$ cargo run
     Running
`/home/jimb/rust/book/fragments/target/debug/fragments`
works by Gesualdo:
  many madrigals
  Tenebrae Responsoria
works by Cellini:
  Perseus with the head of Medusa
  a salt cellar
works by Caravaggio:
  The Musicians
  The Calling of St. Matthew
$
```

But if you've read the previous chapter's section on moves, this definition for `show` should raise a few questions. In particular, `HashMap` is not `Copy`—it can't be, since it owns a dynamically allocated table. So when the program calls `show(table)`, the whole structure gets moved to the function, leaving the variable `table` uninitialized. (It also iterates over its contents in no specific order, so if you’ve gotten a different order, don’t worry.) If the calling code tries to use `table` now, it’ll run into trouble:

```
...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

Rust complains that `table` isn’t available anymore:

```
error: borrow of moved value: `table`
   |
20 |     let mut table = Table::new();
   |         --------- move occurs because `table` has type
   |                   `HashMap<String, Vec<String>>`,
   |                   which does not implement the `Copy` trait
...
31 |     show(table);
   |          ----- value moved here
32 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
   |                ^^^^^ value borrowed here after move
note: consider changing this parameter type in function `show` to
borrow instead
      if owning the value isn't necessary
   |
7  | fn show(table: Table) {
   |    ----        ^^^^^ this parameter takes ownership of the
value
   |    |
   |    in this function
help: consider cloning the value if the performance cost is
acceptable
   |
35 |     show(table.clone());
   |               ++++++++
```

In fact, if we look into the definition of `show`, the outer `for` loop takes ownership of the hash table and consumes it entirely; and the inner `for` loop does the same to each of the vectors. (We saw this behavior earlier, in the “liberté, égalité, fraternité” example.) Because of move semantics, we’ve completely destroyed the entire structure simply by trying to print it out. Thanks, Rust!

The compiler helpfully suggests that we could either create a reference or clone the value to avoid this problem. In this case, the right solution is to use references, which let you access a value without affecting its ownership. References come in two kinds:

- A *shared reference* lets you read but not modify its referent. However, you can have as many shared references to a particular value at a time as you like. The expression `&e` yields a shared reference to `e`’s value; if `e` has the type `T`, then `&e` has the type `&T`, pronounced “ref `T`.” Shared references are `Copy`.
- If you have a *mutable reference* to a value, you may both read and modify the value. However, you may not have any other references of any sort to that value active at the same time. The expression `&mut e` yields a mutable reference to `e`’s value; you write its type as `&mut T`, which is pronounced “ref mute `T`.” Mutable references are not `Copy`.

You can think of the distinction between shared and mutable references as a way to enforce a *multiple readers or single writer* rule at compile time. In fact, this rule doesn’t apply only to references; it covers the borrowed value’s owner as well. As long as there are shared references to a value, not even its owner can modify it; the value is locked down. Nobody can modify `table` while `show` is working with it. Similarly, if there is a mutable reference to a value, it has exclusive access to the value; you can’t use the owner at all, until the mutable reference goes away. Keeping sharing and mutation fully separate turns out to be essential to memory safety, for reasons we’ll go into later in the chapter.

The printing function in our example doesn’t need to modify the table, just read its contents. So the caller should be able to pass it a shared reference to the table, as follows:

```
show(&table);
```

References are non-owning pointers, so the `table` variable remains the owner of the entire structure; `show` has just borrowed it for a bit. Naturally, we'll need to adjust the definition of `show` to match, but you'll have to look closely to see the difference:

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {artist}:");
        for work in works {
            println!("  {work}");
        }
    }
}
```

The type of `show`'s parameter `table` has changed from `Table` to `&Table`: instead of passing the table by value (and hence moving ownership into the function), we're now passing a shared reference. That's the only textual change. But how does this play out as we work through the body?

Whereas our original outer `for` loop took ownership of the `HashMap` and consumed it, in our new version it receives a shared reference to the `HashMap`. Iterating over a shared reference to a `HashMap` is defined to produce shared references to each entry's key and value: `artist` has changed from a `String` to a `&String`, and `works` from a `Vec<String>` to a `&Vec<String>`.

The inner loop is changed similarly. Iterating over a shared reference to a vector is defined to produce shared references to its elements, so `work` is now a `&String`. No ownership changes hands anywhere in this function; it's just passing around non-owning references.

Now, if we wanted to write a function to alphabetize the works of each artist, a shared reference doesn't suffice, since shared references don't permit modification. Instead, the sorting function needs to take a mutable reference to the table:

```
fn sort_works(table: &mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```

And we need to pass it one:

```
sort_works(&mut table);
```

This mutable borrow grants `sort_works` the ability to read and modify our structure, as required by the vectors' `sort` method.

When we pass a value to a function in a way that moves ownership of the value to the function, we say that we have passed it *by value*. If we instead pass the function a reference to the value, we say that we have passed the value *by reference*. For example, we fixed our `show` function by changing it to accept the table by reference, rather than by value. Many languages draw this distinction, but it's especially important in Rust, because it spells out how ownership is affected.

Working with References

The preceding example shows a pretty typical use for references: allowing functions to access or manipulate a structure without taking ownership. But references are more flexible than that, so let's look at some examples to get a more detailed view of what's going on.

Rust References Versus C++ References

If you're familiar with references in C++, they do have something in common with Rust references. Most importantly, they're both just addresses at the machine level. But in practice, Rust’s references have a very different feel.

In C++, references are created implicitly by conversion, and dereferenced implicitly too:

```
// C++ code!
int x = 10;
int &r = x;             // initialization creates reference
implicitly
assert(r == 10);        // implicitly dereference r to see x's
value
r = 20;                 // stores 20 in x, r itself still points
to x
```

In Rust, references are created explicitly with the `&` operator, and dereferenced explicitly with the `*` operator:

```
// Back to Rust code from this point onward.
let x = 10;
let r = &x;             // &x is a shared reference to x
assert_eq!(*r, 10);     // explicitly dereference r
```

To create a mutable reference, use the `&mut` operator:

```
let mut y = 32;
let m = &mut y;        // &mut y is a mutable reference to y
*m += 32;              // explicitly dereference m to set y's
value
assert_eq!(*m, 64);    // or to get y's current value
```

But you might recall that, when we fixed the `show` function to take the table of artists by reference instead of by value, we never had to use the `*` operator. Why is that?

Since references are so widely used in Rust, the `.` operator implicitly dereferences its left operand, if needed:

```
struct Anime { name: &'static str, bechdel_pass: bool }
let aria = Anime { name: "Aria: The Animation", bechdel_pass:
true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// Equivalent to the above, but with the dereference written out:
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

The `println!` macro used in the `show` function expands to code that uses the `.` operator, so it takes advantage of this implicit dereference as well.

The `.` operator can also implicitly borrow a reference to its left operand, if needed for a method call. For example, `Vec`'s `sort` method takes a mutable reference to the vector, so these two calls are equivalent:

```
let mut v = vec![1973, 1968];
v.sort();           // implicitly borrows a mutable reference to
v
(&mut v).sort();    // equivalent, but more verbose
```

In a nutshell, whereas C++ converts implicitly between references and lvalues (that is, expressions referring to locations in memory), with these conversions appearing anywhere they're needed, in Rust you use the `&` and `*` operators to create and follow references, with the exception of the `.` operator, which borrows and dereferences implicitly.

Assigning References

Assigning a reference to a variable makes that variable point somewhere new:

```
let x = 10;
let y = 20;
let mut r = &x;
```

```
if b { r = &y; }

assert!(*r == 10 || *r == 20);
```

The reference `r` initially points to `x`. But if `b` is true, the code points it at `y` instead, as illustrated in Figure 4-1.

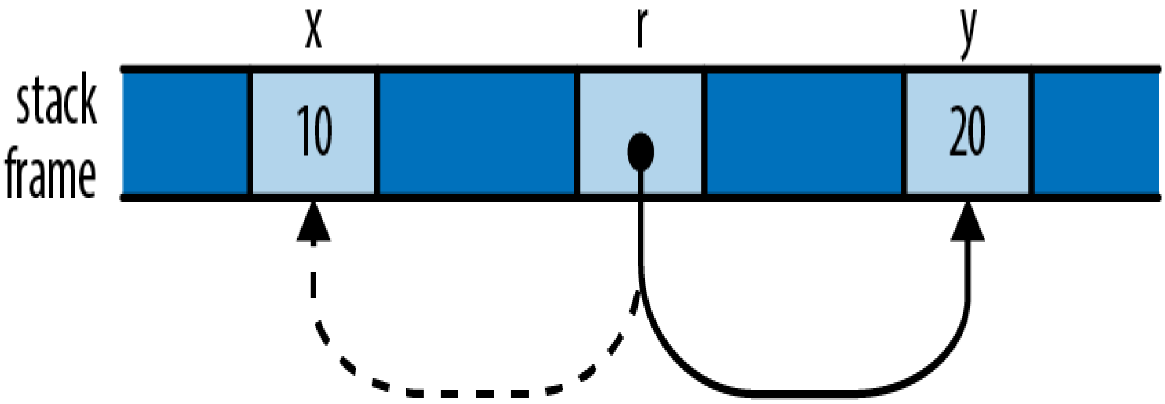


Figure 4-1. The reference `r`, now pointing to `y` instead of `x`

This behavior may seem too obvious to be worth mentioning: of course `r` now points to `y`, since we stored `&y` in it. But we point this out because C++ references behave very differently: as shown earlier, assigning a value to a reference in C++ stores the value in its referent. Once a C++ reference has been initialized, there's no way to make it point at anything else.

References to References

Rust permits references to references:

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr: &&&Point = &rr;
```

(We've written out the reference types for clarity, but you could omit them; there's nothing here Rust can't infer for itself.) The `.` operator follows as many references as it takes to find its target:

```
assert_eq!(rrr.y, 729);
```

In memory, the references are arranged as shown in Figure 4-2.

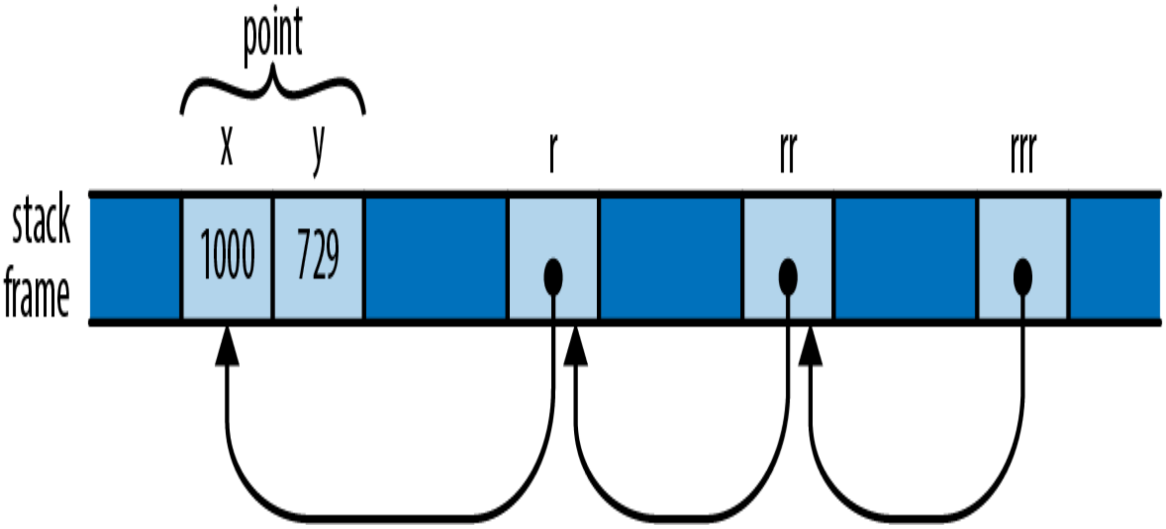


Figure 4-2. A chain of references to references

So the expression `rrr.y`, guided by the type of `rrr`, actually traverses three references to get to the `Point` before fetching its `y` field.

Comparing References

Like the `.` operator, Rust's comparison operators "see through" any number of references:

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

let rx = &x;
let ry = &y;

let rrx = &rx;
let rry = &ry;

assert!(rrx <= rry);
assert!(rrx == rry);
```

The final assertion here succeeds, even though `rrx` and `rry` point at different values (namely, `rx` and `ry`), because the == operator follows all the references and performs the comparison on their final targets, `x` and `y`. This is almost always the behavior you want, especially when writing generic functions. If you actually want to know whether two references point to the same memory, you can use `std::ptr::eq`, which compares them as addresses:

```
assert!(rx == ry);               // their referents are equal
assert!(!std::ptr::eq(rx, ry));  // but occupy different
addresses
```

Note that the operands of a comparison must have exactly the same type, including the references:

```
assert!(rx == rrx);    // error: type mismatch: `&i32` vs `&&i32`
assert!(rx == *rrx);   // this is okay
```

References Are Never Null

Rust references are never null. There's no analogue to C's `NULL` or C++'s `nullptr`. There is no default initial value for a reference (you can't use any variable until it's been initialized, regardless of its type) and Rust won't convert integers to references (outside of `unsafe` code), so you can't convert zero into a reference.

C and C++ code often uses a null pointer to indicate the absence of a value: for example, the `malloc` function returns either a pointer to a new block of memory or `nullptr` if there isn't enough memory available to satisfy the request. In Rust, if you need a value that is either a reference to something or not, use the type `Option<&T>`. At the machine level, Rust represents `None` as a null pointer and `Some(r)`, where `r` is a `&T` value, as the nonzero address, so `Option<&T>` is just as efficient as a nullable pointer in C or C++, even though it's safer: its type requires you to check whether it's None before you can use it.

Borrowing References to Arbitrary Expressions

Whereas C and C++ only let you apply the & operator to certain kinds of expressions, Rust lets you borrow a reference to the value of any sort of expression at all:

```
fn factorial(n: usize) -> usize {
    (1..n+1).product()
}
let r = &factorial(6);
// Arithmetic operators can see through one level of references.
assert_eq!(r + &1009, 1729);
```

In situations like this, Rust simply creates an anonymous variable to hold the expression's value and makes the reference point to that. The lifetime of this anonymous variable depends on what you do with the reference:

- If you immediately assign the reference to a variable in a let statement (or make it part of some struct or array that is being immediately assigned), then Rust makes the anonymous variable live as long as the variable the let initializes. In the preceding example, Rust would do this for the referent of r.

- Otherwise, the anonymous variable lives to the end of the enclosing statement. In our example, the anonymous variable created to hold 1009 lasts only to the end of the assert_eq! statement.

If you're used to C or C++, this may sound error-prone. But remember that Rust will never let you write code that would produce a dangling reference. If the reference could ever be used beyond the anonymous variable's lifetime, Rust will always report the problem to you at compile time. You can then fix your code to keep the referent in a named variable with an appropriate lifetime.

References to Slices and Trait Objects

The references we've shown so far are all simple addresses. However, Rust also includes two kinds of *fat pointers*, two-word values carrying the address of some value, along with some further information necessary to put the value to use.

A reference to a slice is a fat pointer, carrying the starting address of the slice and its length. We described slices in detail in Chapter 2.

Rust's other kind of fat pointer is a *trait object*, a reference to a value that implements a certain trait. A trait object carries a value's address and a pointer to the trait's implementation appropriate to that value, for invoking the trait's methods. We'll cover trait objects in detail in [Link to Come].

Aside from carrying this extra data, slice and trait object references behave just like the other sorts of references we've shown so far in this chapter: they don't own their referents, they are not allowed to outlive their referents, they may be mutable or shared, and so on.

Reference Safety

As we've presented them so far, references look pretty much like ordinary pointers in C or C++. But those are unsafe; how does Rust keep its references under control? Perhaps the best way to see the rules in action is to try to break them.

To convey the fundamental ideas, we'll start with the simplest cases, showing how Rust ensures references are used properly within a single function body. Then we'll look at passing references between functions and storing them in data structures. This entails giving said functions and data types *lifetime parameters*, which we'll explain. Finally, we'll present some shortcuts that Rust provides to simplify common usage patterns.

Throughout, we’ll be showing how Rust points out broken code and often suggests solutions.

Borrowing a Local Variable

Here’s a pretty obvious case. You can’t borrow a reference to a local variable and take it out of the variable’s scope:

```
{
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1);  // bad: reads memory `x` used to occupy
}
```

The Rust compiler rejects this program, with a detailed error message:

```
error: `x` does not live long enough
   |
7  |         r = &x;
   |             ^^ borrowed value does not live long enough
8  |     }
   |     - `x` dropped here while still borrowed
9  |     assert_eq!(*r, 1);  // bad: reads memory `x` used to
occupy
10 | }
```

Rust’s complaint is that `x` lives only until the end of the inner block, whereas the reference remains alive until the end of the outer block, making it a dangling pointer, which is verboten.

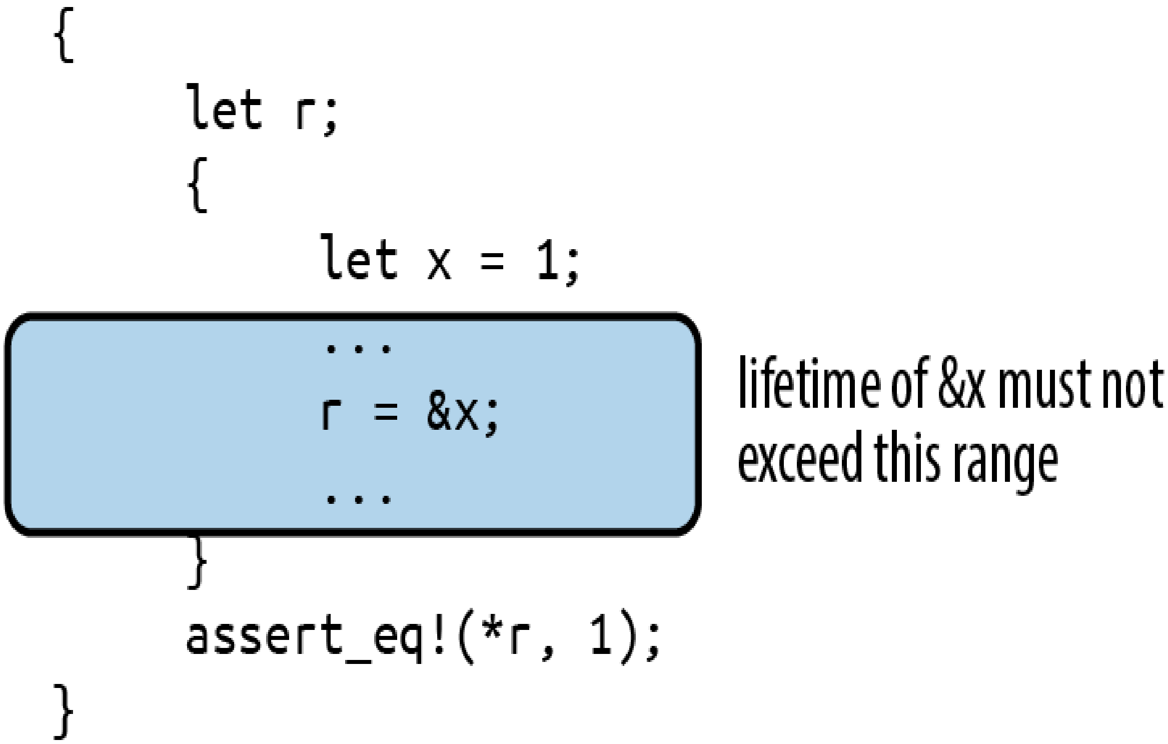
While it’s obvious to a human reader that this program is broken, it’s worth looking at how Rust itself reached that conclusion. Even this simple example shows the logical tools Rust uses to check much more complex code.

Rust tries to assign each reference type in your program a *lifetime* that meets the constraints imposed by how it is used. A lifetime is some stretch of your program for which a reference could be safe to use: a statement, an expression, the scope of some variable, or the like. Lifetimes are entirely figments of Rust's compile-time imagination. At run time, a reference is nothing but an address; its lifetime is part of its type and has no run-time representation.

In this example, there are three lifetimes whose relationships we need to work out. The variables `r` and `x` both have a lifetime, extending from the point at which they're initialized until the point that the compiler can prove they are no longer in use. The third lifetime is that of a reference type: the type of the reference we borrow to `x` and store in `r`.

Here's one constraint that should seem pretty obvious: if you have a variable `x`, then a reference to `x` must not outlive `x` itself, as shown in Figure 4-3.

Beyond the point where `x` goes out of scope, the reference would be a dangling pointer. We say that the variable's lifetime must *contain* or *enclose* that of the reference borrowed from it.

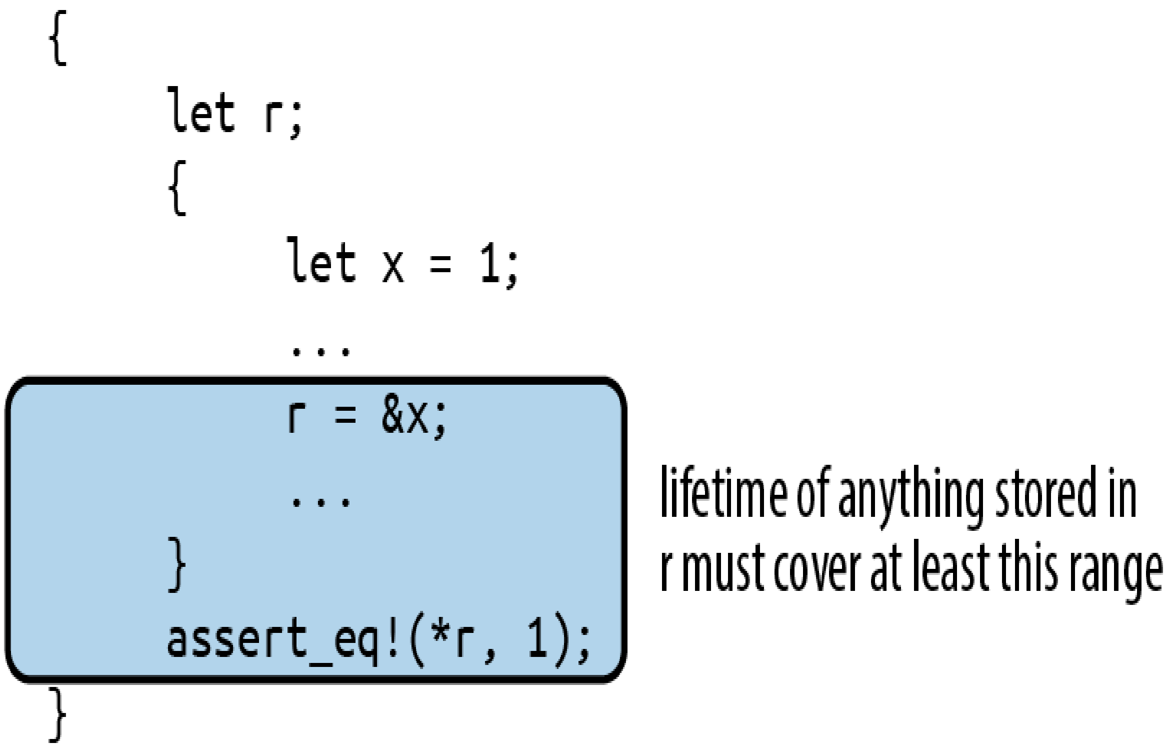


```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```

Figure 4-3. Permissible lifetimes for &x

Here's another kind of constraint: if you store a reference in a variable `r`, the reference's type must be good for the entire lifetime of the variable, from its initialization until its last use, as shown in Figure 4-4.

If the reference can't live at least as long as the variable does, then at some point `r` will be a dangling pointer. We say that the reference's lifetime must contain or enclose the variable's.

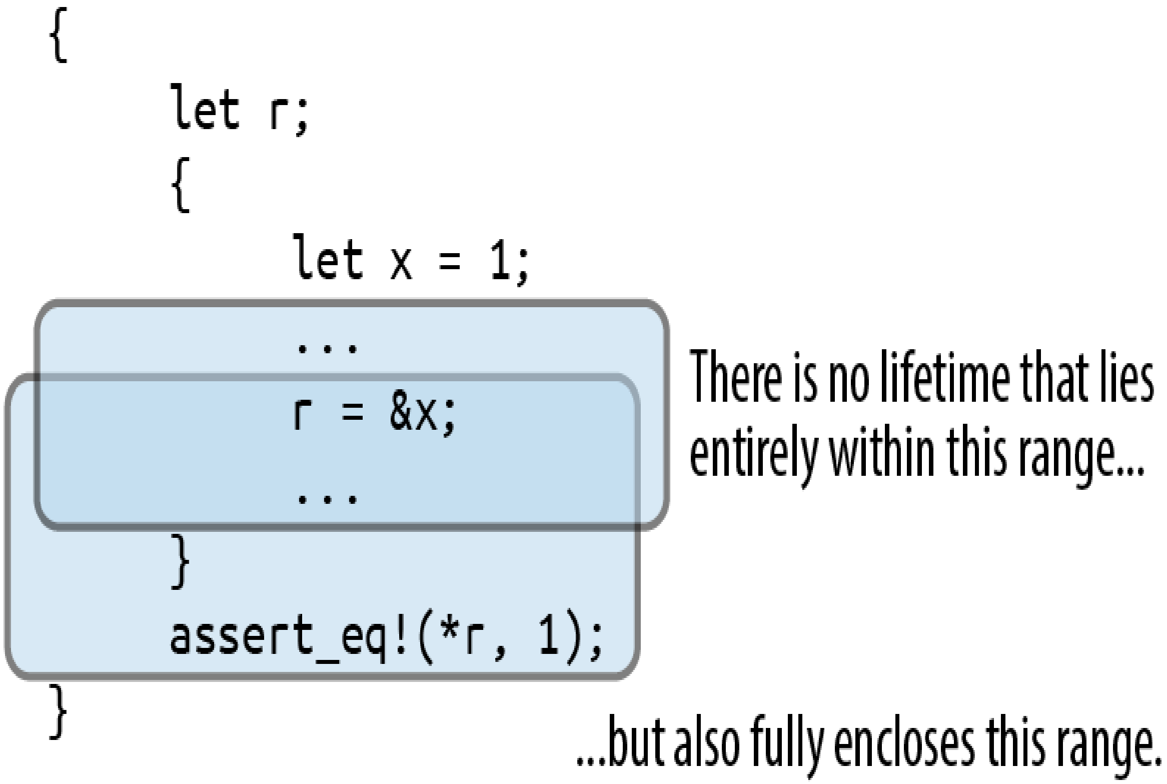


```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```

lifetime of anything stored in r must cover at least this range

Figure 4-4. Permissible lifetimes for reference stored in r

The first kind of constraint limits how large a reference’s lifetime can be, while the second kind limits how small it can be. Rust simply tries to find a lifetime for each reference that satisfies all these constraints. In our example, however, there is no such lifetime, as shown in Figure 4-5.



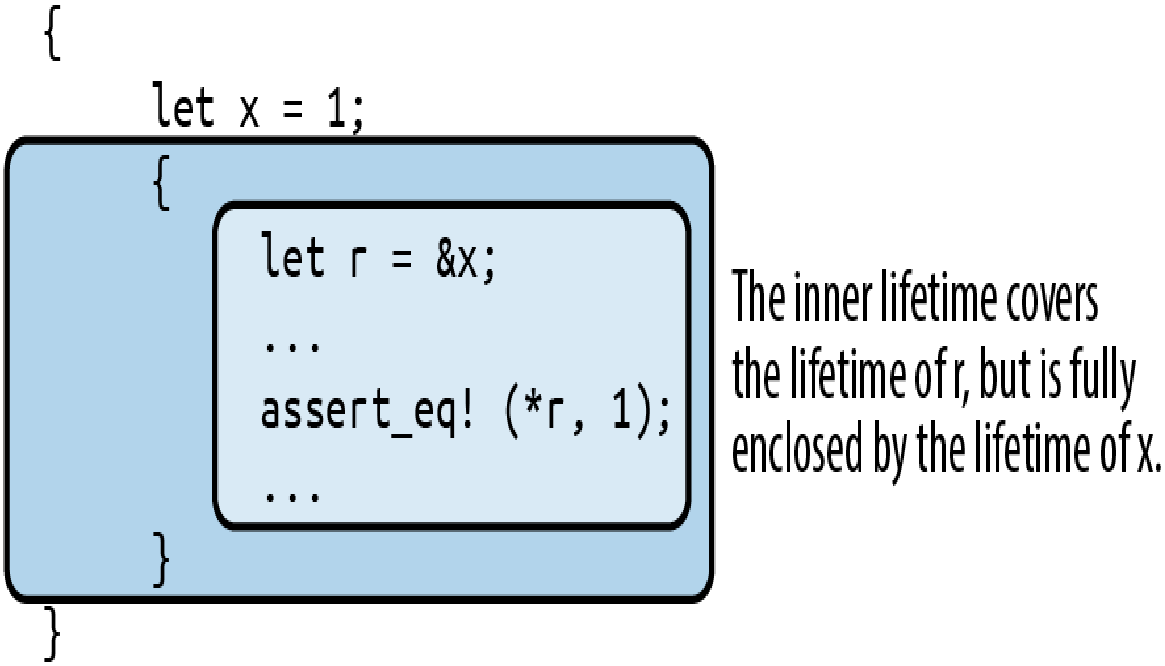
```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```

There is no lifetime that lies entirely within this range...

...but also fully encloses this range.

Figure 4-5. A reference with contradictory constraints on its lifetime

Let’s now consider a different example where things do work out. We have the same kinds of constraints: the reference’s lifetime must be contained by `x`’s, but fully enclose `r`’s. But because `r`’s lifetime is smaller now, there is a lifetime that meets the constraints, as shown in Figure 4-6.



```
{
    let x = 1;
    {
        let r = &x;
        ...
        assert_eq! (*r, 1);
        ...
    }
}
```

The inner lifetime covers the lifetime of r, but is fully enclosed by the lifetime of x.

Figure 4-6. A reference with a lifetime enclosing `r`'s scope, but within `x`'s scope

These rules apply in a natural way when you borrow a reference to some part of some larger data structure, like an element of a vector:

```
let v = vec![1, 2, 3];
let r = &v[1];
```

Since `v` owns the vector, which owns its elements, the lifetime of `v` must enclose that of the reference type of `&v[1]`. Similarly, if you store a reference in some data structure, its lifetime must enclose that of the data structure. For example, if you build a vector of references, all of them must have lifetimes enclosing that of the variable that owns the vector.

This is the essence of the process Rust uses for all code. Bringing more language features into the picture—e.g., data structures and function calls—introduces new sorts of constraints, but the principle remains the same: first, understand the constraints arising from the way the program uses references; then, find lifetimes that satisfy them. This is not so different from the process C and C++ programmers impose on themselves; the difference is that Rust knows the rules and enforces them.

Receiving References as Function Arguments

When we pass a reference to a function, how does Rust make sure the function uses it safely? Suppose we have a function `f` that takes a reference and stores it in a global variable. We'll need to make a few revisions to this, but here's a first cut:

```
// This code has several problems, and doesn't compile.
static mut STASH: &i32;
fn f(p: &i32) { STASH = p; }
```

Rust's equivalent of a global variable is called a *static*: it's a value that's created when the program starts and lasts until it terminates. (Like any other declaration, Rust's module system controls where statics are visible, so they're only "global" in their lifetime, not their visibility.) We cover statics in [Link to Come], but for now we'll just call out a few rules that the code just shown doesn't follow:

- Every static must be initialized.
- Mutable statics are inherently not thread-safe (after all, any thread can access a static at any time), and even in single-threaded programs, they can fall prey to other sorts of reentrancy problems. For these reasons, you may access a mutable static only within an `unsafe` block. In this example we're not concerned with those particular problems, so we'll just throw in an `unsafe` block and move on.

With those revisions made, we now have the following:

```
static mut STASH: &i32 = &128;
fn f(p: &i32) {  // still not good enough
    unsafe {
        STASH = p;
    }
}
```

We’re almost done. To see the remaining problem, we need to write out a few things that Rust is helpfully letting us omit. The signature of `f` as written here is actually shorthand for the following:

```
fn f<'a>(p: &'a i32) { ... }
```

Here, the lifetime `'a` (pronounced “tick A”) is a *lifetime parameter* of `f`. You can read `<'a>` as “for any lifetime `'a`” so when we write `fn f<'a>(p: &'a i32)`, we’re defining a function that takes a reference to an `i32` with any given lifetime `'a`.

Since we must allow `'a` to be any lifetime, things had better work out if it’s the smallest possible lifetime: one just enclosing the call to `f`. This assignment then becomes a point of contention:

```
STASH = p;
```

Since `STASH` lives for the program’s entire execution, the reference type it holds must have a lifetime of the same length; Rust calls this the *`'static` lifetime*. But the lifetime of `p`’s reference is some `'a`, which could be anything, as long as it encloses the call to `f`. So, Rust rejects our code:

```
error: lifetime may not live long enough
  |
3 | fn f(p: &i32) {  // still not good enough
  |         - let's call the lifetime of this reference `'1`
...
5 |         STASH = p;
  |         ^^^^^^^^^ assignment requires that `'1` must outlive
`'static`
```

At this point, it’s clear that our function can’t accept just any reference as an argument. But as Rust points out, it ought to be able to accept a reference that has a `'static` lifetime: storing such a reference in `STASH` can’t create a dangling pointer. And indeed, the following code compiles just fine:

```
static mut STASH: &i32 = &10;

fn f(p: &'static i32) {
    unsafe {
        STASH = p;
    }
}
```

This time, `f`'s signature spells out that `p` must be a reference with lifetime `'static`, so there's no longer any problem storing that in `STASH`. We can only apply `f` to references to other statics, but that's the only thing that's certain not to leave `STASH` dangling anyway. So we can write:

```
static WORTH_POINTING_AT: i32 = 1000;
f(&WORTH_POINTING_AT);
```

Since `WORTH_POINTING_AT` is a static, the type of `&WORTH_POINTING_AT` is `&'static i32`, which is safe to pass to `f`.

Take a step back, though, and notice what happened to `f`'s signature as we amended our way to correctness: the original `f(p: &i32)` ended up as `f(p: &'static i32)`. In other words, we were unable to write a function that stashed a reference in a global variable without reflecting that intention in the function's signature. In Rust, a function's signature always exposes the body's behavior.

Conversely, if we do see a function with a signature like `g(p: &i32)` (or with the lifetimes written out, `g<'a>(p: &'a i32)`), we can tell that it *does not* stash its argument `p` anywhere that will outlive the call. There's no need to look into `g`'s definition; the signature alone tells us what `g` can and can't do with its argument. This fact ends up being very useful when you're trying to establish the safety of a call to the function.

Passing References to Functions

Now that we’ve shown how a function’s signature relates to its body, let’s examine how it relates to the function’s callers. Suppose you have the following code:

```rust
// This could be written more briefly: fn g(p: &i32),
// but let's write out the lifetimes for now.
fn g<'a>(p: &'a i32) { ... }

let x = 10;
g(&x);
```

From `g`’s signature alone, Rust knows it will not save `p` anywhere that might outlive the call: any lifetime that encloses the call must work for `'a`. So Rust chooses the smallest possible lifetime for `&x`: that of the call to `g`. This meets all constraints: it doesn’t outlive `x`, and it encloses the entire call to `g`. So this code passes muster.

Note that although `g` takes a lifetime parameter `'a`, we didn’t need to mention it when calling `g`. You only need to worry about lifetime parameters when defining functions and types; when using them, Rust infers the lifetimes for you.

What if we tried to pass `&x` to our function `f` from earlier that stores its argument in a static?

```rust
fn f(p: &'static i32) { ... }

let x = 10;
f(&x);
```

This fails to compile: the reference `&x` must not outlive `x`, but by passing it to `f`, we constrain it to live at least as long as `'static`. There’s no way to satisfy everyone here, so Rust rejects the code.

Returning References

It’s common for a function to take a reference to some data structure and then return a reference into some part of that structure. For example, here’s a function that returns a reference to the smallest element of a slice:

```
// v should have at least one element.
fn smallest(v: &[i32]) -> &i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

We’ve omitted lifetimes from that function’s signature in the usual way. When a function takes a single reference as an argument and returns a single reference, Rust assumes that the two must have the same lifetime. Writing this out explicitly would give us:

```
fn smallest<'a>(v: &'a [i32]) -> &'a i32 { ... }
```

Suppose we call `smallest` like this:

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0);  // bad: points to element of dropped array
```

From `smallest`’s signature, we can see that its argument and return value must have the same lifetime, `'a`. In our call, the argument `&parabola` must not outlive `parabola` itself, yet `smallest`’s return value must live at least as long as `s`. There’s no possible lifetime `'a` that can satisfy both constraints, so Rust rejects the code:

```
error: `parabola` does not live long enough
   |
11 |         s = smallest(&parabola);
   |                       --------- borrow occurs here
12 |     }
   |     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0);  // bad: points to element of dropped
array
   |                 - borrowed value needs to live until here
14 | }
```

Moving `s` so that its lifetime is clearly contained within `parabola`'s fixes the problem:

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0);  // fine: parabola still alive
}
```

Lifetimes in function signatures let Rust assess the relationships between the references you pass to the function and those the function returns, and they ensure they're being used safely.

Structs Containing References

How does Rust handle references stored in data structures? Here's the same erroneous program we looked at earlier, except that we've put the reference inside a structure:

```
// This does not compile.
struct S {
    r: &i32,
}

let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10);  // bad: reads from dropped `x`
```

The safety constraints Rust places on references can't magically disappear just because we hid the reference inside a struct. Somehow, those constraints must end up applying to S as well. Indeed, Rust is skeptical:

```
error: missing lifetime specifier
  |
7 |         r: &i32
  |            ^ expected lifetime parameter
```

Whenever a reference type appears inside another type's definition, you must write out its lifetime. You can write this:

```
struct S {
    r: &'static i32,
}
```

This says that r can only refer to i32 values that will last for the lifetime of the program, which is rather limiting. The alternative is to give the type a lifetime parameter 'a and use that for r:

```
struct S<'a> {
    r: &'a i32,
}
```

Now the S type has a lifetime, just as reference types do. Each value you create of type S gets a fresh lifetime 'a, which becomes constrained by how you use the value. The lifetime of any reference you store in r had better enclose 'a, and 'a must outlast the lifetime of wherever you store the S.

Turning back to the preceding code, the expression S { r: &x } creates a fresh S value with some lifetime 'a. When you store &x in the r field, you constrain `'a` to lie entirely within `x`’s lifetime.

The assignment `s = S { ... }` stores this `S` in a variable whose lifetime extends to the end of the example, constraining `'a` to outlast the lifetime of `s`. And now Rust has arrived at the same contradictory constraints as before: `'a` must not outlive `x`, yet must live at least as long as `s`. No satisfactory lifetime exists, and Rust rejects the code. Disaster averted!

How does a type with a lifetime parameter behave when placed inside some other type?

```
struct D {
    s: S,  // not adequate
}
```

Rust is skeptical, just as it was when we tried placing a reference in `S` without specifying its lifetime:

```
error: missing lifetime specifier
  |
8 |     s: S  // not adequate
  |        ^ expected named lifetime parameter
  |
```

We can’t leave off `S`’s lifetime parameter here: Rust needs to know how `D`’s lifetime relates to that of the reference in its `S` in order to apply the same checks to `D` that it does for `S` and plain references.

We could give `s` the `'static` lifetime. This works:

```
struct D {
    s: S<'static>,
}
```

With this definition, the s field may only borrow values that live for the entire execution of the program. That's somewhat restrictive, but it does mean that D can't possibly borrow a local variable; there are no special constraints on D's lifetime.

The error message from Rust actually suggests another approach, which is more general:

```
help: consider introducing a named lifetime parameter
  |
7 | struct D<'a> {
8 |     s: S<'a>,
  |
```

Here, we give D its own lifetime parameter and pass that to S:

```
struct D<'a> {
    s: S<'a>,
}
```

By taking a lifetime parameter 'a and using it in s's type, we've allowed Rust to relate D value's lifetime to that of the reference its S holds.

We showed earlier how a function's signature exposes what it does with the references we pass it. Now we've shown something similar about types: a type's lifetime parameters always reveal whether it contains references with interesting (that is, non-'static) lifetimes and what those lifetimes can be.

For example, suppose we have a parsing function that takes a slice of bytes and returns a structure holding the results of the parse:

```
fn parse_record<'i>(input: &'i [u8]) -> Record<'i> { ... }
```

Without looking into the definition of the Record type at all, we can tell that, if we receive a Record from parse_record, whatever references it contains must point into the input buffer we passed in, and nowhere else (except perhaps at `'static` values).

In fact, this exposure of internal behavior is the reason Rust requires types that contain references to take explicit lifetime parameters. There's no reason Rust couldn't simply make up a distinct lifetime for each reference in the struct and save you the trouble of writing them out. Early versions of Rust actually behaved this way, but developers found it confusing: it is helpful to know when one value borrows something from another value, especially when working through errors.

It's not just references and types like `S` that have lifetimes. Every type in Rust has a lifetime, including `i32` and `String`. Most are simply `'static`, meaning that values of those types can live for as long as you like; for example, a `Vec<i32>` is self-contained and needn't be dropped before any particular variable goes out of scope. But a type like `Vec<&'a i32>` has a lifetime that must be enclosed by `'a`: it must be dropped while its referents are still alive.

Distinct Lifetime Parameters

Suppose you've defined a structure containing two references like this:

```
struct S<'a> {
    x: &'a i32,
    y: &'a i32,
}
```

Both references use the same lifetime `'a`. This could be a problem if your code wants to do something like this:

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
println!("{r}");
```

This code doesn't create any dangling pointers. The reference to `y` stays in `s`, which goes out of scope before `y` does. The reference to `x` ends up in `r`, which doesn't outlive `x`.

If you try to compile this, however, Rust will complain that `y` does not live long enough, even though it clearly does. Why is Rust worried? If you work through the code carefully, you can follow its reasoning:

- Both fields of `S` are references with the same lifetime `'a`, so Rust must find a single lifetime that works for both `s.x` and `s.y`.
- We assign `r = s.x`, requiring `'a` to enclose `r`'s lifetime.
- We initialized `s.y` with `&y`, requiring `'a` to be no longer than `y`'s lifetime.

These constraints are impossible to satisfy: no lifetime is shorter than `y`'s scope but longer than `r`'s. Rust balks.

The problem arises because both references in `S` have the same lifetime `'a`. Changing the definition of `S` to let each reference have a distinct lifetime fixes everything:

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32,
}
```

With this definition, `s.x` and `s.y` have independent lifetimes. What we do with `s.x` has no effect on what we store in `s.y`, so it's easy to satisfy the constraints now: `'a` can simply be `r`'s lifetime, and `'b` can be `s`'s. (`y`'s lifetime would work too for `'b`, but Rust tries to choose the smallest lifetime that works.) Everything ends up fine.

Function signatures can have similar effects. Suppose we have a function like this:

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r }  // perhaps too
tight
```

Here, both reference parameters use the same lifetime `'a`, which can unnecessarily constrain the caller in the same way we've shown previously. If this is a problem, you can let parameters' lifetimes vary independently:

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r }  // looser
```

The downside to this is that adding lifetimes can make types and function signatures harder to read. Your authors tend to try the simplest possible definition first and then loosen restrictions until the code compiles. Since Rust won't permit the code to run unless it's safe, simply waiting to be told when there's a problem is a perfectly acceptable tactic.

Omitting Lifetime Parameters

We've shown plenty of functions so far in this book that return references or take them as parameters, but we've usually not needed to spell out which lifetime is which. The lifetimes are there; Rust is just letting us omit them when it's reasonably obvious what they should be.

In the simplest cases, you may never need to write out lifetimes for your parameters. Rust just assigns a distinct lifetime to each spot that needs one. For example:

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32,
}

fn sum_r_xy(r: &i32, s: S) -> i32 {
    r + s.x + s.y
}
```

This function's signature is shorthand for:

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

If you do return references or other types with lifetime parameters, Rust still tries to make the unambiguous cases easy. If there's only a single lifetime that appears among your function's parameters, then Rust assumes any lifetimes in your return value must be that one:

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
    (&point[0], &point[2])
}
```

With all the lifetimes written out, the equivalent would be:

```
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

If there are multiple lifetimes among your parameters, then there's no natural reason to prefer one over the other for the return value, and Rust makes you spell out what's going on.

If your function is a method on some type and takes its `self` parameter by reference, then that breaks the tie: Rust assumes that `self`'s lifetime is the one to give everything in your return value. (A `self` parameter refers to the value the method is being called on. It's Rust's equivalent of `this` in C++, Java, or JavaScript, or `self` in Python. We'll cover methods in [Link to Come].)

For example, you can write the following:

```
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
        for i in 0..self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}
```

The `find_by_prefix` method’s signature is shorthand for:

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) ->
Option<&'a String>
```

Rust assumes that whatever you’re borrowing, you’re borrowing from `self`.

Again, these are just abbreviations, meant to be helpful without introducing surprises. When they’re not what you want, you can always write the lifetimes out explicitly.

Sharing Versus Mutation

So far, we’ve discussed how Rust ensures no reference will ever point to a variable that has gone out of scope. But there are other ways to introduce dangling pointers. Here’s an easy case:

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v;  // move vector to aside
r[0];           // bad: uses `v`, which is now uninitialized
```

The assignment to `aside` moves the vector, leaving `v` uninitialized, and turns `r` into a dangling pointer, as shown in Figure 4-7.

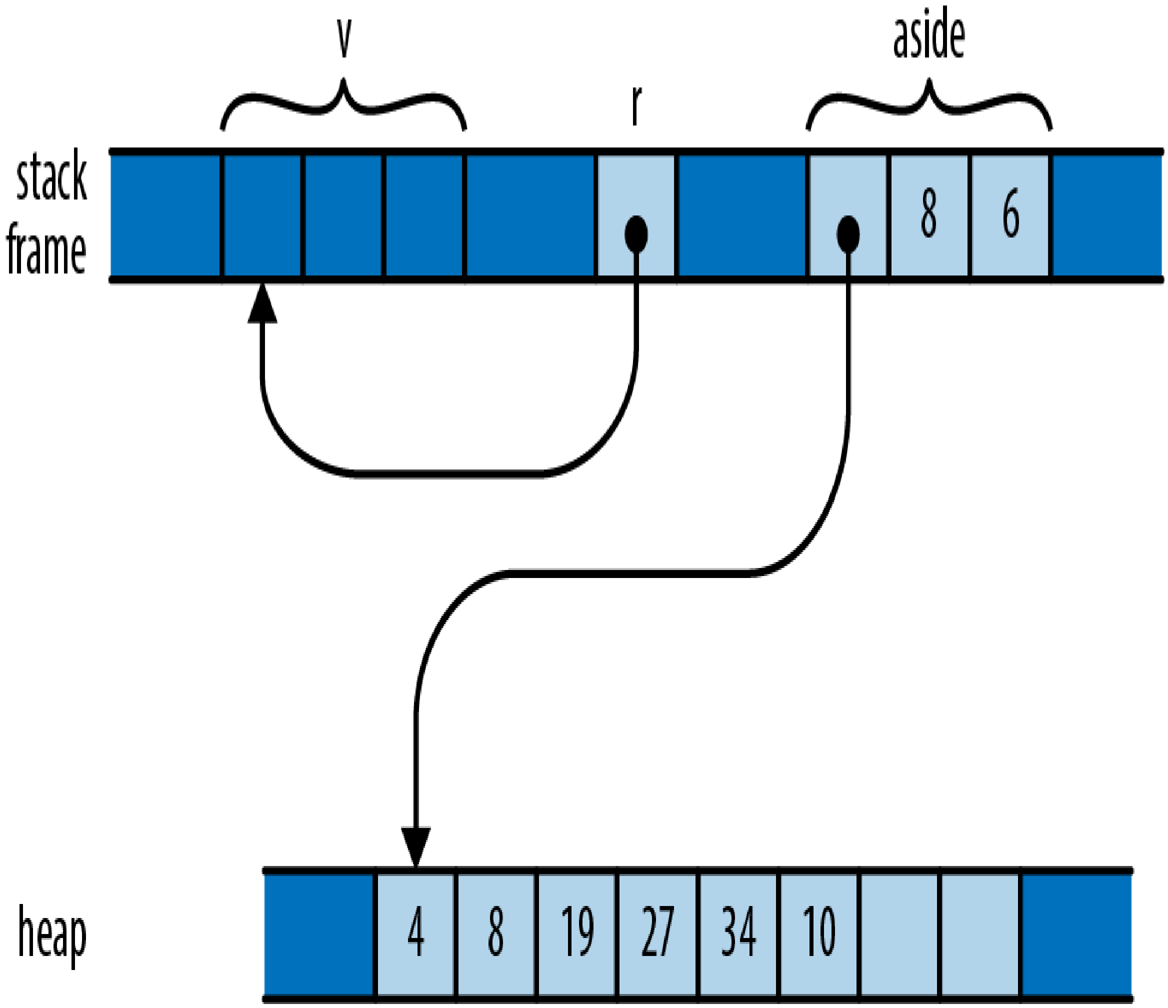


Figure 4-7. A reference to a vector that has been moved away

Although `v` stays in scope for `r`'s entire lifetime, the problem here is that `v`'s value gets moved elsewhere, leaving `v` uninitialized while `r` still refers to it. Naturally, Rust catches the error:

```
error: cannot move out of `v` because it is borrowed
   |
9  |     let r = &v;
   |              - borrow of `v` occurs here
10 |     let aside = v;  // move vector to aside
   |         ^^^^^ move out of `v` occurs here
```

Throughout its lifetime, a shared reference makes its referent read-only: you may not assign to the referent or move its value elsewhere. In this code, `r`'s lifetime contains the attempt to move the vector, so Rust rejects the program. If you change the program as shown here, there's no problem:

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];        // ok: vector is still there
}
let aside = v;
```

In this version, `r` goes out of scope earlier, the reference's lifetime ends before `v` is moved aside, and all is well.

Here's a different way to wreak havoc. Suppose we have a handy function to extend a vector with the elements of a slice:

```
fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

This is a less flexible (and much less optimized) version of the standard library's `extend_from_slice` method on vectors. We can use it to build up a vector from slices of other vectors or arrays:

```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

extend(&mut wave, &head);   // extend wave with another vector
extend(&mut wave, &tail);   // extend wave with an array

assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);
```

So we've built up one period of a sine wave here. If we want to add another undulation, can we append the vector to itself?

```
extend(&mut wave, &wave);
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0, 0.0, 1.0, 0.0, -1.0]);
```

This may look fine on casual inspection. But remember that when we add an element to a vector, if its buffer is full, it must allocate a new buffer with more space. Suppose `wave` starts with space for four elements and so must allocate a larger buffer when `extend` tries to add a fifth. Memory ends up looking like Figure 4-8.

The `extend` function's `vec` argument borrows `wave` (owned by the caller), which has allocated itself a new buffer with space for eight elements. But `slice` continues to point to the old four-element buffer, which has been dropped.

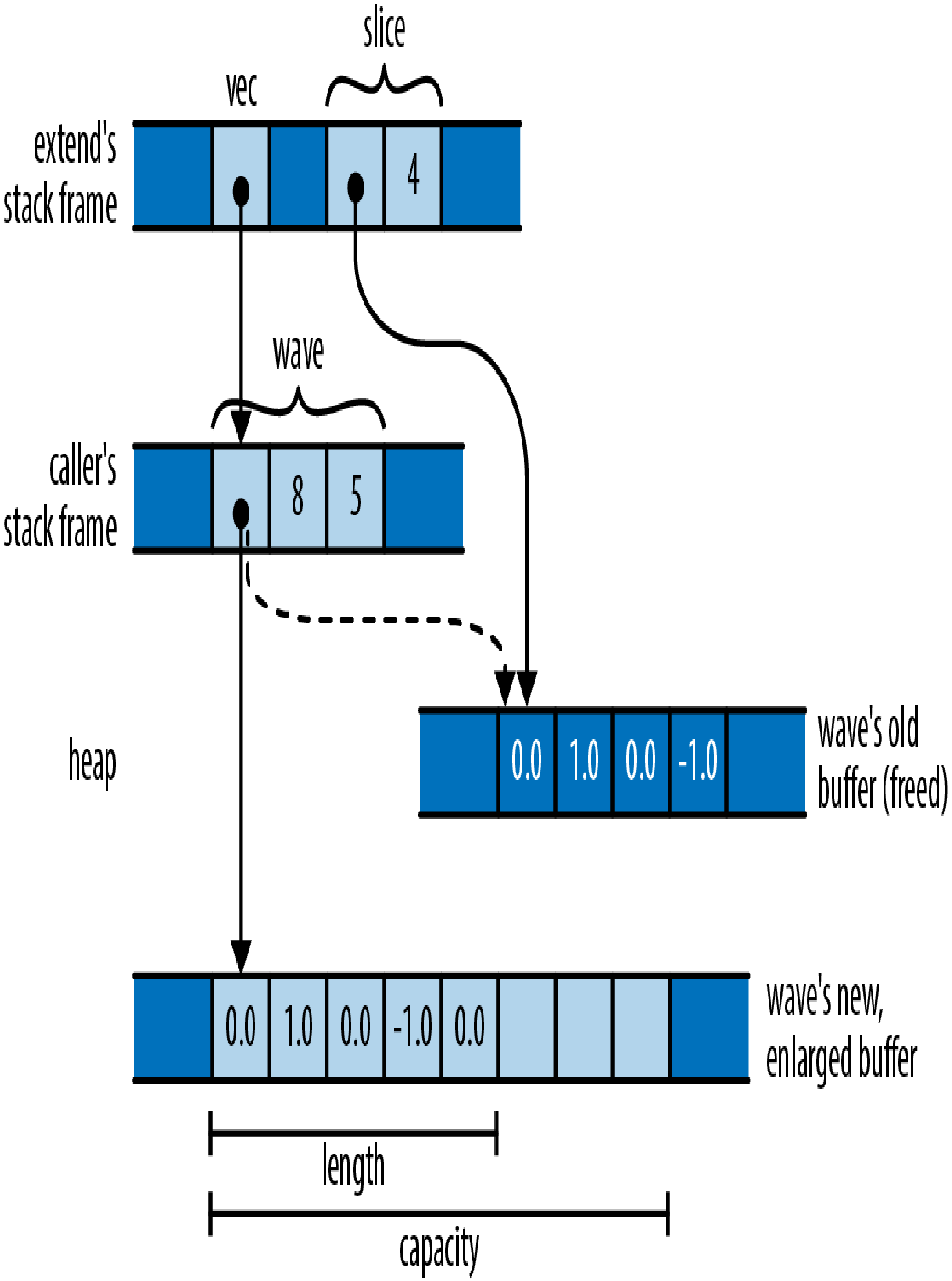


Figure 4-8. A slice turned into a dangling pointer by a vector reallocation

This sort of problem isn’t unique to Rust: modifying collections while pointing into them is delicate territory in many languages. In C++, the `std::vector` specification cautions you that “reallocation [of the vector’s buffer] invalidates all the references, pointers, and iterators referring to the elements in the sequence.” Similarly, Java says, of modifying a `java.util.Hashtable` object:

> *If the Hashtable is structurally modified at any time after the iterator is created, in any way except through the iterator’s own remove method, the iterator will throw a* `ConcurrentModificationException`*.*

What’s especially difficult about this sort of bug is that it doesn’t happen all the time. In testing, your vector might always happen to have enough space, the buffer might never be reallocated, and the problem might never come to light.

Rust, however, reports the problem with our call to `extend` at compile time:

```
error: cannot borrow `wave` as immutable because it is also
       borrowed as mutable
  |
9 |     extend(&mut wave, &wave);
  |                 ----   ^^^^- mutable borrow ends here
  |                 |      |
  |                 |      immutable borrow occurs here
  |                 mutable borrow occurs here
```

In other words, we may borrow a mutable reference to the vector, and we may borrow a shared reference to its elements, but those two references’ lifetimes must not overlap. In our case, both references’ lifetimes contain the call to `extend`, so Rust rejects the code.

These errors both stem from violations of Rust’s rules for mutation and sharing:

Shared access is read-only access.

> Values borrowed by shared references are read-only. Across the lifetime of a shared reference, neither its referent, nor anything reachable from that referent, can be changed *by anything*. There exist no live mutable references to anything in that structure, its owner is held read-only, and so on. It's really frozen.

Mutable access is exclusive access.

> A value borrowed by a mutable reference is reachable exclusively via that reference. Across the lifetime of a mutable reference, there is no other usable path to its referent or to any value reachable from there. The only references whose lifetimes may overlap with a mutable reference are those you borrow from the mutable reference itself.

Rust reported the `extend` example as a violation of the second rule: since we've borrowed a mutable reference to `wave`, that mutable reference must be the only way to reach the vector or its elements. The shared reference to the slice is itself another way to reach the elements, violating the second rule.

But Rust could also have treated our bug as a violation of the first rule: since we've borrowed a shared reference to `wave`'s elements, the elements and the `Vec` itself are all read-only. You can't borrow a mutable reference to a read-only value.

Each kind of reference affects what we can do with the values along the owning path to the referent, and the values reachable from the referent (Figure 4-9).

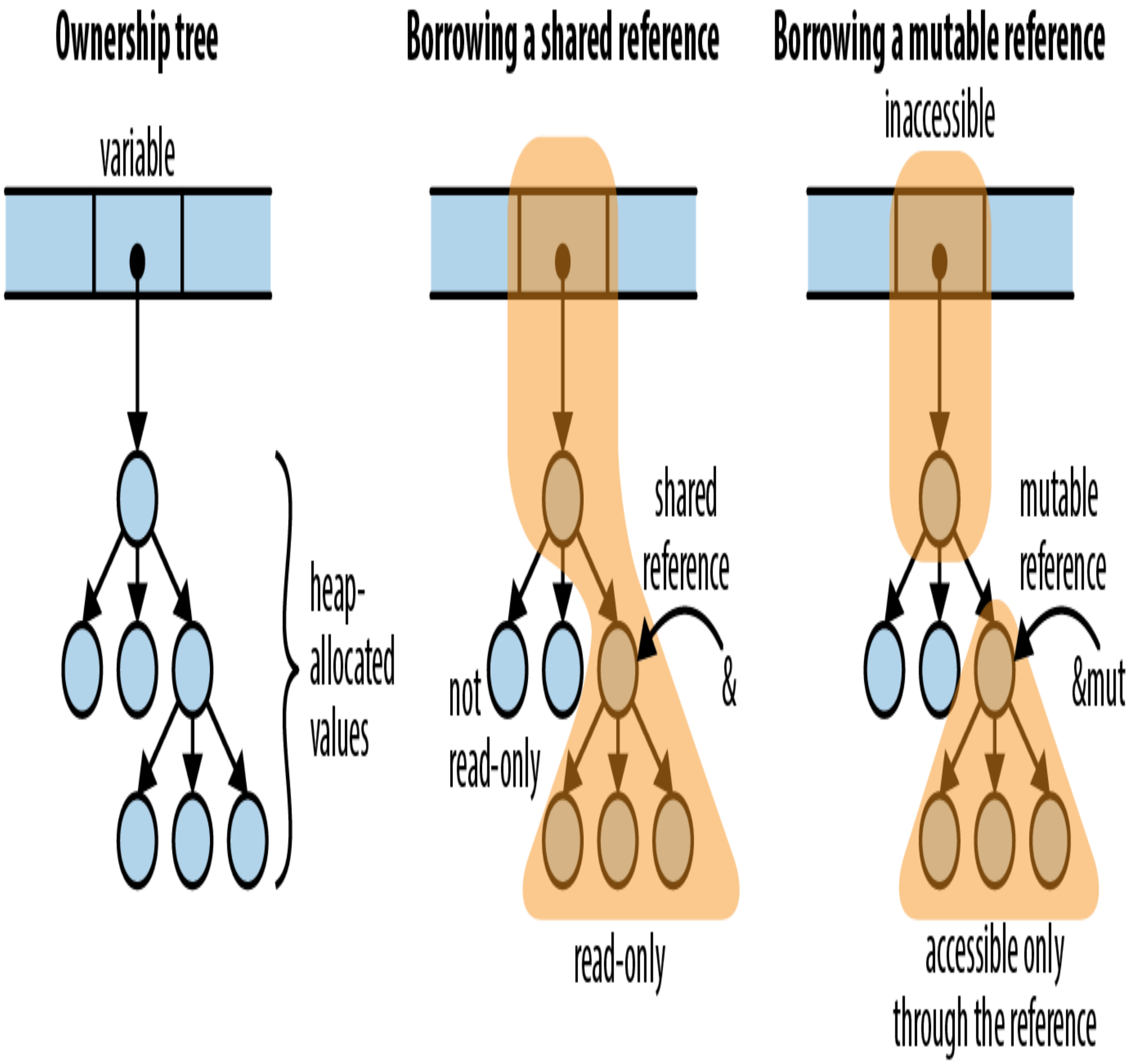


Figure 4-9. Borrowing a reference affects what you can do with other values in the same ownership tree

Note that in both cases, the path of ownership leading to the referent cannot be changed for the reference's lifetime. For a shared borrow, the path is read-only; for a mutable borrow, it's completely inaccessible. So there's no way for the program to do anything that will invalidate the reference.

Paring these principles down to the simplest possible examples:

```
let mut x = 10;
let r1 = &x;
let r2 = &x;     // ok: multiple shared borrows permitted
x += 10;         // error: cannot assign to `x` because it is
```

```
borrowed
let m = &mut x;  // error: cannot borrow `x` as mutable because
it is
                 // also borrowed as immutable
println!("{r1}, {r2}, {m}");  // the references are used here,
                              // so their lifetimes must last
                              // at least this long

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y;  // error: cannot borrow as mutable more than
once
let z = y;        // error: cannot use `y` because it was mutably
borrowed
println!("{m1}, {m2}, {z}");  // references are used here
```

It is OK to reborrow a shared reference from a shared reference:

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;         // ok: reborrowing shared as shared
let m1 = &mut r.1;     // error: can't reborrow shared as mutable
println!("{r0}");      // r0 gets used here
```

You can reborrow from a mutable reference:

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // ok: reborrowing mutable from mutable
*m0 = 137;
let r1 = &m.1;          // ok: reborrowing shared from mutable,
                        // and doesn't overlap with m0
v.1;                    // error: access through other paths
still forbidden
println!("{r1}");       // r1 gets used here
```

These restrictions are pretty tight. Turning back to our attempted call `extend(&mut wave, &wave)`, there’s no quick and easy way to fix up the code to work the way we’d like. And Rust applies these rules everywhere: if we borrow, say, a shared reference to a key in a `HashMap`, we can’t borrow a mutable reference to the `HashMap` until the shared reference’s lifetime ends.

But there’s good justification for this: designing collections to support unrestricted, simultaneous iteration and modification is difficult and often precludes simpler, more efficient implementations. Java’s `Hashtable` and C++’s `vector` don’t bother, and neither Python dictionaries nor JavaScript objects define exactly how such access behaves. Other collection types in JavaScript do, but require heavier implementations as a result. C++’s `std::map` promises that inserting new entries doesn’t invalidate pointers to other entries in the map, but by making that promise, the standard precludes more cache-efficient designs like Rust’s `BTreeMap`, which stores multiple entries in each node of the tree.

Here’s another example of the kind of bug these rules catch. Consider the following C++ code, meant to manage a file descriptor. To keep things simple, we’re only going to show a constructor and a copying assignment operator, and we’re going to omit error handling:

```
struct File {
  int descriptor;

  File(int d) : descriptor(d) {}

  File& operator=(const File &rhs) {
    close(descriptor);
    descriptor = dup(rhs.descriptor);
    return *this;
  }
};
```

The assignment operator is simple enough, but fails badly in a situation like this:

```
File f(open("foo.txt", ...));
...
f = f;
```

If we assign a File to itself, both rhs and *this are the same object, so operator= closes the very file descriptor it’s about to pass to dup. We destroy the same resource we were meant to copy.

In Rust, the analogous code would be:

```
struct File {
    descriptor: i32,
}

fn new_file(d: i32) -> File {
    File { descriptor: d }
}

fn clone_from(this: &mut File, rhs: &File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}
```

(This is not idiomatic Rust. There are excellent ways to give Rust types their own constructor functions and methods, which we describe in [Link to Come], but the preceding definitions work for this example.)

If we write the Rust code corresponding to the use of File, we get:

```
let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);
```

Rust, of course, refuses to even compile this code:

```
error: cannot borrow `f` as immutable because it is also
       borrowed as mutable
   |
18 |     clone_from(&mut f, &f);
   |                     -   ^- mutable borrow ends here
   |                     |   |
   |                     |   immutable borrow occurs here
   |                     mutable borrow occurs here
```

This should look familiar. It turns out that two classic C++ bugs—failure to cope with self-assignment and using invalidated iterators—are the same underlying kind of bug! In both cases, code assumes it is modifying one value while consulting another, when in fact they're both the same value. If you've ever accidentally let the source and destination of a call to `memcpy` or `strcpy` overlap in C or C++, that's yet another form the bug can take. By requiring mutable access to be exclusive, Rust has fended off a wide class of everyday mistakes.

The immiscibility of shared and mutable references really demonstrates its value when writing concurrent code. A data race is possible only when some value is both mutable and shared between threads—which is exactly what Rust's reference rules eliminate. A concurrent Rust program that avoids `unsafe` code is free of data races *by construction*. We'll cover this aspect in more detail when we talk about concurrency in [Link to Come], but in summary, concurrency is much easier to use in Rust than in most other languages.

RUST’S SHARED REFERENCES VERSUS C’S POINTERS TO CONST

On first inspection, Rust’s shared references seem to closely resemble C and C++’s pointers to `const` values. However, Rust’s rules for shared references are much stricter. For example, consider the following C code:

```
int x = 42;              // int variable, not const
const int *p = &x;       // pointer to const int
assert(*p == 42);
x++;                     // change variable directly
assert(*p == 43);        // “constant” referent's value has
changed
```

The fact that `p` is a `const int *` means that you can’t modify its referent via `p` itself: `(*p)++` is forbidden. But you can also get at the referent directly as `x`, which is not `const`, and change its value that way. The C family’s `const` keyword has its uses, but constant it is not.

In Rust, a shared reference forbids all modifications to its referent, until its lifetime ends:

```
let mut x = 42;          // non-const i32 variable
let p = &x;              // shared reference to i32
assert_eq!(*p, 42);
x += 1;                  // error: cannot assign to x because
it is borrowed
assert_eq!(*p, 42);      // if you take out the assignment,
this is true
```

To ensure a value is constant, we need to keep track of all possible paths to that value and make sure that they either don’t permit modification or cannot be used at all. C and C++ pointers are too unrestricted for the compiler to check this. Rust’s references are always tied to a particular lifetime, making it feasible to check them at compile time.

Taking Arms Against a Sea of Objects

Since the rise of automatic memory management in the 1990s, the default architecture of all programs has been the *sea of objects*, shown in Figure 4-10.

This is what happens if you have garbage collection and you start writing a program without designing anything. We’ve all built systems that look like this.

This architecture has many advantages that don’t show up in the diagram: initial progress is rapid, it’s easy to hack stuff in, and a few years down the road, you’ll have no difficulty justifying a complete rewrite. (Cue AC/DC’s “Highway to Hell.”)

Of course, there are disadvantages too. When everything depends on everything else like this, it’s hard to test, evolve, or even think about any component in isolation.

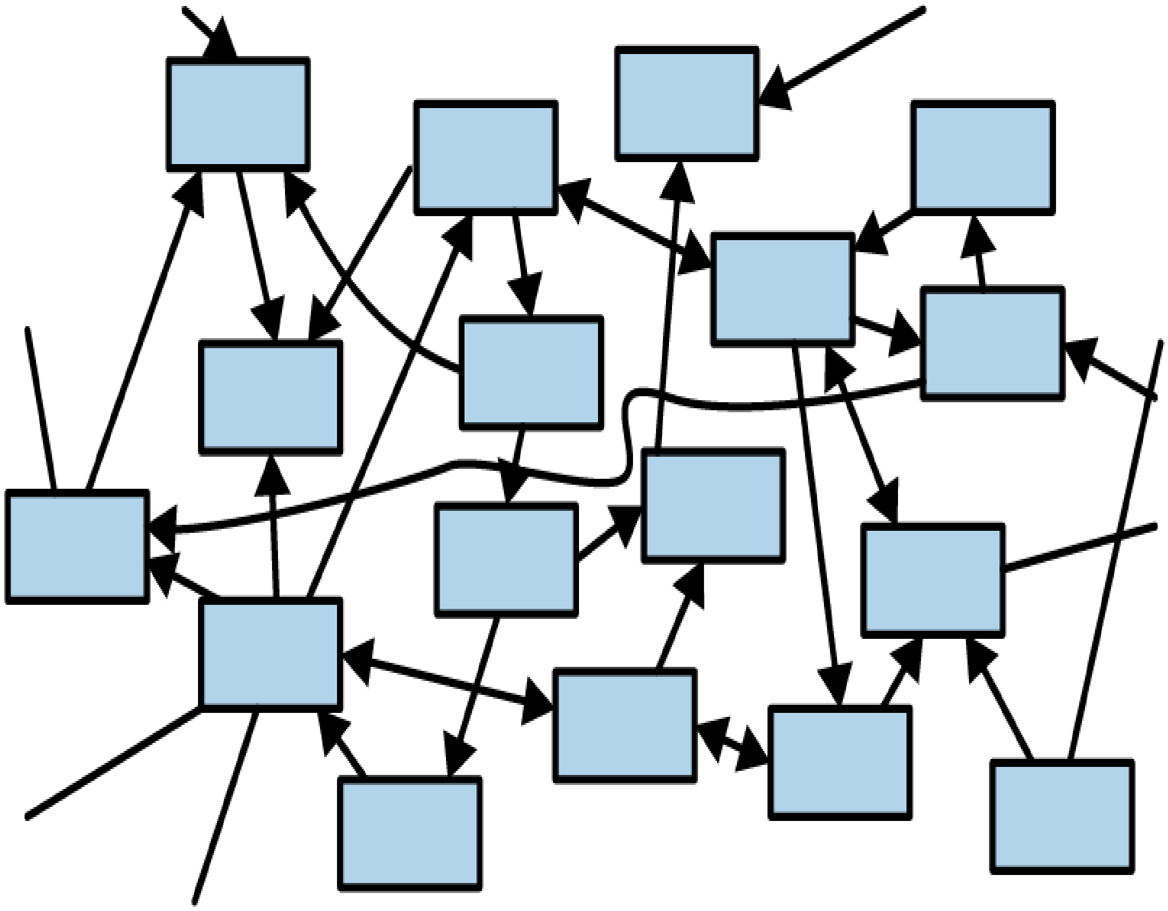


Figure 4-10. A sea of objects

One fascinating thing about Rust is that the ownership model puts a speed bump on the highway to hell. It takes a bit of effort to make a cycle in Rust —two values such that each one contains a reference pointing to the other. You have to use a smart pointer type, such as `Rc`, and interior mutability,a topic we haven't even covered yet. Rust prefers for pointers, ownership, and data flow to pass through the system in one direction, as shown in Figure 4-11.

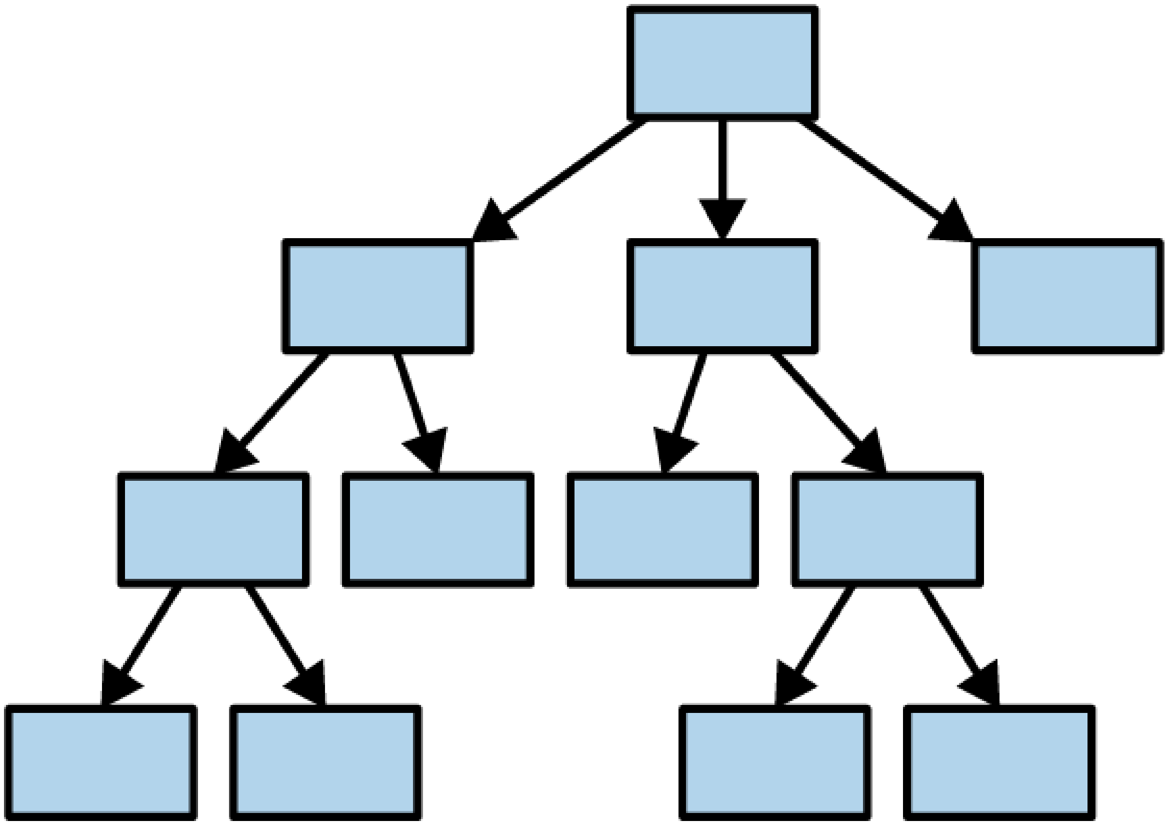


Figure 4-11. A tree of values

The reason we bring this up right now is that it would be natural, after reading this chapter, to want to run right out and create a "sea of structs," all tied together with `Rc` smart pointers, and re-create all the object-oriented antipatterns you're familiar with. This won't work for you right away. Rust's ownership model will give you some trouble. The cure is to do some up-front design and build a better program.

Rust is all about transferring the pain of understanding your program from the future to the present. It works unreasonably well: not only can Rust force you to understand why your program is thread-safe, it can even require some amount of high-level architectural design.

Chapter 5. Expressions

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

> *LISP programmers know the value of everything, but the cost of nothing.*
>
> —Alan Perlis, epigram #55

In this chapter, we'll cover the *expressions* of Rust, the building blocks that make up the body of Rust functions and thus the majority of Rust code. We'll cover control flow, including `if` and `match` expressions, loops, and function calls, which in Rust are all expressions, and we'll examine how Rust's foundational operators work in isolation and in combination.

A few concepts that technically fall into this category, such as closures and iterators, are deep enough that we will dedicate a whole chapter to them later. For now, we aim to cover as much syntax as possible in a few pages.

An Expression Language

Rust visually resembles the C family of languages, but this is a bit of a ruse. In C, there is a sharp distinction between *expressions,* bits of code that look something like this:

```
5 * (fahr-32) / 9
```

and *statements,* which look more like this:

```
for (; begin != end; ++begin) {
    if (*begin == target)
        break;
}
```

Expressions have values. Statements don’t.

Rust is what is called an *expression language*. This means it follows an older tradition, dating back to Lisp, where expressions do all the work.

In C, `if` and `switch` are statements. They don’t produce a value, and they can’t be used in the middle of an expression. In Rust, `if` and `match` *can* produce values. We already saw a `match` expression that produces a numeric value in Chapter 1:

```
pixels[r * bounds.0 + c] =
    match escapes(Complex { re: point.0, im: point.1 }, 255) {
        None => 0,
        Some(count) => 255 - count as u8,
    };
```

An `if` expression can be used to initialize a variable:

```
let status =
    if cpu.temperature <= MAX_TEMP {
        HttpStatus::Ok
    } else {
        HttpStatus::ServerError  // server melted
    };
```

A `match` expression can be passed as an argument to a function or macro:

```
println!("Inside the vat, you see {}.",
    match vat.contents {
        Some(brain) => brain.desc(),
        None => "nothing of interest",
    });
```

This explains why Rust does not have C’s ternary operator (`expr1 ? expr2 : expr3`). In C, it is a handy expression-level analogue to the `if` statement. It would be redundant in Rust: the `if` expression handles both cases.

Most of the control flow tools in C are statements. In Rust, they are all expressions.

Operator Precedence and Associativity

Table 5-1 summarizes Rust expression syntax. We will discuss all of these kinds of expressions in this chapter. Operators are grouped by precedence and ordered from highest precedence to lowest. (Like most programming languages, Rust has *operator precedence* to determine the order of operations when an expression contains multiple adjacent operators. For example, in `limit < 2 * broom.size + 1`, the `.` operator has the highest precedence, so the field access happens first.)

Table 5-1. Expressions

Expression type	Example
Array literal	`[1, 2, 3]`
Repeat array literal	`[0; 50]`
Tuple	`(6, "crullers")`
Grouping	`(2 + 2)`
Block	`{ f(); g() }`
Control flow expressions	`if ok { f() }`
	`if ok { 1 } else { 0 }`
	`if let Some(x) = f() { x } else { 0 }`
	`match x { None => 0, _ => 1 }`
	`for v in e { f(v); }`
	`while ok { ok = f(); }`
	`while let Some(x) = it.next() { f(x); }`
	`loop { next_event(); }`
	`break`
	`continue`
	`return 0`

Expression type	Example
Macro invocation	`println!("ok")`
Path	`std::f64::consts::PI`
Struct literal	`Point {x: 0, y: 0}`
Tuple field access	`pair.0`
Struct field access	`point.x`
Method call	`point.translate(50, 50)`
Function call	`stdin()`
Index	`arr[0]`
`Err`/`None` early return	`create_dir("tmp")?`
Logical/bitwise NOT	`!ok`
Negation	`-num`
Dereference	`*ptr`
Borrow	`&val`
Type cast	`x as u32`
Multiplication	`n * 2`
Division	`n / 2`
Remainder (modulus)	`n % 2`

Expression type	Example
Addition	`n + 1`
Subtraction	`n - 1`
Left shift	`n << 1`
Right shift	`n >> 1`
Bitwise AND	`n & 1`
Bitwise exclusive OR	`n ^ 1`
Bitwise OR	`n | 1`
Less than	`n < 1`
Less than or equal	`n <= 1`
Greater than	`n > 1`
Greater than or equal	`n >= 1`
Equal	`n == 1`
Not equal	`n != 1`
Logical AND	`x.ok && y.ok`
Logical OR	`x.ok || backup.ok`
End-exclusive range	`start..stop`
End-inclusive range	`start..=stop`

Expression type	Example
Assignment	`x = val`
Compound assignment	`x *= 1`
	`x /= 1`
	`x %= 1`
	`x += 1`
	`x -= 1`
	`x <<= 1`
	`x >>= 1`
	`x &= 1`
	`x ^= 1`
	`x |= 1`
Closure	`|x, y| x + y`

The arithmetic and bitwise operations and their associated compound assignments can be overloaded with arbitrary behavior for user-defined types, as discussed in [Link to Come].

All of the operators that can usefully be chained are left-associative. That is, a chain of operations such as `a - b - c` is grouped as `(a - b) - c`, not `a - (b - c)`. The operators that can be chained in this way are all the ones you might expect:

```
*   /   %   +   -   <<   >>   &   ^   |   &&   ||   as
```

The comparison operators, the assignment operators, and the range operators `..` and `..=` can’t be chained at all.

Blocks

Blocks are the most general kind of expression. A block produces a value and can be used anywhere a value is needed:

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

The code after `Some(author) =>` is the simple expression `author.name()`. The code after `None =>` is a block expression. It makes no difference to Rust. The value of the block is the value of its last expression, `ip.to_string()`.

Note that there is no semicolon after the `ip.to_string()` method call. Most lines of Rust code do end with either a semicolon or curly braces, just like C or Java. And if a block looks like C code, with semicolons in all the familiar places, then it will run just like a C block, and its value will be `()`. As we mentioned in Chapter 1, when you leave the semicolon off the last line of a block, that makes the value of the block the value of its final expression, rather than the usual `()`.

In some languages, particularly JavaScript, you’re allowed to omit semicolons, and the language simply fills them in for you—a minor convenience. This is different. In Rust, the semicolon actually means something:

```
let msg = {
    // let-declaration: semicolon is always required
    let dandelion_control = puffball.open();

    // expression + semicolon: method is called, return value
dropped
    dandelion_control.release_all_seeds(launch_codes);

    // expression with no semicolon: method is called,
    // return value stored in `msg`
    dandelion_control.get_status()
};
```

This ability of blocks to contain declarations and also produce a value at the end is a neat feature, one that quickly comes to feel natural. The one drawback is that it leads to an odd error message when you leave out a semicolon by accident:

```
...
if preferences.changed() {
    page.compute_size()  // oops, missing semicolon
}
...
```

If you made this mistake in a C or Java program, the compiler would simply point out that you're missing a semicolon. Here's what Rust says:

```
error: mismatched types
22 |         page.compute_size()  // oops, missing semicolon
   |         ^^^^^^^^^^^^^^^^^^^- help: try adding a semicolon:
`;`
   |         |
   |         expected (), found tuple
   |
   = note: expected unit type `()`
               found tuple `(u32, u32)`
```

With the semicolon missing, the block's value would be whatever `page.compute_size()` returns, but an `if` without an `else` must always return `()`. Fortunately, Rust has seen this sort of thing before and suggests adding the semicolon.

A block can also be *labeled* with a lifetime. This label can be used with `break` to exit early with a value, similarly to `return` in a function. In the following example, `'trim` is a label for the overall block. `break` exits the named block immediately, and it evalues to the given value:

```
let trimmed = 'trim: {
    if string.chars().last() != Some('\n') {
        break 'trim None;
    }
    string.pop();
    if string.chars().last() != Some('\r') {
        break 'trim Some(Newline::Unix);
    }
    string.pop();
    Some(Newline::Windows)
};
```

This is similar to a deeply nested `if ... else` chain, but can be more readable for a series of side effects.

Declarations

In addition to expressions and semicolons, a block may contain any number of declarations. The most common are `let` declarations, which declare local variables:

```
let name: type = expr;
```

The type and initializer are optional. The semicolon is required. Like all identifiers in Rust, variable names must start with a letter or underscore, and can contain digits only after that first character. Rust has a broad definition of “letter”: it includes Greek letters, accented Latin characters, and many more symbols—anything that Unicode Standard Annex #31 declares suitable. Emoji aren't allowed.

A `let` declaration can declare a variable without initializing it. The variable can then be initialized with a later assignment. This is occasionally useful, because sometimes a variable should be initialized from the middle of some sort of control flow construct:

```
let name;
if user.has_nickname() {
    name = user.nickname();
} else {
    name = generate_unique_name();
    user.register(&name);
}
```

Here there are two different ways the local variable `name` might be initialized, but either way it will be initialized exactly once, so `name` does not need to be declared `mut`.

It's an error to use a variable before it's initialized. (This is closely related to the error of using a value after it's been moved. Rust really wants you to use values only while they exist!)

You may occasionally see code that seems to redeclare an existing variable, like this:

```
for line in file.lines() {
    let line = line?;
    ...
}
```

The `let` declaration creates a new, second variable, of a different type. The type of the first variable `line` is `Result<String, io::Error>`. The second `line` is a `String`. Its definition supersedes the first's for the rest of the block. This is called *shadowing* and is very common in Rust programs. The code is equivalent to:

```
for line_result in file.lines() {
    let line = line_result?;
    ...
}
```

In this book, we'll stick to using a `_result` suffix in such situations so that the variables have distinct names. In real-world code, however, shadowing can be useful to prevent future code from incorrectly referring to the previous value. Since that name now refers to the new value instead, it's impossible to do so by mistake.

A block can also contain *item declarations*. An item is simply any declaration that could appear globally in a program or module, such as a `fn`, `struct`, or `use`.

Later chapters will cover items in detail. For now, `fn` makes a sufficient example. Any block may contain a `fn`:

```
use std::io;
use std::cmp::Ordering;

fn show_files() -> io::Result<()> {
    let mut v = vec![];
    ...

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) ->
Ordering {
        a.timestamp.cmp(&b.timestamp)   // first, compare
timestamps
            .reverse()                  // newest file first
            .then(a.path.cmp(&b.path))  // compare paths to break
ties
    }

    v.sort_by(cmp_by_timestamp_then_name);
    ...
}
```

When a `fn` is declared inside a block, its scope is the entire block—that is, it can be *used* throughout the enclosing block. But a nested `fn` cannot access local variables or arguments that happen to be in scope. For example, the function `cmp_by_timestamp_then_name` could not use `v` directly. (Rust also has closures, which do see into enclosing scopes. See [Link to Come].)

A block can even contain a whole module. This may seem a bit much—do we really need to be able to nest *every* piece of the language inside every other piece?—but programmers (and particularly programmers using macros) have a way of finding uses for every scrap of orthogonality the language provides.

if and match

The form of an `if` expression is familiar:

```
if condition1 {
    block1
} else if condition2 {
    block2
} else {
    block_n
}
```

Each `condition` must be an expression of type `bool`; true to form, Rust does not implicitly convert numbers or pointers to Boolean values.

Unlike C, parentheses are not required around conditions. In fact, `rustc` will emit a warning if unnecessary parentheses are present. The curly braces, however, are required.

The `else if` blocks, as well as the final `else`, are optional. An `if` expression with no `else` block behaves exactly as though it had an empty `else` block.

`match` expressions are something like the C `switch` statement, but more flexible. A simple example:

```
match code {
    0 => println!("OK"),
    1 => println!("Wires Tangled"),
    2 => println!("User Asleep"),
    _ => println!("Unrecognized Error {code}"),
}
```

This is something a `switch` statement could do. Exactly one of the four arms of this `match` expression will execute, depending on the value of `code`. The wildcard pattern _ matches everything. This is like the `default:` case in a `switch` statement, except that it must come last; placing a _ pattern before other patterns means that it will have precedence over them. Those patterns will never match anything (and the compiler will warn you about it).

The compiler can optimize this kind of `match` using a jump table, just like a `switch` statement in C++. A similar optimization is applied when each arm of a `match` produces a constant value. In that case, the compiler builds an array of those values, and the `match` is compiled into an array access. Apart from a bounds check, there is no branching at all in the compiled code.

The versatility of `match` stems from the variety of supported *patterns* that can be used to the left of => in each arm. Above, each pattern is simply a constant integer. We've also shown `match` expressions that distinguish the two kinds of `Option` value:

```
match params.get("name") {
    Some(name) => println!("Hello, {name}!"),
    None => println!("Greetings, stranger."),
}
```

This is only a hint of what patterns can do. A pattern can match a range of values. It can unpack tuples. It can match against individual fields of structs. It can chase references, borrow parts of a value, and more. Rust's patterns are a mini-language of their own. We'll dedicate several pages to them in [Link to Come].

The general form of a match expression is:

```
match value {
    pattern => expr,
    ...
}
```

The comma after an arm may be dropped if the expr is a block.

Rust checks the given value against each pattern in turn, starting with the first. When a pattern matches, the corresponding expr is evaluated, and the match expression is complete; no further patterns are checked. At least one of the patterns must match. Rust prohibits match expressions that do not cover all possible values:

```
let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11,
};  // error: nonexhaustive patterns
```

All blocks of an if expression must produce values of the same type:

```
let suggested_pet =
    if with_wings { Pet::Buzzard } else { Pet::Hyena };  // ok

let favorite_number =
    if user.is_hobbit() { "eleventy-one" } else { 9 };  // error

let best_sports_team =
    if is_hockey_season() { "Predators" };  // error
```

(The last example is an error because in July, the result would be ().)

Similarly, all arms of a match expression must have the same type:

```
let suggested_pet =
    match favorites.element {
        Fire => Pet::RedPanda,
        Air => Pet::Buffalo,
        Water => Pet::Orca,
        _ => None,   // error: incompatible types
    };
```

if let

There is one more `if` form, the `if let` expression:

```
if let pattern = expr {
    block1
} else {
    block2
}
```

The given `expr` either matches the `pattern`, in which case `block1` runs, or doesn't match, and `block2` runs. Sometimes this is a nice way to get data out of an `Option` or `Result`:

```
if let Some(cookie) = request.session_cookie {
    return restore_session(cookie);
}

if let Err(err) = show_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    politely_accuse_user_of_being_a_robot();
} else {
    session.mark_as_human();
}
```

It's never strictly *necessary* to use `if let`, because `match` can do everything `if let` can do. An `if let` expression is shorthand for a `match` with just one pattern:

```
match expr {
    pattern => { block1 }
    _ => { block2 }
}
```

let else

When there is only one acceptable pattern, Rust provides a convenient shorthand, the `let ... else` expression:

```
let pattern: type = expr else {
    divergent block
}
```

For instance, application code often needs to open a particular file in order to work, and exit otherwise. With `let ... else`, this can be written concisely as:

```
let Ok(config_file) = File::open(&config_path) else {
    panic!("Unable to open config file {}.",
config_path.display());
};
let config = parse_config(config_file);
```

The else expression must always be something that ends the flow of execution, such as a panic, a `return`, or an infinite loop. (This is called “diverging”, and is discussed further below.) `let ... else` is convenient for many common control flow patterns, but like `if let`, it is always possible to do the same thing via `match`:

```
let name = match expr {
    pattern => { convergent block }
    _ => { divergent block }
}
```

Loops

There are four looping expressions:

```
while condition {
    block
}

while let pattern = expr {
    block
}

loop {
    block
}


for pattern in iterable {
    block
}
```

Loops are expressions in Rust, but the value of a `while` or `for` loop is always `()`, so their value isn't very useful. A `loop` expression can produce a value via the `break` keyword.

A `while` loop behaves exactly like the C equivalent, except that, again, the `condition` must be of the exact type `bool`.

The `while let` loop is analogous to `if let`. At the beginning of each loop iteration, the value of `expr` either matches the given `pattern`, in which case the block runs, or doesn't, in which case the loop exits.

Use `loop` to write infinite loops. It executes the `block` repeatedly forever (or until a `break` or `return` is reached or the thread panics).

A `for` loop evaluates the `iterable` expression and then evaluates the `block` once for each value in the resulting iterator. Many types can be iterated over, including all the standard collections like `Vec` and `HashMap`. The standard C `for` loop:

```
for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}
```

is written like this in Rust:

```
for i in 0..20 {
    println!("{i}");
}
```

As in C, the last number printed is `19`.

The `..` operator produces a *range*, a simple struct with two fields: `start` and `end`. `0..20` is the same as `std::ops::Range { start: 0, end: 20 }`. Ranges can be used with `for` loops because `Range` is an iterable type: it implements the `std::iter::IntoIterator` trait, which we'll discuss in [Link to Come]. The standard collections are all iterable, as are arrays and slices.

In keeping with Rust's move semantics, a `for` loop over a value consumes the value:

```
let strings: Vec<String> = error_messages();
for s in strings {          // each String is moved into s
here...
    println!("{s}");
}                           // ...and dropped here
println!("{} error(s)", strings.len());  // error: use of moved
value
```

This can be inconvenient. The easy remedy is to loop over a reference to the collection instead. The loop variable, then, will be a reference to each item in the collection:

```
for s in &strings {         // `strings` is only borrowed here...
    println!("{s}");
}                           // ...nothing is moved or deallocated
println!("{} error(s)", strings.len());  // ok
```

Here the type of `&strings` is `&Vec<String>`, and the type of `s` is `&String`.

Iterating over a `mut` reference provides a `mut` reference to each element:

```
let mut strings = error_messages();
for s in &mut strings {     // the type of s is &mut String
    s.push('\n');           // add a newline to each string
}
```

[Link to Come] covers `for` loops in greater detail and shows many other ways to use iterators.

break and continue Expressions

A `break` expression exits an enclosing loop. (In Rust, `break` works only in loops and labeled blocks. It is not necessary in `match` expressions, which are unlike `switch` statements in this regard.)

Within the body of a `loop`, you can give `break` an expression, whose value becomes that of the loop:

```
// Each call to `next_line` returns either `Some(line)`, where
// `line` is a line of input, or `None`, if we've reached the end
of
// the input. Return the first line that starts with "answer: ".
// Otherwise, return "answer: nothing".
let answer = loop {
    if let Some(line) = next_line() {
        if line.starts_with("answer: ") {
            break line;
        }
    } else {
        break "answer: nothing";
    }
};
```

A `break` expression can only take a value in a `loop` (or a labeled block), unlike `for` and `while` loops, where `break` is only used to end the loop early. All the `break` expressions within a `loop` must produce values with the same type, which becomes the type of the `loop` itself.

A `continue` expression jumps to the next loop iteration:

```
// Read some data, one line at a time.
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // Jump back to the top of the loop and
        // move on to the next line of input.
        continue;
    }
    ...
}
```

In a `for` loop, `continue` advances to the next value in the collection. If there are no more values, the loop exits. Similarly, in a `while` loop, `continue` rechecks the loop condition. If it's now false, the loop exits.

A loop can be labeled with a lifetime, just like a block. In the following example, `'search:` is a label for the outer `for` loop. Thus, `break 'search` exits that loop, not the inner loop:

```
'search:
for room in apartment {
    for spot in room.hiding_spots() {
        if spot.contains(keys) {
            println!("Your keys are {spot} in the {room}.");
            break 'search;
        }
    }
}
```

A `break` can have both a label and a value expression:

```
// Find the square root of the first perfect square
// in the series.
let sqrt = 'outer: loop {
    let n = next_number();
    for i in 1.. {
        let square = i * i;
        if square == n {
            // Found a square root.
            break 'outer i;
        }
        if square > n {
            // `n` isn't a perfect square, try the next
            break;
        }
    }
};
```

Labels can also be used with `continue`.

return Expressions

A `return` expression exits the current function, returning a value to the caller.

`return` without a value is shorthand for `return ()`:

```
fn f() {     // return type omitted: defaults to ()
    return;  // return value omitted: defaults to ()
}
```

Functions don't have to have an explicit `return` expression. The body of a function works like a block expression: if the last expression isn't followed by a semicolon, its value is the function's return value. In fact, this is the preferred way to supply a function's return value in Rust.

But this doesn't mean that `return` is useless, or merely a concession to users who aren't experienced with expression languages. Like a `break` expression, `return` can abandon work in progress. For example, in

Chapter 1, we used the ? operator to check for errors after calling a function that can fail:

```
let output = File::create(filename)?;
```

We explained that this is shorthand for a match expression:

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err),
};
```

This code starts by calling File::create(filename). If that returns Ok(f), then the whole match expression evaluates to f, so f is stored in output, and we continue with the next line of code following the match.

Otherwise, we'll match Err(err) and hit the return expression. When that happens, it doesn't matter that we're in the middle of evaluating a match expression to determine the value of the variable output. We abandon all of that and exit the enclosing function, returning whatever error we got from File::create().

We'll cover the ? operator more completely in [Link to Come].

Why Rust Has loop

Several pieces of the Rust compiler analyze the flow of control through your program:

- Rust checks that every path through a function returns a value of the expected return type. To do this correctly, it needs to know whether it's possible to reach the end of the function.
- Rust checks that local variables are never used uninitialized. This entails checking every path through a function to make sure there's

no way to reach a place where a variable is used without having already passed through code that initializes it.

- Rust warns about unreachable code. Code is unreachable if *no* path through the function reaches it.

These are called *flow-sensitive* analyses. They are nothing new; Java has had a “definite assignment” analysis, similar to Rust’s, for years.

When enforcing this sort of rule, a language must strike a balance between simplicity, which makes it easier for programmers to figure out what the compiler is talking about sometimes, and cleverness, which can help eliminate false warnings and cases where the compiler rejects a perfectly safe program. Rust went for simplicity. Its flow-sensitive analyses do not examine loop conditions at all, instead simply assuming that any condition in a program can be either true or false.

This causes Rust to reject some safe programs:

```
fn wait_for_process(process: &mut Process) -> i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
}  // error: mismatched types: expected i32, found ()
```

The error here is bogus. This function only exits via the `return` statement, so the fact that the `while` loop doesn’t produce an `i32` is irrelevant.

The `loop` expression is offered as a “say-what-you-mean” solution to this problem.

Rust’s type system is affected by control flow, too. Earlier we said that all branches of an `if` expression must have the same type. But it would be silly to enforce this rule on blocks that end with a `break` or `return` expression, an infinite `loop`, or a call to `panic!()` or `std::process::exit()`. What all those expressions have in common is that they never finish in the usual way, producing a value. A `break` or `return` exits the current block abruptly, a `loop` without a `break` never finishes at all, and so on.

So in Rust, these expressions don't have a normal type. Expressions that don't finish normally are assigned the special type `!`, and they're exempt from the rules about types having to match. You can see `!` in the function signature of `std::process::exit()`:

```
fn exit(code: i32) -> !
```

The `!` means that `exit()` never returns. It's a *divergent function*.

You can write divergent functions of your own using the same syntax, and this is perfectly natural in some cases:

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) ->
! {
    socket.listen();
    loop {
        let s = socket.accept();
        handler.handle(s);
    }
}
```

Of course, Rust then considers it an error if the function can return normally.

With these building blocks of large-scale control flow in place, we can move on to the finer-grained expressions typically used within that flow, like function calls and arithmetic operators.

Function and Method Calls

The syntax for calling functions and methods is the same in Rust as in many other languages:

```
let x = gcd(1302, 462);  // function call

let room = player.location();  // method call
```

In the second example here, `player` is a variable of the made-up type `Player`, which has a made-up `.location()` method. (We'll show how to define your own methods when we start talking about user-defined types in [Link to Come].)

Rust usually makes a sharp distinction between references and the values they refer to. If you pass a `&i32` to a function that expects an `i32`, that's a type error. You'll notice that the `.` operator relaxes those rules a bit. In the method call `player.location()`, `player` might be a `Player`, a reference of type `&Player`, or a smart pointer of type `Box<Player>` or `Rc<Player>`. The `.location()` method might take the player either by value or by reference. The same `.location()` syntax works in all cases, because Rust's `.` operator automatically dereferences `player` or borrows a reference to it as needed.

A third syntax is used for calling type-associated functions, like `Vec::new()`:

```
let mut numbers = Vec::new();  // type-associated function call
```

These are similar to static methods in object-oriented languages: ordinary methods are called on values (like `my_vec.len()`), and type-associated functions are called on types (like `Vec::new()`).

Naturally, method calls can be chained:

```
// From the Actix-based web server in Chapter 2:
server
    .bind("127.0.0.1:3000").expect("error binding server to
address")
    .run().expect("error running server");
```

One quirk of Rust syntax is that in a function call or method call, the usual syntax for generic types, `Vec<T>`, does not work:

```
return Vec<i32>::with_capacity(1000);  // error: something about
chained comparisons

let ramp = (0..n).collect<Vec<i32>>();  // same error
```

The problem is that in expressions, `<` is the less-than operator. The Rust compiler helpfully suggests writing `::<T>` instead of `<T>` in this case, and that solves the problem:

```
return Vec::<i32>::with_capacity(1000);  // ok, using ::<

let ramp = (0..n).collect::<Vec<i32>>();  // ok, using ::<
```

The symbol `::<...>` is affectionately known in the Rust community as the *turbofish*.

Alternatively, it is often possible to drop the type parameters and let Rust infer them:

```
return Vec::with_capacity(10);  // ok, if the fn return type is
Vec<i32>

let ramp: Vec<i32> = (0..n).collect();  // ok, variable's type is
given
```

It’s considered good style to omit the types whenever they can be inferred.

Fields and Elements

The fields of a struct are accessed using familiar syntax. Tuples are the same except that their fields have numbers rather than names:

```
game.black_pawns   // struct field
coords.1           // tuple field
```

If the value to the left of the dot is a reference or smart pointer type, it is automatically dereferenced, just as for method calls.

Square brackets access the elements of an array, slice, or vector:

```
pieces[i]          // array element
```

The value to the left of the brackets is automatically dereferenced.

Expressions like these three are called *lvalues*, because they can appear on the left side of an assignment:

```
game.black_pawns = 0x00ff0000_00000000_u64;
coords.1 = 0;
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

Of course, this is permitted only if `game`, `coords`, and `pieces` are declared as `mut` variables.

Extracting a slice from an array or vector is straightforward:

```
let second_half = &game_moves[midpoint..end];
```

Here `game_moves` may be either an array, a slice, or a vector; the result, regardless, is a borrowed slice of length `end - midpoint`. `game_moves` is considered borrowed for the lifetime of `second_half`.

The `..` operator allows either operand to be omitted; it produces up to four different types of object depending on which operands are present:

```
..      // RangeFull
a..     // RangeFrom { start: a }
```

```
..b       // RangeTo { end: b }
a..b      // Range { start: a, end: b }
```

The latter two forms are *end-exclusive* (or *half-open*): the end value is not included in the range represented. For example, the range `0..3` includes the numbers `0`, `1`, and `2`.

The `..=` operator produces *end-inclusive* (or *closed*) ranges, which do include the end value:

```
..=b      // RangeToInclusive { end: b }
a..=b     // RangeInclusive::new(a, b)
```

For example, the range `0..=3` includes the numbers `0`, `1`, `2`, and `3`.

Only ranges that include a start value are iterable, since a loop must have somewhere to start. But in array slicing, all six forms are useful. If the start or end of the range is omitted, it defaults to the start or end of the data being sliced.

So an implementation of quicksort, the classic divide-and-conquer sorting algorithm, might look, in part, like this:

```
fn quicksort<T: Ord>(slice: &mut [T]) {
    if slice.len() <= 1 {
        return;  // Nothing to sort.
    }

    // Partition the slice into two parts, front and back.
    let pivot_index = partition(slice);

    // Recursively sort the front half of `slice`.
    quicksort(&mut slice[.. pivot_index]);

    // And the back half.
    quicksort(&mut slice[pivot_index + 1 ..]);
}
```

Reference Operators

The address-of operators, `&` and `&mut`, are covered in Chapter 4.

The unary `*` operator is used to access the value pointed to by a reference. As we've seen, Rust automatically follows references when you use the `.` operator to access a field or method, so the `*` operator is necessary only when we want to read or write the entire value that the reference points to.

For example, sometimes an iterator produces references, but the program needs the underlying values:

```
let padovan: Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```

In this example, the type of `elem` is `&u64`, so `*elem` is a `u64`.

Arithmetic, Bitwise, Comparison, and Logical Operators

Rust's binary operators are like those in many other languages. To save time, we assume familiarity with one of those languages, and focus on the few points where Rust departs from tradition.

Rust has the usual arithmetic operators, `+`, `-`, `*`, `/`, and `%`. As mentioned in Chapter 2, integer overflow is detected, and causes a panic, in debug builds. The standard library provides methods like `a.wrapping_add(b)` for unchecked arithmetic.

Integer division rounds toward zero, and dividing an integer by zero triggers a panic even in release builds. Integers have a method `a.checked_div(b)` that returns an `Option` (`None` if `b` is zero) and never panics.

Unary − negates a number. It is supported for all the numeric types except unsigned integers. There is no unary + operator.

```
println!("{}", -100);     // -100
println!("{}", -100u32);  // error: can't apply unary `-` to type
`u32`
println!("{}", +100);     // error: leading `+` is not supported
```

As in C, `a % b` computes the signed remainder, or modulus, of division rounding toward zero. The result has the same sign as the lefthand operand. Note that `%` can be used on floating-point numbers as well as integers:

```
let x = 1234.567 % 10.0;  // approximately 4.567
```

Rust also inherits C's bitwise integer operators, `&`, `|`, ^, <<, and >>. However, Rust uses `!` instead of ~ for bitwise NOT:

```
let hi: u8 = 0xe0;
let lo = !hi;  // 0x1f
```

This means that `!n` can't be used on an integer `n` to mean "n is zero." For that, write `n == 0`.

Bit shifting is always sign-extending on signed integer types and zero-extending on unsigned integer types. Since Rust has unsigned integers, it does not need an unsigned shift operator, like Java's >>> operator.

Bitwise operations have higher precedence than comparisons, unlike C, so if you write `x & BIT != 0`, that means `(x & BIT) != 0`, as you probably intended. This is much more useful than C's interpretation, `x & (BIT != 0)`, which tests the wrong bit!

Rust's comparison operators are ==, !=, <, <=, >, and >=. The two values being compared must have the same type.

Rust also has the two short-circuiting logical operators `&&` and `||`. Both operands must have the exact type `bool`.

Assignment

The = operator can be used to assign to `mut` variables and their fields or elements. But assignment is not as common in Rust as in other languages, since variables are immutable by default.

As described in Chapter 3, if the value has a non-`Copy` type, assignment *moves* it into the destination. Ownership of the value is transferred from the source to the destination. The destination's prior value, if any, is dropped.

Compound assignment is supported:

```
total += item.price;
```

This is equivalent to `total = total + item.price;`. Other operators are supported too: `-=`, `*=`, and so forth. The full list is given in Table 5-1, earlier in this chapter.

Unlike C, Rust doesn't support chaining assignment: you can't write `a = b = 3` to assign the value `3` to both `a` and `b`. Assignment is rare enough in Rust that you won't miss this shorthand.

Rust does not have C's increment and decrement operators `++` and `--`.

Type Casts

Converting a value from one type to another usually requires an explicit cast in Rust. Casts use the `as` keyword:

```
let x = 17;              // x is type i32
let index = x as usize;  // convert to usize
```

Several kinds of casts are permitted:

- Numbers may be cast from any of the built-in numeric types to any other.

  Casting an integer to another integer type is always well-defined. Converting to a narrower type results in truncation. A signed integer cast to a wider type is sign-extended, an unsigned integer is zero-extended, and so on. In short, there are no surprises.

  Converting from a floating-point type to an integer type rounds toward zero: the value of `-1.99 as i32` is `-1`. If the value is too large to fit in the integer type, the cast produces the closest value that the integer type can represent: the value of `1e6 as u8` is `255`.

- Values of type `bool` or `char`, or of a C-like `enum` type, may be cast to any integer type. (We'll cover enums in [Link to Come].)

  Casting in the other direction is not allowed, as `bool`, `char`, and `enum` types all have restrictions on their values that would have to be enforced with run-time checks. For example, casting a `u16` to type `char` is banned because some `u16` values, like `0xd800`, correspond to Unicode surrogate code points and therefore would not make valid `char` values. There is a standard method, `std::char::from_u32()`, which performs the run-time check and returns an `Option<char>`; but more to the point, the need for this kind of conversion has grown rare. We typically convert whole strings or streams at once, and algorithms on Unicode text are often nontrivial and best left to libraries.

  As an exception, a `u8` may be cast to type `char`, since all integers from 0 to 255 are valid Unicode code points for `char` to hold.

- Some casts involving unsafe pointer types are also allowed. See [Link to Come].

We said that a conversion *usually* requires a cast. A few conversions involving reference types are so straightforward that the language performs them even without a cast. One trivial example is converting a `mut` reference to a non-`mut` reference.

Several more significant automatic conversions can happen, though:

- Values of type `&String` auto-convert to type `&str` without a cast.
- Values of type `&Vec<i32>` auto-convert to `&[i32]`.
- Values of type `&Box<Chessboard>` auto-convert to `&Chessboard`.

These are called *deref coercions*, because they apply to types that implement the `Deref` built-in trait. The purpose of `Deref` coercion is to make smart pointer types, like `Box`, behave as much like the underlying value as possible. Using a `Box<Chessboard>` is mostly just like using a plain `Chessboard`, thanks to `Deref`.

User-defined types can implement the `Deref` trait, too. When you need to write your own smart pointer type, see [Link to Come].

Closures

Rust has *closures*, lightweight function-like values. A closure usually consists of an argument list, given between vertical bars, followed by an expression:

```
let is_even = |x| x % 2 == 0;
```

Rust infers the argument types and return type. You can also write them out explicitly, as you would for a function. If you do specify a return type, then the body of the closure must be a block, for the sake of syntactic sanity:

```
let is_even = |x: u64| -> bool x % 2 == 0;  // error

let is_even = |x: u64| -> bool { x % 2 == 0 };  // ok
```

Calling a closure uses the same syntax as calling a function:

```
assert_eq!(is_even(14), true);
```

Closures are one of Rust’s most delightful features, and there is a great deal more to be said about them. We shall say it in [Link to Come].

Onward

Expressions are what we think of as “running code.” They’re the part of a Rust program that compiles to machine instructions. Yet they are a small fraction of the whole language.

The same is true in most programming languages. The first job of a program is to run, but that’s not its only job. Programs have to communicate. They have to be testable. They have to stay organized and flexible so that they can continue to evolve. They have to interoperate with code and services built by other teams. And even just to run, programs in a statically typed language like Rust need some more tools for organizing data than just tuples and arrays.

Coming up, we’ll spend several chapters talking about features in this area: modules and crates, which give your program structure, and then structs and enums, which do the same for your data.

First, we’ll dedicate a few pages to the important topic of what to do when things go wrong.

About the Authors

Jim Blandy has been programming since 1981 and writing free software since 1990. He has been the maintainer of GNU Emacs and GNU Guile, and a maintainer of GDB, the GNU Debugger. He is one of the original designers of the Subversion version control system. Jim now works on Firefox's graphics and rendering for Mozilla.

Jason Orendorff works on undisclosed Rust projects at GitHub. He previously worked on the SpiderMonkey JavaScript engine at Mozilla. He is interested in grammar, baking, time travel, and helping people learn about complicated topics.

Leonora Tindall is a type system enthusiast and software engineer who uses Rust, Elixir, and other advanced languages to build robust and resilient systems software in high-impact areas like healthcare and data ownership. She works on a variety of open source projects, from genetic algorithms that evolve programs in strange languages to the Rust core libraries and crate ecosystem, and enjoys the experience of contributing to supportive and diverse community projects. In her free time, Leonora builds electronics for audio synthesis and is an avid radio hobbyist. Her love of hardware extends to her software engineering practice as well. She has built applications software for LoRa radios in Rust and Python and uses software and DIY hardware to create experimental electronic music on a Eurorack synthesizer.