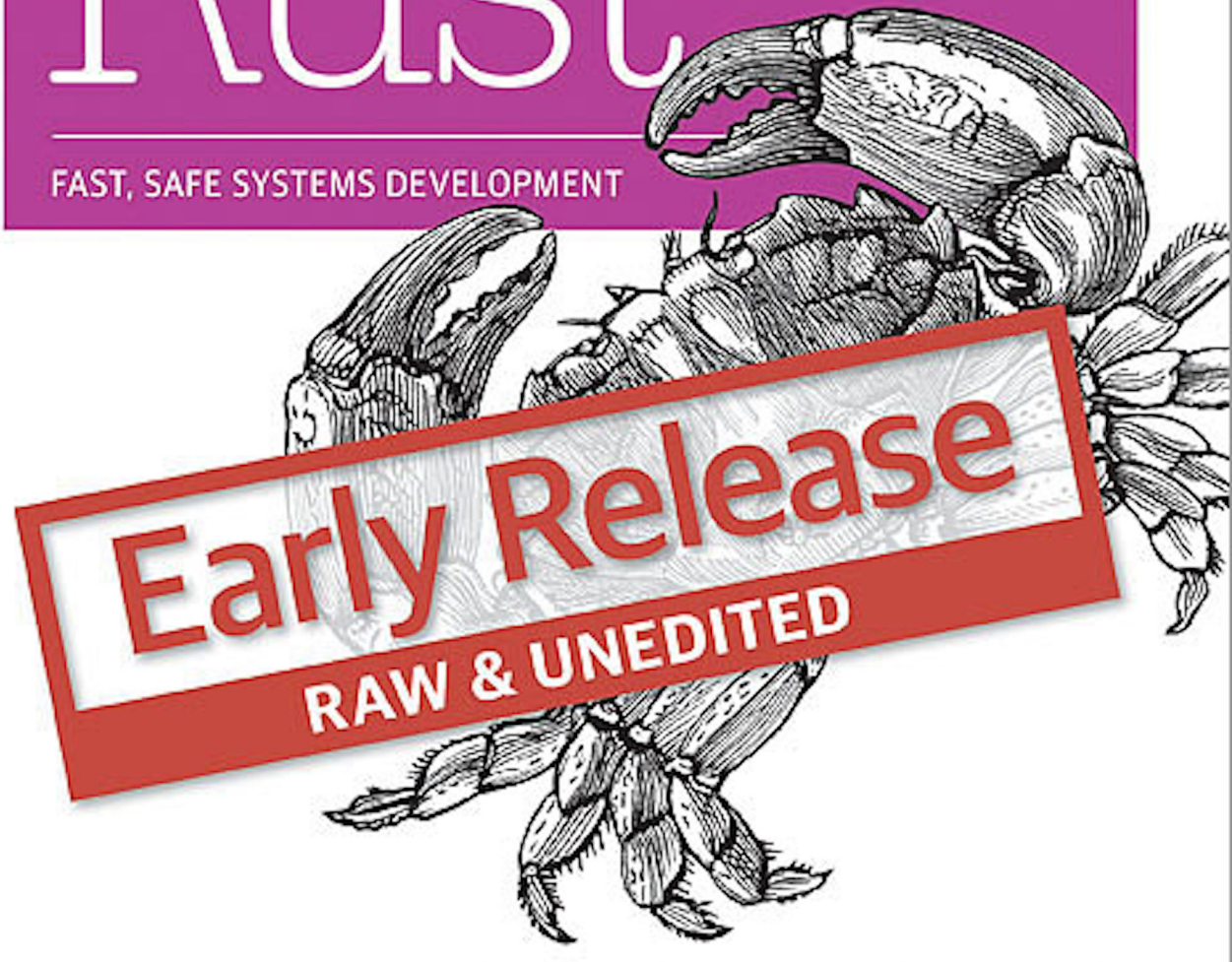
O'REILLY®

Programming Rust

FAST, SAFE SYSTEMS DEVELOPMENT



Early Release

RAW & UNEDITED

Jim Blandy & Jason Orendorff

Programming Rust
by Jim Blandy and Jason Orendorff

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Brian MacDonald and Meghan Blanchette
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2016: First Edition

Revision History for the First Edition
2016-03-07: First Early Release
2016-03-31: Second Early Release
2016-04-28: Third Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491927212* for release details.

978-1-491-92721-2

[FILL IN]

Programming Rust

Jim Blandy and Jason Orendorff

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Table of Contents

CHAPTER 1
Why Rust?

Systems programming languages have come a long way in the 50 years since we started using high-level languages to write operating systems, but two thorny problems in particular have proven difficult to crack:

- It's difficult to write secure code. It's common for security exploits to leverage bugs in the way C and C++ programs handle memory, and has been so for decades. In 1988, the Morris virus, the first Internet virus to be carefully analyzed, took advantage of a buffer overflow bug to propagate itself from one machine to the next.
- It's very difficult to write multithreaded code, which is the only way to exploit the abilities of modern machines. Each new generation of hardware brings us, instead of faster processors, more of them; now even midrange mobile devices have multiple cores. Taking advantage of this entails writing multithreaded code, but even experienced programmers approach that task with caution: concurrency introduces broad new classes of bugs, and can make ordinary bugs much harder to reproduce.

These are the problems Rust was made to address.

Rust is a new systems programming language designed by Mozilla. Like C and C++, Rust gives the developer fine control over the use of memory, and maintains a close relationship between the primitive operations of the language and those of the machines it runs on, helping developers anticipate their code's costs. Rust shares the ambitions Bjarne Stroustrup articulates for C++ in his paper "Abstraction and the C++ machine model":

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

To these Rust adds its own goals of memory safety and data-race-free concurrency.

The key to meeting all these promises is Rust's novel system of ownership, moves, and borrows, checked at compile time and carefully designed to complement Rust's flexible static type system. The ownership system establishes a clear lifetime for each value, making garbage collection unnecessary in the core language, and enabling sound but flexible interfaces for managing other sorts of resources like sockets and file handles.

These same ownership rules also form the foundation of Rust's trustworthy concurrency model. Most languages leave the relationship between a mutex and the data it's meant to protect to the comments; Rust can actually check at compile time that your code locks the mutex while it accesses the data. Most languages admonish you to be sure not to use a data structure yourself after you've sent it via a channel to another thread; Rust checks that you don't. Rust is able to prevent data races at compile time.

Mozilla and Samsung have been collaborating on an experimental new web browser engine named Servo, written in Rust. Servo's needs and Rust's goals are well matched: as programs whose primary use is handling untrusted data, browsers must be secure; and as the Web is the primary interactive medium of the modern Net, browsers must perform well. Servo takes advantage of Rust's sound concurrency support to exploit as much parallelism as its developers can find, without compromising its stability. As of this writing, Servo is roughly 100,000 lines of code, and Rust has adapted over time to meet the demands of development at this scale.

Type Safety

But what do we mean by "type safety"? Safety sounds good, but what exactly are we being kept safe from?

Here's the definition of "undefined behavior" from the 1999 standard for the C programming language, known as "C99":

> 3.4.3
>
> **undefined behavior**
>
> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

Consider the following C program:

```
int main(int argc, char **argv) {
  unsigned long a[1];
  a[3] = 0x7ffff7b36cebUL;
  return 0;
}
```

According to C99, because this program accesses an element off the end of the array `a`, its behavior is undefined, meaning that it can do anything whatsoever. This morning, running the program on Jim's laptop produces the output:

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

Then it crashes. This computer don't even have a `.netrc` file.

The machine code the C compiler generated for this `main` function happens to place the array `a` on the stack three words before the return address, so storing `0x7ffff7b36cebUL` in `a[3]` changes poor `main`'s return address to point into the midst of code in the C standard library that consults one's `.netrc` file for a password. When `main` returns, execution resumes not in `main`'s caller, but at the machine code for these lines from the library:

```
warnx(_("Error: .netrc file is readable by others."));
warnx(_("Remove password or make file unreadable by others."));
            goto bad;
```

In allowing an array reference to affect the behavior of a subsequent `return` statement, the C compiler is fully standards-compliant. An "undefined" operation doesn't just produce an unspecified result: it is allowed to cause the program to do *anything at all*.

The C99 standard grants the compiler this carte blanche to allow it to generate faster code. Rather than making the compiler responsible for detecting and handling odd behavior like running off the end of an array, the standard makes the C programmer responsible for ensuring those conditions never arise in the first place.

Empirically speaking, we're not very good at that. The 1988 Morris virus had various ways to break into new machines, one of which entailed tricking a server into executing an elaboration on the technique shown above; the "undefined behavior" produced in that case was to download and run a copy of the virus. (Undefined behavior is often sufficiently predictable in practice to build effective security exploits from.) The same class of exploit remains in widespread use today. While a student at the University of Utah, researcher Peng Li modified C and C++ compilers to make the programs they translated report when they executed certain forms of undefined behavior. He found that nearly all programs do, including those from well-respected projects that hold their code to high standards.

In light of that example, let's define some terms. If a program has been written so that no possible execution can exhibit undefined behavior, we say that program is *well defined*. If a language's compile time checks ensure that every program is well defined, we say that language is *type safe*.

C and C++ are not type safe: the program shown above has no type errors, yet exhibits undefined behavior. By contrast, Python is type safe. Python is willing to spend

processor time to detect and handle out-of-range array indices in a friendlier fashion than C:

```
>>> a = [0]
>>> a[3] = 0x7ffff7b36ceb
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

Python raised an exception, which is not undefined behavior: the Python documentation specifies that the assignment to `a[3]` should raise an `IndexError` exception, as we saw. As a type-safe language, Python assigns a meaning to every operation, even if that meaning is just to raise an exception. Java, JavaScript, Ruby, and Haskell are also type safe: every program those languages will accept at all is well defined.

Note that being type safe is mostly independent of whether a language checks types at compile time or at run time: C checks at compile time, and is not type safe; Python checks at runtime, and is type safe. Any practical type-safe language must do at least some checks (array bounds checks, for example) at runtime.

It is ironic that the dominant systems programming languages, C and C++, are not type safe, while most other popular languages are. Given that C and C++ are meant to be used to implement the foundations of a system, entrusted with implementing security boundaries and placed in contact with untrusted data, type safety would seem like an especially valuable quality for them to have.

This is the decades-old tension Rust aims to resolve: it is both type safe and a systems programming language. Rust is designed for implementing those fundamental system layers that require performance and fine-grained control over resources, yet still guarantees the basic level of predictability that type safety provides. We'll look at how Rust manages this unification in more detail in later parts of this book.

Type safety might seem like a modest promise, but it starts to look like a surprisingly good deal if you consider its consequences for multithreaded programming. Concurrency is notoriously difficult to use correctly in C and C++; developers usually turn to concurrency only when single-threaded code has proven unable to achieve the performance they need. But Rust's particular form of type safety guarantees that concurrent code is free of data races, catching any misuse of mutexes or other synchronization primitives at compile time. In Rust, you can exploit parallelism without worrying that you've made your code impossible for any but the most accomplished programmers to work on.

Rust does provide for *unsafe code*, functions or lexical blocks that the programmer has marked with the `unsafe` keyword, within which some of Rust's type rules are relaxed. In an unsafe block, you can use unrestricted pointers, treat blocks of raw memory as if they contained any type you like, call any C function you want, use inline assembly language, and so on.

Whereas in ordinary Rust code the compiler guarantees your program is well defined, in unsafe blocks it becomes the programmer's responsibility to avoid undefined behavior, as in C and C++. As long as the programmer succeeds at this, unsafe blocks don't affect the safety of the rest of the program. Rust's standard library uses unsafe blocks to implement features that are themselves safe to use, but which the compiler isn't able to recognize as such on its own.

The great majority of programs do not require unsafe code, and Rust programmers generally avoid it, since it must be reviewed with special care. Except where explicitly noted otherwise, you may assume that this book is discussing the safe portion of the language.

CHAPTER 2
A Tour of Rust

In this chapter we'll look at several short programs to see how Rust's syntax, types, and semantics fit together to support safe, concurrent, and efficient code. We'll walk through the process of downloading and installing Rust; show some simple mathematical code; try out a web server based on a third-party library; and use multiple threads to speed up the process of traversing a directory tree.

Downloading and installing Rust

The best way to install Rust, as of this writing, is to visit the language's web site, `https://www.rust-lang.org`, and follow the instructions for downloading and installing the language. The site provides pre-built packages for Linux, Macintosh OSX, and Windows. Install the package in the usual way for your system; on Linux, you may need to unpack a `tar` file and follow the instructions in the `README.md` file.

Once you've completed the installation, you should have three new commands available at your command line:

```
$ cargo --version
cargo 0.4.0-nightly (553b363 2015-08-03) (built 2015-08-02)
$ rustc --version
rustc 1.3.0 (9a92aaf19 2015-09-15)
$ rustdoc --version
rustdoc 1.3.0 (9a92aaf19 2015-09-15)
$
```

In the command-line interactions in this book, the `$` character at the beginning of a line is the command prompt; on Windows, the command line would be `C:\>` or something similar. The command appears to the right, and the program's output appears on the following lines.

Here we've 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 --bin hello
```

This command creates a new package directory named `hello`, and the `--bin` flag directs Cargo to prepare this as an executable, not a library. 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 ..
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
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"
authors = ["Jim Blandy <jimb@red-bean.com>"]
```

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 ???.

Cargo has set up our package for use with the `git` version control system, creating a `.git` metadata subdirectory, and a `.gitignore` file. Cargo also supports the Mercu-

rial version control system (sometimes known as `hg`). By passing the flag `--vcs none` to the `cargo` command, you can request that no version control be prepared.

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 started 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. 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 (file:///home/jimb/rust/hello)
     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
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 native
$ ../target/debug/hello
Hello, world!
$
```

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

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
$
```

A simple function

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:

```
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 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`, a unsigned 64-bit integer. The function's return type appears after the `->`: 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 eight-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.

Normally, once a variable's value has been established, it can't be changed, but placing the `mut` keyword (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; requiring the `mut` keyword on those that do flags them for careful attention from the reader.

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, crashes the program with a helpful message including the source location of the failing check; this kind of controlled crash 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.

The heart of our function is a `while` loop containing an `if` statement and an assignment. Unlike C and C++, Rust does not require parenthesis 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've done here. If we had wanted to spell out `t`'s type, we could have written:

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

Rust has a `return` statement, but we didn't need one to return our value here. In Rust, a block surrounded by curly braces can be an expression; its value is that of the last expression it contains. The body of our function is such a block, and its last expression is `n`, so that's our return value. Likewise, `if` is an expression whose value is that of the branch that was taken. Rust has no need for a separate `?:` conditional operator as in C; one just writes the `if-else` structure right into the expression.

Writing and running unit tests

Rust has simple support for testing built into the language. To test our `gcd` function, we can write:

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(2 * 5 * 11 * 17,
                   3 * 7 * 13 * 19),
               1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                   3 * 7 * 11 * 13 * 19),
               3 * 11);
}
```

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. Let's assume we've edited our `gcd` and `test_gcd` definitions into the `hello` package we created at the beginning of the chapter. If our current directory is somewhere within the package's subtree, we can run the tests as follows:

```
$ cargo test
   Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
     Running /home/jimb/rust/hello/target/debug/hello-2375a82d9e9673d7

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

```
$
```

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, somewhat like attributes in C# or annotations in Java. They're used to control compiler warnings and code style checks, include code conditionally (like `#if` in C and C++), tell Rust how to interact with code written in other languages, and much else. We'll see more examples of attributes as we go.

Handling command-line arguments

If we want our program to take a series of numbers as command-line arguments and print their greatest common divisor, we can replace the `main` function with the following:

```
use std::io::Write;
use std::str::FromStr;

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

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

    if numbers.len() == 0 {
        writeln!(std::io::stderr(), "Usage: gcd NUMBER ...").unwrap();
        std::process::exit(1);
    }

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

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

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

```
use std::io::Write;
use std::str::FromStr;
```

The `use` declarations bring the two *traits* `Write` and `FromStr` into scope. We'll cover traits in detail in Chapter 10, but for now, we'll simply say that a trait is a collection of methods a type can implement. Any type that implements the `Write` trait has a `write_fmt` method, which the `writeln!` macro uses. A type that implements `FromStr` has a `from_str` associated function; we'll use this method on `u64` to parse our command-line arguments.

```
fn main() {
```

Our `main` function doesn't return a value, so we can simply omit the `->` and 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. Our variable's type is `Vec<u64>`, a vector of `u64` values, but as before, we don't need to write it out: Rust will infer it for us. Since we intend to push numbers onto the end of this vector as we parse our command-line arguments, we use the `mut` keyword to make the vector mutable.

```
for arg in std::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::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.

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 `std::env::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. Each `Result` is one of two variants:

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

Almost every function in Rust's standard library that might encounter an error returns a `Result` value, carrying values of appropriate types in its `Ok` and `Err` variants. Functions that perform input or output or otherwise interact with the operating system all return `Result` types whose `Ok` variants carry successful results—a count of bytes transferred, a file opened, and so on—and whose `Err` variants carry an error code from the system.

We check the success of our parse using `Result`'s `expect` method. If the result is some `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 {
        writeln!(std::io::stderr(), "Usage: gcd NUMBER ...").unwrap();
        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 `writeln!` macro to write our error message to the standard error output stream, provided by `std::io::stderr()`. The `.unwrap()` call is a terse way to check that the attempt to print the error message did not itself fail; an `expect` call would work here too, but that's probably not worth it.

```
    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 of all, 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, mak-

ing 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 vector lasts until `numbers` goes out of scope, at the end of `main`; then Rust frees its memory. The `&` operator in `&numbers[1..]` borrows a *reference* to the vector's elements from the second onwards. 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 borrows; this is the next `u64` we want to pass to `gcd`.

Rust's rules for ownership and references are key to Rust's memory management and safe concurrency; we discuss them in detail in Chapter Chapter 4 and its companion, Chapter Chapter 5. 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 {:?} is {}",
             numbers, d);
```

Having iterated over `numbers`' elements, the program prints the results to the standard output stream. The `println!` macro takes a template string, substitutes formatted versions of the remaining arguments for the `{...}` forms as they appear in the template string, and writes the result to the standard output stream. The `{:?}` form requests that `println!` show us the "debugging" form of the corresponding argument. Most types in Rust can be printed this way; we use it here to print the vector of numbers. The `{}` form requests the "display" form of the argument; we use it here to print our final result, `d`.

Unlike C and C++, which require `main` to return zero if the program finished successfully, or a non-zero 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:

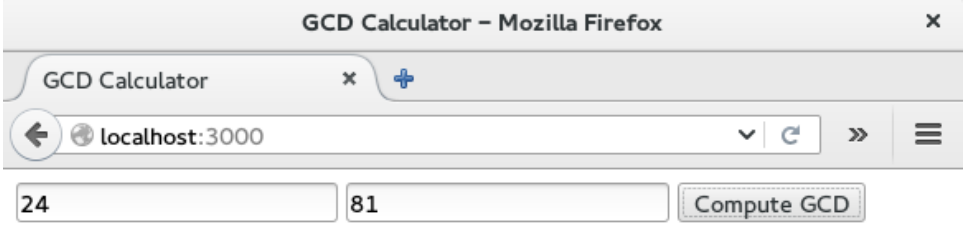
```
$ cargo run 42 56
   Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
     Running `/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
```

```
$ cargo run 83
     Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
Process didn't exit successfully: `/home/jimb/rust/hello/target/debug/hello` (exit code: 1)
$
```

A simple web server

One of Rust's strengths is the collection of library packages written by the Rust user community and freely available for anyone to use, many of which are published on the web site `https://crates.io`. The `cargo` command makes it easy to use a `crates.io` package from our own code: 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 the `iron` web framework, the `hyper` HTTP server, and various other crates on which they depend. Our web site will prompt the user for two numbers, and compute their greatest common divisor:



Web page offering to compute GCD

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

```
$ cargo new --bin iron-gcd
$ cd iron-gcd
$
```

Then, we'll edit our new project's `Cargo.toml` file to list the packages we want to use; its contents should be as follows:

```
[package]
name = "iron-gcd"
version = "0.1.0"
```

```
authors = ["Jim Blandy <jimb@red-bean.com>"]

[dependencies]
iron = "0.2.2"
mime = "0.1.0"
router = "0.0.15"
urlencoded = "0.2.0"
```

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. There may well be versions of these crates on `cargo.io` newer than those shown here, but by naming the specific versions we tested this code against, we can ensure the code will continue to compile even as new versions of the packages are published. We'll discuss version management in more detail in ???.

Note that we need only name those packages we'll use directly; `cargo` takes care of bringing in whatever other packages 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 `iron-gcd/src/main.rs`, we'll place the following text:

```
extern crate iron;
#[macro_use] extern crate mime;

use iron::prelude::*;
use iron::status;

fn main() {
    println!("Serving on http://localhost:3000...");
    Iron::new(get_form).http("localhost:3000").unwrap();
}

#[allow(unused_variables)]
fn get_form(request: &mut Request) -> IronResult<Response> {
    let mut response = Response::new();

    response.set_mut(status::Ok);
    response.set_mut(mime!(Text/Html; Charset=Utf8));
    response.set_mut(r#"
        <title>GCD Calculator</title>
        <form action="/gcd" method="post">
          <input type="text" name="n"/>
          <input type="text" name="n"/>
          <button type="submit">Compute GCD</button>
        </form>
    "#);

    Ok(response)
}
```

We start with two `extern crate` directives, which make the `iron` and `mime` crates that we cited in our `Cargo.toml` file available to our program. The `#[macro_use]` attribute before the `extern crate mime` item alerts Rust that we plan to use macros exported by this crate.

Next, we have `use` declarations to bring in some of those crates' bindings. The declaration `use iron::prelude::*` makes all the public bindings of the `iron::prelude` module directly visible in our own code. Generally, one should prefer to spell out the names one wishes to use, as we did for `iron::status`; but by convention, when a module is named `prelude`, that means that its exports are intended to provide the sort of general facilities that any user of the crate will probably need. So here, a wildcard `use` directive makes a bit more sense.

Our `main` function is simple: it prints a message reminding us how to connect to our server, calls `Iron::new` to create a server, and then sets it listening on TCP port 3000 on the local machine. We pass the `get_form` function to `Iron::new`, indicating that the server should use that function to handle all requests; we'll refine this shortly.

The `get_form` function itself takes a mutable reference, written `&mut`, to a `Request` value representing the HTTP request we've been called to handle. While this particular handler function never uses its `request` parameter, we'll see one later that does. For the time being, we put the attribute `#[allow(unused_variables)]` atop the function to prevent Rust from printing a warning message about it.

In the body of the function, we build a `Response` value, set its HTTP status, indicate the media type of the content returned (using the handy `mime!` macro), and finally supply the actual text of the response. Since the response text is several lines long, 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, and no escape sequences are recognized; we can always ensure the string ends where we want it to by supplying enough hash marks.

Our function's return type, `IronResult<Response>`, is another variant of the `Result` type we encountered earlier: here it is either `Ok(r)` for some successful `Response` value `r`, or `Err(e)` for some error value `e`. We construct our return value `Ok(response)` at the bottom of the function body, using the "last expression" syntax to implicitly establish the value of the entire body.

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 registry `https://github.com/rust-lang/crates.io-index`
 Downloading kernel32-sys v0.1.4
```

```
 Downloading winapi-build v0.1.1
 Downloading libressl-pnacl-sys v2.1.6
 Downloading mime v0.1.0
 Downloading pnacl-build-helper v1.4.10
 Downloading plugin v0.2.6
....
   Compiling hyper v0.6.14
   Compiling iron v0.2.2
   Compiling router v0.0.15
   Compiling persistent v0.0.7
   Compiling bodyparser v0.0.6
   Compiling urlencoded v0.2.0
   Compiling iron-gcd v0.1.0 (file:///home/jimb/iron-gcd)
     Running `target/debug/iron-gcd`
Serving on http://localhost:3000...
```

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

Unfortunately, clicking on "Compute GCD" doesn't do anything, other than navigate our browser to the URL `http://localhost:3000/gcd`, which then shows the same page; every URL on our server does. Let's fix that next, using the `Router` type to associate different handlers with different paths.

First, let's arrange to be able to use `Router` without qualification, by adding the following declarations to `iron-gcd/src/main.rs`:

```
extern crate router;
use router::Router;
```

Rust programmers typically gather all their `extern crate` and `use` declarations together towards 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. (Macro definitions and imports are an important exception to this rule; they must appear before they are are used.)

We can then modify our `main` function to read as follows:

```
fn main() {
    let mut router = Router::new();

    router.get("/", get_form);
    router.post("/gcd", post_gcd);

    println!("Serving on http://localhost:3000...");
    Iron::new(router).http("localhost:3000").unwrap();
}
```

We create a `Router`, establish handler functions for two specific paths, and then pass this `Router` as the request handler to `Iron::new`, yielding a web server that consults the URL path to decide which handler function to call.

Now we are ready to write our `post_gcd` function:

```
extern crate urlencoded;

use std::str::FromStr;
use urlencoded::UrlEncodedBody;

fn post_gcd(request: &mut Request) -> IronResult<Response> {
    let mut response = Response::new();

    let hashmap;
    match request.get_ref::<UrlEncodedBody>() {
        Err(e) => {
            response.set_mut(status::BadRequest);
            response.set_mut(format!("Error parsing form data: {:?}\n", e));
            return Ok(response);
        }
        Ok(map) => { hashmap = map; }
    }

    let unparsed_numbers;
    match hashmap.get("n") {
        None => {
            response.set_mut(status::BadRequest);
            response.set_mut(format!("form data has no 'n' parameter\n"));
            return Ok(response);
        }
        Some(nums) => { unparsed_numbers = nums; }
    }

    let mut numbers = Vec::new();
    for unparsed in unparsed_numbers {
        match u64::from_str(&unparsed) {
            Err(_) => {
                response.set_mut(status::BadRequest);
                response.set_mut(format!("Value for 'n' parameter not a number: {:?}\n", unparsed)
                return Ok(response);
            }
            Ok(n) => { numbers.push(n); }
        }
    }

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

    response.set_mut(status::Ok);
    response.set_mut(mime!(Text/Html; Charset=Utf8));
    response.set_mut(format!("The greatest common divisor of the numbers {:?} is <b>{}</b>\n",
                             numbers, d));
```

```
        Ok(response)
    }
```

The bulk of this function is a series of `match` statements, which will be unfamiliar to C, C++, Java, and JavaScript programmers, but a welcome sight to Haskell and OCaml developers. We've mentioned that a `Result` is either a value `Ok(s)` for some success value `s`, or `Err(e)` for some error value `e`. Given some `Result` `res`, we can check which variant it is and access whichever value it holds with a `match` statement of the form:

```
match res {
    Ok(success) => { ... },
    Err(error)  => { ... }
}
```

This is a conditional structure, like an `if` statement or a `switch` statement: if `res` is `Ok(v)`, then it runs the first branch, with `v` assigned to the variable `success`, which is local to its branch. Similarly, if `res` is `Err(e)`, it assigns `e` to `error`, and run the second branch. The beauty of a `match` statement is that the program can only access the value of a `Result` by first checking which variant it is; one can never misinterpret a failure value as a successful completion.

Rust allows you to define your own types with value-carrying variants, and use match statements to analyze them: Rust calls such types "enumerations", but these are generally known as "algebraic data types". We describe enumerations in detail in Chapter 9.

Now that you can read `match` statements, the structure of `post_gcd` should be clear:

- It retrieves a table that maps query parameter names to arrays of values, by calling `request.get_ref::<UrlEncodedBody>()`, checking for an error and sending an appropriate response back to the client.
- Within that table, it finds the value of the parameter named `"n"`, which is where the HTML form places the numbers entered into the web page. This value will be not a single string but a vector of strings, as query parameter names can be repeated.
- It walks the vector of strings, parsing each one as an unsigned 64-bit number, and returning an appropriate failure page if any of the strings fail to parse.
- Finally, it computes the numbers' greatest common divisor as before, and constructs a response describing the results. The `format!` macro uses the same kind of string template as the `writeln!` and `println!` macros, but returns a string value, rather than writing the text to a stream.

The last remaining piece is the `gcd` function we wrote earlier. With that in place, you can interrupt any servers you might have left running, and re-build and re-start the program:

```
$ cargo run
   Compiling iron-gcd v0.1.0 (file:///home/jimb/iron-gcd)
     Running `target/debug/iron-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:

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 only share memory 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 have always locked the mutex before you access the data. 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.

In this section we'll walk you through the process of writing your second multithreaded program.

You may not have noticed it, but you've already written your first: the Iron 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_form` and `post_gcd` functions in several threads at once.

Because those particular functions are so simple, this is obviously safe; but no matter how elaborate the handler functions become, Rust guarantees that any and all data they share is managed in a thread-safe way. This allows Iron to exploit concurrency without worrying that naive handler functions may be unprepared for the consequences: all Rust functions are ready.

This section's program plots a section of 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 "Concurrency" on page 30, but this task demonstrates some of the essentials.

However, before we can talk about its concurrent implementation, we need to describe the computation we're going to perform.

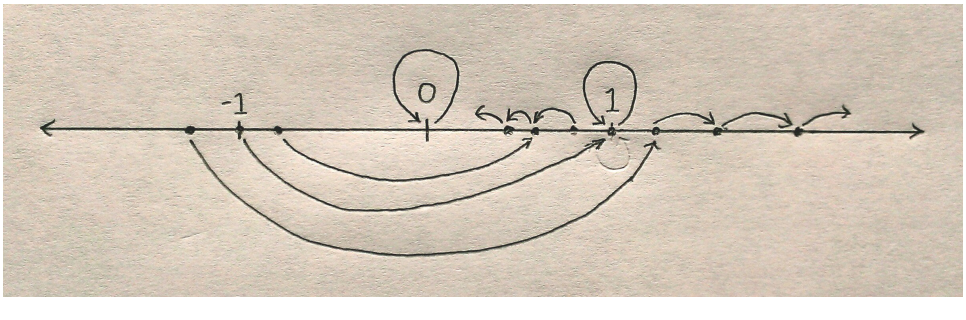
What the Mandelbrot set actually is

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.

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 as one of the prior cases:



So depending on the value you pass to `square_loop`, `x` either approaches zero, stays at one, 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. Here's the entire file, up to this point (we'll be adding more later):

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["Jim Blandy <jimb@red-bean.com>"]

[dependencies]
num = "0.1.27"
```

Now we can write the penultimate version of our loop:

```
extern crate num;
use num::Complex;

#[allow(dead_code)]
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
}
```

This 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`". So `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: R, im: I }` produces a `Complex` value with its `re` field initialized to `R`, and its `im` field initialized to `I`.

The `num` crate arranges for `*`, `+`, and the 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 Chapter 10.

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. To plot the Mandelbrot set, for each pixel in our image, we must run the above loop on the corresponding point on the complex plane, and see whether it escapes to infinity, or orbits around the origin forever.

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 two 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:

```
extern crate num;
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 two 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 escapes(c: Complex<f64>, limit: u32) -> Option<u32> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        z = z*z + c;
        if z.norm_sqr() > 4.0 {
            return Some(i);
```

```
            }
        }

        return None;
    }
```

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

The function's return value is an `Option<u32>`. 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, `escapes` returns an `Option<u32>` 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, `escapes` returns `Some(i)`, where `i` is the number of the iteration at which `z` left the circle of radius two. Otherwise, `c` is apparently in the set, and `escapes` 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 two, instead of computing a square root, we just compare the squared distance with `4.0`, which is faster.

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

Parsing pair command-line arguments

The program needs to take several command-line arguments controlling the resolution of the bitmap we'll write, and the portion of the complex plane that bitmap

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`.
///
/// 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` here is the first example we've seen of 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::<u32>` is a function that parses pairs of `u32` 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`. Often Rust will be able to infer type parameters for you, and you won't need to write them out as we've done here.

Our return type is `Option<(T, T)>`: either `None`, or a value `Some((v1, v2))`, where `(v1, v2)` is a tuple of two values 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 matching `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 two `Result` values.

The `match` expression compares this tuple against the following pattern:

```
(Ok(l), Ok(r))
```

In this pattern, `Ok(l)` and `Ok(r)` are sub-patterns that match only if the corresponding tuple element is an `Ok` result—that is, only if `from_str` succeeded in parsing that slice as a value of type `T`. If the whole pattern matches, `match` sets the variables `l` and `r` to the values carried by the two `Ok` variants, and runs the rest of that line:

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

Here, `Some((l, r))` is the value of the match expression, and hence the return value of the function.

```
_ => None
```

If the first pattern doesn't match, the wildcard pattern `_` matches anything, and ignores its value. If we reach this point, one of the `T::from_str` calls must have failed, so the match expression evaluates to `None`, which becomes the value that `parse_pair` returns.

Mapping from pixels to complex numbers

The program needs to work in two related coordinate spaces: each pixel in the output bitmap corresponds to a number on the complex plane. The relationship between these two spaces is determined by command-line arguments. The following function converts from "bitmap space" to "complex number space":

```
/// Return the point on the complex plane corresponding to a given pixel in the
/// bitmap.
///
/// `bounds` is a pair giving the width and height of the bitmap. `pixel` is a
/// pair indicating a particular pixel in that bitmap. The `upper_left` and
/// `lower_right` parameters are points on the complex plane designating the
/// area our bitmap covers.
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: (f64, f64),
                  lower_right: (f64, f64))
    -> (f64, f64)
{
    // It might be nicer to find the position of the *middle* of the pixel,
    // instead of its upper left corner, but this is easier to write tests for.
    let (width, height) = (lower_right.0 - upper_left.0,
                           upper_left.1 - lower_right.1);
    (upper_left.0 + pixel.0 as f64 * width  / bounds.0 as f64,
     upper_left.1 - pixel.1 as f64 * height / bounds.1 as f64)
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 100), (25, 75),
                              (-1.0, 1.0), (1.0, -1.0)),
               (-0.5, -0.5));
}
```

This is simply calculation, so we won't explain it in detail. However, there are a few things to point out.

```
lower_right.0
```

Expressions with this form refer to tuple elements; this refers to the first element of the tuple `lower_right`.

```
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 making explicit which conversions occur and when is suprisingly helpful. Implicit integer conversions are a frequent source of bugs and security holes in real-world C and C++ code.

Plotting the set

To plot the Mandelbrot set, for every pixel in the bitmap, we simply apply `escapes` 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: (f64, f64),
          lower_right: (f64, f64))
{
    assert!(pixels.len() == bounds.0 * bounds.1);

    for r in 0 .. bounds.1 {
        for c in 0 .. bounds.0 {
            let point = pixel_to_point(bounds, (c, r),
                                       upper_left, lower_right);
            pixels[r * bounds.0 + c] =
                match escapes(Complex { re: point.0, im: point.1 }, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}
```

This should all look pretty familiar at this point.

```
Complex { re: point.0, im: point.1 }
```

This is another use of the syntax for constructing a value of type `Complex`, in this case producing a complex number from a plain `(f64, f64)` tuple.

```
pixels[r * bounds.0 + c] =
    match escapes(Complex { re: point.0, im: point.1 }, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
```

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

Writing bitmap 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. To use image, add the following line to the `[dependencies]` section of `Cargo.toml`:

```
image = "0.6.1"
```

With that in place, we can write:

```
extern crate image;

use image::ColorType;
use image::png::PNGEncoder;
use std::fs::File;

/// Write the buffer `pixels`, whose dimensions are given by `bounds`, to the
/// file named `filename`.
fn write_bitmap(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{
    let output = match File::create(filename) {
        Ok(f) => { f }
        Err(e) => { return Err(e); }
    };

    //let output = try!(File::create(filename));

    let encoder = PNGEncoder::new(output);
    try!(encoder.encode(&pixels,
                        bounds.0 as u32, bounds.1 as u32,
                        ColorType::Gray(8)));

    Ok(())
}
```

The operation of this function is pretty straightforward: it opens a file and tries to write the bitmap image to it. 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 `ColorType::Gray(8)` indicates that each byte is an eight-bit grayscale value.

That's all straightforward. 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_bitmap`'s success and error types?

When all goes well, our `write_bitmap` 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_bitmap` function to do the same.

Consider the call to `File::create`. If that returns `Ok(f)` for some successfully opened `File` value `f`, then `write_bitmap` can proceed to write the image data to `f`. But if `File::create` returns `Err(e)` for some error code `e`, `write_bitmap` should immediately return `Err(e)` as its own return value. The call to `encoder.encode` must be handled similarly: failure should result in an immediate return, passing along the error code.

The `try!` macro exists to make these checks convenient. Instead of spelling everything out, and writing:

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

you can use the equivalent and much more legible:

```
let output = try!(File::create(filename));
```

It's a common beginner's mistake to attempt to use `try!` in the `main` function. However, since `main` has no return value, this won't work; you should use `Result`'s `expect` method instead. The `try!` macro is only useful for checking for errors reported by an expression of type `Result`, from within functions that themselves return `Result`.

There's another shorthand we could use here. Because return types of the form `Result<T, std::io::Error>` for some type `T` are so common—this is often the right type for a function that does I/O—the Rust standard library defines a shorthand for it. In the `std::io` module, we have the definitions:

```
// The std::io::Error type.
struct Error { ... };

// The std::io::Result type, equivalent to the usual `Result`, but
```

```
// specialized to use std::io::Error as the error type.
type Result<T> = std::result::Result<T, Error>
```

If we bring this definition into scope with a `use std::io::Result` declaration, we can write `write_bitmap`'s return type more tersely as `Result<()>`. This is the form you will often see when reading the documentation for functions in `std::io`, `std::fs`, and elsewhere.

A concurrent Mandelbrot program

Finally, 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 non-concurrent version for simplicity:

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

fn main() {
    let args: Vec<String> = std::env::args().collect();

    if args.len() != 5 {
        writeln!(std::io::stderr(),
                 "Usage: mandelbrot FILE PIXELS UPPERLEFT LOWERRIGHT")
            .unwrap();
        writeln!(std::io::stderr(),
                 "Example: {} mandel.png 1000x750 -1.20,0.35 -1,0.20",
                 args[0])
            .unwrap();
        std::process::exit(1);
    }

    let bounds = parse_pair(&args[2], 'x')
        .expect("error parsing image dimensions");
    let upper_left = parse_pair(&args[3], ',')
        .expect("error parsing upper left corner point");
    let lower_right = parse_pair(&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_bitmap(&args[1], &pixels, bounds)
        .expect("error writing PNG file");
}
```

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

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

This uses the `vec!` macro to create a buffer of one-byte grayscale pixel values, whose size is given by `bounds`, parsed from the command line. Rust doesn't permit programs to ever read uninitialized values, so `vec!` fills the buffer with zeros.

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

Finally, we write the pixel buffer out to disk as a PNG file. In this case we pass the buffer as a shareable (non-mutable) slice, since `write_bitmap` should have no need to modify the buffer's contents.

The natural way to distribute this calculation across multiple processors is to divide the image up into sections, one per processor, and let each procesor color the pixels assigned to it. Only when all processors have finished should we write out the pixels to disk.

The `crossbeam` crate provides a number of valuable concurrency facilities, including a *scoped thread* facility which does exactly what we need here. To use it, we must add the following line to our `Cargo.toml` file:

```
crossbeam = "0.2.8"
```

Then, we must add the following line to the top of our `main.rs` file:

```
extern crate crossbeam;
```

Then we need to take out the single line calling `render`, and replace it with the following:

```
let threads = 8;
let band_rows = bounds.1 / threads + 1;

{
    let bands: Vec<_> = pixels.chunks_mut(band_rows * bounds.0).collect();
    crossbeam::scope(|scope| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = band_rows * 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);
```

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

Breaking this down in the usual way:

```
let threads = 8;
let band_rows = bounds.1 / threads + 1;
```

Here we decide to use eight threads. Then we compute how many rows of pixels each band should have. Since the height of a band is `band_rows` and the overall width of the image is `bounds.0`, the area of a band, in pixels, is `band_rows * bounds.0`. We round the row count upwards, to make sure the bands cover the entire bitmap even if the height isn't a multiple of `threads`.

```
let bands: Vec<_> = pixels.chunks_mut(band_rows * bounds.0).collect();
```

Here we divide the pixel buffer into bands. The buffer's `chunks_mut` method returns an iterator producing mutable, non-overlapping slices of the buffer, each of which encloses `band_rows * bounds.0` pixels—in other words, `band_rows` complete rows of pixels. The last slice that `chunks_mut` produces may be shorter, but since all the other slices enclosed complete rows from the buffer, the last slice will too. Finally, the iterator's `collect` method builds a vector holding these mutable, non-overlapping slices.

Now we can put the `crossbeam` library to work:

```
crossbeam::scope(|scope| { ... });
```

The expression `|scope| { ... }` is a Rust *closure* expression. A closure is a value that can be called as if it were a function; here, `|scope|` is the argument list, and `{ ... }` is the body of the function. 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, `crossbeam::scope` takes the closure and applies it to a `Scope` object, representing the lifetime of the group of threads we'll create to render our horizontal bands.

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

Here we iterate over the buffer's bands. By using the `into_iter()` iterator, we ensure that each iteration of the loop body takes ownership of its band; and the `enumerate` adapter attaches an index `i` to each value produced.

```
let top = band_rows * 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 bitmap. 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.

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

Finally, we create a thread, running the closure `move || { ... }`. This syntax is a bit strange to read: it denotes a closure of no arguments whose body is the `{ ... }` form. 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`.

The `crossbeam::scope` call ensures that all threads have completed before it returns, meaning that it is safe to save the bitmap to a file, which is our next action.

Running the Mandelbrot plotter

We've used several external crates in this program: `num` for complex number arithmetic; `image` for writing `PNG` files; and `crossbeam` for the scoped thread creation primitives. Here's the final `Cargo.toml` file including all those dependencies:

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["Jim Blandy <jimb@red-bean.com>"]

[dependencies]
crossbeam = "0.2.8"
image = "0.6.1"
num = "0.1.27"
```

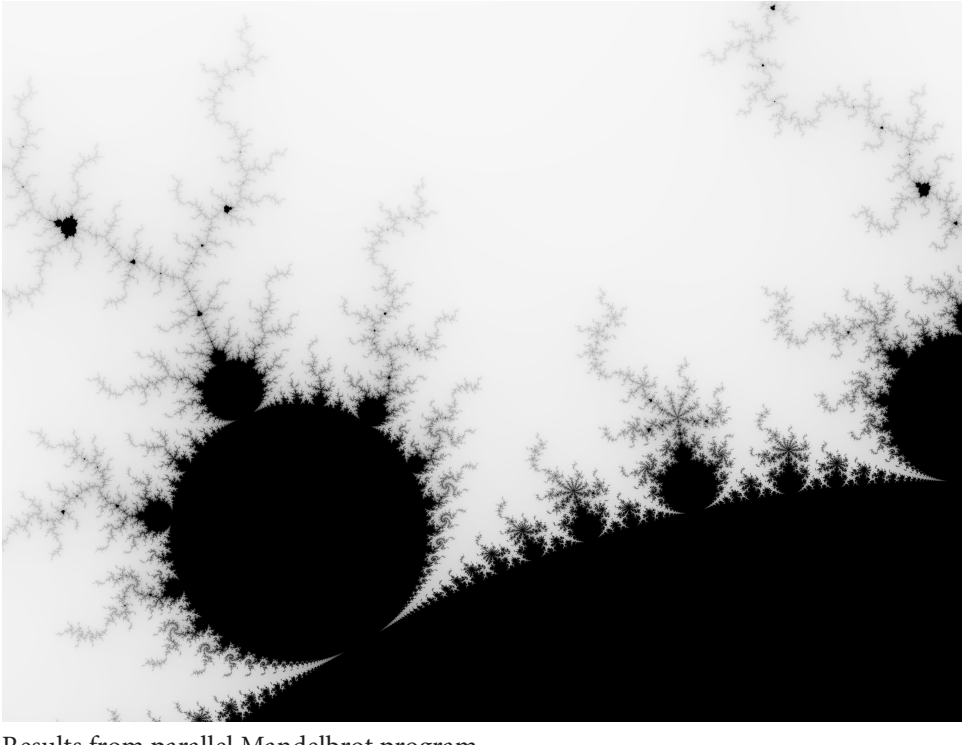
With that in place, we can build and run the program:

```
$ cargo build --release
   Compiling bitflags v0.3.3
   ...
   Compiling png v0.4.2
   Compiling image v0.6.1
   Compiling mandelbrot v0.1.0 (file:///home/jimb/rust/book/tests/mandelbrot)
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20

real    0m2.525s
user    0m6.254s
```

```
sys 0m0.013s
$
```

Here, we've used the Unix `time` program to see how long the program took to run; note that even though we spent more than six seconds of processor time computing the image, the elapsed real time was only two and a half seconds. You can verify that a substantial portion of that real time is spent writing the image file by commenting out the code that does so; on the laptop where this code was tested, the concurrent version reduces the Mandelbrot calculation time proper by a factor of almost four.

This command should create a file called `mandel.png`, which you can view with your system's image viewing program, or visit in a web browser. If all has gone well, it should look like this:



Results from parallel Mandelbrot program

Safety is invisible

In the end, the program we've written here is not substantially different from what we might write in any other language: we apportion pieces of the pixel buffer out amongst the processors; let each one work on its piece separately; and when they've all finished, present the result. So then, 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 above partitions the buffer amongst the threads correctly, but there are many small variations on that code that do not, and 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 Chapter 5 and "Concurrency" on page 30, we'll describe Rust's rules for memory safety, and explain how these rules also ensure proper concurrency hygiene.

CHAPTER 3
Basic types

Rust's types help the language meet several goals:

- *Safety*: A program's types provide enough information about its behavior to allow the compiler to ensure that the program is well defined.
- *Efficiency*: The programmer has fine-grained control over how Rust programs represent values in memory, and can choose types she knows the processor will handle efficiently. Programs needn't pay for generality or flexibility they don't use.
- *Concision*: Rust manages the above without requiring too much guidance from the programmer in the form of types written out in the code. Rust programs are usually less cluttered with types than the analogous C++ program would be.

Rather than using an interpreter or a just-in-time compiler, Rust is designed to use ahead-of-time compilation: the translation of your entire program to machine code is completed before it ever begins execution. Rust's types help an ahead-of-time compiler choose good machine-level representations for the values your program operates on: representations whose performance you can predict, and which give you full access to the machine's capabilities.

Rust is a *statically typed* language: without actually running the program, the compiler checks that every possible path of execution will use values only in ways consistent with their types. This allows Rust to catch many programming mistakes early, and is crucial to Rust's safety guarantees.

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 functions' parameters and return values, members of struct types, and a few other situations. However, two features of Rust make this less trouble than you might expect:

- Given the types that you did spell out, Rust will *infer* most of the rest for you. In practice, there's often only one type that will work for a given variable or expression; when this is the case, Rust lets you leave out the type. For example, you could spell out every type in a function, like this:

```
fn build_vector() -> Vec<i16> {
    let mut v: Vec<i16> = Vec::<i16>::new();
    v.push(10i16);
    v.push(20i16);
    return v;
}
```

  But this is cluttered and repetitive. Given the function's return type, it's obvious that `v` must be a `Vec<i16>`, a vector of 16-bit signed integers; no other type would work. And from that it follows that each element of the vector must be an `i16`. This is exactly the sort of reasoning Rust's type inference applies, allowing you to instead write:

```
fn build_vector() -> Vec<i16> {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    return v;
}
```

  These two definitions are exactly equivalent; 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.

- Functions can be *generic*: when a function's purpose and implementation are general enough, you can define it to work on any set of types that meet the necessary criteria. A single definition can cover an open-ended set of use cases.

  In Python and JavaScript, all functions work this way naturally: a function can operate on any value that has the properties and methods the function will need. (This is the characteristic often called "duck typing": if it quacks like a duck, it's a duck.) But it's exactly this flexibility that makes it so difficult for those languages to detect type errors early; testing is often the only way to catch such mistakes. Rust's generic functions give the language a degree of the same flexibility, while still catching all type errors at compile time.

  Despite their flexibility, generic functions are just as efficient as their non-generic counterparts. We'll discuss generic functions in detail in Chapter 10.

The rest of this chapter covers Rust's types from the bottom up, starting with simple machine types like integers and floating-point values, and then showing how to compose them into more complex structures. Where appropriate, we'll describe how Rust represents values of these types in memory, and their performance characteristics.

Here's a summary of all Rust's types, brought together in one place.

Type	Description	Values
`i8`, `i16`, `i32`, `i64` `u8`, `u16`, `u32`, `u64`	signed and unsigned integers, of given bit width	`-5i8`, `0x400u16`, `0o100i16`, `20_922_789_888_000u64`, `b'*'` (`u8` byte literal), `42` (type is inferred)
`isize`, `usize`	signed and unsigned integers, size of address on target machine (32 or 64 bits)	`-0b0101_0010isize`, `0xffff_fc00usize`, `137` (type is inferred)
`f32`, `f64`	IEEE floating-point numbers, single and double precision	`3.14f32`, `6.0221e23f64`, `1.61803` (float type is inferred)
`bool`	Boolean	`true`, `false`
`(char, u8, i32)`	tuple: mixed types allowed	`('%', 0x7f, -1)`
`()`	"unit" (empty) tuple	`()`
`struct S { x: f32, y: f32 }`	named-field structure	`S { x: 120.0, y: 209.0 }`
`struct T (i32, char);`	tuple-like structure	`T(120, 'X')`
`struct E;`	unit-like structure; has no fields	`E`
`enum Attend { OnTime, Late(u32) }`	enumeration, algebraic data type	`Late(5)`, `OnTime`
`Box<Attend>`	box: owning pointer that frees referent when dropped	`Box::new(Late(15))`
`&i32`, `&mut i32`	shared and mutable references: non-owning pointers that must not outlive their referent	`&s.y`, `&mut v`
`char`	Unicode character, 32 bits wide	`'*'`, `'\n'`, `' '`, `'\x7f'`, `'\u{CA0}'`
`String`	UTF-8 string, dynamically sized	`"    : ramen".to_string()`
`&str`	reference to `str`: non-owning pointer to UTF-8 text	`"  : soba"`, `&s[0..12]`
`[f64; 4]`, `[u8; 256]`	array, fixed length; elements all of same type	`[1.0, 0.0, 0.0, 1.0]`, `[b' '; 256]`
`Vector<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[..]`
`&Any`, `&mut Read`	trait object: comprises pointer to value and vtable of trait methods	`value as &Any`, `&mut file as &mut Read`
`fn(&str, usize) -> isize`	pointer to function (not a closure)	`i32::saturating_add`

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

- We give `struct` types their own chapter, ???.

- We give enumerated types their own chapter, Chapter 9.
- We describe trait objects in Chapter 10

Machine 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, and the boolean and character types.

The names of Rust's numeric types follow a regular pattern, spelling out their width in bits, and the representation they use:

Table 3-1. Rust's numeric types

size (bits)	unsigned integer	signed integer	floating-point
8	`u8`	`i8`	
16	`u16`	`i16`	
32	`u32`	`i32`	`f32`
64	`u64`	`i64`	`f64`
machine word	`usize`	`isize`	

Integer types

Rust's unsigned integer types use their full range to represent positive values and zero:

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, 18 quintillion)
`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:

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)

type	range
`i64`	-2^{63} to $2^{63}-1$ (−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
`isize`	either -2^{31} to $2^{31}-1$, or -2^{63} to $2^{63}-1$

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

Unlike C and C++, Rust treats characters as distinct from the numeric types; a `char` is neither a `u8` nor an `i8`. We describe Rust's `char` type in its own section below.

The precision of the `usize` and `isize` types depends on 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. The `usize` type is analogous to the `size_t` type in C and C++. 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. The `isize` type is the signed analog of the `usize` type, similar to the `ssize_t` type in C and C++.

In debug builds, Rust checks for integer overflow in arithmetic.

```
let big_val = std::i32::MAX;
let x = big_val + 1;  // panic: arithmetic operation overflowed
```

In a release build, this addition would wrap to a negative number (unlike C++, where signed integer overflow is undefined behavior). But unless you want to give up debug builds forever, it's a bad idea to count on it. When you want wrapping arithmetic, use the methods:

```
let x = big_val.wrapping_add(1);  // ok
```

Integer literals in Rust can take a suffix indicating their type: `42u8` is a `u8` value, and `1729isize` is an `isize`. You can omit the suffix on an integer literal, in which case Rust will try to infer a unique type for it from the context. If more than one type is possible, Rust defaults to `i32`, if that is among the possibilities. Otherwise, Rust reports the ambiguity as an error.

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; for example, this permits breaking hexadecimal or binary numbers into groups of four digits, which is often more natural than groups of three.

Some examples of integer literals:

literal	type	decimal value
`116i8`	`i8`	116
`0xcafeu32`	`u32`	51966
`0b0010_1010`	inferred	42
`0o106`	inferred	70

Although numeric types and the `char` type are distinct, Rust does provide "byte literals", character-like literals for `u8` values: `b'X'` represents the ASCII code for the character `X`, as a `u8` value. For example, since the ASCII code for `A` is 65, the literals `b'A'` and `65u8` are exactly equivalent. Byte literals are limited to ASCII values, from 0 through 127.

There are a few characters that you cannot simply place after the single quote, because that would be either syntactically ambiguous or hard to read. The following characters require a backslash placed in front of them:

character	byte literal	numeric equivalent
single quote, '	`b'\''`	`39u8`
backslash, \	`b'\\'`	`92u8`
newline	`b'\n'`	`10u8`
carriage return	`b'\r'`	`13u8`
tab	`b'\t'`	`9u8`

For characters that are hard to write or read, you can write their ASCII code in hexadecimal instead. A byte literal of the form `b'\xHH'`, where `HH` is a two-digit hexadecimal number, represents the character whose ASCII code is `HH`. The number `HH` must be between `00` and `7F` (127 decimal). For example, the ASCII "escape" control character has a code of 27 decimal, or 1B hexadecimal, so you can write a byte literal for "escape" as `b'\x1b'`. But since byte literals are just another notation for `u8` values, `b'\x1b'` and `0x1b` are equivalent (letting Rust infer the type). Since the simple numeric literal is more legible, it probably only makes sense to use hexadecimal byte literals when you want to emphasize that the value represents an ASCII code.

Like any other sort of value, integers can have methods. The standard library provides some basic operations, which you can look up in the on-line documentation by searching for `std::i32`, `std::u8`, and so on.

```
assert_eq!(2u16.pow(4), 16);            // exponentiation
assert_eq!((-4i32).abs(), 4);           // absolute value
assert_eq!(0b101101u8.count_ones(), 4); // population count
```

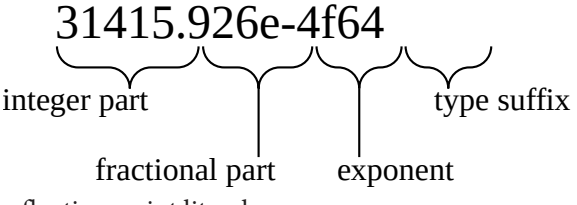
The type suffixes on the literals are required here: Rust can't look up a value's methods until it knows its type. In real code, however, there's usually additional context to disambiguate the type, so the suffixes aren't needed.

Floating-point types

Rust provides IEEE single- and double-precision floating-point types. Following the IEEE 754-2008 specification, these types include positive and negative infinities, distinct positive and negative zero values, and a "not a number" value.

type	precision	range
`f32`	IEEE single precision (at least 6 decimal digits)	roughly -3.4×10^{38} to $+3.4 \times 10^{38}$
`f64`	IEEE double precision (at least 15 decimal digits)	roughly -1.8×10^{308} to $+1.8 \times 10^{308}$

Floating-point literals have the general form:



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. 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 will infer whether it is an `f32` or `f64` from the context, defaulting to the latter if both would be possible. For the purposes of type inference, Rust treats integer literals and floating-point literals as distinct classes: it will never infer a floating-point type for an integer literal, or vice versa.

Some examples of floating-point literals:

literal	type	mathematical value
`-1.125`	inferred	−(1 9/16)
`2.`	inferred	2
`0.25`	inferred	1/4
`125e-3`	inferred	1/8
`1e4`	inferred	1000
`40f32`	`f32`	40

literal	type	mathematical value
`271.8281e-2f64`	`f64`	2.718281

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. The standard library documentation describes these under the module name "`std::f32` (primitive type)" and "`std::f64` (primitive type)".

The standard library's `std::f32` and `std::f64` modules define constants for the IEEE-required special values, as well as the largest and smallest finite values. The `std::f32::consts` and `std::f64::consts` modules provide various commonly used constants like `E`, `PI`, and the square root of two.

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 the key word here is "implicitly": 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; in our experience, the act of writing out numeric conversions in Rust has alerted us to problems we would otherwise have missed.

Like any other type, floating-point types can have methods. The standard library provides the usual selection of arithmetic operations, transcendental functions, IEEE-specific manipulations, and general utilities, which you can look up in the on-line documentation by searching for `std::f32` and `std::f64`. Some examples:

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.);
assert_eq!(1f64.asin(), std::f64::consts::PI/2.);
assert!((-1. / std::f32::INFINITY).is_sign_negative());
```

The type suffixes on the literals are required here: Rust can't look up a value's methods until it knows its type. In real code, however, there's usually additional context to disambiguate the type, so the suffixes aren't needed.

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 permits strings, lists, dictionaries, and even sets in boolean contexts, treating such values as true if they're non-empty. 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` only needs 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. But naturally, if the compiler can prove that a given `bool` never has its address taken, it can choose whatever representation it likes for it, since the programmer will never know the difference.

Characters

Rust's character type `char` represents a single Unicode character, as a 32-bit value.

Rust uses the `char` type for single characters in isolation, but uses the UTF-8 encoding for strings and streams of text. So, a `String` represents its text as a sequence of UTF-8 bytes; but iterating over a string with a `for` loop produces `char` values.

Character literals are characters enclosed in single quotes, like `'8'` or `'!'`. You can use any Unicode character you like: `' '` is a `char` literal representing the Japanese kanji for "tetsu" (iron).

As with byte literals, backslash escapes are required for a few characters:

character	Rust character literal
single quote, '	`'\''`
backslash, \	`'\\'`
newline	`'\n'`
carriage return	`'\r'`
tab	`'\t'`

If you prefer, you can write out a character's Unicode scalar value in hexadecimal:

- If the character's scalar value is in the range `U+0000` to `U+007F` (that is, if it is drawn from the ASCII character set), then you can write the character as `'\xHH'`, where `HH` is a two-digit hexadecimal number. For example, the character literals

`'*'` and `'\x2A'` are equivalent, because the scalar value of the character `*` is 42, or 2A in hexadecimal.

- You can write any Unicode character as `'\u{HHHHHH}'`, where `HHHHHH` is a hexadecimal number between one and six digits long. For example, the character literal `'\u{CA0}'` represents the character “ ”, a Kannada character used in the Unicode Look of Disapproval, “ _ ”. The same literal could also be simply written as `' '`.

A `char` always holds a Unicode scalar value, in the range 0x0000 to 0xD7FF or 0xE000 to 0x10FFFF. A `char` is never a surrogate pair half (that is, a code point in the range 0xD800 to 0xDFFF), or a value outside the Unicode code space (that is, greater than 0x10FFFF). Rust uses the type system and dynamic checks to ensure `char` values are always in the permitted range.

Rust never implicitly converts between `char` and any other type. You can use the `as` conversion operator to convert a `char` to an integer type; for types smaller than 32 bits, the upper bits of the character's value are truncated:

```
assert_eq!('*' as i32, 42);
assert_eq!(' ' as u16, 0xca0);
assert_eq!(' ' as i8, -0x60); // U+0CA0 truncated to eight bits, signed
```

Going in the other direction, `u8` is the only type the `as` operator will convert to `char`: Rust intends the `as` operator to perform only cheap, infallible conversions, but every integer type other than `u8` includes values that are not permitted Unicode scalar values, so those conversions would require run-time checks. Instead, the standard library function `std::char::from_u32` takes any `u32` value and returns an `Option<char>`: if the `u32` is not a permitted Unicode scalar value, then `from_u32` returns `None`; otherwise, it returns `Some(c)`, where `c` is the `char` result.

The standard library provides some useful methods on characters, which you can look up in the on-line documentation by searching for `std::char`. For example:

```
assert_eq!('*'.is_alphabetic(), false);
assert_eq!('8'.to_digit(10), Some(8));
assert_eq!(' '.len_utf8(), 3);
```

Naturally, single characters in isolation are not as interesting as strings and streams of text. We'll describe Rust's standard `String` type and text handling in general below.

Tuples

A *tuple* is a pair, or triple, or quadruple, ... of values of assorted types. You can write a tuple as a sequence of elements, separated by commas and surrounded by parentheses. For example, `("Brazil", 1985)` is a tuple whose first element is a statically allocated string, and whose second is an integer; its type is `(&str, i32)` (or whatever

integer type Rust infers for `1985`). Given a tuple value `t`, you can access its elements as `t.0`, `t.1`, and so on.

Tuples differ from arrays: for one thing, each element of a tuple can have a different type, whereas an array's elements must be all the same type. Further, tuples allow only constants as indices: if `t` is a tuple, you can't write `t[i]` to refer to the `i`'th element of a tuple. A tuple element expression always refers to some fixed element, like `t.4`.

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 element 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 2, we need 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`.
///
/// `bounds` is a pair giving the width and height of the bitmap. ...
fn write_bitmap(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<()>
{ ... }
```

The type of the `bounds` parameter is `(usize, usize)`, a tuple of two `usize` values. Using a tuple lets us manage the width and height as a single parameter, making the code more legible.

The other commonly used tuple type, perhaps surprisingly, 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 which returns no value has a return type of `()`. The standard library's `std::mem::swap` function has no meaningful return value; it just exchanges the values of its two arguments. The declaration for `std::mem::swap` reads:

```
fn swap<T>(x: &mut T, y: &mut T);
```

The `<T>` means that `swap` is *generic*: you can use it on references to values of any type `T`. But the signature omits the `swap`'s return type altogether, which is shorthand for returning the unit type:

```
fn swap<T>(x: &mut T, y: &mut T) -> ();
```

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

If you like, you may include a comma after a tuple's last element: the types `(&str, i32,)` and `(&str, i32)` are equivalent, as are the expressions `("Brazil", 1985,)` and `("Brazil", 1985)`. Human programmers will probably find trailing commas distracting, but tolerating them in the language's syntax can simplify programs that generate Rust code. Rust consistently permits an extra trailing comma everywhere commas are used: function arguments, arrays, enum definitions, and so on.

For completeness' sake, there are even tuples that contain a single value. The literal `("lonely hearts",)` is a tuple containing a single string; its type is `(&str,)`. Here, the comma after the value is necessary to distinguish the singleton tuple from a simple parenthetic expression. Like the trailing commas, singleton tuples probably don't make much sense in code written by humans, but their admissibility can be useful to generated code.

Pointer types

Rust has several types that represent memory addresses.

This is a big difference between Rust and most languages with garbage collection. In Java, if `class Tree` contains a field `Tree left;`, then `left` is a reference to another separately-created `Tree` object. Objects never physically contain other objects in Java.

Rust is different. The language is designed to help keep allocations to a minimum. 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, when a Rust program needs values to point to other values, it must use pointer types explicitly. The good news is that the pointer types used in safe Rust are constrained to eliminate undefined behavior, so pointers are much easier to use correctly in Rust than in C++.

We'll discuss three pointer types here: references, boxes, and unsafe pointers.

References

A value of type `&String` is a reference to a `String` value, an `&i32` is a reference to an `i32`, and so on.

It's easiest to get started by thinking of references as Rust's basic pointer type. A reference can point to any value anywhere, stack or heap. The address-of operator, `&`, and the deref operator, `*`, work on references in Rust, just as their counterparts in C work on pointers. And like a C pointer, a reference does not automatically free any resources when it goes out of scope.

One difference is that Rust references are immutable by default:

- `&T` - immutable reference, like `const T*` in C
- `&mut T` - mutable reference, like `T*` in C

Another major difference is that Rust tracks the ownership and lifetimes of values, so many common pointer-related mistakes are ruled out at compile time. Chapter 5 explains Rust's rules for safe reference use.

Boxes

The simplest way to allocate a value in the heap is to use `Box::new`.

```
let t = (12, "eggs");
let b = Box::new(t);  // allocate a tuple in the heap
```

The type of `t` is `(i32, &str)`, so the type of `b` is `Box<(i32, &str)>`. `Box::new()` allocates enough memory to contain the tuple on the heap. When `b` goes out of scope, the memory is freed immediately, unless `b` has been *moved*—by returning it, for example.

Raw pointers

Rust also has the raw pointer types `*mut T` and `*const T`. Raw pointers really are just like pointers in C++. Using a raw pointer is unsafe, because Rust makes no effort to track what a raw pointer points to. For example, the pointer may be null; it may point to memory that has been freed or now contains a value of a different type. All the classic pointer mistakes of C++ are offered for your enjoyment in unsafe Rust. For details, see ???.

Arrays, Vectors, and Slices

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

- The type `[T; N]` represents an array of `N` values, each of type `T`. An array's size is a constant, and is part of the type; you can't append new elements, 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 on 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" or "mutable slice of `T`", is a reference 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 amongst 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 thread panics. Of course, v's length 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; 100000]` is an array of 100000 `bool` elements, all set to `true`:

```
let mut sieve = [true; 100000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
```

```
            while j < 100000 {
                sieve[j] = false;
                j += i;
            }
        }
    }

    assert!(sieve[211]);
    assert!(!sieve[30031]);
```

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

The useful methods you'd like to see on arrays—iterating over elements, searching, sorting, filling, filtering, and so on—all appear as methods of slices, not arrays. But since those methods take their operands by reference, and taking a reference to an array produces a slice, you can actually 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 `sort` takes its operand by reference, we can use it directly on `chaos`: the call implicitly produces a `&mut [i32]` slice referring to the entire array. In fact, the `len` method we mentioned earlier is a slice method as well.

We cover slices in more detail in **section slices** below.

Vectors

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

```
let mut v = vec![2, 3, 5, 7];
assert_eq!(v.iter().fold(1, |a, b| a * b), 210);
```

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

```
v.push(11);
v.push(13);
assert_eq!(v.iter().fold(1, |a, b| a * b), 30030);
```

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

```
let mut v = Vec::new();
v.push("step");
v.push("on");
v.push("no");
```

```
v.push("pets");
assert_eq!(v, 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 above, as `collect` can build many different sorts of collections, not just vectors. By making the type for `v` explicit, we've made it unambiguous which sort of collection we want.

`Vec` is a fairly fundamental 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. To explore other options, consult the online documentation for `std::vec::Vec`.

A vector stores its contents in the dynamically allocated heap. A `Vec<T>` consists of three values: a pointer to the block of memory allocated to hold the elements; the number of elements that block has the capacity to store; and the number it actually contains now (in other words, its length). When the block has reached its capacity, adding another element to the vector entails allocating a larger block, copying the present contents into it, updating the vector's pointer and capacity to describe the new block, 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 block of memory 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. Note that this only establishes the 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 above, 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.

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);
assert_eq!(v.capacity(), 4);
```

The capacities you'll see in your code may differ from those shown here, depending on what sizes `Vec` and the system's heap allocator decide would be best.

You can insert and remove elements wherever you like in a vector, although these operations copy all the elements after the insertion point:

```
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 2.
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!["carmen", "miranda"];
assert_eq!(v.pop(), Some("miranda"));
assert_eq!(v.pop(), Some("carmen"));
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 fragments v0.1.0 (file:///home/jimb/rust/book/fragments)
     Running `.../target/debug/fragments Lisp Scheme C C++ Fortran`
Lisp: functional
Scheme: functional
C: imperative
C++: imperative
Fortran: imperative
$
```

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

As with arrays, many useful methods you'd like to see on vectors, like iterating over elements, searching, sorting, filling, and filtering, all appear as methods of slices, not arrays. But since those methods take their operands by reference, and taking a reference to a vector produces a slice, you can actually call any slice method on an vector directly:

```
let mut v = vec!["a man", "a plan", "a canal"];
v.reverse();
assert_eq!(v, ["a canal", "a plan", "a man"]); // disappointing
```

Here, the `reverse` method is actually defined on slices, but since `reverse` takes its operand by reference, we can use it directly on `v`: the call implicitly produces a `&mut [&str]` slice referring to the entire array.

Building vectors element by element

Building a vector one element at a time isn't as bad as it might sound. Whenever a vector outgrows its capacity by a single element, it chooses a new block twice as large as the old one. By the time it has reached its final size of 2^n for some n, the total number of elements copied in the course of reaching that size is the sum of each of the powers of two smaller than 2^n—that is, the sizes of the blocks we left behind. But if you think about how powers of two work, that total is simply 2^n-1, meaning that the number of elements copied is always within a factor of two of the final size. Since the number of copies is linear in the final size, the cost per element is constant—the same as it would be if you had allocated the vector with the correct size to begin with!

What this means is that using `Vec::with_capacity` instead of `Vec::new` is a way to gain a constant factor improvement in speed, rather than an algorithmic improvement. For small vectors, avoiding a few calls to the heap allocator can make an observable difference in performance.

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.

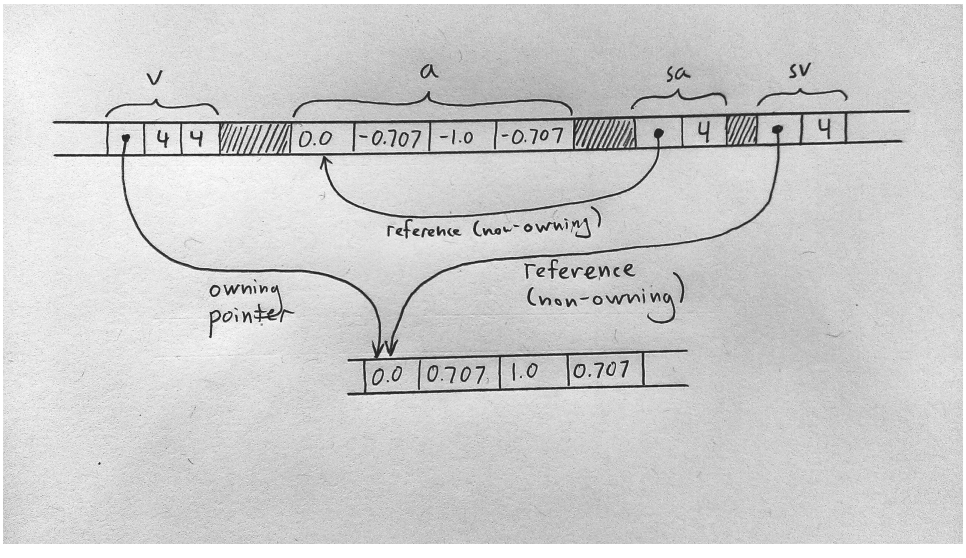
Suppose you run the following code:

```
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;
```

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

By the end, memory looks like this:



Whereas an ordinary reference is a non-owning pointer to a single value, a reference to a slice is a non-owning pointer to several values. This makes slice references a good choice when you want to write a function that operates on any homogeneous data series, regardless of whether it's stored in an array or a vector, stack or heap. 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(&v);  // works on vectors
print(&a);  // works on arrays
```

Because this function takes a slice reference as an argument, you can apply it to either a vector or an array, as shown. In fact, many methods you might think of as belonging to vectors or arrays are actually methods defined on slices: for example, the `sort` and `reverse` methods, which sort or reverse a sequence of elements in place, are actually methods on the slice type `[T]`.

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 take a slice that extends past the end of the data results in a thread panic.

String 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 all the ways to write string literals, then talk about Rust's two string types and how to use them.

String literals

String literals are enclosed in double quotes. They use the same backslash escape sequences as `char` literals.

```
let speech = "\"Ouch!\" said the well.\n";
```

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.

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 after the dash.

In a few cases, the need to double every backslash in a string is a nuisance. (The classic examples are regular expressions and Windows filenames.) 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 = Pcre::compile(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.

Byte strings

A string literal with the `b` prefix is a byte string. Such a string is a slice of `u8` values—that is, bytes—rather than Unicode text.

```
let method = b"GET";
assert_eq!(method, &[b'G', b'E', b'T']);
```

This combines with all the other string syntax we've shown above: byte strings can span multiple lines, use escape sequences, and use backslashes to join lines. Raw byte strings start with `br"`.

Byte strings can't contain arbitrary Unicode characters. They must make do with ASCII and escape sequences that denote values in the range 0-255.

The type of `method` above is `&[u8; 3]`: it's a reference to an array of 3 bytes. It doesn't have any of the string methods we'll discuss in a minute. The most string-like thing about it is the syntax we used to write it.

Strings in memory

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.

```
assert_eq!(" _ ".as_bytes(),
           [0xe0, 0xb2, 0xa0, b'_', 0xe0, 0xb2, 0xa0]);
```

The type of a string literal is `&str`, meaning it is a reference to a `str`, a slice of memory that's guaranteed to contain valid UTF-8 data.

Like other slice references, an `&str` is a fat pointer. It contains both the address of the actual data and a length field. The `.len()` method of an `&str` returns the length. Note that it's measured in bytes, not characters:

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

A string literal is a reference to an immutable string of text, typically stored in memory that is mapped as read-only. It is impossible to modify a `str`:

```
let mut s = "hello";
s[0] = 'c';    // error: the type `str` cannot be mutably indexed
s.push('\n');  // error: no method named `push` found for type `&str`
```

For creating new strings at run time, there is the standard `String` type.

`String`

`&str` is very much like `&[T]`: a fat pointer to some data. `String` is analogous to `Vec<T>`.

	`Vec<T>`	`String`
automatically frees buffers	yes	yes
growable	yes	yes
`::new()` and `::with_capacity()` static methods	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 an `&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.
  ```
  assert_eq!(format!("{}°{:02} {:02} N", 24, 5, 23),
             "24°05 23 N".to_string());
  ```

As it happens, this would work fine without the `.to_string()` call, because the `String` can automatically convert to `&str`.

- Arrays, slices, and vectors of strings have two methods, `.concat()` and `.join(sep)`, that form a new `String` from many strings:

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

The choice sometimes arises of which type to use: `&str` or `String`. Chapter 4 addresses this question in detail. For now it will do to point out that an `&str` can refer to any slice of any string, whether it is a string literal (stored in the executable) or a `String` (allocated and freed at run time). This means that `&str` is more appropriate for function arguments when the caller should be allowed to pass either kind of string.

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_eq!("ONE".to_lowercase() == "one", true);
```

Strings also support the comparison operators `<`, `<=`, `>`, and `>=`, as well as many useful methods which you can find in the on-line documentation by searching for `str`. (Or just flip to ???.)

```
assert_eq!("peanut".contains("nut"), true);
assert_eq!(" _ ".replace(" ", " "), " _ ");
assert_eq!("    clean\n".trim(), "clean");

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

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 for these cases is to offer a few string-like types for these particular situations. Stick to `String` and `&str` for Unicode text; but

- when working with filenames, use `std::path::PathBuf` and `&Path` instead;
- when working with binary data that isn't character data at all, use `Vec<u8>` and `&[u8]`;
- when interoperating with C libraries that use null-terminated strings, use `std::ffi::CString` and `&CStr`.

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 type, and we'll cover them in three successive chapters: structs in ???, enums in Chapter 9, and traits in Chapter 10.

Functions and closures have their own types, covered in ???. And the types that make up the standard library are covered throughout the book. For example, ??? 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 4
Ownership and moves

Rust makes the following pair of promises, both essential to a safe systems programming language:

- You decide the lifetime of each value in your program. Rust frees memory and other resources belonging to a value promptly, at a point under your control.
- Even so, your program will never use a pointer to an object after it has been freed. Using a "dangling pointer" is a common mistake in C and C++: if you're lucky, your program crashes; if you're unlucky, your program has a security hole. Rust catches these mistakes at compile time.

C and C++ keep the first promise: you can call `free` or `delete` on any object on the dynamically-allocated heap you like, whenever you like. But in exchange, the second promise is set aside: it is entirely your responsibility to ensure that no pointer to the value you freed is ever used. There's ample empirical evidence that this is a difficult responsibility to meet, in the unfortunate form of fifteen years' worth of crashes and reported security vulnerabilities caused by pointer misuse.

Plenty of languages fulfill the second promise using garbage collection, automatically freeing objects only when all reachable pointers to them are gone. But in exchange, you relinquish control to the collector over exactly when objects get freed. In general, garbage collectors are surprising beasts; understanding why memory wasn't freed when you expected can become a challenge. And if you're working with objects that represent files, network connections, or other operating system resources, not being able to trust that they'll be freed at the time you intended, and their underlying resources cleaned up along with them, is a disappointment.

None of these compromises are acceptable for Rust: the programmer should have control over values' lifetimes, *and* the language should be safe. But this is a pretty

well-explored area of language design; you can't make major improvements without some fundamental changes.

Rust breaks the deadlock in a surprising way: by restricting how your programs can use pointers. Ownership relations must be made explicit in the types; non-owning pointers must have restricted lifetimes; mutation and sharing must be kept segregated; and so on. Some common structures you are accustomed to using may not fit within the rules, and you'll need to look for alternatives. But the net effect of these restrictions is to bring just enough order to the chaos that Rust's compile-time checks can promise that your program is free of memory management errors: dangling pointers, double frees, using uninitialized memory, and so on. 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.

These same rules also form the basis of Rust's support for safe concurrent programming. Given a carefully designed set of library routines for starting new threads and communicating between them, the same checks that ensure your code uses memory correctly also serve to prove that it is free of data races.

Rust's radical wager 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 motivate Rust's rules by showing how the same underlying issues play out in other languages. Then, we'll explain Rust's rules in detail. Finally, we'll talk about some exceptions and almost-exceptions.

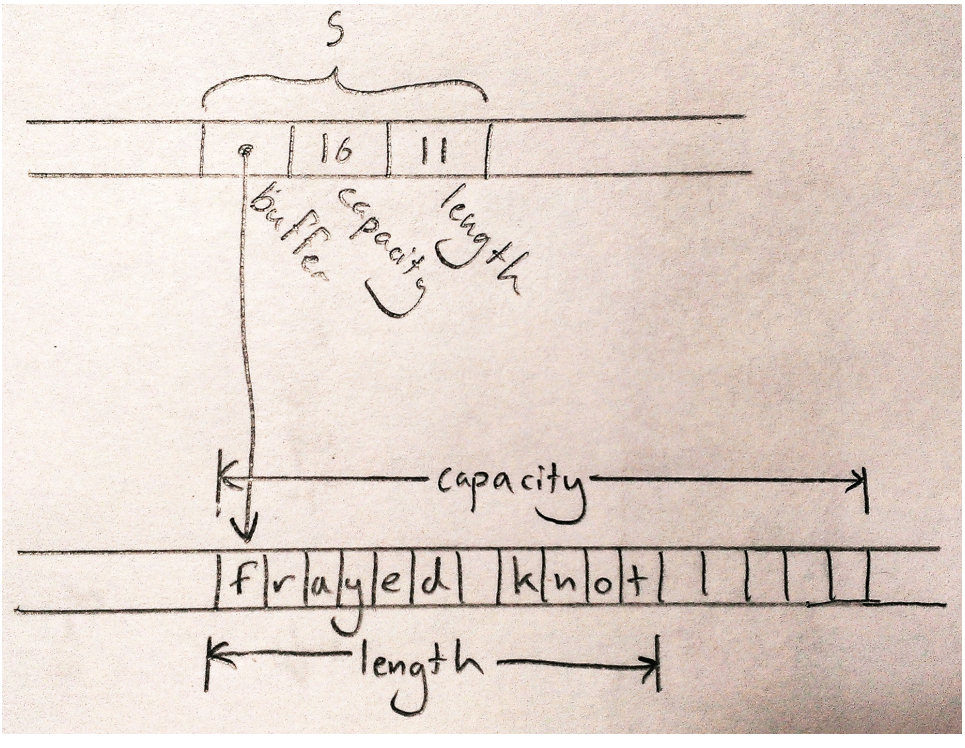
Ownership

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 will probably destroy 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 like this:



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.

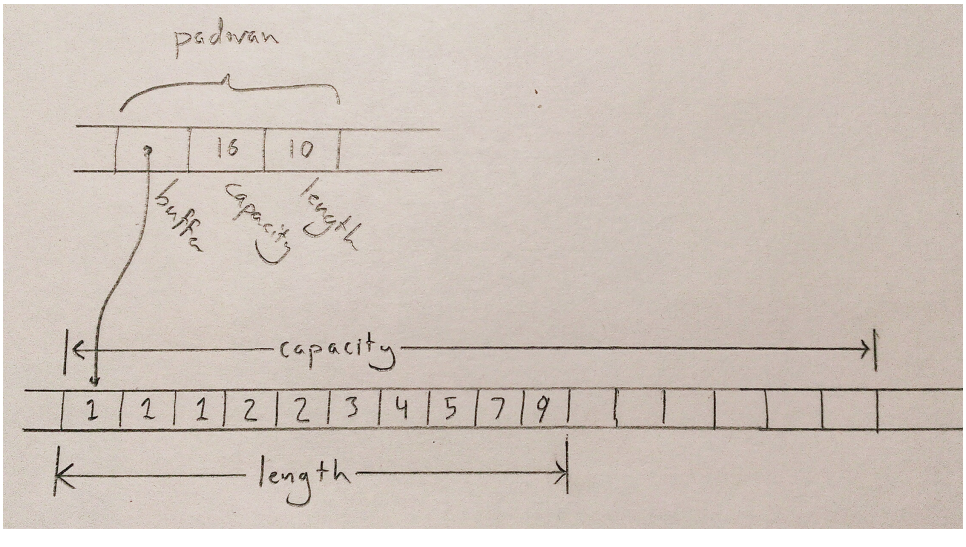
A `std::string` owns its buffer: when the program destroys the string, the string's destructor frees the buffer[1]. In these situations it's generally understood that, although it's fine for other code to create temporary pointers to the owned memory, it is such code's own responsibility to make sure 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 any more. The owner determines the lifetime of the owned, and everyone else must respect its decisions.

Rust takes this principle out of the comments and makes it explicit in the language. In Rust, every value has a clear owner; and when we say that one value *owns* another, we mean that when the owner is freed—or "dropped", in Rust terminology—the owned value gets dropped along with it. 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:

```
{
    let mut padovan = vec![1,1,1];  // vector allocated here
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);         // vector grown here, possibly
    }
    println!("P(1..10) = {:?}", padovan);
}                                   // dropped here
```

The type of `padovan` is `std::vec::Vec<i32>`, a vector of 32-bit integers. In memory, the final value of `padovan` will look something like this:



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 function (not shown) that contains this code; only the vector's buffer is allocated on the heap.

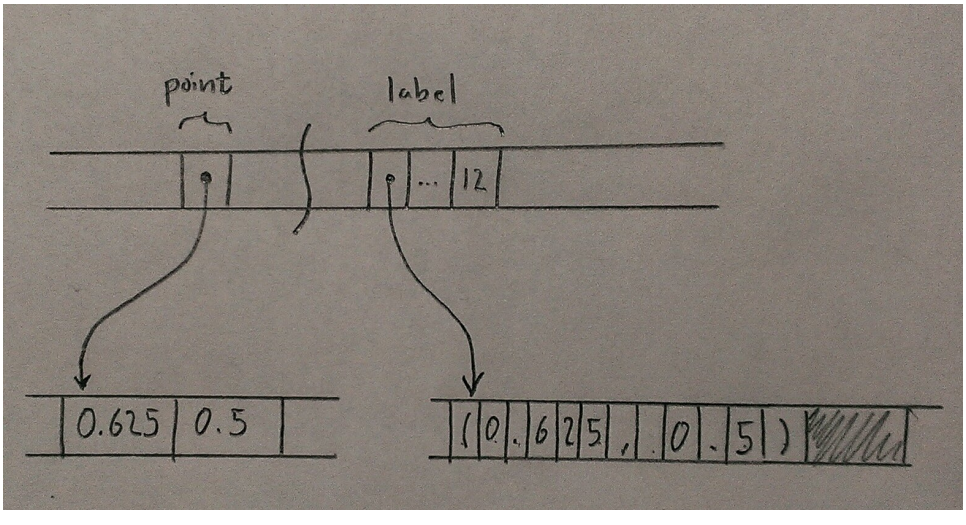
As with the string `s` earlier, the vector owns the buffer holding its elements. When the variable `padovan` goes out of scope at the end of the block, the program drops the vector. And since the vector owns its buffer, the buffer goes with it.

As another example of ownership, the pointer type `Box` simply owns a value stored on the heap. The `Box::new` function allocates an appropriately-sized block of heap space, and stores its argument there. Since a `Box` owns its referent, when the `Box` is dropped, the referent goes with it. So you can allocate a tuple in the heap like so:

```
{
    let point = Box::new((0.625, 0.5));
```

```
    let label = format!("{:?}", point);
    assert_eq!(label, "(0.625, 0.5)");
}
```

When the program calls `Box::new`, it allocates space for a tuple of two `f64` values on 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 this:

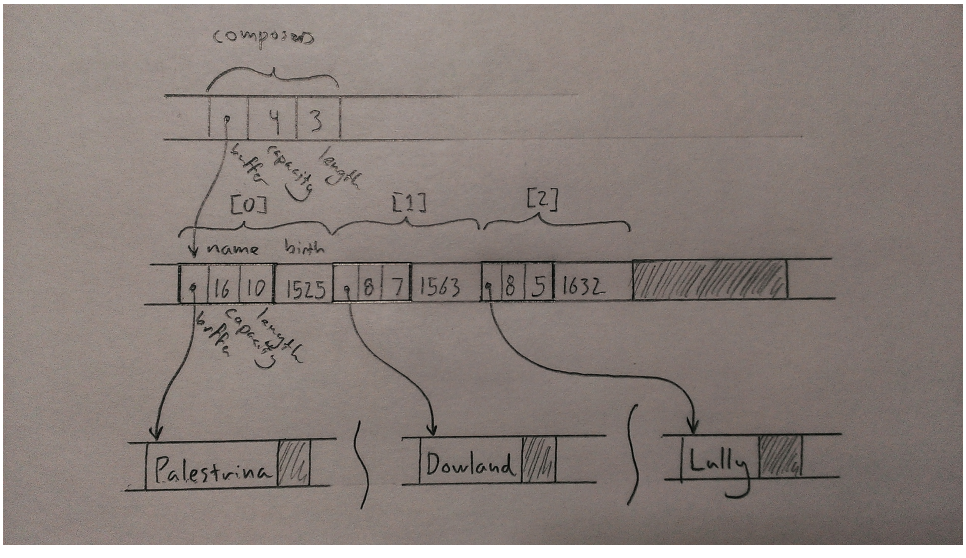


The stack frame itself holds the variables `point` and `label`, each of which refers to a block of memory on the heap that it owns.

Just as variables own their values, structures and enumerated types own their members, and tuples, 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 structures, each of which holds a string and a number. In memory, the final value of `composers` looks like this:



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 containers 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; otherwise, we couldn't be sure 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 (or at least, Rust without `unsafe` blocks) 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 the language is less powerful that the analyses Rust 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, the story we've told so far is still much too rigid to be usable. Rust extends this picture in several ways:

- You can move values from one owner to another. This allows you to build, rearrange, and tear down the tree.
- You can "borrow a reference" to a value; references are non-owning pointers, with limited lifetimes.
- The standard library provides the reference-counted pointer types `Rc` and `Arc`, which allow values to have multiple owners, under some restrictions.

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

Moves

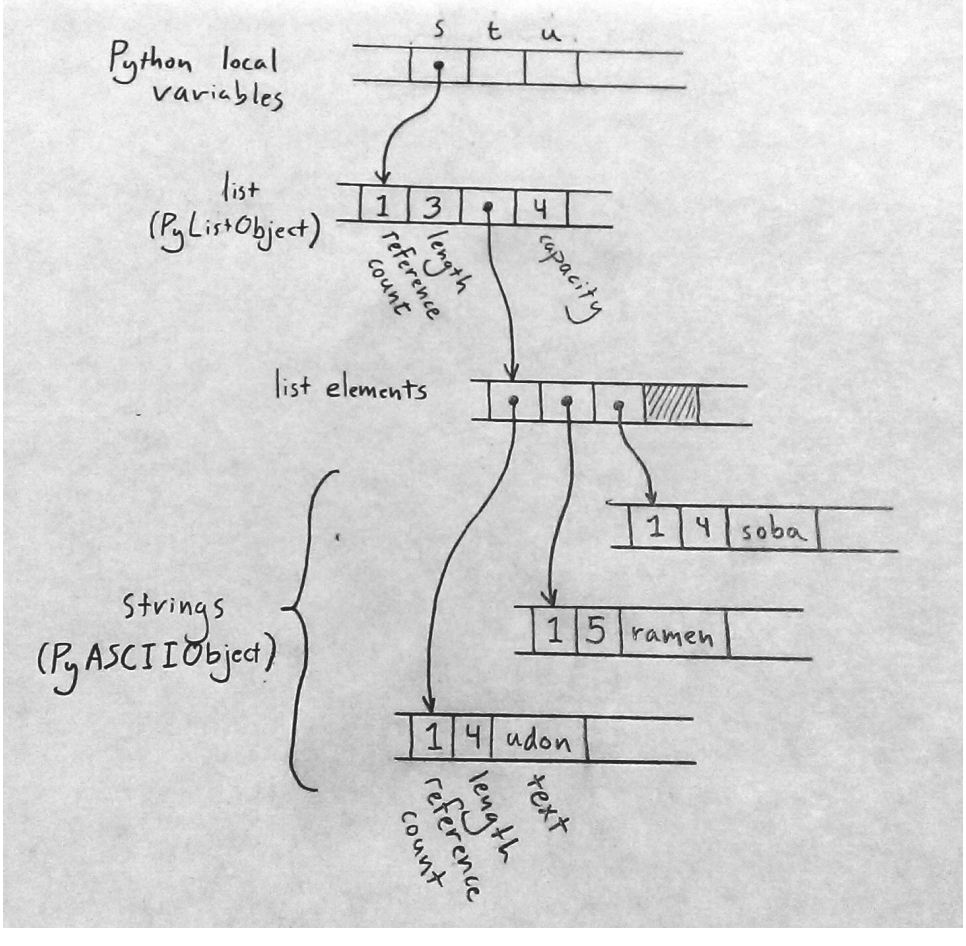
In Rust, for most types, operations like assigning a value to a variable, passing it to a function, or returning it from a function don't copy the value: they *move* it. The source relinquishes ownership of the value to the destination, and becomes uninitialized; the destination now controls the value's 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.

So, consider the following Python code:

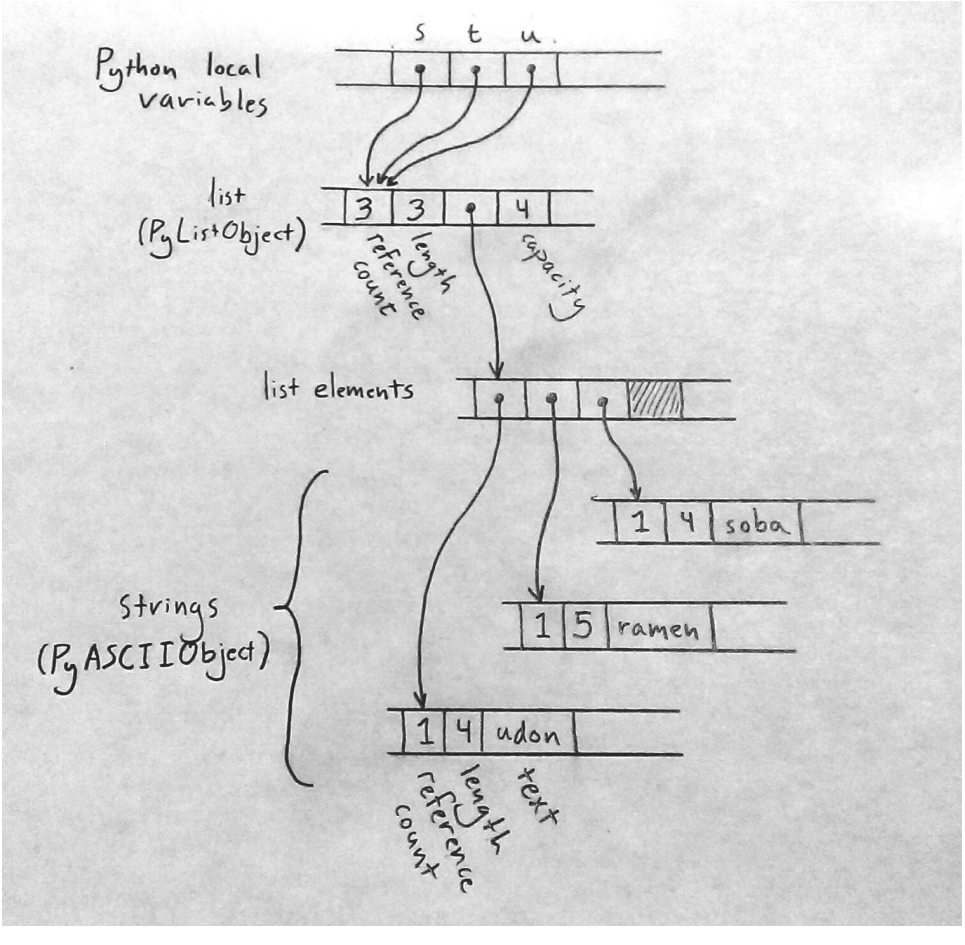
```
s = ['udon', 'ramen', 'soba']
t = s
u = s
```

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 this (with some fields left out):



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 this:

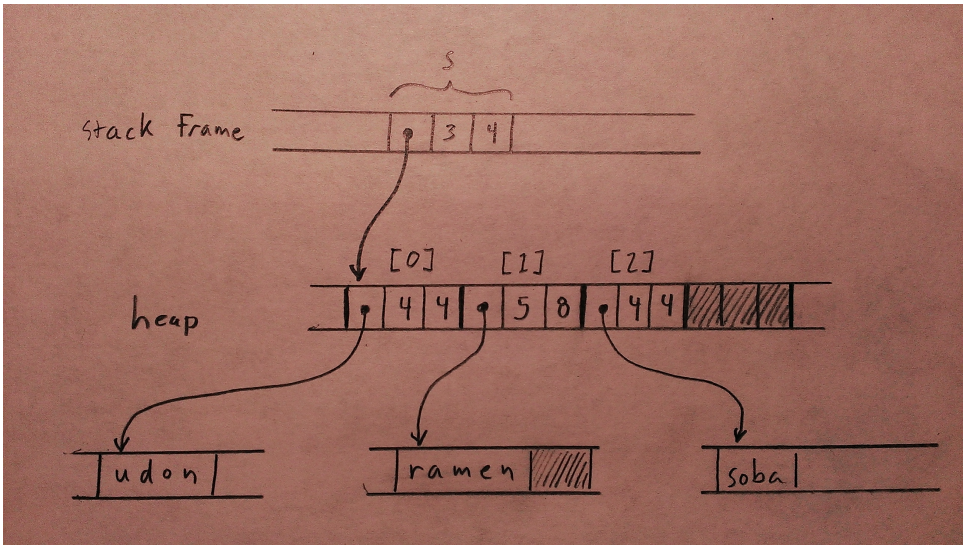


Python has copied the pointer from `s` into `t` and `u`, and updated the list's reference count to 3. Assignment in Python is cheap, but because it creates a new reference to the object, we must maintain reference counts to know when we can free the value.

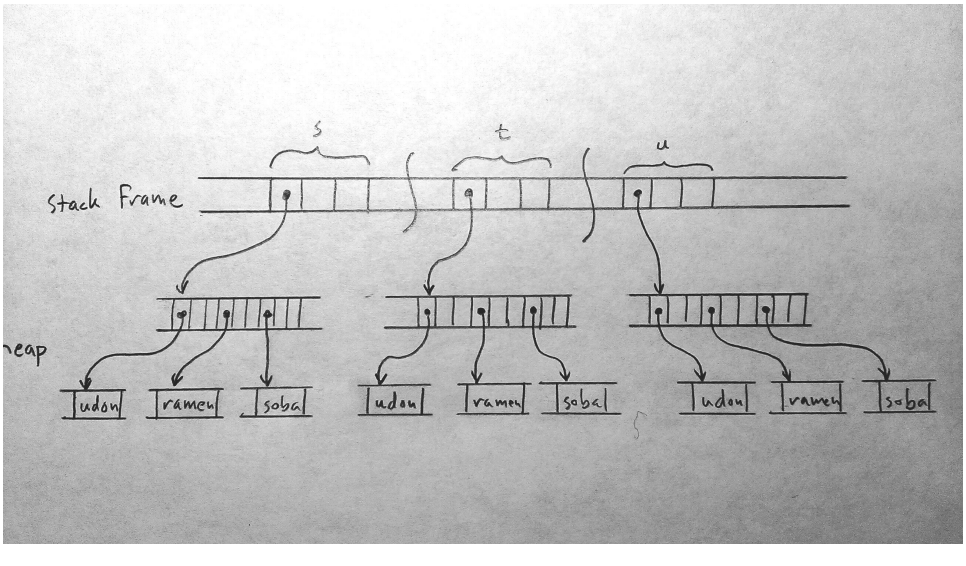
Now consider the analogous C++ code:

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

The original value of `s` looks like this in memory:



What happens when the program assigns `s` to `t` and `u`? In C++, assigning a `std::vector` produces a copy of the vector; `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:



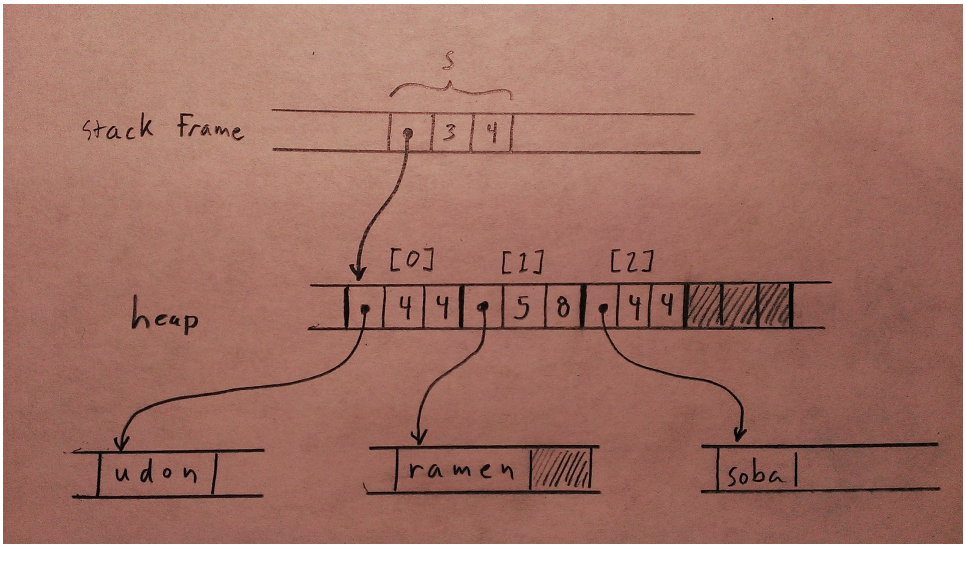
Depending on the values involved, assignment in C++ can consume unbounded amounts of memory and processor time. The advantage, however, is that it's easy for the program to decide when to free all this memory: when the variables go out of scope, everything allocated here gets cleaned up automatically.

In a sense, C++ and Python have chosen opposite tradeoffs: 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.

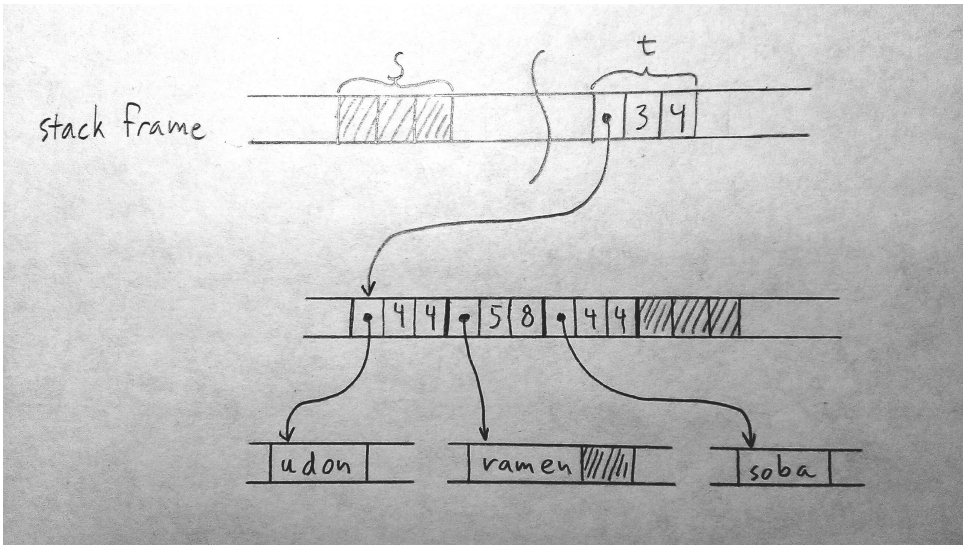
So what would the analogous program do in Rust? Here's the code:

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

After carrying out the initialization of `s`, since Rust and C++ use similar representations for vectors and strings, the situation looks just like it did in C++:



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 this:



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`
let u = s;
    ^
note: `s` moved here because it has type `Vec<String>`,
which is moved by default
let t = s;
    ^
```

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:

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

You could also recreate Python's behavior using Rust's reference-counted pointer types; we'll discuss those shortly in **section reference counting**.

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; returning a value from a function moves ownership to the caller. Calling a constructor moves the arguments into the constructed value. And so on.

You may now have a 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 it by value: 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 is passed by value to the vector's `push` method, which moves it onto the end of the structure. 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 of all, 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 very 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 examples above 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. So, 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); // ... okay to move from x here
} else {
    g(x); // ... and okay 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: after first iteration, x is uninitialized
}
```

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);
```

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];
let fifth = v[4];
```

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 above code with the error:

```
error: cannot move out of indexed content
let third = v[2];
            ^~~~
note: attempting to move value to here
let third = v[2];
    ^~~~~
```

It also makes a similar complaint about the move to `fifth`. These errors simply mean that you must find a way to move out the values you need that respects the limitations of the type. For example, here are three ways to move individual values out of a vector:

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

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

// Move a value out of the middle of the vector, and move the last
// element into its spot:
let third = v.swap_remove(2);
assert_eq!(third, "103");

// Swap in another value for the one we're taking out.
let second = std::mem::replace(&mut v[1], "substitute".to_string());
```

```
assert_eq!(second, "102");

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

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 indexed content" 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);
```

In fact, using `Option` this way is common enough that the type provides a `take` method for this very purpose. You could write the manipulation above more legibly as:

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

This has exactly the same effect as the original `let`.

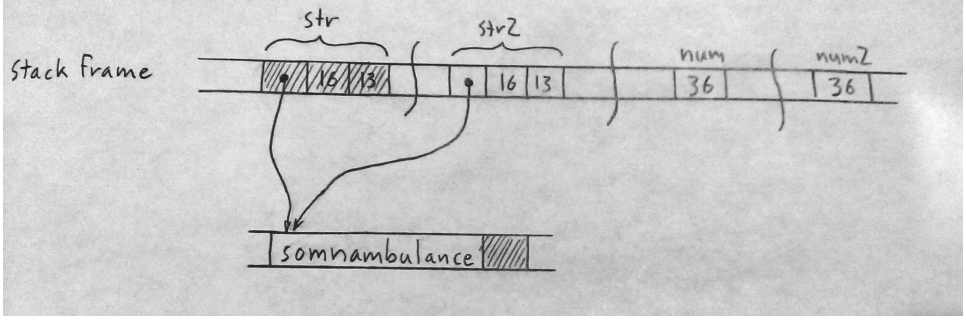
Copy types: the exception to moves

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 circumspection really isn't necessary.

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

```
let str = "somnambulance".to_string();
let str2 = str;

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

After running this code, memory looks like this:



As with the vectors earlier, assignment *moves* `str` to `str2`, so that we don't end up with two strings responsible for freeing the same buffer. However, the situation with `num` 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 `num`.

Moving a value leaves the source of the move uninitialized. But whereas it serves an essential purpose to treat `str` as valueless, treating `num` 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 explained above, `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.

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: use of moved value: `l.number`
println!("My label number is: {}", l.number);
                                   ^~~~~~~~
note: `l` moved here because it has type `Label`, which is non-copyable
print(l);
      ^
```

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 an `i32` with pretensions. There's no reason passing `l` to `print` should move the value.

But types being non-`Copy` is only the default. If all the fields of a structure or enumerated type are themselves `Copy`, then you can make the type itself `Copy` by placing the the attribute `#[derive(Copy, Clone)]` above the definition like so:

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

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

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

elicits the error:

```
error: the trait `Copy` may not be implemented for this type; field `name`
does not implement `Copy`
#[derive(Copy, Clone)]
         ^~~~
```

Why aren't user-defined types `Copy` by default, 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 custom copy and move constructors, Rust doesn't include any way to provide custom code to make a type `Copy`; it's either a plain byte-for-byte copy, or an explicit call to the `clone` method (which you must implement yourself). Rust's moves can't be customized either; a move is always just a byte-for-byte, shallow copy that leaves its source uninitialized.

`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, or 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++.

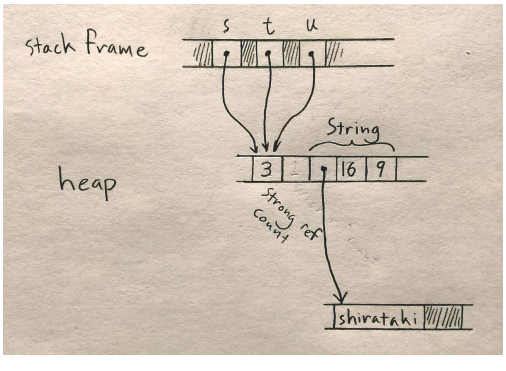
The `Rc` and `Arc` types are very similar; the only difference between them is that an `Arc` is safe to share between threads directly—the name `Arc` is short for "Atomic Reference Count"—whereas a plain `Rc` uses faster non-thread-safe code to update its reference count. If you don't need to share the 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 in the chapter 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 above code produces the following situation in memory:



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.

Just as with a `Box`, you can use any of `String`'s usual methods directly on an `Rc<T>`:

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

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

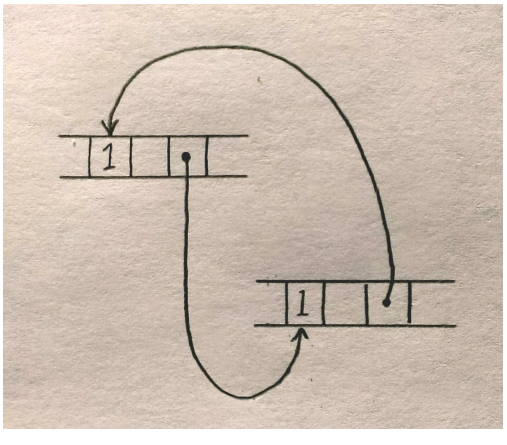
```
s.push_str(" noodles");
```

Rust will decline:

```
error: cannot borrow immutable borrowed content as mutable
    s.push_str(" noodles");
    ^~
```

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 5.

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:



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; if you combine those techniques with `Rc` pointers, you can create a cycle and leak memory.

1. 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. The 2011 C++ specification effectively precludes that representation; all modern C++ libraries use roughly the approach shown here.↩

CHAPTER 5

References and borrowing

All the pointer types we've seen so far—the simple `Box<T>` heap pointer, and the pointers internal to `String` and `Vec` values—are owning pointers: when the owner is dropped, the referent goes with it. 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. Rust requires it to be apparent simply from inspecting the code that no reference will 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 "requires it to be apparent" phrase there, you're in excellent company. The references themselves are nothing special; under the hood, they're just pointers. 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 multi-threaded programming is exciting enough to keep you up late thinking about the possibilities. This is Rust's radical wager, again.

As an example, let's suppose we're going to build a table of murderous Renaissance artists and their most celebrated works. 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 which 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` for debugging:

```
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:

```
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:
  Tenebrae Responsoria
  many madrigals
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)` above, the whole structure gets moved to the function, leaving the variable `table` uninitialized. 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 any more:

```
error: use of moved value: `table`
assert_eq!(table["Gesualdo"][0], "many madrigals");
           ^~~~~
note: `table` moved here because it has type `HashMap<String, Vec<String>>`,
which is non-copyable
show(table);
     ^~~~~
```

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 right way to handle this is to use references. A reference lets 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 "reference to `T`". Shared references are `Copy`.
- A *mutable reference* lets you both read and modify its referent. However, you may only have one mutable reference to a particular value active at a time. The expression `&mut e` yields a mutable reference to `e`'s value; you write its type as `&mut T`, which is pronounced "mutable reference to `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. This 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 the `sort_works` function the ability to read and modify our structure, as required by the vectors' `sort` method.

References as values

The above 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 very constrained examples to get a more detailed view of what's going on.

Implicit dereferencing

Superficially, Rust references resemble C++ references: they're both just pointers under the hood; and in the examples we showed, there was no need to explicitly dereference them. However, Rust references are really closer to C or C++ pointers.

You must use the `&` operator to create references, and in the general case, dereferencing does require explicit use of the `*` operator:

```
let x = 10;
let r = &x;
assert!(*r == 10);

let mut y = 32;
let mr = &mut y;
*mr *= 32;
assert!(*mr == 1024);
```

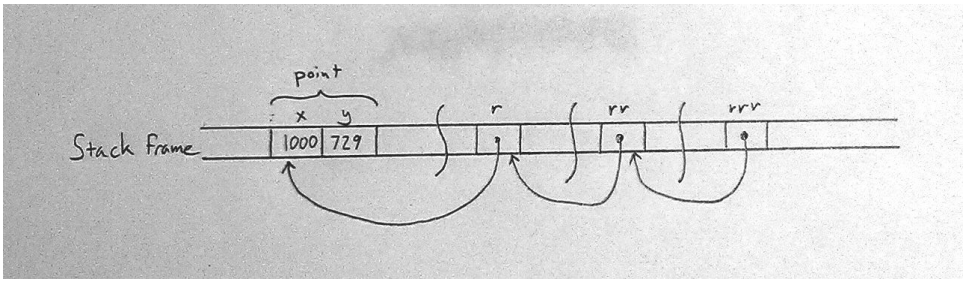
So why were there no uses of `*` in the artist-handling code? The `.` operator automatically follows references for you, so you can omit the `*` in many cases:

```
let point = (1000, 729);
let r = &point;
assert_eq!(r.0, 1000);
```

In the above, the reference `r.0` automatically dereferences `r`, as if you'd written `(*r).0`. This is why Rust has no analog to C and C++'s `->` operator: the `.` operator handles that case itself. In fact, the `.` operator will follow as many references as you give it:

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

(We've only written out the types here for clarity's sake; there's nothing here Rust couldn't infer.) In memory, that code builds a structure like this:



So the expression `rrr.y` actually traverses three pointers to get to its target, as directed by the type of `rrr`, before fetching its `y` field.

Method calls automatically follow references as well:

```
let mut v = vec![1968, 1973];
let mut r : &mut Vec<i32> = &mut v;
let rr : &mut &mut Vec<i32> = &mut r;
```

```
rr.sort_by(|a, b| b.cmp(a)); // reverse sort order
assert_eq!(**rr, [1973, 1968]);
```

Even though the type of `rr` is `&mut &mut Vec<i32>`, we can still invoke methods available on `Vec` directly on `rr`, like `sort_by`.

Rust's comparison operators "see through" any number of references as well, as long as both operands have the same type:

```
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 object, you must cast the references to raw pointers, which the comparison operators will *not* automatically dereference:

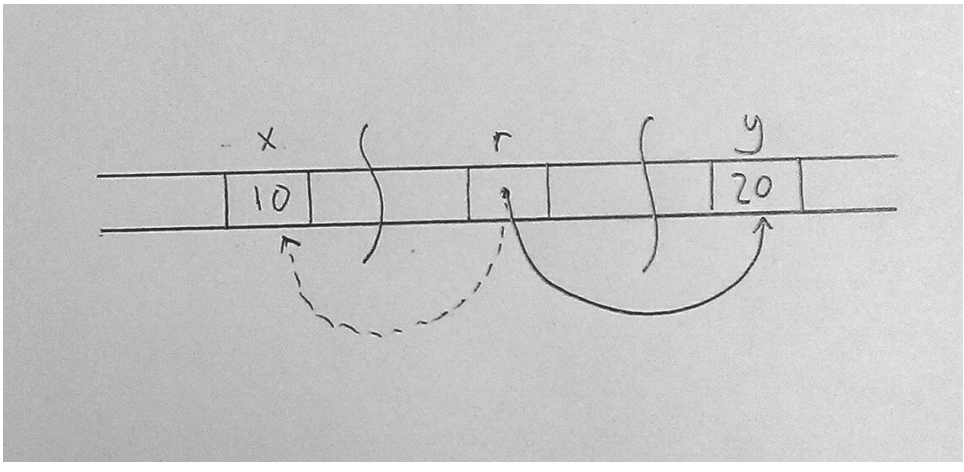
```
assert!(rx as *const i32 != ry as *const i32);
```

Assigning references

Like a C or C++ pointer, and unlike a C++ reference, assigning to a Rust reference makes it point at a new value:

```
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 `if` expression will change `r` to point to `y` instead:



References to slices and trait objects

The references we've shown so far are all simple pointers. However, as explained in the sections on slices and trait objects in Chapter 3, some references are "fat pointers": two-word values that include both a pointer and some additional information about its referent.

A reference to a slice of an array, vector, or string, written `&[T]` for some type `T`, or `&str` for a slice of a `String`, comprises a pointer to the first element included in the slice, and the length of the slice in elements. A reference to a trait object `&Tr`, for some trait `Tr`, comprises a pointer to a value that implements the trait `Tr`, and a pointer to an implementation of `Tr`'s methods appropriate for the referent's true type.

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 are non-owning pointers, which are not allowed to outlive their referents; they may be mutable or shared; and so on.

References are never null

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

C and C++ code often uses a null pointer to indicate the absence of a value: for example, the `malloc` function either returns 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

an `&T` value, as the non-zero address, so `Option<&T>` is just as efficient as a nullable pointer would be 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 their `&` 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);
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 we do with the reference:

- If the reference is being immediately assigned to a variable in a `let` statement (or is 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 our 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.

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. We'll start with the simplest example possible, and then add in interesting complications and explain how they work out.

Borrowing a local variable

Here's a pretty obvious case. You can't borrow a reference to a local and take it out of the local'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:

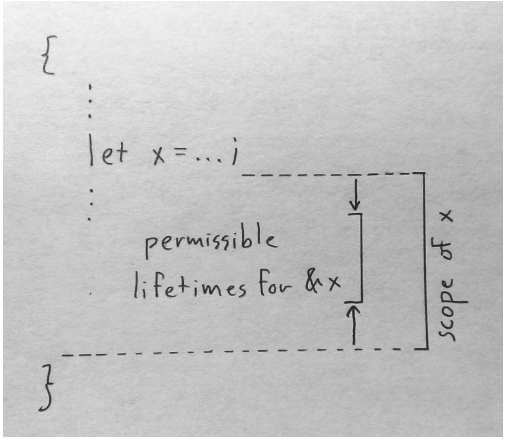
```
error: `x` does not live long enough
        r = &x;
             ^~
note: reference must be valid for the block suffix following statement 0 at ...
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1);  // bad: reads memory `x` used to occupy
...
note: ...but borrowed value is only valid for the block suffix following
statement 0 at ...
        let x = 1;
        r = &x;
    }
```

The "block suffix following statement 0" phrase isn't very clear. It generally refers to the portion of the program that some variable is in scope, from the point of its declaration to the end of the block that contains it. Here, the error messages talk about the "block suffixes" of the lifetimes of `r` and `x`. The compiler's complaint is that the reference `r` is still live when its referent `x` goes out of scope, 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.

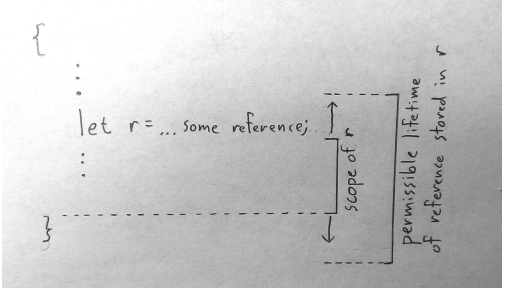
Rust tries to assign each reference in your program a *lifetime* that meets the constraints imposed by how the reference is used. A lifetime is some stretch of your program for which a reference could live: a lexical block, a statement, an expression, the scope of some variable, or the like.

Here's one constraint which should seem pretty obvious: If you have a variable `x`, then a reference to `x` must not outlive `x` itself:



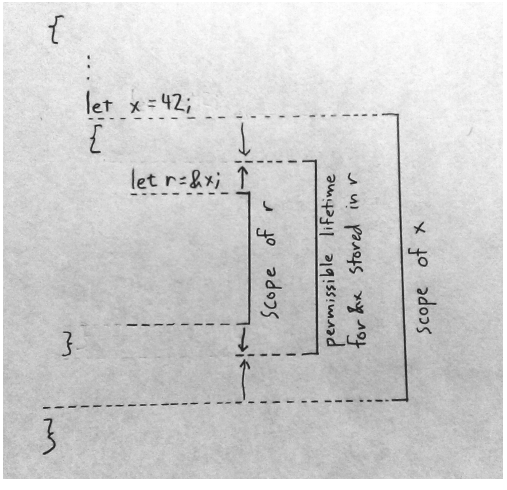
Beyond the point where `x` goes out of scope, the reference would be a dangling pointer. This is true even if `x` is some larger data structure instead of a simple `i32`, and you've borrowed a reference to some part of it: `x` owns the whole structure, so when `x` goes, every value it owns goes along with it.

Here's another constraint: If you store a reference in a variable `r`, the reference must be good for the entire lifetime of `r`:



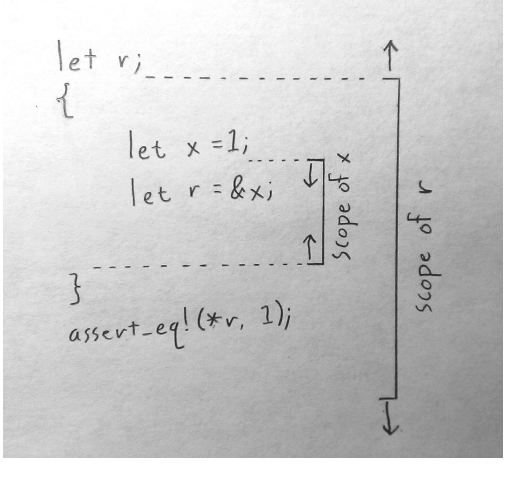
If the reference can't live at least as long as `r` does, then at some point `r` will be a dangling pointer. As before, this is true even if `r` is some larger data structure that contains the reference; if you build a vector of references, say, all of them must have lifetimes that enclose the vector's.

So we've got some situations that limit how long a reference's lifetime can be, and others that limit how short it can be. Rust simply tries to find a lifetime for every reference that satisfies these constraints. For example, the following code fragment shows a lifetime that satisfies the constraints placed on it:



Since we've borrowed a reference to `x`, the reference's lifetime must not extend beyond `x`'s scope. Since we've stored it in `r`, its lifetime must cover `r`'s scope. Since the latter scope lies within the former, Rust can easily find a lifetime that meets the constraints.

But in our original example, the constraints are contradictory:



There is simply no lifetime that is contained by `x`'s scope, and yet contains `r`'s scope. Rust recognizes this, and rejects the program.

This is the process Rust uses for all code. Data structures and function calls introduce 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 those constraints. This is not so different from the process C and C++ pro-

grammers impose on themselves; the difference is that Rust knows the rules, and enforces them.

Lifetimes are entirely figments of Rust's compile-time imagination. At run time, a reference is nothing but a pointer; its lifetime has been checked and discarded, and has no run-time representation.

Receiving references as parameters

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 which 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 Chapter 7, but for now we'll just call out a few rules that our code above 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 only access a mutable static 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.
- If you use a reference in a static's type, you must explicitly write out its lifetime. Lifetime names in Rust are lower-case identifiers with a `'` affixed to the front, like `'a` or `'party`. Our static `STASH` is alive for the program's entire execution; Rust names this maximal lifetime `'static`. In a reference type, the lifetime name goes between the `&` and the referent type, so `STASH`'s type must be `&'static i32`.

With those revisions made, we now have:

```
static mut STASH: &'static 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 above is actually shorthand for the following:

```
fn f<'a>(p: &'a i32) { ... }
```

Here, the lifetime `'a` is a "lifetime parameter" of `f`. When we write `fn f<'a>`, we're defining a function that will work for any given lifetime `'a`. So, the signature above says that `f` is 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 body of `f`. This assignment then becomes a point of contention:

```
STASH = p;
```

When we assign one reference to another, the source's lifetime must be at least as long as the destination's. But `p`'s lifetime is clearly not guaranteed to be as long as `STASH`'s, which is `'static`, so Rust rejects our code:

```
error: cannot infer an appropriate lifetime for automatic coercion due to
conflicting requirements
        STASH = p;
              ^~

note: first, the lifetime cannot outlive the lifetime 'a as defined on the
block at ...
fn f<'a>(p: &'a i32) {
    unsafe {
        STASH = p;
    }
}
note: ...so that reference does not outlive borrowed content
        STASH = p;
              ^~
note: but, the lifetime must be valid for the static lifetime...
note: ...so that expression is assignable (expected `&'static i32`, found `&i32`)
        STASH = p;
              ^~
```

At this point it's clear that our function can't accept just any reference as an argument. But 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: &'static 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 safe to do anyway.

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 as arguments

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:

```
// 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 covers 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 covers 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`. In general, 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?

```
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, but writing them out 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
        s = smallest(&parabola);
                     ^~~~~~~~
note: reference must be valid for the block suffix following statement 0 at...
    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
...
note: ...but borrowed value is only valid for the block suffix following
statement 0 at ...
        let parabola = [9, 4, 1, 0, 1, 4, 9];
        s = smallest(&parabola);
    }
```

Lifetimes in function signatures let Rust assess the relationships between the references you pass to the function and those the function returns, and ensure they're being used safely.

Structures 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`
```

Rust is skeptical about our type `S`:

```
error: missing lifetime specifier
            r: &i32
               ^~~
```

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
}
```

The amended definition reads: "`S` is a struct that, for any given lifetime `'a`, has a field `r` holding a reference with lifetime `'a` to an `i32`."

Each time you create a value of the type `S`, Rust tries to decide what lifetime would work for that value's lifetime parameter `'a`. Consider the expression `S { r: &x }`: this initializes `r` with `&x`, constraining `'a` to be no larger than `x`'s scope.

Recall that assigning a reference to a variable constrains the reference's lifetime to cover the variable's scope. This rule actually applies not just to references, but to any

type that takes a lifetime parameter, like our struct `S`. So the assignment `s = S { r: &x }` constrains the lifetime `'a` to be at least as long as that 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`. The type's lifetime parameter relates the containing value's lifetime to those of the references it contains. In this case, `'a` relates the lifetime of our `S` value to its `r` member, allowing Rust to detect the dangling pointer. Disaster averted!

Note that when a type has lifetime parameters, the lifetimes for each value of that type are distinct. If we had several values of type `S` running around in our example, each one would have its own independent `'a` lifetime; the values wouldn't necessarily constrain each other. (Naturally, if we assigned references back and forth between the values, that would introduce constraints following the usual rules.)

How does a type with a lifetime parameter behave when placed inside some other type?

```
struct T {
    s: S  // not adequate
}
```

Rust is skeptical, just as it was when we tried placing a reference in `S` without specifying its lifetime:

```
error: wrong number of lifetime parameters: expected 1, found 0
s: S
   ^~
```

We can't leave off `S`'s lifetime parameter here: Rust needs to know how a `T`'s lifetime relates to that of the reference in its `S`, in order to apply the same checks to `T` that it does for `S` and plain references.

We could give `s` the `'static` lifetime. This works:

```
struct T {
    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 a `T` can't possibly borrow a local variable; there are no special constraints on a `T`'s lifetime.

The other approach would be to give `T` its own lifetime parameter, and pass that to `S`:

```
struct T<'a> {
    s: S<'a>
}
```

By taking a lifetime parameter `'a` and using it in `s`'s type, we've allowed Rust to relate a `T` 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 parseRecord<'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 `parseRecord`, whatever references it contains must point into the input buffer we passed in, and nowhere else (except perhaps at `'static` values).

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;
    }
}
```

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 members 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 cover `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.

Newcomers to Rust often encounter difficulties of this sort: they know how to add lifetime parameters to their structures, and write lifetime names into their reference types, but don't recognize when they've placed tighter constraints on the references than they really need. Often the stricter definition works fine in simple situations, but once they encounter an unlucky arrangement of lifetimes, Rust rejects their correct program, citing risks the programmer can see will never come to pass. If this happens to you, check whether your lifetime parameters aren't entangling things you'd prefer to let vary independently.

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 above. When possible, let parameters' lifetimes vary independently:

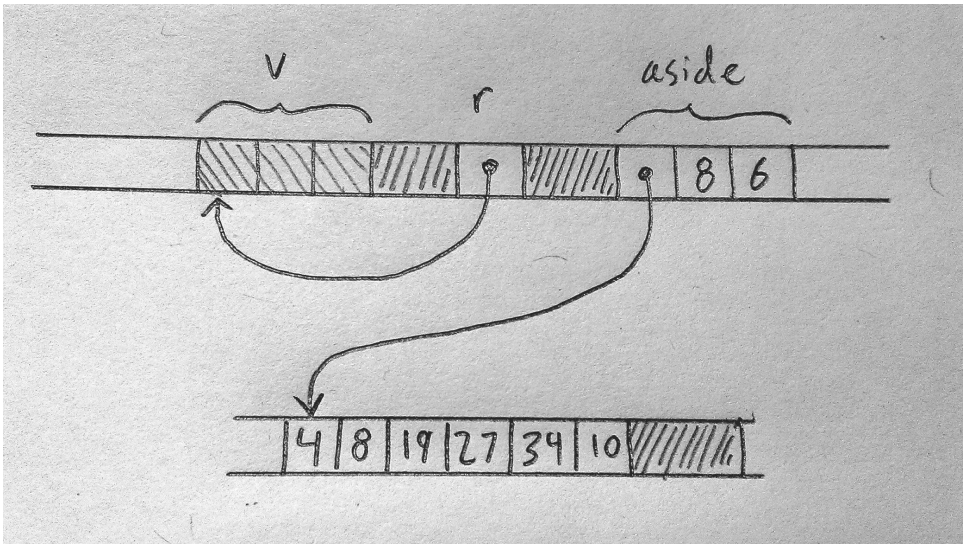
```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r } // looser
```

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, turning `r` into a dangling pointer:



The problem here is not that `v` goes out of scope while `r` still refers to it, but rather that `v`'s value gets moved elsewhere, leaving `v` uninitialized. Naturally, Rust catches the error:

```
error: cannot move out of `v` because it is borrowed
let aside = v;
    ^~~~
note: borrow of `v` occurs here
let r = &v;
        ^~
```

Throughout its lifetime, a shared reference makes its referent read-only: you may not assign to the referent or move its value elsewhere. In the code above, `r`'s lifetime covers the attempt to move the vector, so Rust rejects the program. If you change the program as shown below, there's no problem:

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];       // okay: 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);
    }
}
```

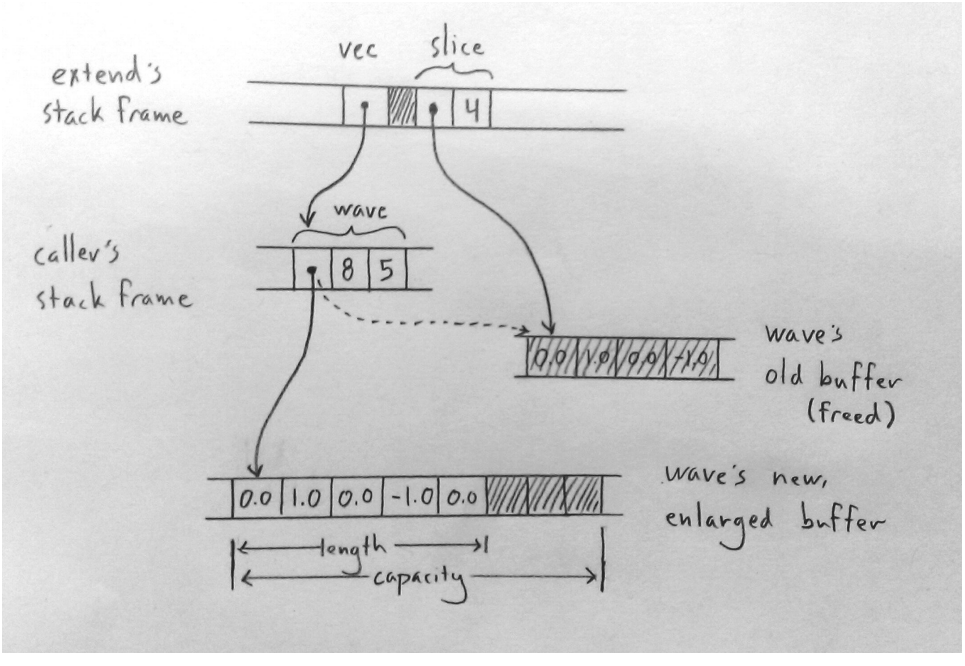
This is a slightly less flexible (and much less optimized) version of the standard library's `extend_from_slice` method on vectors. We can use it to build up a vector from slices of other vectors or arrays:

```
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 on 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 whose buffer is full, the vector 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 this:



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.

This sort of problem isn't unique to Rust: modifying collections while pointing into them is delicate territory in many languages. In C++, the specification of `std::vector` 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:

> [I]f 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.

Rust reports the problem with our call to `extend` at compile time:

```
error: cannot borrow `wave` as immutable because it is also borrowed as mutable
    extend(&mut wave, &wave);
                      ^~~~
note: previous borrow of `wave` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `wave` until the borrow ends
    extend(&mut wave, &wave);
                  ^~~~
note: previous borrow ends here
    extend(&mut wave, &wave);
                           ^
```

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 may not overlap. In our case, both references' lifetimes cover the call to `extend`, so Rust rejects the code.

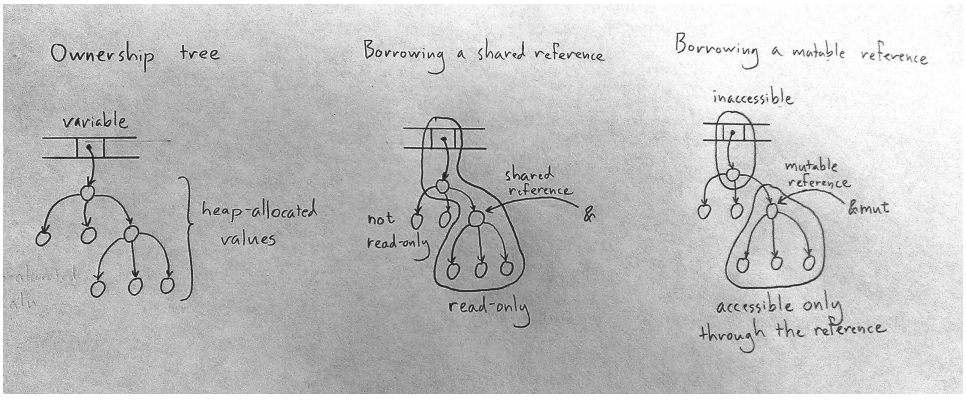
The errors above both stem from violations of Rust's rules for mutation and sharing:

- *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.
- Conversely, *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 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.

But Rust could also have treated our bug 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.

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:



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;     // okay: 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

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
```

It is okay to reborrow a shared reference from a shared reference:

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;         // okay: reborrowing shared as shared
let m1 = &mut r.1;     // error: can't reborrow shared as mutable
```

Reborrowing from a mutable reference is considered just another way of accessing the value through that reference, so it is permitted:

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // okay: reborrowing mutable from mutable
*m0 = 137;
let r1 = &mut m.1;      // okay: reborrowing shared from mutable,
                        // and doesn't overlap with m0
v.1;                    // error: access through other paths still forbidden
```

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 containers 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 container types in JavaScript do, but require heavier implementations as a result. C++'s `std::map` container 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);
  }
};
```

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 ???, but the above definitions work for this example.

If we write the Rust code corresponding to the use of `File` above, 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
    clone_from(&mut f, &f);
```

```
                        ^~
note: previous borrow of `f` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `f` until the borrow ends
    clone_from(&mut f, &f);
                       ^~
note: previous borrow ends here
    clone_from(&mut f, &f);
                         ^
```

This should look familiar. It turns out that two classic C++ bugs—failure to cope with self-assignment, and using invalidated iterators—are actually both 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. By requiring mutable access to be exclusive, Rust has fended off a wide class of everyday mistakes.

The immiscibility of shared and mutable references also really shines when writing concurrent code. A data race is only possible when some value is both shared shared between threads and mutable—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 concurrency in detail in "Concurrency" on page 30.

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.

XXX Lifetime elision

CHAPTER 6

Expressions

In this chapter, we'll cover the *expressions* of Rust, the building blocks that make up the body of Rust functions. A few concepts, such as closures and iterators, are deep enough that we will dedicate a whole chapter to them later on. 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 which 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 this in Chapter 2:

```
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 (*e*₁ `?` *e*₂ `:` *e*₃). 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.

Blocks and statements

Blocks, too, are expressions. 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 = try!(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. It makes no difference to Rust. The value of the block is the value of its last expression, `ip.to_string()`.

Rust's rules regarding semicolons cause mild but recurring perplexity in many programmers. Sometimes a semicolon is required, sometimes it may be dropped, sometimes it *must* be dropped. The rules are intuitive enough that no one ever seems to struggle with them, but to set your mind at ease, we present the full rules below. The key is that Rust does have statements after all.

The syntax of a block is:

```
{ stmt* expr? }
```

That is, a block contains zero or more statements, followed by an optional final expression. This final expression, if present, gives the block its value and type. Note that there is no semicolon after this expression.

As almost everything is an expression in Rust, there are only a few kinds of statements:

- empty statements
- declarations
- expression statements

An empty statement consists of a stray semicolon, all by itself. Rust follows the tradition of C in allowing this. Empty statements do nothing except convey a slight feeling of melancholy. We mention them only for completeness.

Declarations are described in a separate section below.

That leaves the expression statement, which is simply an expression followed by a semicolon. Here are three expression statements:

```
dandelion_control.release_all_seeds(launch_codes);

stats.launch_count += 1;

status_message =
    if self.stuck_seeds.is_empty() {
        "launch ok!".to_string()
    } else {
        format!("launch error: {} stuck seeds", self.stuck_seeds.len())
    };
```

The semicolon that marks the end of an expression statement may be omitted if the expression ends with `}` and its type is `()`. (This rule is necessary to permit, for example, an `if` block followed by another statement, with no semicolon between.)

And that is all: those are Rust's semicolon rules. The resulting language is both flexible and readable. 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 is `()`. To make a block produce a value, add an expression at the end without a trailing semicolon.

Declarations

A `let` declaration is the most common kind of declaration. We have already shown many of these. They declare local variables.

```
let binding: type = expr;
```

The type and initializer are optional. The semicolon is required.

The scope of a `let` variable starts immediately *after* the `let` declaration and extends to the end of the block. This matters when you have two different variables with the same name:

```
for line in buf_read.lines() {
    let line = try!(line);
    ...
}
```

Here the loop variable, `line`, is an `io::Result<String>`. Inside the loop, `try!()` returns early if an IO error occurred. Otherwise, the `String` is stored in a new variable, also called `line`. The new variable shadows the old one. We could have given the two variables different names, but in this case, using `line` for both is fine. Taking the wrapper off of something doesn't necessarily mean it needs a new name.

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 self.has_nickname() {
    name = self.nickname();
} else {
    name = generate_unique_name();
    self.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!)

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`:

```
fn show_files() -> Result<()> {
    let mut v = vec![];
    ...

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering {
        if a.mtime != b.mtime {
            a.mtime.cmp(&b.mtime).reverse()
        } else {
            a.path.cmp(&b.path)
        }
    }
```

```
    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` is not a closure. It cannot access local variables or arguments that happen to be in scope.

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 as we'll see, macros have a way of finding a use for every scrap of orthogonality the language provides.

`if` and `match`

The form of an `if` statement is familiar:

```
if condition₁ {
    block₁
} else if condition₂ {
    block₂
} else {
    blockₙ
}
```

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 analogous to 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, so it serves as the `default:` case.

For this kind of `match`, the compiler generally uses a jump table, just like a `switch` statement. 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. In 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 Chapter 9.

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: non-exhaustive 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
```

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
        };
```

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 $block_1$ runs, or it doesn't, and $block_2$ runs. This is shorthand for a `match` expression with just one pattern:

```
match expr {
    pattern => { block1 }
    _ => { block2 }
}
```

Loops

There are four looping expressions:

```
while condition {
    block
}

while let pattern = expr {
    block
}

loop {
    block
}

for binding in collection {
    block
}
```

Loops are expressions in Rust, but they don't produce useful values. The value of a loop is `()`.

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 result of *expr* either matches the given *pattern*, in which case the block runs, or it 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 *`collection`* expression, then evaluates the *`block`* once for each value in the collection. Many collection types are supported. 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 ???. 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 ps in &strings {
    println!("String {:?} is at address {:p}.", *ps, ps);
}
```

Here the type of `&strings` is `&Vec<String>` and the type of `ps` is `&String`.

Iterating over a `mut` reference provides a `mut` reference to each element:

```
for p in &mut strings {
    p.push('\n');  // add a newline to each string
}
```

Chapter iterators covers `for` loops in greater detail and shows many other ways to use iterators.

A `break` expression exits an enclosing loop. (In Rust, `break` works only in loops. It is not necessary in `match` expressions, which are unlike `switch` statements in this regard.)

A `continue` expression jumps to the next loop iteration.

A loop can be *labeled* with a lifetime. In the example below, `'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 {} in the {}.", spot, room);
            break 'search;
        }
    }
}
```

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 ()
}
```

`return` is familiar enough in statement-oriented code. In expression-oriented code it can be puzzling at first, so it's worth spending a moment on an example. Back in Chapter 2, we used the `try!()` macro to check for errors after calling a function that can fail:

```
let output = try!(File::create(filename));
```

This is shorthand for the following `match` expression:

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

The `match` expression first calls `File::create(filename)`. If that returns `Ok(val)`, then the whole `match` expression evaluates to `val`, so `val` is stored in `output` and we continue with the next line of code.

Otherwise, we hit the `return` expression, and it doesn't matter that we're in the middle of evaluating a `match` expression in order 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()`. This is how `try!()` magically propagates errors. There is nothing magic about the control flow; it's just using standard Rust parts.

Chapter macros explains in detail how `try!(File::create(filename))` is expanded into a `match` expression.

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 or not 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: not all control paths return a value
```

The error here is bogus. It is not actually possible to reach the end of the function without returning a value.

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; an infinite `loop` 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()`:

```
pub 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.

Function and method calls

We turn now from control flow to the other building blocks of Rust expressions: operators, function calls, and so on.

Function calls and method calls are much like those in other languages:

```
let room = player.location();
```

Rust typically distinguishes between references and the values they refer to, but in this case, as a convenience, Rust allows either a value or a reference to the left of the dot.

So, for example, if the type of `player` is `&Creature`, then it inherits the methods of the type `Creature`. You don't have to write `(*player).location()`. (Or `player->location()`, as you would in C++; there is no such syntax in Rust.) The same is true for smart pointer types, like `Box`. If `player` is a `Box<Creature>`, you may call `Creature` methods on it directly.

When calling a method that takes its `self` parameter by reference, Rust implicitly borrows the appropriate kind of reference, so you don't have to write `(&player).location()` either.

These conveniences apply only to the `self` argument. Static method calls have no `self` argument, so they do not automatically dereference or borrow their arguments. They are really just plain function calls:

```
String::from_utf8(bytes)  // `bytes` must be a Vec<u8>
```

Occasionally a program needs to refer to a generic method or type in an expression. The usual syntax for generic types, `Vec<T>`, doesn't work there:

```
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 advises 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 "Space Invader smiley".

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
```

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, a slice, or a vector:

```
pieces[i]          // array element
```

The value to the left of the brackets is automatically dereferenced.

Expressions like the three shown above 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` bindings.

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 }
```

Only the last form is useful as an iterator in a `for` loop, since a loop must have somewhere to start and we typically also want it, eventually, to end. But in array slicing, the other forms are useful too. 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]) {
    ...

    // 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 5.

The unary `*` operator is used to access the value pointed to by a reference. As we've already pointed out, in many places, Rust automatically follows references, 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`.

Occasionally, a method uses `*self =` to overwrite all of a value's fields at once:

```
impl Chessboard {
    fn restart_game(&mut self) {
        *self = Chessboard::new();
    }
}
```

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 3, debug builds check for integer overflow, and it causes a thread panic. The standard library provides methods like `a.wrapping_add(b)` for unchecked arithmetic.

Dividing an integer by zero causes a thread panic even in release builds. Integers have a method `a.checked_div(b)` that returns the result as an `Option` and never panics.

Unary `-` negates a number. It is supported only for signed integers. There is no unary `+`.

```
println!("{}", -100);     // -100
println!("{}", -100u32);  // error: unary negation of unsigned integer
println!("{}", +100);     // error: unexpected `+`
```

As in C, `a % b` computes the remainder, or modulus, of division. The result has the same sign as the left-hand 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 both, it does not need the `>>>` operator from Java and JavaScript.

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 `>=`. As with all the binary operators, the two operands 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 4, assignment *moves* values of non-copyable types, rather than implicitly copying them.

Compound assignment is supported:

```
total += item.price;
```

The value of any assignment is `()`, not the value being assigned.

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 builtin 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.

  However, as of this writing, casting a large floating-point value to an integer type that is too small to represent it can lead to undefined behavior. This can cause crashes even in safe Rust. It is a bug in the compiler.

- Values of type `bool`, `char`, or of a C-like `enum` type, may be cast to any integer type.

  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`, do not correspond to Unicode code points and therefore would not make valid `char` values. There is a standard method, `std::char::from_u32()`, which performs the runtime 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.

- Pointer casts and conversions between pointers and integers are also allowed. Such casts are of little use in safe code. See ???.

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` builtin trait. Rust performs these conversions automatically for values that otherwise wouldn't quite be the right type for the function argument they're being passed to, the variable they're being assigned to, and so on. We'll revisit the `Deref` trait in ???.

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
```

Closures can be called using ordinary function-call syntax:

```
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 ???.

Precedence and associativity

Table 1 gives a summary of Rust expression syntax.

Table 6-1. Table 1 - Expressions

Expression type	Example	Related traits
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); }`	`std::iter::IntoIterator`
	`while ok { ok = f(); }`	
	`while let Some(x) = it.next() { f(x); }`	
	`loop { next_event(); }`	
	`break`	
	`continue`	
	`return 0`	
macro invocation	`println!("ok")`	
closure	`|x, y| x + y`	
path	`std::f64::consts::PI`	
struct literal	`Point {x: 0, y: 0}`	
tuple field access	`pair.0`	`Deref`, `DerefMut`
struct field access	`point.x`	`Deref`, `DerefMut`
method call	`point.translate(50, 50)`	`Deref`, `DerefMut`

Expression type	Example	Related traits
function call	`stdin()`	`Fn(Arg0, ...) -> T`, `FnMut(Arg0, ...) -> T`, `FnOnce(Arg0, ...) -> T`
index	`arr[0]`	`Index`, `IndexMut` `Deref`, `DerefMut`
logical/bitwise NOT	`!ok`	`Not`
negation	`-num`	`Neg`
dereference	`*ptr`	`Deref`, `DerefMut`
borrow	`&val`	
type cast	`x as u32`	
multiplication	`n * 2`	`Mul`
division	`n / 2`	`Div`
remainder (modulus)	`n % 2`	`Rem`
addition	`n + 1`	`Add`
subtraction	`n - 1`	`Sub`
left shift	`n << 1`	`Shl`
right shift	`n >> 1`	`Shr`
bitwise AND	`n & 1`	`BitAnd`
bitwise exclusive OR	`n ^ 1`	`BitXor`
bitwise OR	`n | 1`	`BitOr`
less than	`n < 1`	`std::cmp::PartialOrd`
less than or equal	`n <= 1`	`std::cmp::PartialOrd`
greater than	`n > 1`	`std::cmp::PartialOrd`
greater than or equal	`n >= 1`	`std::cmp::PartialOrd`
equal	`n == 1`	`std::cmp::PartialEq`
not equal	`n != 1`	`std::cmp::PartialEq`
logical AND	`x.ok && y.ok`	
logical OR	`x.ok || backup.ok`	
range	`start .. stop`	
assignment	`x = val`	
compound assignment	`x += 1`	

All of the operators that can usefully be chained are left-associative. That is, a chain of operations like `a - b - c` groups like `(a - b) - c`, not `a - (b - c)`. The operators that can be chained in this way are all the ones you might expect:

```
*   /   %   +   -   <<   >>   &   ^   |   &&   ||
```

Unlike C, assignment can't be chained: `a = b = 3` does not assign the value `3` to both *a* and *b*. Assignment is rare enough in Rust that you won't miss this shorthand.

The comparison operators, `as`, and the range operator `..` can't be chained at all.

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.

The next few chapters cover features in this area: first modules and crates, which give your program structure, then structs and enums, which do the same for your data.

CHAPTER 7
Program structure

Suppose you're writing a program that simulates the growth of ferns, from the level of individual cells on up. Your program, like a fern, will start out very simple, with all the code, perhaps, in a single file—just the spore of an idea. As it grows, it will start to have internal structure. Different pieces will have different purposes. It will branch out into multiple files. It may cover a whole directory tree. In time it may become a significant part of a whole software ecosystem.

This chapter covers the features of Rust that help keep your program organized: crates and modules. We'll also show how to document and test Rust code and how to publish open-source libraries on crates.io.

Crates

Rust programs are made of *crates*. Each crate is a single Rust library or executable. For your fern simulator, you might use third-party libraries for 3D graphics, bioinformatics, parallel computation, and so on. These libraries are distributed as crates.

The easiest way to see what crates are and how they work together is to use `cargo build` with the `--verbose` flag to build an existing project that has some dependencies. We did this, using the Mandelbrot set program from "Concurrency" on page 30 as our example. The results are shown in Example 7-1.

Example 7-1: Verbose `cargo build` output

```
$ cd mandelbrot
$ cargo clean    # delete previously compiled code
$ cargo build --verbose
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading image v0.6.1
 Downloading crossbeam v0.2.9
 Downloading gif v0.7.0
```

```
 Downloading png v0.4.2

 ... (downloading and compiling many more crates)

   Compiling png v0.4.2
     Running `rustc .../png-0.4.2/src/lib.rs
                 --crate-name png
                 --crate-type lib
                 --extern num=.../libnum-a2e6e61627ca7fe5.rlib
                 --extern inflate=.../libinflate-331fc425bf167339.rlib
                 --extern flate2=.../libflate2-857dff75f2932d8a.rlib
                 ...`
   Compiling image v0.6.1
     Running `rustc .../image-0.6.1/./src/lib.rs
                 --crate-name image
                 --crate-type lib
                 --extern png=.../libpng-16c24f58491a5853.rlib
                 ...`
   Compiling mandelbrot v0.1.0 (file://.../mandelbrot)
     Running `rustc src/main.rs
                 --crate-name mandelbrot
                 --crate-type bin
                 --extern crossbeam=.../libcrossbeam-ba292320058da7df.rlib
                 --extern image=.../libimage-254ec48c8f0684f2.rlib
                 ...`
$
```

We reformatted the `rustc` command lines for readability, and we deleted a lot of compiler options that aren't relevant to the discussion below, replacing them with "`...`".

You might recall that by the time we were done, the Mandelbrot program's `main.rs` contained three `extern crate` declarations:

```
extern crate num;
extern crate image;
extern crate crossbeam;
```

These lines simply tell Rust that `num`, `image`, and `crossbeam` are external libraries, not part of the Mandelbrot program itself.

We also specified in our `Cargo.toml` file which version of each crate we wanted:

```
[dependencies]
num = "0.1.27"
image = "0.6.1"
crossbeam = "0.2.8"
```

Example 7-1 tells the story of how this information is used. When we run `cargo build`, Cargo starts by downloding source code for the specified versions of these crates from `crates.io`. Then, it reads those crates' `Cargo.toml` files, downloads *their*

dependencies, and so on recursively. For example, the source code for version 0.6.1 of the `image` crate contains a `Cargo.toml` file that includes this:

```
[dependencies]
byteorder = "0.4.0"
num = "0.1.27"
enum_primitive = "0.1.0"
glob = "0.2.10"
```

Seeing this, Cargo knows that before it can use `image`, it must fetch these crates as well. Later on, we'll see how to tell Cargo to fetch source code from a git repository or the local filesystem rather than `crates.io`.

Once it has obtained all the source code, Cargo compiles all the crates. It runs `rustc`, the Rust compiler, once per crate in the project's dependency graph. When compiling libraries, Cargo uses the `--crate-type lib` option. This tells `rustc` not to look for a `main()` function but instead to produce an `.rlib` file containing compiled code in a form that later `rustc` commands can use as input. When compiling a program, Cargo uses `--crate-type bin`, and the result is a binary executable for the target platform: `mandelbrot.exe` on Windows, for example.

With each command, Cargo passes `--extern` options to tell `rustc` the filename of each library the crate will use. That way, when `rustc` sees a line of code like `extern crate crossbeam;`, it knows where to find that crate on disk. The Rust compiler needs access to that `.rlib` in order to check that the library features used in the code actually exist in the library and that they're being used correctly.

Modules

Modules are Rust's namespaces. They're containers for the functions, types, constants, and so on that make up your Rust program or library. Whereas crates are about code sharing across projects, modules are about code organization *within* a project. They look like this:

```
mod spores {
    /// A cell made by an adult fern. It disperses on the wind as part of
    /// the fern life cycle. A spore grows into a prothallus -- a whole
    /// separate organism, up to 5mm across -- which produces a zygote,
    /// which becomes a new fern. (Plant sex is complicated.)
    pub struct Spore {
        ...
    }

    /// Simulate the production of a spore by meiosis.
    pub fn produce_spore(factory: &mut Sporangium) -> Spore {
        ...
    }
```

```
    /// Mix genes to prepare for meiosis (part of interphase).
    fn recombine(parent: &mut Cell) {
        ...
    }
}
```

A module is a collection of *items*, named features like the `Spore` struct and the two functions shown above. The `pub` keyword makes an item public, so it can be accessed from outside the module. Anything that isn't marked `pub` is private.

```
let s = spores::produce_spore(frond);  // ok

spores::recombine(&mut cell);  // error: `recombine` is private
```

Modules can nest, and it's fairly common to see a module that's just a collection of submodules:

```
mod plant_structures {
    pub mod roots {
        ...
    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}
```

In this way, we could write out a whole program, with a huge amount of code and a whole hierarchy of modules, all in a single source file. Actually working that way is a pain, though, so there's an alternative.

Modules in separate files

A module can also be written like this:

```
mod spores;
```

Earlier, we included the body of the `spores` module, wrapped in curly braces. Here, we're instead telling the Rust compiler that the `spores` module lives in a separate file, which we'll call `spores.rs`:

```
// spores.rs

/// A cell made by an adult fern...
pub struct Spore {
    ...
}

/// Simulate the production of a spore by meiosis.
pub fn produce_spore(frond: &mut Frond) -> Spore {
```

```
    ...
}

/// Mix genes to prepare for meiosis (part of interphase).
fn recombine(parent: &mut Cell) {
    ...
}
```

`spores.rs` contains only the items that make up the module. It doesn't need any kind of boilerplate to declare that it's a module.

The location of the code is the *only* difference between this `spores` module and the version we showed in the previous section. The rules about what's public and what's private are exactly the same either way. And Rust never compiles modules separately, even if they're in separate files: when you build a Rust crate, you're recompiling all of its modules.

A module can have its own directory. When Rust sees `mod spores;`, it checks for both `spores.rs` and `spores/mod.rs`; if neither file exists, or both exist, that's an error. For this example, we used `spores.rs`, because the `spores` module did not have any submodules. But consider the `plant_structures` module we wrote out earlier. If we decide to split that module and its three submodules into their own files, the resulting project would look like this:

```
fern_sim/
├── Cargo.toml
└── src/
    ├── main.rs
    └── plant_structures/
        ├── mod.rs
        ├── leaves.rs
        ├── roots.rs
        └── stems.rs
```

In `main.rs`, we declare the `plant_structures` module:

```
mod plant_structures;
```

This would cause Rust to load `plant_structures/mod.rs`, which declares the three submodules:

```
// in plant_structures/mod.rs
pub mod roots;
pub mod stems;
pub mod leaves;
```

The content of those three modules is stored in separate files named `leaves.rs`, `roots.rs`, and `stems.rs`, located alongside `mod.rs` in the `plant_structures` directory.

Paths and imports

The `::` operator is used to access features of a module. Code anywhere in your project can refer to any standard library feature by writing out its *absolute path*:

```
if s1 > s2 {
    ::std::mem::swap(&mut s1, &mut s2);
}
```

This function name, `::std::mem::swap`, is an absolute path, because it starts with `::`. `::std` refers to the top-level module of the standard library. `::std::mem` is a submodule within the standard library, and `::std::mem::swap` is a public function in that module.

You could write all your code this way, spelling out `::std::f64::consts::PI` and `::std::collections::HashMap::new` every time you want a circle or a dictionary. The alternative is to *import* features into the modules where they're used.

```
use std::mem::swap;

if s1 > s2 {
    swap(&mut s1, &mut s2);
}
```

The `use` declaration causes the name `swap` to be a local alias for `::std::mem::swap` throughout the enclosing block or module. Paths in `use` declarations are automatically absolute paths, so there is no need for a leading `::`.

Several names can be imported at once:

```
use std::collections::{HashMap, HashSet};  // import both

use std::io::prelude::*;  // import everything
```

This is just shorthand for writing out all the individual imports:

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

// all the public features of std::io::prelude:
use std::io::prelude::Read;
use std::io::prelude::Write;
use std::io::prelude::BufRead;
use std::io::prelude::Seek;
```

Modules do *not* automatically inherit names from their parent modules. For example, suppose we have this in our `main.rs`:

```
// main.rs
enum AminoAcid { ... }
mod proteins;
```

Then the code in `proteins.rs` does not automatically see the type `AminoAcid`:

```
// proteins.rs
fn synthesize(sequence: &[AminoAcid])  // error: AminoAcid is not defined
...
```

Instead, each module starts with a blank slate and must import the names it uses:

```
// proteins.rs
use super::AminoAcid;  // explicitly import from parent

fn synthesize(sequence: &[AminoAcid])  // ok
...
```

The keyword `super` has a special meaning in imports: it's an alias for the parent module. Similarly, `self` is an alias for the current module. While paths in imports are treated as absolute paths by default, these keywords let you override that and import from relative paths.

While modules aren't the same thing as files, there is a natural analogy between modules and the files and directories of a Unix filesystem. The `use` keyword creates aliases, just as the `ln` command creates links. Paths, like filenames, come in absolute and relative forms. `self` and `super` are like the `.` and `..` special directories. And `extern crate` grafts another crate's root module into your project. It is a lot like mounting a filesystem.

The standard prelude

We said a moment ago that each module starts with a "blank slate", as far as imported names are concerned. But the slate is not *completely* blank.

For one thing, the standard library `std` is automatically included in every project. It's as though your `lib.rs` or `main.rs` contained an invisible declaration for it:

```
extern crate std;
```

Furthermore, a few particularly handy names, like `Vec` and `Result`, are automatically imported into each module: the *standard prelude*. Rust behaves as though every module, including the root module, started with the following import:

```
use std::prelude::v1::*;
```

The standard prelude contains a few dozen commonly used traits and types. One thing it does *not* contain is `std` itself. So the occasional module that needs to refer to `std` directly (a rare need) must import it, like this:

```
use std;
```

In Chapter 2, we mentioned that libraries sometimes provide modules named `prelude`. But `std::prelude::v1` is the only prelude that is ever imported automatically.

Naming a module `prelude` is just a convention that tells users it's meant to be imported using `*`.

Items, the building blocks of Rust

A module is made up of *items*, the individual features that make up a Rust program. Every item has a name and can be either `pub` or private, with the exception of `impl` blocks, which we'll get to in a second. There are several kinds of item, and the list is really a list of the language's major features:

- **Functions.** We have seen a great many of these already.
- **Types.** User-defined types are introduced using the `struct`, `enum`, and `trait` keywords. We'll dedicate a chapter to each of them, in good time; a simple struct looks like this:

```
pub struct Point2d {
    pub x: f64,
    pub y: f64
}
```

  A struct's fields, even private fields, are accessible throughout the module where the struct is declared. Visibility is always per-module. This turns out to be great for software design. It cuts down on boilerplate "getter" and "setter" methods, and it largely eliminates the need for C++ `friend` declarations (which Rust does not have). A single module can define several types that work closely together. Code in that module can access implementation details of all the types, while still hiding those implementation details from the rust of your program.

- **`impl` blocks.** Methods are attached to types using `impl` blocks:

```
impl Point2d {
    pub fn distance_from_origin(self) -> f64 {
        f64::hypot(self.x, self.y)
    }
}
```

  The syntax is explained in ???. An `impl` block can't be marked `pub`: the type itself is either public or it isn't. Individual methods can be marked `pub`, though. Private methods, like private struct fields, are visible throughout the module where they're declared.

- **Constants.** The `const` keyword introduces a constant. The syntax is just like `let` except that it may be marked `pub`, and the type is required. Also, `UPPER_CASE_NAMES` are conventional for constants:

```
pub const ROOM_TEMPERATURE: f64 = 20.0;  // degrees Celsius
```

  The `static` keyword introduces a static item, which is very nearly the same thing:

```
pub static ROOM_TEMPERATURE: f64 = 68.0;  // degrees Fahrenheit
```

A constant is a bit like a C++ `#define`: the value is compiled into your code every place it's used. A static is a variable that's set up before your program starts running and lasts until it exits. Use constants for magic numbers and strings in your code. Use statics for larger amounts of data.

There are no `mut` constants, so you must also use statics on those very rare occasions when you must have a global variable:

```
/// How many threads are running. Always use atomic operations
/// to access this variable!
static mut ACTIVE_THREAD_COUNT: usize = 0;
```

As discussed in Chapter 5, reading and writing a `mut` static are unsafe operations. There is no global mutable state in safe Rust.

??? offers a few alternatives.

- **Modules.** We're talking about these right now, of course.
- **Imports.** Even though they're just aliases, imports can be public too:

  ```
  // in plant_structures/mod.rs
  ...
  pub use self::leaves::Leaf;
  pub use self::roots::Root;
  ```

  This means that `Leaf` and `Root` are public features of the `plant_structures` module. They're still simple aliases for `plant_structures::leaves::Leaf` and `plant_structures::roots::Root`.

  The standard prelude is written as just such a series of `pub` imports.

Attributes

Attributes are Rust's catch-all syntax for writing miscellaneous instructions and advice to the compiler. For example, suppose you're getting this warning:

```
libgit2.rs: warning: type `git_revspec` should have a camel case name
    such as `GitRevspec`, #[warn(non_camel_case_types)] on by default
```

But you chose this name for a reason, and you just wish Rust would shut up about it. You can disable the warning by adding an `#[allow]` attribute on the type:

```
#[allow(non_camel_case_types)]
pub struct git_revspec {
    ...
}
```

Conditional compilation is another feature that's written using an attribute, the `#[cfg]` attribute:

```
// Only include this module in the project if we're building for Android.
#[cfg(target_os="android")]
mod mobile;
```

Very occasionally, we need to micromanage the inline expansion of functions, an optimization that we're usually happy to leave to the compiler. We can use the `#[inline]` attribute for that:

```
/// Adjust levels of ions etc. in two adjacent cells
/// due to osmosis between them.
#[inline]
fn do_osmosis(c1: &mut Cell, c2: &mut Cell) {
    ...
}
```

The complier treats `#[inline]` as a suggestion. Rust also supports the more insistent `#[inline(always)]`, to request that a function be expanded inline at every call site, and `#[inline(never)]`, to ask that a function never be inlined.

Some attributes, like `#[cfg]` and `#[allow]`, can be attached to a whole module and apply to everything in it. Others, like `#[inline]`, must be attached to individual items. As you might expect for a catch-all feature, each attribute is custom-made and has its own set of supported arguments. The Rust Reference documents the full set of supported attributes in detail.

To attach an attribute to a whole crate, add it at the top of the `main.rs` or `lib.rs` file, before any items, and write `#!` instead of `#`.

```
// libgit2_sys/lib.rs
#![allow(non_camel_case_types)]

pub struct git_revspec {
    ...
}

pub struct git_error {
    ...
}
```

The `#!` tells Rust to attach an attribute to the enclosing item rather than whatever comes next: in this case, the `#![allow]` attribute attaches to the whole `libgit2_sys` crate, not `struct git_revspec`.

`#!` can also be used inside functions, structs, and so on, but it's only typically used at the beginning of a file, to attach an attribute to the whole module or crate.

Unit tests

A simple unit testing framework is built into Rust. Tests are ordinary functions marked with the `#[test]` attribute.

```
#[test]
fn math_works() {
    let x: i32 = 1;
```

```
    assert!(x.is_positive());
    assert_eq!(x + 1, 2);
}
```

`cargo test` runs all the tests in your project.

```
$ cargo test
   Compiling math_test v0.1.0 (file:///.../math_test)
     Running target/release/math_test-e31ed91ae51ebf22

running 1 test
test math_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

(You'll also see some output about "doc-tests", which we'll get to in a minute.)

This works the same whether your crate is an executable or a library. You can run specific tests by passing arguments to Cargo: `cargo test math` runs all tests that contain `math` somewhere in their name.

Tests commonly use the `assert!` and `assert_eq!` macros from the Rust standard library. `assert!(expr)` succeeds if `expr` is true. Otherwise it causes a thread panic, which causes the test to fail. `assert_eq!(v1, v2)` is just like `assert!(v1 == v2)` except that if the assertion fails, the error message shows both values.

You can use these macros in ordinary code, to check invariants, but note that `assert!` and `assert_eq!` are included even in release builds. Use `debug_assert!` and `debug_assert_eq!` instead to write assertions that are checked only in debug builds.

To test error cases, add the `#[should_panic]` attribute to your test:

```
/// This test passes only if division by zero causes a panic,
/// as we claimed in the previous chapter.
#[test]
#[should_panic(expected="divide by zero")]
fn test_divide_by_zero_error() {
    1 / 0;  // should panic!
}
```

Functions marked with `#[test]` are conditionally compiled. When you run `cargo test`, Cargo builds a copy of your program with your tests and the test harness enabled. A plain `cargo build` or `cargo build --release` skips the testing code. This means your unit tests can live right alongside the code they test, accessing internal implementation details if they need to, and yet there's no runtime cost. However, it can result in some warnings. For example:

```
fn roughly_equal(a: f64, b: f64) -> bool {
    (a - b).abs() < 1e-6
}
```

```
#[test]
fn trig_works() {
    assert!(roughly_equal(std::f64::consts::PI.cos(), -1.0));
}
```

In a testing build, this is fine. In a non-testing build, `roughly_equal` is unused, and Rust will complain:

```
$ cargo build
   Compiling math_test v0.1.0 (file:///.../math_test)
src/main.rs:1:1: 4:2 warning: function is never used: `roughly_equal`,
#[warn(dead_code)] on by default
```

So the convention, when your tests get substantial enough to require support code, is to put them in a `tests` module and declare the whole module to be testing-only using the `#[cfg]` attribute:

```
#[cfg(test)]   // include this module only when testing
mod tests {
    fn roughly_equal(a: f64, b: f64) -> bool {
        (a - b).abs() < 1e-6
    }

    #[test]
    fn trig_works() {
        assert!(roughly_equal(std::f64::consts::PI.cos(), -1.0));
    }
}
```

Rust's test harness uses multiple threads to run several tests at a time, a nice side benefit of your Rust code being thread-safe by default. This means that, technically, the Mandelbrot program we showed in Chapter 2 was not the second multithreaded program in that chapter, but the third! The `cargo test` run in "Writing and running unit tests" on page 19 was the first.

Integration tests

Your fern simulator continues to grow. You've decided to put all the major functionality into a library that can be used by multiple executables. It would be nice to have some tests that link with the library the way an end user would, using `fern_sim.rlib` as an external crate. Also, you have some tests that start by loading a saved simulation from a binary file, and it is awkward having those large test files in your `src` directory. Integration tests help with these two problems.

Integration tests are `.rs` files that live in a `tests` directory alongside your project's `src` directory. When you run `cargo test`, Cargo compiles each integration test as a separate, standalone crate, linked with your library and the Rust test harness. Here is an example:

```
// tests/unfurl.rs - Fiddleheads unfurl in sunlight

extern crate fern_sim;
use fern_sim::{Fern, Terrarium};

const ONE_HOUR: f64 = 60.0 * 60.0;

#[test]
fn test_unfurl() {
    let mut world = Terrarium::load("tests/unfurl_files/fiddlehead.tm");
    assert!(world.fern(0).is_furled());
    world.apply_sunlight(ONE_HOUR);
    assert!(world.fern(0).is_fully_unfurled());
}
```

Note that the integration test includes an `extern crate` declaration, since it uses `fern_sim` as a library.

`cargo test` runs both unit tests and integration tests. To run only a specific integration test, use the `--test` option:

```
$ cargo test --test unfurl
   Compiling fern_sim v0.1.0 (file:///home/jorendorff/dev/rustbook/tests/fern_sim)
     Running target/debug/unfurl-c1e66668db2a636f

running 1 test
test test_unfurl ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

CHAPTER 8
Structures

Rust structures, usually just called "structs", resemble `struct` types in C and C++; Python's class instances and JavaScript's objects play analogous roles in those languages. Rust has three kinds of struct types: *named-field*, *tuple-like*, and *unit-like*.

Named-field structs

The definition of a named-field struct type looks like this:

```
// A rectangle of eight-bit grayscale pixels.
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

This declares a type `GrayscaleMap` with two fields named `pixels` and `size`, of the given types. You can construct a value of this type with a "struct expression", like this:

```
let width = 1024;
let height = 576;
let image = GrayscaleMap { size: (width, height),
                           pixels: vec![0; width * height] };
```

You can refer to a struct's fields using the familiar `.` operator:

```
assert_eq!(image.size, (1024, 576));
assert_eq!(image.pixels.len(), 1024 * 576);
```

It's a convention in Rust for all types, structs included, to have names that capitalize the first letter of each word, and use lower-case letters otherwise, like `GrayscaleMap`.

When creating a struct value, you can use another struct to supply values for fields you omit. In a struct expression, if the named fields are followed by `..E` for some expression `E`, then `E` must be another value of the same struct type, from which Rust

initialized any unmentioned fields. Suppose we have a struct representing a monster in a game:

```
struct Broom {
    name: String,
    height: u32,
    health: u32,
    position: (f32, f32, f32),
    intent: BroomIntent
}
```

Chopping a monster broom in half with an axe naturally produces two brooms, each of half the size, but with the same health and other characteristics as the original:

```
// Receive the input Broom by value, taking ownership.
fn chop(b: Broom) -> (Broom, Broom) {
    // Since String is not Copy, broom1 takes ownership of b's name.
    let mut broom1 = Broom { height: b.height / 2, .. b };

    // Here broom2 must clone broom1's name, to avoid stealing it.
    let mut broom2 = Broom { name: broom1.name.clone(), .. broom1 };

    // Give each fragment a distinct name.
    broom1.name.push_str(" I");
    broom2.name.push_str(" II");

    (broom1, broom2)
}
```

With that definition in place, we can create a broom, chop it in two, and see what we get:

```
let hokey = Broom {
    name: "Hokey".to_string(),
    height: 60,
    health: 100,
    position: (100.0, 200.0, 0.0),
    intent: BroomIntent::FetchWater
};

let (hokey1, hokey2) = chop(hokey);
assert_eq!(hokey1.name, "Hokey I");
assert_eq!(hokey1.health, 100);

assert_eq!(hokey2.name, "Hokey II");
assert_eq!(hokey2.health, 100);
```

Tuple-like structs

The second kind of struct type is called a “tuple-like struct”, because it resembles a tuple:

```
struct Bounds(usize, usize);
```

You construct a value of this type much as you would construct a tuple, except that you must include the struct name:

```
let image_bounds = Bounds(1024, 768);
```

The values held by a tuple-like struct are called "elements", just as the values of a tuple are. You access them just as you would a tuple's:

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

At the most fundamental level, named-field and tuple-like structs are very similar. The choice of which to use boils down to questions of legibility, ambiguity, and brevity. Identifying fields by name provides the reader more information, and is perhaps more robust against typos. But when a type is widely used, and has few fields whose ordering is well-established by convention (say, *x* and *y* coordinates, or width and height pairs), the brevity of tuple-like structs is valuable.

Unit-like structs

The third kind of struct is a little obscure: it declares a struct type with no elements at all:

```
struct Onesuch;
```

A value of such a type occupies no memory, much like the unit type `()`. Rust doesn't bother actually storing unit-like struct values in memory or generating code to operate on them, because it can tell everything it might need to know about the value from its type alone. But logically, an empty struct is a type with values like any other—or more precisely, a type of which there is only a single value:

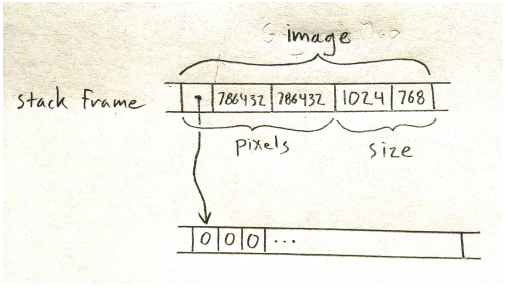
```
let o = Onesuch;
```

You've already encountered a unit-like struct when reading about "Fields and elements" on page 132. Whereas an expression like `3..5` is shorthand for the struct value `Range { start: 3, end: 5 }`, the expression `..`, a range omitting both endpoints, is shorthand for the unit-like struct value `RangeFull`.

Unit-like structs can also be useful when working with traits, which we'll describe in Chapter 10.

Struct layout

In memory, both named-field and tuple-like structs are the same thing: a collection of values, of possibly mixed types, laid out in a particular way in memory. For example, the `GrayscaleMap` value we built above looks like this:



Unlike C and C++, Rust doesn't make specific promises about how it will order a struct's fields or elements in memory; this diagram shows only one possible arrangement. However, Rust does promise to store fields' values directly in the struct's block of memory: whereas JavaScript, Python, and Java would have a `GrayscaleMap` hold pointers to a vector and bounds, each allocated in their own blocks, Rust embeds the `Vec<u8>` header and the bounds directly in the `GrayscaleMap` value.

Defining methods with `impl`

Throughout the book we've been calling methods on all sorts of values. We've pushed elements onto vectors with `v.push(e)`; fetched their length with `v.len()`; checked `Result` values for errors with `r.expect("msg")`; and so on.

Rust lets all types have methods, not just designated "object" types. For example, the signed numeric types like `i32` all have an `abs()` method that returns their absolute value. Enumerations can have methods as well. So in Rust we generally talk about calling methods on values, and the term "object" doesn't come up much.

You can define methods on any struct type you define. Rather than appearing inside the struct definition, as in C++ or Java, Rust methods appear in a separate `impl` block. For example:

```
/// A last-in, first-out queue of characters.
struct Queue {
    older: Vec<char>,   // older elements, eldest last.
    younger: Vec<char>  // younger elements, youngest last.
}

impl Queue {
    /// Push a character onto the back of a queue.
    fn push(self: &mut Queue, c: char) {
        self.younger.push(c);
    }

    /// Pop a character off the front of a queue. Return `Some(c)` if there
    /// was a character to pop, or `None` if the queue was empty.
    fn pop(self: &mut Queue) -> Option<char> {
```

```
            if self.older.is_empty() {
                if self.younger.is_empty() {
                    return None;
                }

                // Bring the elements in younger over to older, and put them in
                // the promised order.
                use std::mem::swap;
                swap(&mut self.older, &mut self.younger);
                self.older.reverse();
            }

            // Now older is guaranteed to have something. Vec's pop method
            // already returns an Option, so we're set.
            self.older.pop()
        }
    }
```

An `impl` block is simply a collection of `fn` definitions, each of which becomes a method on the type named at the top of the block. Here we've defined a struct type `Queue`, and then given it two methods, `push` and `pop`.

The comments starting with `///` above the `Queue` type itself and the two methods are documentation comments. The `rustdoc` program knows how to extract these from your program and produce on-line and printed documentation for your modules. We describe `rustdoc` in more detail in Chapter 7.

Rust passes a method the value it's being called on as its first argument, which must have the special name `self`. In our example, `push` and `pop` access their `Queue` fields as `self.older` and `self.younger`. Unlike C++ and Java, where the members of the "this" object are directly visible in method bodies as unqualified identifiers, a Rust method must use `self` to refer to the value it was called on, similar to the way Python methods use `self`, and the way JavaScript methods use `this`.

Since `push` and `pop` need to modify the `Queue`, they both declare `self` to have the type `&mut Queue`. However, when you call a method, you don't need to write out borrowing that mutable reference yourself; the ordinary method call syntax takes care of that implicitly. So with these definitions in place, you can use `Queue` like this:

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('0');
q.push('1');
assert_eq!(q.pop(), Some('0'));

q.push('∞');
assert_eq!(q.pop(), Some('1'));
assert_eq!(q.pop(), Some('∞'));
assert_eq!(q.pop(), None);
```

Simply writing `q.push(...)` borrows a mutable reference to `q`, since that's what the `push` method's `self` requires. But if a method doesn't need to modify its `self`, then you can define it to take a shared reference instead. For example:

```
impl Queue {
    fn is_empty(self: &Queue) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }
}
```

Again, the method call expression knows which sort of reference to borrow:

```
assert!(q.is_empty());
q.push(' ');
assert!(!q.is_empty());
```

Or, if a method wants to take ownership of `self`, it can take `self` by value:

```
impl Queue {
    fn split(self: Queue) -> (Vec<char>, Vec<char>) {
        (self.older, self.younger)
    }
}
```

Calling this `split` method looks like the other method calls:

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('P');
q.push('D');
assert_eq!(q.pop(), Some('P'));
q.push('X');

let (older, younger) = q.split();
// q is now uninitialized.
assert_eq!(older, vec!['D']);
assert_eq!(younger, vec!['X']);
```

But note that, since `split` takes its `self` by value, this *moves* the `Queue` out of `q`, leaving `q` uninitialized. Since `split`'s `self` now owns the queue, it's able to move the individual vectors out of it, and return them to the caller.

Since it's so common for methods in an `impl Queue` block to have a first argument of `self: &Queue`, `self: &mut Queue`, or `self: Queue`, Rust provides a shorthand: for those three cases, simply writing `&self`, `&mut self` or `self` as the first argument, with no type, serves just as well. So we could write our `impl` block a bit more succinctly this way:

```
impl Queue {
    fn push(&mut self, c: char) { ... }
    fn pop(&mut self) -> Option<char> { ... }
    fn is_empty(&self) -> bool { ... }
```

```
    fn split(mut self) -> (Vec<char>, Vec<char>) { ... }
}
```

Omitting `self`'s type is the usual style in Rust code, since it's generally obvious.

You can also define methods that don't take `self` as an argument at all. These become functions associated with the type itself, not with any specific value of the type. Following the tradition established by C++ and Java, Rust calls these *static methods*. They're often used to provide constructor functions, like this:

```
impl Queue {
    fn new() -> Queue {
        Queue { older: Vec::new(), younger: Vec::new() }
    }
}
```

To use this method, we refer to it as `Queue::new`: the type name, a double colon, and then the method name. Now our example code becomes a bit more svelte:

```
let mut q = Queue::new();

q.push('*');
...
```

It's conventional in Rust for constructor functions to be named `new`; we've already seen `Vec::new`, `Box::new`, `HashMap::new`, and others. But there's nothing special about the name `new`. It's not a keyword, and types often have other static methods that serve as constructors, like `Vec::with_capacity`.

Although you can have many separate `impl` blocks for a single type, they must all be in the same crate that defines that type. However, Rust does let you attach your own methods to other types; we'll explain how in Chapter 10.

If you're used to C++ or Java, separating a type's methods from its definition may seem unusual, but there are several advantages to doing so:

- It's always easy to find a type's data members. In large C++ class definitions, one may need to scour pages of member function definitions to be sure one hasn't missed any of the class's data members; in Rust, they're all in one place.
- Rust uses the same `impl` block syntax for defining methods on other types. Tuple-like structs, unit-like structs, and `enum` types can all have methods. Separate `impl` blocks provide a uniform way to define them, regardless of what the definition looks like.
- Rust also uses `impl` blocks when implementing a trait's methods for some type or family of types. Since Rust code uses traits so heavily, trait `impl`s are probably more common than "inherent `impl`s", that define methods directly on a type, like our `impl Queue` block above.

Generic structs

Our definition of `Queue` above is unsatisfying: it is written to store characters, but there's nothing about its structure or methods that is specific to characters at all. If we were to define another queue type that held, say, `String` values, the code could be identical, except that `char` would be replaced with `String`. That would be a waste of time.

Fortunately, Rust types can be *generic*, meaning that their definition is a template into which you can plug whatever types you like. For example, here's a definition for `Queue` that can hold values of any type:

```
struct Queue<T> {
    older: Vec<T>,
    younger: Vec<T>
}
```

You can read the `<T>` in `Queue<T>` as "for any element type `T` ...". So this definition reads, "For any type `T`, a `Queue<T>` is two vectors holding two fields of type `Vec<T>`." So, for example, in `Queue<String>`, `T` is `String`, so this is a queue of `String` values. In `Queue<char>`, `T` is `char`, and we get a type identical to the `char`-specific definition we started with. Perhaps obviously, `Vec` itself is a generic type, defined in just this way.

In generic type definitions, the type names used in <angle brackets> are called "type parameters". They're conventionally uppercase letters, but any `CamelCase` name will do. Single letters work well because anyone reading your code can see at a glance that `T` or `W` or `X` is not the name of a type they can look up in the documentation; it's a type parameter, so they need to look at the first line of the generic item it appears in. (Your team, we hope, trusts you not to name a struct `X`.)

An `impl` blocks for a generic type looks like like this:

```
impl<T> Queue<T> {
    fn new() -> Queue<T> {
        Queue { older: Vec::new(), younger: Vec::new() }
    }

    fn push(&mut self, t: T) {
        self.younger.push(t);
    }

    fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }

    ...
}
```

You can read the line `impl<T> Queue<T>` as something like, "for any type `T`, here are some methods available on `Queue<T>`." Then, you can use the type parameter `T` as a type in the method definitions.

We've used Rust's shorthand for `self` parameters above; writing out `Queue<T>` everywhere becomes a mouthful and a distraction. As another shorthand, every `impl` block, generic or not, defines the special type parameter `Self` (note the `CamelCase` name) to be whatever type we're adding methods to. In the above, `Self` would be `Queue<T>`, so we can abbreviate `Queue::new`'s definition a bit further:

```
fn new() -> Self {
    Queue { older: Vec::new(), younger: Vec::new() }
}
```

You might have noticed that, in the body of `new`, we didn't need to write the type parameter in the construction expression `Queue<T> { ... }`; simply writing `Queue { ... }` was good enough. This is Rust's type inference at work: since there's only one type that works for that expression—namely, `Queue<T>`—Rust supplies it for us. However, you'll always need to supply type parameters in function signatures and type definitions. Rust doesn't infer those; instead, it uses those explict types as the basis from which it infers types within function bodies.

Structs with lifetime parameters

As we discussed in the earlier section "Structures containing references" on page 108, if a struct type contains references, you must name those references' lifetimes. For example, here's a structure that might hold references to the greatest and least elements of some slice:

```
struct Extrema<'elt> {
    greatest: &'elt i32,
    least: &'elt i32
}
```

Earlier, we invited you to think of a declaration like `struct Queue<T>` as meaning that, given any specific type `T`, you can make a `Queue<T>` that holds that type. Similarly, you can think of `struct Extrema<'elt>` as meaning that, given any specific lifetime `'elt`, you can make an `Extrema<'elt>` that holds references with that lifetime.

Here's a function to scan a slice and return an `Extrema` value whose fields refer to its elements:

```
fn find_extrema<'s>(slice: &'s [i32]) -> Extrema<'s> {
    let mut greatest = &slice[0];
    let mut least = &slice[0];

    for i in 1..slice.len() {
```

```
            if slice[i] < *least    { least    = &slice[i]; }
            if slice[i] > *greatest { greatest = &slice[i]; }
        }
        Extrema { greatest: greatest, least: least }
    }
```

Here, since `find_extrema` borrows elements of `slice`, which has lifetime `'s`, the `Extrema` struct we return also uses `'s` as the lifetime of its references.

Because it's so common for the return type to use the same lifetime as the arguments, Rust lets us omit the lifetimes in some cases. We could also have written `find_extrema`'s signature like this, with no change in meaning:

```
fn find_extrema(slice: &[i32]) -> Extrema {
    ...
}
```

Granted, we *might* have meant `Extrema<'static>`, but that's pretty unusual. Rust provides a shorthand for the common case.

Deriving common traits for struct types

For convenience, Rust can automatically implement some common traits for your struct types. Of course, you can implement all of these with your own custom code if you like; we'll show you how in Chapter 10. But implementations for these basic traits are often so formulaic that letting Rust write them for you is perfectly adequate.

`Copy` and `Clone`

In the "Ownership and moves" chapter, we introduced "`Copy` types: the exception to moves" on page 87: types whose values get copied, rather than moved, by assignment and similar operations. As we explained there, only types with a simple ownership structure can be `Copy`; and even among those types where it's possible, `Copy` isn't the default.

If all your structs' fields or elements are `Copy`, then you can ask Rust to make your struct `Copy` too by placing a `derive` attribute above the definition. For example:

```
#[derive(Copy, Clone)]
struct Complex { r: f64, i: f64 }
```

Here, since `f64` is `Copy`, Rust permits this derivation, and now `Complex` is `Copy` as well.

`Debug`: printing struct values

While you're developing your code, it's handy to be able to print out values of the types you've defined, for diagnostics and logging. The easiest way to accomplish this

is to have Rust derive an implementation of the `Debug` trait for your type. Adding `Debug` to the prior example:

```
#[derive(Copy, Clone, Debug)]
struct Complex { r: f64, i: f64 }
```

With this definition in place, `println!`, `format!`, and related macros can handle `Complex` values, if you use the `{:?}` formatting argument:

```
use std::f64::consts::PI;
let sixth = Complex { r: 0.5, i: (PI / 3.0).sin() };
println!("A sixth root of 1.0 is {:?}.", sixth);
```

This code produces the output:

```
A sixth root of 1.0 is Complex { r: 0.5, i: 0.8660254037844386 }.
```

This isn't the ideal way to print a complex number; most people would find something like `0.5 + 0.8660254037844386i` more legible. But for user-facing output you should implement the `Display` trait, intended for that purpose. `Debug` output is geared towards people working with the code itself, so the form generated for the `#[derive(Debug)]` attribute is often perfectly adequate.

`PartialEq`: checking structs for equality

To make a struct type comparable with the `==` and `!=` operators, you can derive the `PartialEq` trait automatically, assuming the struct's fields or elements are all comparable in that way themselves. For example:

```
#[derive(Copy, Clone, Debug, PartialEq)]
struct Complex { r: f64, i: f64 }

// Return the complex conjugate of `z`: the same number, with the
// imaginary part negated.
fn conj(z: &Complex) -> Complex { Complex { i: -z.i, ..*z } }

let z = Complex { r: -1.0, i: 0.5 };
assert!(z != conj(&z));
assert_eq!(conj(&z), Complex { r: -1.0, i: -0.5 });
```

Since we've implemented `PartialEq`, we can now use the `!=` operator and the `assert_eq!` macro on `Complex` values. The `assert_eq!` macro also needs its arguments to be `Debug`, so that it can print them if the assertion fails.

The automatically generated equality comparison considers all the type's components, so if your type has fields which must be ignored, you'll need to provide your own implementation of the `PartialEq` trait.

`PartialOrd`: ordering structs

If your type is `PartialEq`, then you can also support the ordered comparison operators like `<` and `>=` by deriving an implementation of the `PartialOrd` trait. Rust's generated `PartialOrd` implementations take a blindly mechanical approach to comparing two values of a struct type: working through the struct's fields or elements in the order they appear in the definition, the ordering is that of the first pair that is not equal; and if all of them are equal, then the pair of structs are equal. Obviously, this requires all the component types to be `PartialOrd` themselves.

If we have a type representing an IPv4 address, placing the most significant octet first in the structure causes Rust's derived `PartialOrd` implementation to order addresses lexicographically:

```
#[derive(Copy, Clone, PartialEq, PartialOrd)]
struct IPv4(u8, u8, u8, u8);

assert!(IPv4(127, 0, 0, 1) < IPv4(127, 0, 1, 0));
assert!(IPv4(66, 146, 219, 98) >= IPv4(63, 245, 213, 24));
```

The function Rust generates for ordering `IPv4` values is equivalent to the following:

```
fn partial_cmp(l: &IPv4, r: &IPv4) -> Option<Ordering> {
    let c = l.0.cmp(&r.0);
    if c != Ordering::Equal { return Some(c); }

    let c = l.1.cmp(&r.1);
    if c != Ordering::Equal { return Some(c); }

    let c = l.2.cmp(&r.2);
    if c != Ordering::Equal { return Some(c); }

    return Some(l.3.cmp(&r.3));
}
```

This compares the addresses' bytes from left to right using the `cmp` method, which returns `Ordering::Equal`, `Ordering::Less` or `Ordering::Greater` to indicate their relationship.

CHAPTER 9

Enums and patterns

Like the devil, our next topic is potent, as old as the hills, happy to help you get a lot done in short order (for a price), and known by many names in many cultures. Unlike the devil, it really is quite safe, and the price it asks is no great privation. It is a kind of data type, known in other languages as sum types, discriminated unions, or algebraic data types. In Rust, they are called enumerations, or simply enums.

In the simplest case, Rust enums are like those in C++ or C#. The values of such an enum are simply constants. But Rust takes enums much further. A Rust enum can also contain data, even data *of varying types,* like a C `union`, but type-safe.

It is in this capacity, as type-safe unions, that enums truly shine. They are just the right tool for modeling situations where a value might be either one thing or another. They can be used to build rich tree-like data structures with very little code, compared to Java or C++. If that's not enough, they are the perfect complement to Rust's fast and expressive pattern-matching, our topic for the second half of this chapter.

Patterns, too, may be familiar if you've used unpacking in Python or destructuring in JavaScript, but Rust takes patterns to extremes of usefulness. Rust patterns are a little like regular expressions for all your data. They're used to test whether or not a value has a particular desired shape. They can extract several fields from a struct or tuple into local variables all at once. And like regular expressions, they are concise, typically doing it all in a single line of code.

Enums

Simple, C-style enums are straightforward:

```
enum Ordering {
    Less,
    Equal,
```

```
    Greater
}
```

This declares a type `Ordering` with three possible values, called *variants* or *constructors*: `Ordering::Less`, `Ordering::Equal`, and `Ordering::Greater`. This particular enum is part of the standard library, so Rust code can import either the type by itself:

```
use std::cmp::Ordering;
talk(Ordering::Less);
smile(Ordering::Greater);
```

or all its constructors:

```
use std::cmp::Ordering::*;    // `*` to import all children
talk(Less);
smile(Greater);
```

To do the same for an enum declared in the current module, use a `self` import:

```
enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;
```

The constructors of a public enum are automatically public.

In memory, values of C-style enums are stored as integers. Occasionally it's useful to tell Rust which integers to use:

```
enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}
```

Otherwise Rust will assign the numbers for you, starting at 0.

By default, Rust stores C-style enums using the smallest built-in integer type that can accomodate them. Most fit in a single byte.

```
use std::mem::size_of;
assert_eq!(size_of::<Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2);  // 404 doesn't fit in a u8
```

You can override Rust's choice of representation by adding a `#[repr]` attribute to the enum. For details, see ???.

Casting a C-style enum to an integer is allowed:

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

However, casting in the other direction, from the integer to the enum, is not. Unlike C and C++, Rust guarantees that an enum value is only ever one of the values spelled out in the `enum` declaration. An unchecked cast from an integer type to an enum type could break this guarantee, so it's not allowed. Short of unsafe code, the only built-in way to convert an integer to an `enum` is to write the desired function yourself:

```
fn to_http_status(i: u32) -> Option<HttpStatus> {
    match i {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None
    }
}
```

As with structs, the compiler will implement features like the == operator for you, but you have to ask.

```
#[derive(Copy, Clone, Debug, PartialEq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years
}
```

In a similar vein, several crates on crates.io offer macros to autogenerate integer-to-enum methods like `to_http_status` for you.

So much for C-style enums. The more interesting sort of enum is the kind that contains data.

Tuple and struct variants

```
/// A timestamp that has been deliberately rounded off, so our program
/// says "6 months ago" instead of "February 9, 2016, at 9:49 AM".
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}
```

Two of the variants in this enum, `RoughTime::InThePast` and `RoughTime::InTheFuture`, take arguments. These are called *tuple variants*. Like tuple structs, these constructors are functions that create new `RoughTime` values.

```
let four_score_and_seven_years_ago =
    RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);

let three_hours_from_now =
    RoughTime::InTheFuture(TimeUnit::Hours, 3);
```

Enums can also have *struct variants*, which contain named fields, just like ordinary structs:

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
    Box { point1: Point3d, point2: Point3d }
}

let unit_sphere = Shape::Sphere { center: ORIGIN, radius: 1.0 };
```

In all, Rust has three kinds of enum variant, echoing the three kinds of struct we showed in the previous chapter. Variants with no data correspond to unit-like structs. Tuple variants look and function just like tuple structs. Struct variants have curly braces and named fields. A single enum can have variants of all three kinds.

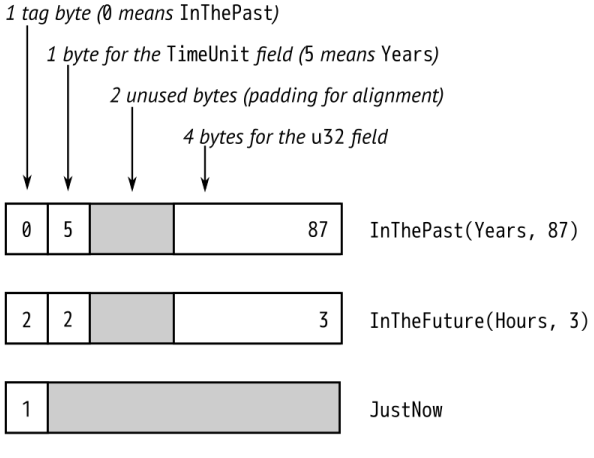
```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
        car: DifferentialEquation,
        cdr: EarlyModernistPoem
    }
}
```

Enums in memory

In memory, enums with data are stored as a small integer *tag*, plus enough memory to hold all the fields of the largest variant. The tag field is for Rust's internal use. It tells which constructor created the value, and therefore which fields it has.

As of Rust 1.8, `RoughTime` fits in 8 bytes, as shown in Figure 1.

Figure 1 - `RoughTime` values in memory



Rust makes no promises about enum layout, however, in order to leave the door open for future optimizations. In some cases, it would be possible to pack an enum more efficiently than Figure 1 suggests. We'll show later in this chapter how Rust can already optimize away the tag field for some enums.

Rich data structures using enums

Enums are also useful for quickly implementing tree-like data structures. For example, suppose a Rust program needs to work with arbitrary JSON data. In memory, any JSON document can be represented as a value of this Rust type:

```
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

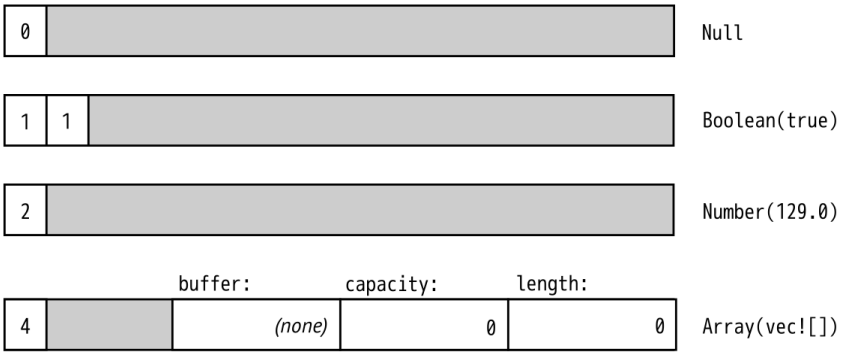
The explanation of this data structure in English can't improve much upon the Rust code. The JSON standard specifies the various data types that can appear in a JSON document: `null`, boolean values, numbers, strings, arrays of JSON values, and objects with string keys and JSON values. The `Json` enum simply spells out these types.

This is not a hypothetical example. A very similar enum can be found in serde_json, a serialization library for Rust structs that is one of the most-downloaded crates on crates.io.

The `Box` around the `HashMap` that represents an `Object` serves only to make all `Json` values more compact. In memory, values of type `Json` take up four machine words. `String` and `Vec` values are three words, and Rust adds a tag byte. `Null` and `Boolean` values don't have enough data in them to use up all that space, but all `Json` values must be the same size. The extra space goes unused. Figure 2 shows some examples of how `Json` values actually look in memory.

A `HashMap` is larger still. If we had to leave room for it in every `Json` value, they would be quite large, eight words or so. But a `Box<HashMap>` is a single word: it's just a pointer to heap-allocated data. We could make `Json` even more compact by boxing more fields.

Figure 2 - `Json` values in memory



What's remarkable here is how easy it was to set this up. In C++, one might write a class for this:

```
class JSON {
private:
    enum Tag {
        Null, Boolean, Number, String, Array, Object
    };
    union Data {
        bool boolean;
        double number;
        shared_ptr<string> str;
        shared_ptr<vector<JSON>> array;
        shared_ptr<unordered_map<string, JSON>> object;

        Data() {}
        ~Data() {}
        ...
    };

    Tag tag;
    Data data;

public:
    bool is_null() const { return tag == Null; }
    bool is_boolean() const { return tag == Boolean; }
    bool get_boolean() const {
        assert(is_boolean());
        return data.boolean;
    }
    void set_boolean(bool value) {
        this->~JSON();  // clean up string/array/object value
        tag = Boolean;
        data.boolean = value;
    }
    ...
};
```

At 30 lines of code, we have barely begun the work. This class will need constructors, a destructor, and an assignment operator. An alternative would be to create a class hierarchy with a base class `JSON` and subclasses `JSONBoolean`, `JSONString`, and so on. Either way, when it's done, our C++ JSON library will have more than a dozen methods. It will take a bit of reading for other programmers to pick it up and use it. The entire Rust enum is 8 lines of code.

Generic enums

Enums can be generic. Two examples from the standard library are among the most-used data types in the language:

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

enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

These types are familiar enough by now, and the syntax for generic enums is the same as for generic structs. One unobvious detail is that Rust can eliminate the tag field of `Option<T>` when the type `T` is a `Box` or some other smart pointer type. An `Option<Box<i32>>` is stored in memory as a single machine word, 0 for `None` and nonzero for `Some` boxed value.

Generic data structures can be built with just a few lines of code:

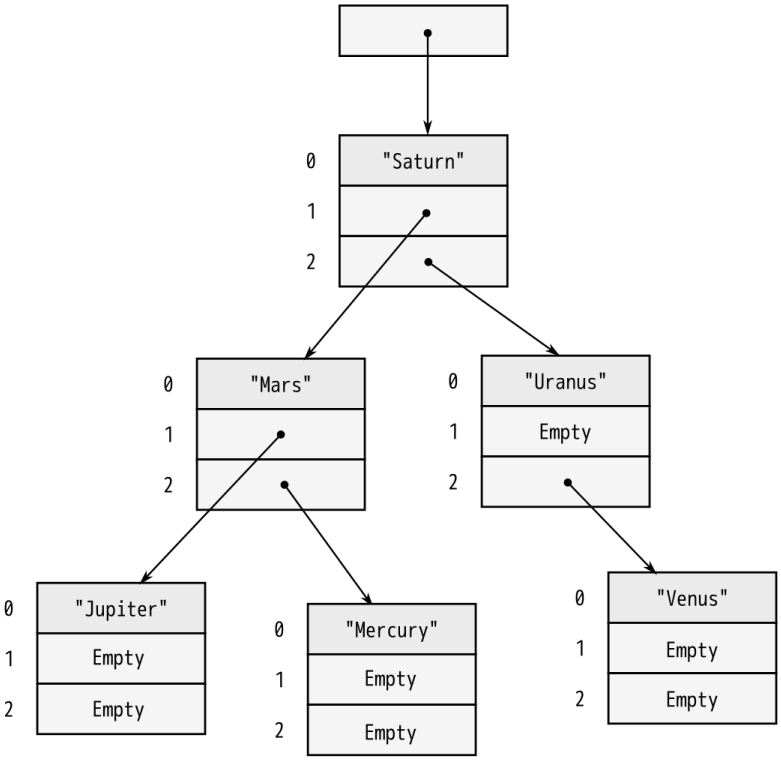
```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<(T, BinaryTree<T>, BinaryTree<T>)>)
}
```

These four lines of code define a `BinaryTree` type that can store any number of values of type `T`.

A great deal of information is packed into this definition, so we will take the time to translate the code word-for-word into English. Each `BinaryTree` value is either `Empty` or `NonEmpty`. If it's `Empty`, then it contains no data at all. If `NonEmpty`, then it is a `Box`, a pointer to heap-allocated data. Since the boxed data contains a value of type `T` and two more `BinaryTree` values, a `NonEmpty` tree can have any number of descendents.

A sketch of a value of type `BinaryTree<&str>` is shown in Figure 3. As with `Option<Box<T>>`, Rust eliminates the tag field, so a `BinaryTree` value is just one machine word.

Figure 3 - A BinaryTree containing six strings



Building any particular node in this tree is straightforward:

```
use self::BinaryTree::*;
let jupiter_node = Box::new(("Jupiter", Empty, Empty));
```

Larger trees can be built from smaller ones:

```
let mars_node = Box::new(("Mars",
                          NonEmpty(jupiter_node),
                          NonEmpty(mercury_node)));
```

Naturally, this assignment transfers ownership of `jupiter_node` and `mercury_node` to their new parent node.

The remaining parts of the tree follow the same patterns. The root is a `BinaryTree::NonEmpty` value.

```
let tree = NonEmpty(saturn_node);
```

Later in this chapter, we will show how to implement an `add` method on the `BinaryTree` type, so that we can instead simply write:

```
let mut tree = Empty;
for planet in planets {
```

```
        tree.add(planet);
    }
```

Now we come to the diabolical "price" mentioned in the introduction. The tag field of an enum costs a little memory, up to 8 bytes in the worst case, but that is usually negligible. The downside to enums (if it can be called that) is that Rust code cannot throw caution to the wind and try to access fields regardless of whether they are actually present in the value.

```
let r = shape.radius;  // error: no field with that name
```

The only way to access the data in an enum is the safe way: using patterns.

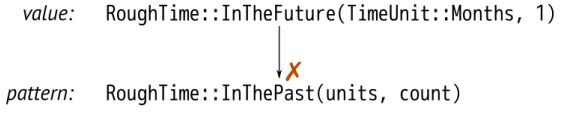
Patterns

We've shown examples of pattern matching throughout the book. Now it's time to show in detail how it works.

```
 1  fn rough_time_to_english(rt: RoughTime) -> String {
 2      match rt {
 3          RoughTime::InThePast(units, count) =>
 4              format!("{} {} ago", count, units.plural()),
 5          RoughTime::JustNow =>
 6              format!("just now"),
 7          RoughTime::InTheFuture(units, count) =>
 8              format!("{} {} from now", count, units.plural())
 9      }
10  }
```

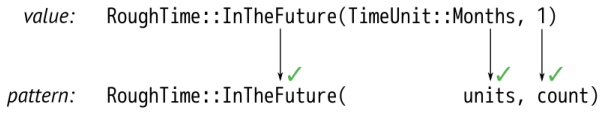
Lines 3, 5, and 7 consist of a pattern followed by `=>`. Patterns that match `RoughTime` values look just like the expressions used to create `RoughTime` values. They both look just like function calls. This is no coincidence. Expressions *produce* values; patterns *consume* values. The two use a lot of the same syntax.

Suppose `rt` is the value `RoughTime::InTheFuture(TimeUnit::Months, 1)`. Rust first tries to match this value against the pattern on line 3. It doesn't match:



Pattern matching on an enum, struct, or tuple works as though Rust is doing a simple left-to-right scan, checking each component of the pattern to see if the value matches it. If it doesn't, Rust moves on to the next pattern.

The patterns on lines 3 and 5 fail to match. But the pattern on line 7 succeeds:



When a pattern contains simple identifiers like `units` and `count`, those become local variables in the code following the pattern. Whatever is present in the value is copied or moved into the new variables. Rust stores `TimeUnit::Months` in `units` and `1` in `count`, runs line 8, and returns the string `"1 months from now"`.

The output has a minor grammatical issue which can be fixed by adding another arm to the `match`:

```
RoughTime::InTheFuture(unit, 1) =>
    format!("a {} from now", unit.singular()),
```

This arm matches only if the `count` field is exactly 1. Note that this new code must be added before line 7. If we add it at the end, Rust will never get to it, because the pattern on line 7 matches all `InTheFuture` values. The Rust compiler notices this kind of bug and flags it as an error.

Unfortunately, even with the new code, there is still a problem with `RoughTime::InTheFuture(TimeUnit::Hours, 1)`. Such is the English language. (This too can be fixed by adding another arm to the `match`. Try it out.)

A common beginner mistake with pattern matching is trying to use an existing variable in a pattern:

```
fn print_if_equal(x: i32, y: i32) {
    match x {
        y =>  // trying to match only if x == y
              // (it doesn't work: see explanation below)
            println!("{} == {}", x, y),
        _ =>  // error: unreachable pattern
            println!("not equal")
    }
}
```

This fails because identifiers introduce *new* bindings. The pattern `y` here creates a new local variable `y`, shadowing the argument `y`.

Tuple and struct patterns

Tuple patterns contain a subpattern for each element. A tuple can be used to get multiple pieces of data involved in a single `match`:

```
fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;
    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "at the origin",
```

```
            (_, Equal) => "on the x axis",
            (Equal, _) => "on the y axis",
            (Greater, Greater) => "in the first quadrant",
            (Less, Greater) => "in the second quadrant",
            _ => "somewhere else"
        }
    }
```

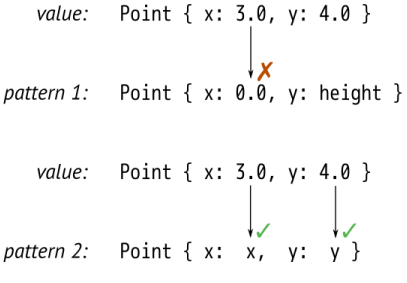
As the example shows, _ works as a subpattern, and it does exactly the same thing it does as a pattern: match any value. Because it doesn't store the value anywhere, it also provides a hint to other programmers that the program doesn't care about that value. _ is called the wildcard pattern, or sometimes the "don't-care" pattern.

Struct patterns use curly braces, just like struct expressions. They contain a subpattern for each field:

```
match balloon.location {
    Point { x: 0.0, y: height } =>
        println!("straight up {} meters", height),
    Point { x: x, y: y } =>
        println!("at ({}m, {}m)", x, y)
}
```

In this example, if the first arm matches, then `balloon.location.y` is bound to the new local variable `height`.

Suppose `balloon.location` is `Point { x: 3.0, y: 4.0 }`. As always, Rust checks each component of each pattern in turn:



Patterns like `Point { x: x, y: y }` are common when matching structs, and the redundant names are visual clutter, so Rust has a shorthand for this: `Point {x, y}`. The meaning is the same. This pattern still stores a point's `x` field in a new local `x` and its `y` field in a new local `y`.

Even with the shorthand, it is cumbersome to match a large struct when we only care about a few fields:

```
match get_account(id) {
    ...
    Some(Account {
            name, language,  // <--- the 2 things we care about
```

```
            id: _, status: _, address: _, birthday: _, eye_color: _,
            pet: _, security_question: _, hashed_innermost_secret: _,
            is_adamantium_preferred_customer: _ }) =>
        language.show_custom_greeting(name)
}
```

To avoid this, use `..` to tell Rust you don't care about any of the other fields.

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name)
```

Reference patterns

Rust patterns support two features for working with references. `ref` patterns borrow parts of a matched value. `&` patterns match references.

Matching on a non-copyable value moves the value. Continuing with the account example above, this code would be invalid:

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        account.show_hats(ui);  // error: use of moved value
    }
}
```

`account` cannot be used after pattern matching, because its `name` and `language` fields were moved into local variables, and the rest were dropped. The program needs to borrow `account.name` and `account.language` instead of moving them. The `ref` keyword does just that:

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        account.show_hats(ui);  // ok
    }
}
```

In this example, the local bindings `name` and `language` are references to the corresponding fields in `account`. Since `account` is only being borrowed, not consumed, it's OK to continue calling methods on it.

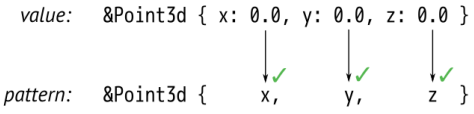
The opposite kind of reference pattern is the `&` pattern. A pattern starting with `&` matches a reference.

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

In this example, suppose `sphere.center()` returns a reference to a private field of `sphere`, a common pattern in Rust. The value returned is the address of a `Point3d`. If the center is at the origin, we could say:

```
    assert_eq!(sphere.center(), &Point3d { x: 0.0, y: 0.0, z: 0.0 });
```

So pattern matching proceeds like this:



This is a bit tricky because Rust is following a pointer here, an action we usually associate with the `*` operator, not the `&` operator. The thing to remember is that patterns and expressions are natural opposites. The expression `(x, y)` makes two values into a new tuple, but the pattern `(x, y)` does the opposite: it matches a tuple and breaks out the two values. It's the same with `&`. In an expression, `&` produces a reference. In a pattern, `&` consumes a reference.

Let's look at one more example of an `&` pattern. Suppose we have an iterator `chars` over the characters in a string, and it has a method `chars.peek()` that returns an `Option<&char>`: a reference to the next character, if any. (Peekable iterators do in fact return an `Option<&ItemType>`, as we'll see in ???.)

A program can use an `&` pattern to get the pointed-to character:

```
match chars.peek() {
    Some(&c) => println!("coming up: {:?}", c),
    None => println!("end of chars")
}
```

Matching multiple possibilities

The vertical bar, '`|`', can be used to combine several patterns in a single `match` arm.

```
let at_end =
    match chars.peek() {
        Some(&'\r') | Some(&'\n') | None => true,
        _ => false
    };
```

In an expression, `|` is the bitwise OR operator, but here it works more like the `|` symbol in a regular expression. `at_end` is set to `true` if `chars.peek()` matches any of the three patterns.

Use `...` to match a whole range of values. Range patterns include the begin and end values, so `'0' ... '9'` matches all the ASCII digits.

```
match next_char {
    '0' ... '9' =>
        self.read_number(),
    'a' ... 'z' | 'A' ... 'Z' =>
        self.read_word(),
    ' ' | '\t' | '\n' =>
```

```
            self.skip_whitespace(),
        _ =>
            self.handle_punctuation()
    }
```

Pattern guards

Use the `if` keyword to add a *guard* to a `match` arm. The match succeeds only if the guard evaluates to `true`:

```
match robot.last_known_location() {
    Some(point) if self.distance_to(point) < 10 =>
        short_distance_strategy(point),
    Some(point) =>
        long_distance_strategy(point),
    None =>
        searching_strategy()
}
```

@ patterns

Lastly, *`identifier @ pattern`* creates a new variable, like *`identifier`*, but only succeeds if the given *`pattern`* matches. It's used to grab a value and simultaneously check something about its internal structure:

```
match self.job_status() {
    // bind the result to 'success', but only if it's an Ok result
    success @ Ok(_) => return success,
    Err(err) => report_error(err)
}
```

It's also useful with range patterns:

```
match chars.next() {
    Some(digit @ '0' ... '9') => read_number(digit, chars),
    ...
}
```

Table 9-1 summarizes Rust's pattern language.

Table 9-1. Patterns

Pattern type	Example	Notes

Constant	`100`
	`"name"`
	`None`
		matches an exact value

Range	`0 ... 100` `'a' ... 'k'`	matches any value in range, including the end value
Wildcard	`_`	matches any value and ignores it

Pattern type	Example	Notes
Binding	`name` `mut count`	like _ but moves the value into a new local variable
Binding with subpattern	`val @ 0 ... 99` `ref circle @ Shape::Circle { .. }`	match the pattern to the right of @, use the variable name to the left
Borrow	`ref field`	match all or part of a value without moving it

Enum pattern	`Some(value)`
	`None`
	`Pet::Orca`	

Tuple pattern	`(key, value)` `(r, g, b)`	

Struct pattern	`Color(r, g, b)`
	`Point { x, y }`
	`Card { suit: Clubs, rank: n }`
	`Account { id, name, .. }`
		always matches unless a subpattern fails to match

Dereference	`&value` `&(k, v)`	matches only reference values
Multiple patterns	`'a' | 'A'`	in `match` only (not valid in `let`, etc.)
Guard expression	`x if x * x <= r2`	in `match` only (not valid in `let`, etc.)

Where patterns are allowed

Although patterns are most prominent in `match` expressions, they are also allowed in several other places, typically in place of an identifier. The meaning is always the same: instead of just storing a value in a single variable, Rust uses pattern matching to take the value apart.

This means patterns can be used to...

```
// ...unpack a struct into three new local variables
let Point3d { x, y, z } = center;

// ...iterate over keys and values of a HashMap
for (id, document) in &cache_map {
    println!("Document #{}: {}", id, document.title);
}

// ...run some code only if a table lookup succeeds
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// ...manually loop over an iterator
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
```

```
}

// ...automatically dereference an argument to a closure
// (handy because sometimes other code passes you a reference
// when you'd rather have a copy)
let sum = numbers.fold(0, |a, &num| a + num);
```

Each of these saves two or three lines of boilerplate code.

Populating a binary tree

Earlier we promised to show how to implement a method, `BinaryTree::add()`, that adds a node to a `BinaryTree` of this type:

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<(T, BinaryTree<T>, BinaryTree<T>)>)
}
```

We now know enough about patterns to write this method. An explanation of binary search trees is beyond the scope of this book, but for readers already familiar with the topic, it's worth seeing how it plays out in Rust.

```
 1  impl<T: Ord> BinaryTree<T> {
 2      fn add(&mut self, value: T) {
 3          match *self {
 4              Empty =>
 5                  *self = NonEmpty(Box::new((value, Empty, Empty))),
 6              NonEmpty(ref mut node) =>
 7                  if value <= node.0 {
 8                      node.1.add(value);
 9                  } else {
10                      node.2.add(value);
11                  }
12          }
13      }
14  }
```

Line 1 tells Rust that we're defining a method on `BinaryTree`s of ordered types. This is exactly the same syntax we use to define methods on structs, explained in "Defining methods with `impl`" on page 158.

The pattern on line 5 is the key:

```
NonEmpty(ref mut node) =>
```

This pattern matches if `*self` is a non-empty tree. A successful match borrows a mutable reference to the `Box`, so we can access and modify data inside the box. That reference is named `node`, and it's in scope from line 7 to line 11. The rest is simply a matter of accessing the three fields of that boxed triple.

The big picture

Enums and structs are complementary. Structs group together values that are so closely related that they occur together; enums are just the opposite, grouping values that are so closely related they *don't* occur together.

```
enum ThatGuyStandingOverThere {
    ClarkKent(Glasses, Notebook, Pencil),
    Superman(SpecialPowers, Cape, Tights)
}
```

Rust's enums may be new to systems programming, but they are not a new idea. Traveling under various academic-sounding names, like "algebraic data types", they have been used in functional programming languages for more than forty years. It's unclear why so few other languages in the C tradition have ever had them. Perhaps it is simply that for a programming language designer, combining variants, references, mutability, and memory safety is extremely challenging. Functional programming languages dispense with mutability. C `unions`, by contrast, have variants, pointers, and mutability—but are so spectacularly unsafe that even in C, they're a last resort. Rust's borrow checker is the magic that makes it possible to combine all four without compromise.

Programming is data processing. Getting data into the right shape can be the difference between a small, fast, elegant program and a slow, gigantic tangle of duct tape and virtual method calls.

This is the problem space enums address. They are a design tool for getting data into the right shape. For cases when a value may be one thing, or another thing, or perhaps nothing at all, enums are better than class hierarchies on every axis: faster, safer, less code, easier to document.

The limiting factor is flexibility. End users of an enum can't extend it to add new variants. Variants can only be added by changing the enum declaration. And when that happens, existing code breaks. Every `match` expression that individually matches each variant of the enum must be revisited—it needs a new arm to handle the new variant. In some cases, trading flexibility for simplicity is just good sense. After all, the structure of JSON is not expected to change. And in some cases, revisiting all uses of an enum when it changes is exactly what we want. For example, when an `enum` is used in a compiler to represent the various operators of a programming language, adding a new operator *should* involve touching all code that handles operators.

But sometimes more flexibility is needed. For those situations, Rust has traits, the topic of our next chapter.

CHAPTER 10
Traits and generics

One of the great discoveries in programming is that it's possible to write code that operates on values of many different types, *even types that haven't been invented yet.* Two examples:

- `Vec<T>` is generic: you can create a vector of any type of value, including types defined in your program that the authors of `Vec` never anticipated.
- Many things have `.write()` methods, including `File`s and `TcpStream`s. Your code can take a writer by reference, any writer, and send data to it. Your code doesn't have to care what type of writer it is. Later, if someone adds a new type of writer, your code will already support it.

Of course, this capability is hardly new with Rust. It's called *polymorphism*, and it was the hot new programming language technology of the 1970s. By now it's effectively universal. Rust supports polymorphism with two related features: traits and generics. C++ programmers will find both of these very familiar, but there are still a few surprises in store.

Traits are Rust's take on interfaces, or abstract base classes. The trait for writing bytes is called `std::io::Write`, and its definition in the standard library starts out like this:

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    ...
}
```

This trait provides two methods: `stream.write(buf)` writes the bytes in `buf` to `stream`, and `stream.flush()` writes any bytes `stream` may be holding back out to the system.

The standard types `File` and `TcpStream` both implement this trait. So does `Vec<u8>`. This means that all three types provide `write()` and `flush()` methods. Code that uses a writer without caring about its type looks like this:

```
fn say_hello(out: &mut Write) -> std::io::Result<()> {
    try!(out.write_all(b"hello world\n"));
    out.flush()
}
```

The `write()` method only writes as much data as the operating system is willing to accept in one call, so here we use the `write_all()` method instead, which calls `write()` repeatedly until data is written. The `Write` trait includes both methods.

The type of `out` is `&mut Write`, meaning "a mutable reference to any value that implements the `Write` trait".

```
let mut local_file = try!(File::create("hello.txt"));
try!(say_hello(&mut local_file));  // works

let mut bytes = vec![];
try!(say_hello(&mut bytes));  // also works
```

Interfaces are a delight in every language, and Rust is no exception. In this chapter, we'll show how to use them, how they work, and how to define your own. We'll see how traits can be used to add extension methods to existing types, even built-in types like `str` and `bool`. And built-in traits are the hook into the language that Rust provides for operator overloading and other features.

Generics are the other flavor of polymorphism in Rust. Like a C++ template, a generic function or type can be used with values of many different types.

```
/// Given two values, pick whichever one is less.
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

The `<T: Ord>` in this function means that `min` can be used with arguments of any type `T` that implements the `Ord` trait—that is, any ordered type. The compiler generates custom machine code for each type `T` that you actually use.

Generics and traits are closely related. Rust makes us declare the `T: Ord` requirement (called a *bound*) up front, before using the `<=` operator to compare two values of type `T`. So we'll also talk about how `&mut Write` and `<T: Write>` are similar, how they're different, and how to choose between these two ways of using traits.

Using traits

A trait is a feature that some types support and some don't. Most often, a trait represents a capability: something a type can do.

- A value that implements `std::io::Write` can write out bytes.
- A value that implements `std::iter::Iterator` can iterate over a sequence of values.
- A value that implements `std::clone::Clone` can produce clones of itself in memory.
- A value that implements `std::fmt::Debug` can be printed using `println!()` and the `{:?}` format specifier.

These traits are all part of Rust's standard library, and many standard types implement them.

- `std::fs::File` implements the `Write` trait; it writes bytes to a local file. `std::net::TcpStream` writes to a network connection. `Vec<u8>` also implements `Write`. Each `.write()` call on a vector of bytes appends some data to the end.
- `Range<i32>` (the type of `0..10`) implements the `Iterator` trait, as do some iterator types associated with slices, hash tables, and so on.
- Most standard library types implement `Clone`. The exceptions are mainly types like `TcpStream` that represent more than just data in memory.
- Likewise, most standard library types support `Debug`.

There is one unusual rule about trait methods: the trait itself must be in scope. Otherwise all its methods are hidden.

```
let mut buf: Vec<u8> = vec![];
try!(buf.write_all(b"hello"));  // error: no method named `write_all`
```

In this case, the compiler prints a friendly error message that suggests adding `use std::io::Write;` and indeed that fixes the problem.

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

let mut buf: Vec<u8> = vec![];
try!(buf.write_all(b"hello"));  // ok
```

Rust has this rule because traits can add methods to any type. You can even use traits to add new methods to standard library types, and thanks to this rule, you don't need to worry about naming conflicts. Each module imports the traits it needs to use. In short, trait methods are opt-in.

The reason `Clone` and `Iterator` methods work without any special imports is that they're always in scope by default: they're part of the standard prelude, names that Rust automatically imports into every module. In fact, the prelude is mostly a carefully chosen selection of traits. We'll cover many of them in the next chapter.

C++ programmers will already have noticed that trait methods are like C++ virtual member functions. Still, calls like the one shown above are fast, as fast as any other method call. Simply put, there's no polymorphism here. It's obvious that `buf` is a vector, not a file or a network connection. The compiler can emit a simple call to `Vec<u8>::write()`. It can even inline the method. (C++ will often do the same, although the possibility of subclassing sometimes precludes this.) Only calls through `&mut Write` incur the overhead of a virtual method call.

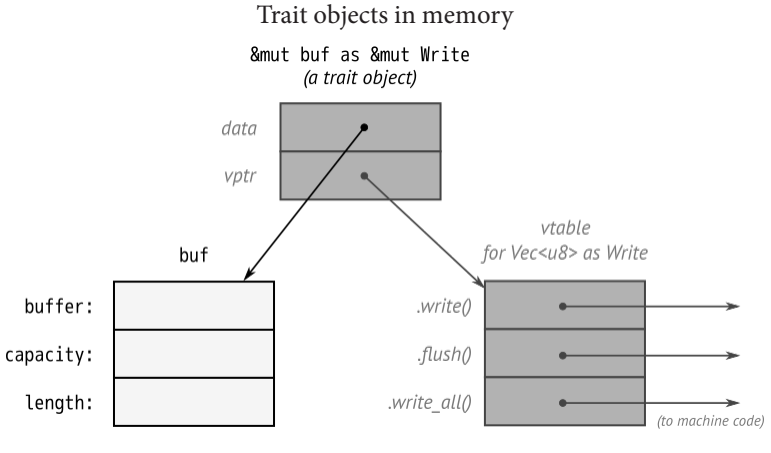
Trait objects

There are two ways of using traits to write polymorphic code in Rust: trait objects and generics. We'll present trait objects first and turn to generics in the next section.

When we use the name of a trait in a reference type, like `&mut Write`, we're talking about a special kind of reference called a *trait object*. The word "object" means different things in different languages, so it's worth taking a second to see how they work in Rust.

To start with, a trait object is a reference. Like any other reference, it points to some value, it has a lifetime, and it can be either `mut` or shared.

However, a trait object is a reference to a value of unknown type. All Rust knows about the value is that it implements a particular trait, like `Write`. In memory, the trait object is a fat pointer consisting of a pointer to the value, plus a pointer to a table representing that value's type. Each trait object therefore takes up two machine words, as shown in Figure 10-0.



C++ has this kind of run-time type information, too. It's called a *virtual table*, or *vtable*. In Rust, as in C++, the vtable is generated once, at compile time, and shared by all objects of the same type. Everything shown in dark gray in Figure 10-0, including the vtable, is a private implementation detail of Rust. These aren't fields and data structures that you can access directly. Instead, the language automatically uses the vtable when you call a method of a trait object. That's how Rust knows which method to call.

Rust automatically converts ordinary references into trait objects when needed. This is why we're able to pass `&mut local_file` to `say_hello` in this example:

```
let mut local_file = try!(File::create("hello.txt"));
try!(say_hello(&mut local_file));
```

The type of `&mut local_file` is `&mut File`, and the type of the argument to `say_hello` is `&mut Write`. Since a `File` is a kind of writer, Rust allows this, automatically converting the plain reference to a trait object.

Likewise, Rust will happily convert a `Box<File>` to a `Box<Write>`, a value that owns a writer in the heap.

```
let w: Box<Write> = Box::new(local_file);
```

Generic functions

At the beginning of this chapter, we showed a `say_hello()` function that took a trait object as an argument. Let's rewrite that function as a generic function:

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    try!(out.write_all(b"hello world\n"));
    out.flush()
}
```

Only the type signature has changed:

```
fn say_hello(out: &mut Write)         // plain function

fn say_hello<W: Write>(out: &mut W)   // generic function
```

The phrase `<W: Write>` is what makes the function generic. This is a *type parameter*. It means that throughout the body of this function, `W` stands for a type that implements the `Write` trait. Which type? It depends on how the generic function is used.

```
try!(say_hello(&mut local_file));  // calls say_hello::<File>
try!(say_hello(&mut bytes));       // calls say_hello::<Vec<u8>>
```

When you pass `&mut local_file` to the generic `say_hello()` function, you're calling `say_hello::<File>()`. Rust generates machine code for this function that calls `File::write_all()` and `File::flush()`. When you pass `&mut bytes`, you're calling `say_hello::<Vec<u8>>()`. Rust generates separate machine code for this version of

the function, calling the corresponding `Vec<u8>` methods. In both cases, Rust infers the type `W` from the type of the argument. You can always spell out the type parameters:

```
try!(say_hello::<File>(&mut local_file));
```

but it's seldom necessary.

Type parameters are conventionally uppercase letters, but any `CamelCase` name will do. Single letters work well because anyone reading your code can see at a glance that `T` or `W` or `X` is not the name of a type they can look up in the documentation; it's a type parameter, so they need to look at the first line of the generic item it appears in. (Your team, we hope, trusts you not to name a struct `X`.)

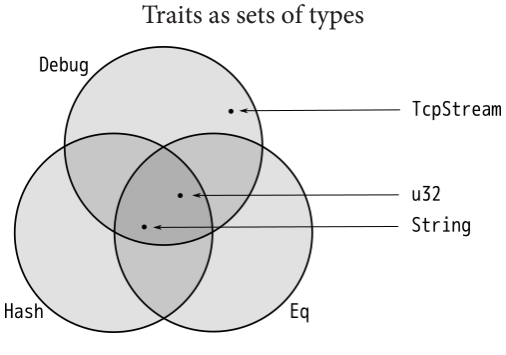
Sometimes we need multiple abilities from a type parameter. For example, if we want to print out the top ten most common values in a vector, we'll need for those values to be printable:

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

But this isn't good enough. How are we planning to determine which values are the most common? The usual way is use the values as keys in a hash table. That means the values need to support the `Hash` and `Eq` operations. The bounds on `T` must include these as well as `Debug`. The syntax for this uses the `+` sign:

```
fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

Some types implement `Debug`, some implement `Hash`, some support `Eq`; and a few, like `u32` and `String`, implement all three.



It's also possible for a type parameter to have no bounds at all, though you can't do much with a value if you haven't specified any bounds for it. You can move it. You can put it in a box or vector. That's about it.

```
fn one_item_vector<A>(item: A) -> Vec<A> {
    let mut vector = Vec::new();
    vector.push(item);
    vector
}
```

Generic functions can have multiple type parameters:

```
/// Run a query on a large, partitioned data set.
/// See <http://research.google.com/archive/mapreduce.html>.
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

As this example shows, the bounds can get to be so long that they are hard on the eyes. Rust provides an alternative syntax using the keyword `where`:

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

The type parameters `M` and `R` are still declared up front, but the bounds are moved to separate lines. This kind of `where` clause is also allowed on generic structs, enums, type aliases, and methods—anywhere bounds are permitted.

Of course, an alternative to `where` clauses is to keep it simple: find a way to write the program without using generics quite so intensively.

"Receiving references as parameters" on page 104 showed the syntax for lifetime parameters. A generic function can have both lifetime parameters and type parameters. Lifetime parameters come first.

```
fn nearest<'c, P>(target: &P, candidates: &'c [P]) -> &'c P
    where P: Distance
{ ... }
```

Two calls to `nearest()` using the same type `P` but different lifetimes will call the same compiled function. Lifetimes never have any impact on machine code. Only differing types cause Rust to generate multiple copies of a generic function.

Of course, functions are not the only kind of generic code in Rust.

- We've already covered generic types in ??? and "Generic enums" on page 173.
- An individual method can be generic, even if the type it's defined on is not generic:

```
impl PancakeStack {
    fn push<T: Topping>(&mut self, goop: T) -> PancakeResult<()> {
        ...
    }
}
```

- Type aliases can be generic, too:
  ```
  type PancakeResult<T> = Result<T, PancakeError>;
  ```
- We'll cover generic traits later in this chapter.

All the features introduced in this section—bounds, where clauses, lifetime parameters, and so forth—can be used on all generic items, not just functions.

Which to use

The choice of whether to use trait objects or generic code is subtle. Since both features are based on traits, they have a lot in common.

Trait objects are the natural choice whenever you need a collection of values of mixed types, all together. It is technically possible to make generic salad:

```
struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

but this is a rather severe design. Each such salad consists entirely of a single type of vegetable. Not everyone is cut out for this sort of thing. One of your authors once paid fourteen dollars for a `Salad<IcebergLettuce>` and has never quite gotten over the experience. Trait objects would have been the solution:

```
struct Salad {
    veggies: Vec<Box<Vegetable>>
}
```

Apart from the unfortunate mixed metaphor of having boxes in one's food, this is precisely what's called for, and it would work out just as well for shapes in a drawing app, monsters in a game, pluggable routing algorithms in a network router, and so on.

Another possible reason to use trait objects is to reduce the total amount of compiled code. Rust may have to compile a generic function many times, once for each type it's used with. This could make the binary large, a phenomenon called "code bloat" in C++ circles. These days, memory is plentiful, and most of us have the luxury of ignoring code size; but constrained environments do exist.

Outside of situations involving salad or microcontrollers, generics have two important advantages over trait objects, with the result that in Rust, generics are the more common choice.

The first advantage is speed. Each time the Rust compiler generates machine code for a generic function, it knows which types it's working with, so it knows at that time which `write` method to call. There's no need for dynamic dispatch.

The generic `min()` function shown in the introduction is just as fast as if we had written separate functions `min_u8`, `min_i64`, `min_String`, and so on. The compiler can

inline it, like any other function, so in a release build, `min_i32` is likely just two or three instructions. A call with constant arguments, like `min(5, 3)`, will be even faster: Rust can evaluate it at compile time, so that there's no run-time cost at all.

Or consider this generic function call:

```
let mut sink = std::io::sink();
try!(say_hello(&mut sink));
```

`std::io::sink()` returns a writer of type `Sink` which quietly discards all bytes written to it.

When Rust generates machine code for this, it could emit code that calls `Sink::write_all`, checks for errors, then calls `Sink::flush`. That's what the body of the generic function says to do.

Or, Rust could look at those methods and realize that

- `Sink::write_all()` does nothing;
- `Sink::flush()` does nothing; and
- neither method ever returns an error.

In short, Rust has all the information it needs to optimize away this function entirely.

Compare that to the behavior with trait objects. Rust never knows what type of value a trait object points to until run time. So even if you pass a `Sink`, the overhead of calling virtual methods and checking for errors still applies.

The second advantage of generics is that not every trait can support trait objects. Traits support several features that are useful in combination with generics, but which rule out trait objects altogether: static methods, generic methods, associated types, and the `Self` type. These features are covered in detail later in this chapter. Let's begin with the basics.

Defining and implementing traits

Defining a trait is simple. Give it a name and list the type signatures of the trait methods. If we're writing a game, we might have a trait like this:

```
/// A trait for characters, items, and scenery -
/// anything in the game world that's visible on screen.
trait Visible {
    /// Draw this object on the given canvas.
    fn render(&self, canvas: &mut Canvas);

    /// Return true if clicking at (x, y) should
    /// select this object.
```

```
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

To implement a trait, use the syntax `impl` *`TraitName`* `for` *`Type`*.

```
impl Visible for Broom {
    fn render(&self, canvas: &mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
            canvas.write_at(self.x, y, '|');
        }
        canvas.write_at(self.x, self.y, 'M');
    }

    fn hit_test(&self, x: i32, y: i32) -> bool {
        self.x == x
        && self.y - self.height - 1 <= y
        && y <= self.y
    }
}
```

Note that this `impl` contains an implementation for each method of the `Visible` trait, and nothing else. Everything defined in a trait `impl` must actually be a feature of the trait; if we wanted to add a helper method in support of `Broom::render()`, we would have to define it in a separate `impl` block:

```
impl Broom {
    /// Helper function used by Broom::render() below.
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}

impl Visible for Broom {
    fn render(&self, canvas: &mut Canvas) {
        for y in self.broomstick_range() {
            ...
        }
        ...
    }
    ...
}
```

Default methods

The `Sink` writer type we discussed earlier can be implemented in a few lines of code. First, we define the type:

```
/// A Writer that ignores whatever data you write to it.
pub struct Sink;
```

`Sink` is an empty struct, since we don't need to store any data in it. Next, we provide an implementation of the `Write` trait for `Sink`:

```
use std::io::{Write, Result};

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

So far, this is very much like the `Visible` trait. But we have also seen that the `Write` trait has a `write_all` method:

```
try!(out.write_all(b"hello world\n"));
```

Why does Rust let us `impl Write for Sink` without defining this method? The answer is that the standard library's definition of the `Write` trait contains a *default implementation* for `write_all`:

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> {
        let mut bytes_written = 0;
        while bytes_written < buf.len() {
            bytes_written += try!(self.write(&buf[bytes_written..]));
        }
        Ok(())
    }

    ...
}
```

The `write` and `flush` methods are the basic methods which every writer must implement. A writer may also implement `write_all`, but if not, the default implementation shown above will be used.

Your own traits can include default implementations using the same syntax.

The most dramatic use of default methods in the standard library is the `Iterator` trait, which has one required method (`.next()`) and dozens of default methods. ??? explains why.

Traits and other people's types

Rust lets you implement any trait on any type, as long as either the trait or the type is introduced in the current crate.

This means that any time you want to add a method to any type, you can use a trait to do it.

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// Implement IsEmoji for the built-in character type.
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

Like any other trait method, this new `is_emoji` method is only visible when `IsEmoji` is in scope.

The `serde` library offers a nice example of how useful it can be to implement user-defined traits on standard types. `serde` is a serialization library. That is, you can use it to write Rust data structures to disk and reload them later. The library defines a trait, `Serialize`, that's implemented for every data type the library supports. So in the `serde` source code, there is code implementing `Serialize` for `bool`, `i8`, `i16`, `i32`, array and tuple types, and so on, through all the standard data structures like `Vec` and `HashMap`.

The `Serialize` trait makes it possible to write a single generic function that serializes any supported type:

```
pub fn to_string<T: Serialize>(value: &T) -> Result<String> {
    ...
}
```

A function with this signature is defined in the `serde_json` library.

```
assert_eq!(serde_json::to_string(&(11.25, true)).unwrap(),
           "[11.25,true]".to_string());
assert_eq!(serde_json::to_string(&vec!["hello"]).unwrap(),
           "[\"hello\"]".to_string());
```

(Another is in the `serde_yaml` library. `serde` supports pluggable serialization formats.)

We said earlier that when you implement a trait, either the trait or the type must be new in the current crate. This rule helps Rust ensure that trait implementations are unique. Your code can't `impl Write for u8`, because both `Write` and `u8` are defined in the standard library. If Rust let crates do that, there could be multiple implementations in different crates, and Rust would have no reasonable way to decide which implementation to use for a given method call.

(C++ has a similar uniqueness restriction: the One Definition Rule. In typical C++ fashion, it isn't enforced by the compiler, except in the simplest cases, and you get undefined behavior if you break it.)

`Self` in traits

A trait can use the keyword `Self` as a type. The standard `Clone` trait, for example, looks like this (slightly simplified):

```
pub trait Clone {
    fn clone(&self) -> Self;
    ...
}
```

Using `Self` as the return type here means that the type of `x.clone()` is the same as the type of `x`, whatever that might be. If `x` is a `String`, then the type of `x.clone()` is `String`—not `Clone` or any other cloneable type.

Likewise, if we define this trait:

```
pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}
```

with two implementations:

```
impl Spliceable for CherryTree {
    ...
}

impl Spliceable for Mammoth {
    ...
}
```

then this means that we can splice together two cherry trees or two mammoths, not that we can create a mammoth-cherry hybrid. The type of `self` and the type of `other` must match.

A trait that uses the `Self` type is incompatible with trait objects:

```
// error: the trait `Spliceable` cannot be made into an object
fn splice_anything(left: &Spliceable, right: &Spliceable) {
    let combo = left.splice(right);
    ...
}
```

The reason is something we'll see again and again as we dig into the advanced features of traits. Rust rejects this code because it has no way to type-check the call `left.splice(right)`. The whole point of trait objects is that the type isn't known until run time. Rust has no way to know if `left` and `right` will be the same type, as required.

Trait objects are really intended for the simplest kinds of traits, the kinds that could be implemented using interfaces in Java or pure virtual methods in C++. The more advanced features of traits are useful, but they can't coexist with trait objects because with trait objects, you lose the type information Rust needs to type-check your program.

Now, had we wanted genetically improbable splicing, we could have designed a trait-object-friendly trait:

```
pub trait MegaSpliceable {
    fn splice(&self, other: &MegaSpliceable) -> Box<MegaSpliceable>;
}
```

This trait is compatible with trait objects. There's no problem type-checking calls to this `.splice()` method because the type of the argument `other` is not required to match the type of `self`, as long as both types are `MegaSpliceable`.

Subtraits

We can declare that a trait is an extension of another trait:

```
/// Someone in the game world, either the player or some other
/// pixie, gargoyle, squirrel, ogre, etc.
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

The phrase `trait Creature: Visible` means that all creatures are visible. Every type that implements `Creature` must also implement the `Visible` trait:

```
impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}
```

We can implement the two traits in either order, but it's an error to implement `Creature` for a type without also implementing `Visible`.

Static methods

In most object-oriented languages, interfaces can't include static methods or constructors. Rust traits can:

```
trait StringSet {
    /// Return a new empty set.
```

```
    fn new() -> Self;

    /// Return a set that contains all the strings in `strings`.
    fn from_slice(strings: &[&str]) -> Self;

    /// Find out if this set contains a particular `value`.
    fn contains(&self, string: &str) -> bool;

    /// Add a string to this set.
    fn add(&mut self, string: &str);
}
```

Every type that implements the `StringSet` trait must implement the two functions `new()` and `from_slice()`, which serve as constructors. These are *static methods*, a kind of method we first saw in "Defining methods with `impl`" on page 158. Here, they work exactly like the other methods of `trait StringSet` except that they don't take a `self` argument.

In non-generic code, these methods can be called using `::` syntax, just like any static method:

```
// Create sets of two hypothetical types that impl StringSet:
let set1 = SortedStringSet::new();
let set2 = HashedStringSet::new();
```

In generic code, it's the same, except the type is often a type variable, as in the call to `S::new()` below:

```
/// Return the set of words in `document` that aren't in `wordlist`.
fn unknown_words<S: StringSet>(document: &Vec<String>, wordlist: &S) -> S {
    let mut unknowns = S::new();
    for word in document {
        if !wordlist.contains(word) {
            unknowns.add(word);
        }
    }
    unknowns
}
```

Traits with static methods, like `StringSet`, are incompatible with trait objects.

Traits and related types

Traits are often used to support generic code in situations where they're not just about a single type but about multiple types that have to work together.

- The `std::iter::Iterator` trait relates each iterator type with the type of value it produces.

- The `std::ops::Mul` trait relates types that can be multiplied. Rust permits the value on the left-hand side of multiplication to be the same type as the one on the right, or a different type.
- The `rand` crate includes both a trait for random number generators (`rand::Rng`) and a trait for types that can be randomly generated (`rand::Rand`).

You won't need to create this kind of trait every day, but you'll come across examples like these throughout the standard library and in third-party crates. In this section, we'll show how each of the examples above is implemented, picking up relevant Rust language features as we need them. The key skill here is the ability to read traits and method signatures and figure out what they say about the types involved.

Associated types (or, how iterators work)

We'll start with iterators. By now every object-oriented language has some sort of built-in support for iterators, objects that represent the traversal of some sequence of values.

Rust has a standard `Iterator` trait, defined like this:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

The first feature of this trait, `type Item;` is an *associated type*. Each type that implements `Iterator` must specify what type of item it produces.

The second feature, the `next()` method, uses the associated type in its return value. `next()` returns an `Option<Self::Item>`: either `Some(item)`, the next value in the sequence, or `None` when there are no more values to visit.

The type is written as `Self::Item`, not just plain `Item`, because `Item` is a feature of each type of iterator, not a standalone type. As always, `self` and `Self` show up explicitly in the code everywhere their fields, methods, and so on are used.

Generic code can use associated types:

```
/// Loop over an iterator, storing the values in a new vector.
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

Inside the body of this function, Rust infers the type of `value` for us, which is nice; but we must spell out the return type of `collect_into_vector`, and the `Item` associated type is the only way to do that. (`Vec<I>` would be simply wrong: we would be claiming to return a vector of iterators!)

The above example is not code that you would write out yourself, because after reading `???`, you'll know that iterators already have a standard method that does this: `iter.collect()`. So let's look at one more example before moving on.

```
/// Print out all the values produced by an iterator
fn dump<I: Iterator>(iter: I) {
    let mut index: usize = 0;
    for value in iter {
        println!("{}: {}", index, value);   // error
        index += 1;
    }
}
```

This almost works. There is just one problem: `value` might not be a printable type.

```
dump.rs:6:35: 6:40 error: the trait bound `<I as std::iter::Iterator>::Item:
std::fmt::Display` is not satisfied
dump.rs:6         println!("{}: {}", index, value);
                                             ^~~~
```

The error message is slightly obfuscated by Rust's use of the syntax `<I as std::iter::Iterator>::Item`, which is a long, maximally explicit way of saying `I::Item`. This is valid Rust syntax, but you'll rarely actually need to write a type out that way.

The gist of the error message is that to make this generic function compile, we must ensure that `I::Item` implements the `Display` trait, the trait for printable types. We can do this by placing a bound on `I::Item`.

```
fn dump<I: Iterator>(iter: I)
    where I::Item: Display
{
    ...
}
```

Or, we could write, "`I` must be an iterator over `String` values".

```
fn dump<I: Iterator<Item=String>>(iter: I) {
    ...
}
```

`Iterator<Item=String>` is itself a trait. If you think of `Iterator` as the set of all iterator types, then `Iterator<Item=String>` is a subset of `Iterator`: the set of iterator types that produce `String`s. This syntax can be used anywhere the name of a trait can be used, including trait object types:

```
fn dump(iter: &mut Iterator<Item=String>) {
    for s in iter {
        println!("{}", s);
    }
}
```

Traits with associated types, like `Iterator`, are compatible with trait methods, but only if all the associated types are spelled out, as shown above. Otherwise, the type of `s` could be anything, and again, Rust would have no way to type-check this code.

We've shown a lot of examples involving iterators. It's hard not to; they're by far the most prominent use of associated types. But associated types are generally useful whenever a trait needs to cover more than just methods.

- In a thread pool library, a `Task` trait, representing a unit of work, could have an associated `Output` type.
- A `Pattern` trait, representing a way of searching a string, could have an associated `Match` type, representing all the information gathered by matching the pattern to the string.

  ```
  trait Pattern {
      type Match;

      fn search(&self, string: &str) -> Option<Self::Match>;
  }

  /// You can search a string for a particular character.
  impl Pattern for char {
      /// A "match" is just the location where the
      /// character was found.
      type Match = usize;

      fn search(&self, string: &str) -> Option<usize> {
          ...
      }
  }
  ```

  If you're familiar with regular expressions, it's easy to see how `impl Pattern for RegExp` would have a more elaborate `Match` type, probably a struct that would include the start and length of the match, the locations where parenthesized groups matched, and so on.
- A library for working with relational databases might have a `DatabaseConnection` trait with associated types representing transactions, cursors, prepared statements, and so on.

Associated types are perfect for cases where each implementation has *one* specific related type: each type of `Task` produces a particular type of `Output`; each type of `Pat`

tern looks for a particular type of Match. However, as we'll see, some relationships among types are not like this.

Generic traits (or, how operator overloading works)

Multiplication in Rust uses this trait:

```
/// std::ops::Mul, the trait for types that support `*`.
pub trait Mul<RHS=Self> {
    /// The resulting type after applying the `*` operator
    type Output;

    /// The method for the `*` operator
    fn mul(self, rhs: RHS) -> Self::Output;
}
```

Mul is a generic trait. The type parameter, RHS, is short for "right-hand side". In Rust, the expression lhs * rhs is shorthand for lhs.mul(rhs). (In fact, the only difference between the two is that the * syntax always uses the Mul trait, whether you imported it or not.) There are similar traits for all the other arithmetic operators.

Maybe you have already guessed how to overload the * operator in Rust: just implement the Mul trait. Here's an example:

```
use std::ops::Mul;

/// The size of a rectangle (width, height).
#[derive(Copy, Clone, Debug, PartialEq)]
struct Size(f64, f64);

/// Multiplying (Size * f64) scales width and height by the same factor.
impl Mul<f64> for Size {
    /// The result is a new Size.
    type Output = Size;

    fn mul(self, scale: f64) -> Size {
        Size(self.0 * scale, self.1 * scale)
    }
}

assert_eq!(Size(3.0, 4.0) * 20.0, Size(60.0, 80.0));
```

We bring this up because of a curious thing about the design of Mul. Take a look back at the definition of trait Mul<RHS> above. The type RHS is a type parameter, but Output is an associated type. Why are there two different things? What do they mean?

The type parameter here means the same thing that it means on a struct or function: Mul is a generic trait, and its instances Mul<f64>, Mul<String>, Mul<Size>, etc. are all different traits, just as min<i32>() and min<String>() are different functions.

The definition says `trait Mul<RHS=Self>`, meaning that `RHS` defaults to `Self`. If I say that `i32` implements `Mul`, I mean it implements `Mul<i32>`: you can multiply an integer by another integer.

So how is this different from an associated type? It's pretty simple: a single type can implement not only `Mul<f64>` but `Mul<i32>` and `Mul<u8>` as well. The relationship between left-hand and right-hand types is not one-to-one; it's potentially many-to-many. But for each implementation of `Mul`, there is always a single `mul()` method, and a single type `Output` which is its return type. The type `<i32 as Mul<i32>>::Output` is always and forever `i32`; There is no way to get integer multiplication to produce any other type.

When you see an associated type in a trait, you know that exactly one type is associated, per implementation. When you see a generic trait, you know that a single type could implement several instances of it.

Buddy traits (or, how `rand::random()` works)

There's one more way to use traits to express relationships between types. This way is perhaps the simplest of the bunch, since you don't have to learn any new language features to understand it: what we'll call "buddy traits" are simply traits that are designed to work together.

There's a good example inside the `rand` crate, a popular crate for generating random numbers. The main feature of `rand` is the `random()` function, which returns a random value:

```
use rand::random;
let x = random();
```

If Rust can't infer the type of the random value, which is often the case, you must specify it:

```
let x = random::<f64>();   // a number, 0.0 <= x < 1.0
let b = random::<bool>();  // true or false
```

For many programs, this one generic function is all you need. But the `rand` crate also offers several different, but interoperable, random number generators. All the random number generators in the library implement a common trait:

```
/// A random number generator.
pub trait Rng {
    fn next_u32(&mut self) -> u32;
    ...
}
```

An `Rng` is simply a value that can spit out integers on demand.

The related trait is called `Rand`:

```
/// A type that can be randomly generated using an `Rng`.
pub trait Rand: Sized {
    fn rand<R: Rng>(rng: &mut R) -> Self;
}
```

Pass any random number generator to this method, and it returns a random value.

```
let x = f64::rand(rng);
let b = bool::rand(rng);
```

In fact `random()` works by passing a globally allocated `Rng` to this `rand` method. If you implement `Rand` for a new type you've introduced, then `random()` automatically supports it. The function's type reflects this.

```
pub fn random<T: Rand>() -> T
```

When you see traits that use other traits as bounds, the way `Rand::rand()` uses `Rng`, you know that those two traits are mix-and-match: any `Rng` can generate values of every `Rand` type. Since the methods involved are generic, Rust generates optimized machine code for each combination of `Rng` and `Rand` that your program actually uses.

The two traits also serve to separate concerns. Whether you're implementing `Rand` for your `Monster` type or implementing a spectacularly fast but not-so-random `Rng`, you don't have to do anything special for those two pieces of code to be able to work together.

The standard library's support for computing hash codes provides a similar pair of traits. Types that implement `Hash` are hashable, so they can be used as hash table keys. Types that implement `Hasher` are hashing algorithms. The two are linked in the same way as `Rand` and `Rng`: `Hash` has a generic method `.hash()` that accepts any type of `Hasher` as an argument.

In the last three sections, we've shown several ways traits can describe relationships between types. All of these can also be seen as ways of avoiding virtual method overhead and downcasts, since they allow Rust to know more concrete types at compile time. Surprisingly many traits benefit from these features.

Reverse-engineering bounds

Writing generic code can be a real slog when there's no single trait that does everything you need. Suppose we have written this non-generic function to add up some numbers:

```
fn sum(values: &Vec<i32>) -> i32 {
    let mut total = 0;
    for v in values {
        total += *v;
    }
```

```
    total
}
```

Now we want to use the same code to add floating-point values. We might try something like this:

```
fn sum<N>(values: &Vec<N>) -> N {
    let mut total: N = 0;
    for v in values {
        total += *v;
    }
    total
}
```

No such luck: Rust complains about the use of `+=` and the type of `0`. The traits representing these two language features (`AddAssign` and `Zero`) are unstable, so we must find stable alternatives. We can rewrite `+=` using the simple `+` operator, and there is a standard `Default` trait for types that have "default values", which for numeric types is always 0:

```
use std::ops::Add;

fn sum<N: Add + Default>(values: &Vec<N>) -> N {
    let mut total = N::default();
    for v in values {
        total = total + *v;
    }
    total
}
```

This is closer, but still does not quite work:

```
sum.rs:6:21: 6:31 error: mismatched types:
 expected `N`,
    found `<N as core::ops::Add>::Output`
sum.rs:6             total = total + *v;
                             ^~~~~~~~~
```

Our new code assumes that that adding two values of type `N` produces another value of type `N`. This isn't necessarily the case. You can overload the addition operator to return whatever type you want. We need to somehow tell Rust that this generic function only works with types that have the normal flavor of addition, where adding `N + N` returns an `N`. We do this by replacing `Add` with `Add<Output=N>`.

```
fn sum<N: Add<Output=N> + Default>(values: &Vec<N>) -> N {
    ...
}
```

At this point, the bounds are starting to pile up, making the code hard to read. Let's move the bounds into a `where` clause:

```
fn sum<N>(values: &Vec<N>) -> N
    where N: Add<Output=N> + Default
{
    ...
}
```

Great. But Rust still complains about this line of code:

```
total = total + *v;  // error: cannot move `*v` out of borrowed content
```

This one might be a real puzzle, even though by now we're familiar with all these terms. Yes, `v` is a reference, and yes, it would be illegal to move the value `*v` out of its vector. But numbers are copyable. So what's the problem?

The answer is that *Rust doesn't know* `*v` is a number. In fact, it isn't—the type `N` can be any type that satisfies the bounds we've given it. If we also want `N` to be a copyable type, we must say so:

```
where N: Add<Output=N> + Default + Copy
```

With this, the code compiles and runs. The final code looks like this:

```
use std::ops::Add;

fn sum<N>(values: &Vec<N>) -> N
    where N: Add<Output=N> + Default + Copy
{
    let mut total = N::default();
    for v in values {
        total = total + *v;
    }
    total
}

#[test]
fn test_sum() {
    assert_eq!(sum(&vec![1, 2, 3, 4]), 10);
    assert_eq!(sum(&vec![53.0, 35.0]), 88.0);
}
```

This occasionally happens in Rust: there is a period of intense arguing with the compiler, at the end of which the code looks rather nice, as if it had been a breeze to write, and runs beautifully.

What we've been doing here is reverse-engineering the bounds on `N`, using the compiler to guide and check our work. The reason it was a bit of a pain is that there wasn't a single `Number` trait in the standard library that included all the operators and methods we wanted to use. As it happens, there's a popular open source crate called `num` which defines such a trait! Had we known, we could have added `num` to our `Cargo.toml` and written:

```
use num::Num;

fn sum<N: Num + Copy>(values: &Vec<N>) -> N {
    let mut total = N::zero();
    for v in values {
        total = total + *v;
    }
    total
}
```

Just as in object-oriented programming, the right interface makes everything nice, in generic programming, the right trait makes everything nice.

Still, why go to all this trouble? Why didn't Rust's designers make the generics more like C++ templates, where the constraints are left implicit in the code, a la "duck typing"?

One advantage of Rust's approach is forward compatibility of generic code. You can change the implementation of a public generic function or method, and if you didn't change the signature, you haven't broken any of its users.

Another advantage of bounds is that when you do get a compiler error, at least the compiler can tell you where the trouble is. C++ compiler error messages involving templates can be much longer than Rust's, pointing at many different lines of code, because the compiler has no way to tell who's to blame for a problem: the template—or its caller, which might also be a template—or *that* template's caller...

Perhaps the most important advantage of writing out the bounds explicitly is simply that they are there, in the code and in the documentation. You can look at the signature of a generic function in Rust and see exactly what kind of arguments it accepts. The same can't be said for templates. The work that goes into fully documenting argument types in C++ libraries like Boost is even *more* arduous than what we went through here. The Boost developers don't have a compiler that checks their work.

Conclusion

Traits are one of the main organizing features in Rust, and with good reason. There's nothing better to design a program or library around than a good interface.

This chapter was a blizzard of syntax, rules, and explanations. Now that we've laid a foundation, we can start talking about the many ways traits and generics are used in Rust code. The fact is, we've only begun to scratch the surface. The next chapter covers common traits provided by the standard library. Upcoming chapters cover closures, iterators, input/output, and concurrency. Traits and generics play a central role in all of these topics.

CHAPTER 11

Built-in traits

Some Rust language features use specific traits to decide how to handle the types they encounter. For example, the `+` operator uses the `std::ops::Add` trait to add its operands, and a `for` loop uses the `Iterator` and `IntoIterator` traits to iterate over its operand. You can implement these traits for your own types, and extend arithmetic, `for` loops, and other parts of Rust to handle your own types in a natural way. We call these *built-in traits*. The effect is very much like operator overloading in C++, although Rust's version is a bit more restrictive.

Built-in traits fall into a few categories depending on what part of the language they support, as shown in Table 11-1. The remaining sections of this chapter cover each category in turn.

Built-in traits for arithmetic and bitwise operators

You can use built-in traits to extend the behavior of Rust's arithmetic and bitwise operators. For example, in the Mandelbrot set plotter we showed in Chapter 2, we used the `num` crate's `Complex` type to represent a number on the complex plane:

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

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

Table 11-1. Summary of built-in traits

category	trait	infix operator
arithmetic operations	`std::ops::Add`	`x + y`
	`std::ops::Sub`	`x - y`
	`std::ops::Mul`	`x * y`
	`std::ops::Div`	`x / y`
	`std::ops::Rem`	`x % y`
bitwise operations	`std::ops::BitAnd`	`x & y`
	`std::ops::BitOr`	`x | y`
	`std::ops::BitXOr`	`x ^ y`
	`std::ops::Shl`	`x << y`
	`std::ops::Shr`	`x >> y`
compound assignment arithmetic operations	`std::ops::AddAssign`	`x += y`
	`std::ops::SubAssign`	`x -= y`
	`std::ops::MulAssign`	`x *= y`
	`std::ops::DivAssign`	`x /= y`
	`std::ops::RemAssign`	`x %= y`
compound assignment bitwise operations	`std::ops::BitAndAssign`	`x &= y`
	`std::ops::BitOrAssign`	`x |= y`
	`std::ops::BitXorAssign`	`x ^= y`
	`std::ops::ShlAssign`	`x <<= y`
	`std::ops::ShrAssign`	`x >>= y`
comparison	`std::cmp::PartialEq`	`x == y`, `x != y`
	`std::cmp::PartialOrd`	`x < y`, `x <= y`, `x > y`, `x >= y`
indexing	`std::ops::Index`	`x[y]`, `&x[y]`
	`std::ops::IndexMut`	`x[y] = z`, `&mut x[y]`
iteration	`std::iter::IntoIterator`	`for v in e { ... }`
function application	`std::ops::Fn`	`x()`, callee needs only to read from closure
	`std::ops::FnMut`	`x()`, callee needs to read and modify closure
	`std::ops::FnOnce`	`x()`, callee needs to take ownership of closure
dereferencing, reference coercion	`std::ops::Deref`	`*x`, converting `&T` to `&U`
dropping values	`std::ops::Drop`	whenever a value is dropped

Given two `Complex<i32>` values, we can perform arithmetic on them using the same syntax we would for any other numeric type:

```
let z = Complex { re: -2, im: 6 };
let c = Complex { re:  1, im: 2 };
assert_eq!(z + c,    Complex { re: -1, im: 8 });
assert_eq!(z * c,    Complex { re: -14, im: 2 });
```

In Rust, the expression `z + c` is actually shorthand for `z.add(c)`, a call to the `add` method of the standard library's `std::ops::Add` trait. To make the expression `z + c` work for `Complex` values, the `num` crate implements `std::ops::Add` for `Complex`. Similar traits cover the other operators: `z * c` is shorthand for `z.mul(c)`, a method from the `std::ops::Mul` trait; `std::ops::Neg` covers the prefix negation operator `-`; and so on.

If you want to actually try writing out `z.add(c)`, you'll need to bring the `Add` trait into scope, so that its method is visible. That done, you can treat all arithmetic as function calls[1]:

```
use std::ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

The definition of `std::ops::Add` is simple:

```
trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Implementations must define `Output`, the type of the result. Then, they must implement the trait's `add` method, which has the type you'd expect: two arguments of type `Self` and `RHS`, and a result of the claimed `Output` type. Since `RHS` defaults to `Self`, writing only `Add` with no type parameters refers to adding two values of the same type. (If this all looks familiar, it may be because `Add` follows the same pattern as the `Mul` trait we presented in the Chapter 10.)

So to add two `Complex<i32>` values, `Complex<i32>` must implement `Add<Complex<i32>>`, or more briefly, `Add`:

```
use std::ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

Of course, we shouldn't have to implement `Add` separately for `Complex<i32>`, `Complex<f32>`, `Complex<f64>`, and so on. All the definitions would look exactly the same except for the types involved, so we should be able to write a single generic imple-

mentation that covers all types `Complex<T>`, as long as `T` itself is a type that can be added:

```
use std::ops::Add;

impl<T> Add for Complex<T>
    where T: Add<Output=T>
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

The clause `where T: Add<Output=T>` restricts `T` to types that can be added to themselves, yielding another `T` value. This is reasonable, but in general, the `Add` trait doesn't require both operands of the `+` operator to have the same type, nor does it constrain the result type. So the maximally generic implementation would let the right hand operand's type vary independently of the left's, and produce a `Complex` value of whatever component type that addition produces:

```
use std::ops::Add;

impl<L, R, O> Add<Complex<R>> for Complex<L>
    where L: Add<R, Output=O>
{
    type Output = Complex<O>;
    fn add(self, rhs: Complex<R>) -> Self::Output {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

Although the `Add` trait supports it, in practice, Rust tends to avoid implementing addition where the operands have different types. Implementations of `Add` usually have `Self` as their `RHS` and `Output` types: the implementing type can only be added with itself, and the addition yields another value of the same type. So even in our maximally generic implementation above, since `L` must implement `Add<R, Output=O>`, it usually follows that `L`, `R` and `O` must all the same type. In the end, this maximally generic version may not be much more useful than the prior, simpler generic definition.

Rust's built-in traits for arithmetic and bitwise operations come in three groups: unary operators, binary operators, and compound assignment operators. Within each group, the traits and their methods all have the same form, so we'll cover one example from each.

Unary operators

Aside from the dereferencing operator `*`, which we'll cover separately in ???, Rust has two unary operators that you can overload, shown in Table 11-2.

Table 11-2. Built-in traits for unary operators

trait name	expression	equivalent expression
std::ops::Neg	-x	x.neg()
std::ops::Not	!x	x.not()

These traits' definitions are simple:

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

Note that Rust's `!` operator complements `bool` values, and also performs a bitwise complement (that is, flips the bits) of integer types; it's a combination of the `!` and `~` operators from C and C++.

Here's how we might write a generic implementation of negation for `Complex` values:

```
use std::ops::Neg;

impl<T, O> Neg for Complex<T>
    where T: Neg<Output=O>
{
    type Output = Complex<O>;
    fn neg(self) -> Complex<O> {
        Complex { re: -self.re, im: -self.im }
    }
}
```

Binary operators

Rust's binary arithmetic and bitwise operators and their corresponding built-in traits appear in Table 11-3:

Table 11-3. Built-in traits for binary operators

category	trait name	expression	equivalent expression
arithmetic operations	`std::ops::Add`	`x + y`	`x.add(y)`
	`std::ops::Sub`	`x - y`	`x.sub(y)`
	`std::ops::Mul`	`x * y`	`x.mul(y)`
	`std::ops::Div`	`x / y`	`x.div(y)`
	`std::ops::Rem`	`x % y`	`x.rem(y)`
bitwise operations	`std::ops::BitAnd`	`x & y`	`x.bitand(y)`
	`std::ops::BitOr`	`x | y`	`x.bitor(y)`
	`std::ops::BitXor`	`x ^ y`	`x.bitxor(y)`
	`std::ops::Shl`	`x << y`	`x.shl(y)`
	`std::ops::Shr`	`x >> y`	`x.shr(y)`

All of the traits here have the same general form; the definition of `std::ops::BitXor`, for the `^` operator, looks like this:

```
trait BitXor<RHS = Self> {
    type Output;
    fn bitxor(self, rhs: RHS) -> Self::Output;
}
```

We showed an example implementation of a trait from this category, `std::ops::Add`, at the beginning of this section.

Compound assignment operators

A compound assignment expression is one like `x += y` or `x &= y`: it takes two operands, performs some operation on them like addition or a bitwise AND, and stores the result back in the left operand. In Rust, the value of a compound assignment expression is always `()`, never the value stored.

Many languages have operators like these, and usually define them as shorthand for expressions like `x = x + y` or `x = x & y`. However, Rust doesn't take that approach. Instead, `x += y` is shorthand for the method call `x.add_assign(y)`, where `add_assign` is the sole method of the `std::ops::AddAssign` trait:

```
trait AddAssign<RHS = Self> {
    fn add_assign(&mut self, RHS);
}
```

A generic implementation of `AddAssign` for our `Complex` type is straightforward:

```
use std::ops::AddAssign;

impl<T> AddAssign for Complex<T>
    where T: AddAssign<T>
```

```
{
    fn add_assign(&mut self, rhs: Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}
```

Table 11-4 shows all of Rust's compound assignment operators, and the built-in traits that implement them.

Table 11-4. Built-in traits for compound assignment operators

category	trait name	expression	equivalent expression
arithmetic operations	`std::ops::AddAssign`	`x += y`	`x.add_assign(y)`
	`std::ops::SubAssign`	`x -= y`	`x.sub_assign(y)`
	`std::ops::MulAssign`	`x *= y`	`x.mul_assign(y)`
	`std::ops::DivAssign`	`x /= y`	`x.div_assign(y)`
	`std::ops::RemAssign`	`x %= y`	`x.rem_assign(y)`
bitwise operations	`std::ops::BitAndAssign`	`x &= y`	`x.bitand_assign(y)`
	`std::ops::BitOrAssign`	`x |= y`	`x.bitor_assign(y)`
	`std::ops::BitXorAssign`	`x ^= y`	`x.bitxor_assign(y)`
	`std::ops::ShlAssign`	`x <<= y`	`x.shl_assign(y)`
	`std::ops::ShrAssign`	`x >>= y`	`x.shr_assign(y)`

The built-in trait for a compound assignment operator is completely independent of the built-in trait for its binary operator. Implementing `std::ops::Add` does not automatically implement `std::ops::AddAssign`; if a type does not implement `AddAssign`, it must not appear as the left-hand operand of a `+=` operator, even if it does implement `Add`. Rust's standard library cannot supply a provisional generic implementation of `AddAssign` in terms of `Add`, because it would then conflict with any implementation the user might want to provide.

Using distinct traits for the compound assignment operators makes it possible to use them on non-`Copy` values stored in locations that can't be moved from. For example, suppose we have an arbitrary-precision integer type, `BitInt`, which allocates memory from the heap to store as many digits as it needs. Since `BigInt` owns memory on the heap, it can't be `Copy`. So if we have a vector `v` of `BigInt` values, we can't write:

```
v[i] = v[i] + a;
```

The addition operator takes its operands by value, meaning that it will try to take ownership of `v[i]`—but you can't move an individual element out of a vector, even if you promise to put it back immediately. However, this does work:

```
v[i] += a;
```

This is shorthand for `v[i].mul_assign(a)`. Since `mul_assign` takes a mutable reference to its `self` value, the compound assignment simply borrows a mutable reference to `v[i]` and performs the addition in place. The element is never moved out of the vector, so this is fine.

Built-in traits for equality tests

Rust's equality operators, `==` and `!=`, are shorthand for calls to the `std::cmp::PartialEq` trait's `eq` and `ne` methods:

```
assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));
```

Here's the definition of `std::cmp::PartialEq`:

```
pub trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { !self.eq(other) }
}
```

Since `ne` has a default definition, you only need to define `eq` to implement the `PartialEq` trait. Here's a complete implementation for `Complex`:

1. Lisp programmers will be pleased!↩